# Практическая работа №11 «JavaScript. Массивы. Методы массивов. Деструктуризация. Рекурсия и стек. Глобальный объект. Планирование. Декораторы. Флаги и дескрипторы. Геттеры и сеттеры. Прототипное наследование. Классы»

# Массивы

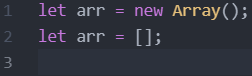
Объекты позволяют хранить данные со строковыми ключами. Но довольно часто мы понимаем, что нам необходима упорядоченная коллекция данных, в которой присутствуют 1-й, 2-й, 3-й элементы и т.д. Например, она понадобится нам для хранения списка чего-либо: пользователей, товаров, элементов HTML и т.д.

В этом случае использовать объект неудобно, так как он не предоставляет методов управления порядком элементов. Мы не можем вставить новое свойство «между» уже существующими. Объекты просто не предназначены для этих целей.

Для хранения упорядоченных коллекций существует особая структура данных, которая называется массив, Array.

## **[Объявление](https://learn.javascript.ru/array" \l "obyavlenie)**

Существует два варианта синтаксиса для создания пустого массива:

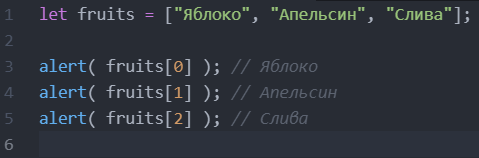


Практически всегда используется второй вариант синтаксиса. В скобках мы можем указать начальные значения элементов:

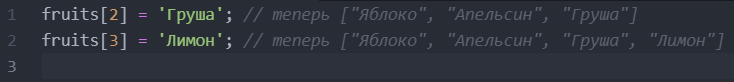


Элементы массива нумеруются, начиная с нуля.

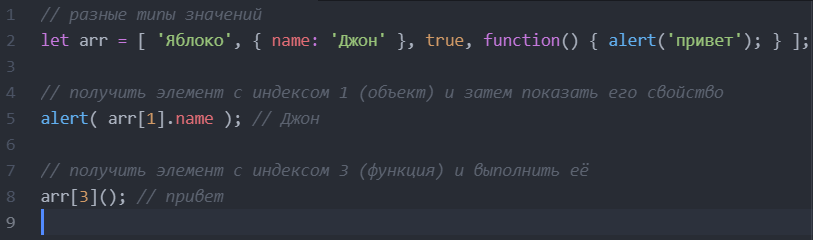
Мы можем получить элемент, указав его номер в квадратных скобках:



Мы можем заменить элемент или добавить новый к существующему массиву:



В массиве могут храниться элементы любого типа. Например:



## **[Методы pop/push, shift/unshift](https://learn.javascript.ru/array" \l "metody-pop-push-shift-unshift)**

[Очередь](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

* push добавляет элемент в конец.
* shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

Массивы поддерживают обе операции.

На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране.

Существует и другой вариант применения для массивов – структура данных, называемая [стек](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA).

Она поддерживает два вида операций:

* push добавляет элемент в конец.
* pop удаляет последний элемент.

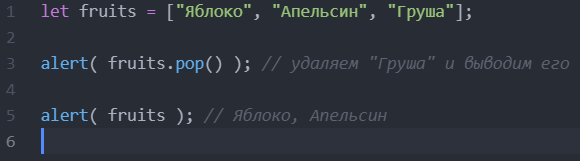
Таким образом, новые элементы всегда добавляются или удаляются из «конца». Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху:

* Массивы в JavaScript могут работать и как очередь, и как стек. Мы можем добавлять/удалять элементы как в начало, так и в конец массива.
* В компьютерных науках структура данных, делающая это возможным, называется [двусторонняя очередь](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D1%83%D1%85%D1%81%D1%82%D0%BE%D1%80%D0%BE%D0%BD%D0%BD%D1%8F%D1%8F_%D0%BE%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C).

**Методы, работающие с концом массива:**

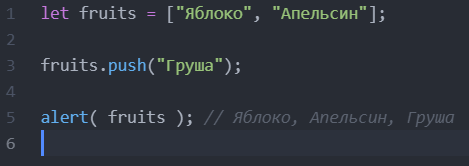
**pop**

Удаляет последний элемент из массива и возвращает его:

И fruits.pop() и fruits.at(-1) возвращают последний элемент массива, но fruits.pop() также изменяет массив, удаляя его.

**push**

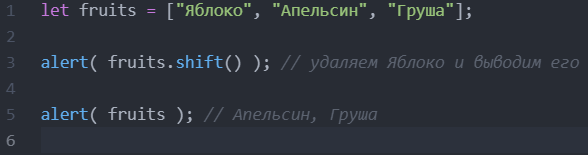
Добавляет элемент в конец массива:

Вызов fruits.push(...) равнозначен fruits[fruits.length] = ....

**Методы, работающие с началом массива:**

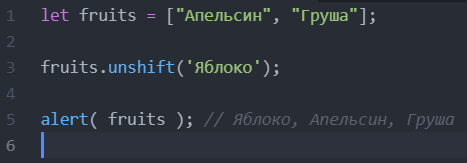
**shift**

Удаляет из массива первый элемент и возвращает его:



**unshift**

Добавляет элемент в начало массива:



Методы push и unshift могут добавлять сразу несколько элементов:

## 

## [**Внутреннее устройство массива**](https://learn.javascript.ru/array#vnutrennee-ustroystvo-massiva)

Массив – это особый подвид объектов. Квадратные скобки, используемые для того, чтобы получить доступ к свойству arr[0] – это по сути обычный синтаксис доступа по ключу, как obj[key], где в роли obj у нас arr, а в качестве ключа – числовой индекс.

Массивы расширяют объекты, так как предусматривают специальные методы для работы с упорядоченными коллекциями данных, а также свойство length. Но в основе всё равно лежит объект.

Следует помнить, что в JavaScript существует 8 основных типов данных. Массив является объектом и, следовательно, ведёт себя как объект.

Варианты неправильного применения массива:

* Добавление нечислового свойства, например: arr.test = 5.
* Создание «дыр», например: добавление arr[0], затем arr[1000] (между ними ничего нет).
* Заполнение массива в обратном порядке, например: arr[1000], arr[999] и т.д.

Массив следует считать особой структурой, позволяющей работать с упорядоченными данными. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому, пожалуйста, используйте их именно в таких случаях. Если вам нужны произвольные ключи, вполне возможно, лучше подойдёт обычный объект {}.

## **[Эффективность](https://learn.javascript.ru/array" \l "effektivnost)**

Методы push/pop выполняются быстро, а методы shift/unshift – медленно.

Например, операция shift должна выполнить 3 действия:

1. Удалить элемент с индексом 0.
2. Сдвинуть все элементы влево, заново пронумеровать их, заменив 1 на 0, 2 на 1 и т.д.
3. Обновить свойство length .

**Чем больше элементов содержит массив, тем больше времени потребуется для того, чтобы их переместить, больше операций с памятью.**

То же самое происходит с unshift: чтобы добавить элемент в начало массива, нам нужно сначала сдвинуть существующие элементы вправо, увеличивая их индексы.

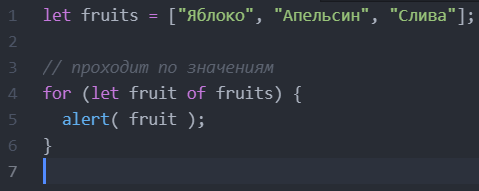
push/pop не нужно ничего перемещать. Чтобы удалить элемент в конце массива, метод pop очищает индекс и уменьшает значение length.

**Метод pop не требует перемещения, потому что остальные элементы остаются с теми же индексами. Именно поэтому он выполняется очень быстро.**

Аналогично работает метод push.

## **[Перебор элементов](https://learn.javascript.ru/array" \l "perebor-elementov)**

Одним из самых старых способов перебора элементов массива является цикл for по цифровым индексам, но для массивов возможен и другой вариант цикла, for..of:



Цикл for..of не предоставляет доступа к номеру текущего элемента, только к его значению, но в большинстве случаев этого достаточно.

## **[Многомерные массивы](https://learn.javascript.ru/array" \l "mnogomernye-massivy)**

Массивы могут содержать элементы, которые тоже являются массивами. Это можно использовать для создания многомерных массивов, например, для хранения матриц:

## 

Вызов new Array(number) создаёт массив с заданной длиной, но без элементов.

* Свойство length отражает длину массива или, если точнее, его последний цифровой индекс плюс один. Длина корректируется автоматически методами массива.
* Если мы уменьшаем length вручную, массив укорачивается.

Получение элементов:

* Мы можем получить элемент по его индексу, например arr[0].
* Также мы можем использовать метод at(i) для получения элементов с отрицательным индексом, для отрицательных значений i, он отступает от конца массива. В остальном он работает так же, как arr[i], если i >= 0.

Мы можем использовать массив как двустороннюю очередь, используя следующие операции:

* push(...items)добавляет items в конец массива.
* pop() удаляет элемент в конце массива и возвращает его.
* shift() удаляет элемент в начале массива и возвращает его.
* unshift(...items) добавляет items в начало массива.

Чтобы пройтись по элементам массива:

* for (let i=0; i<arr.length; i++) – работает быстрее всего, совместим со старыми браузерами.
* for (let item of arr) – современный синтаксис только для значений элементов (к индексам нет доступа).
* for (let i in arr) – никогда не используйте для массивов!

# Методы массивов

Массивы предоставляют множество методов.

## [Добавление/удаление элементов](https://learn.javascript.ru/array-methods" \l "dobavlenie-udalenie-elementov)

Методы, которые добавляют и удаляют элементы из начала или конца:

* arr.push(...items) – добавляет элементы в конец,
* arr.pop() – извлекает элемент из конца,
* arr.shift() – извлекает элемент из начала,
* arr.unshift(...items) – добавляет элементы в начало.

Удаление элемента из массива?

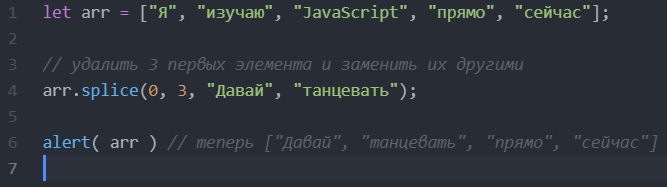
Метод [arr.splice(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы.

Его синтаксис:

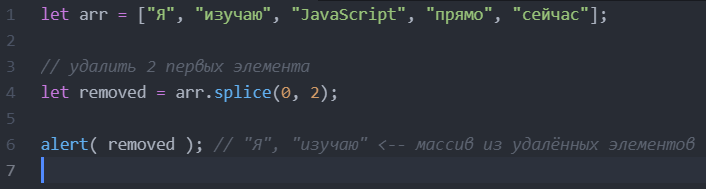


Он начинает с позиции index, удаляет deleteCount элементов и вставляет elem1, ..., elemN на их место. Возвращает массив из удалённых элементов.

Пример:



Здесь видно, что splice возвращает массив из удалённых элементов:



**Отрицательные индексы разрешены**

В этом и в других методах массива допускается использование отрицательного индекса. Он позволяет начать отсчёт элементов с конца, как тут:

### 

### [slice](https://learn.javascript.ru/array-methods#slice)

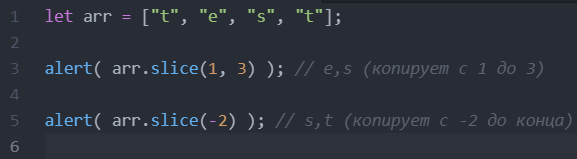
Метод [arr.slice](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) намного проще, чем похожий на него arr.splice.

Его синтаксис:



Он возвращает новый массив, в который копирует элементы, начиная с индекса start и до end (не включая end). Оба индекса start и end могут быть отрицательными. В таком случае отсчёт будет осуществляться с конца массива. Это похоже на строковый метод str.slice, но вместо подстрок возвращает подмассивы.

Например:



Можно вызвать slice и вообще без аргументов: arr.slice() создаёт копию массива arr. Это часто используют, чтобы создать копию массива для дальнейших преобразований, которые не должны менять исходный массив.

### [concat](https://learn.javascript.ru/array-methods" \l "concat)

Метод [arr.concat](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/concat) создаёт новый массив, в который копирует данные из других массивов и дополнительные значения.

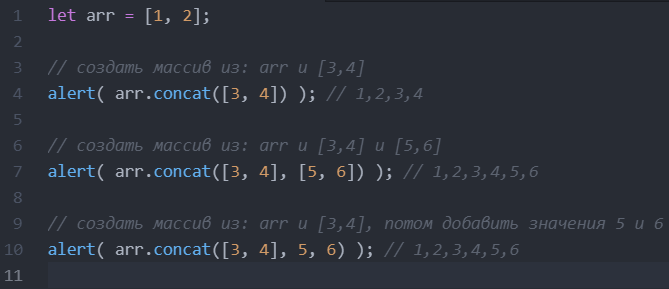
Его синтаксис:



Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями. В результате мы получаем новый массив, включающий в себя элементы из arr, а также arg1, arg2 и т.д.

Если аргумент argN – массив, то все его элементы копируются. Иначе скопируется сам аргумент.

Например:

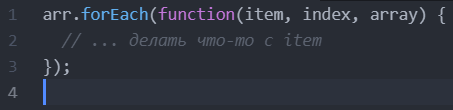


Обычно он копирует только элементы из массивов. Другие объекты, даже если они выглядят как массивы, добавляются как есть.

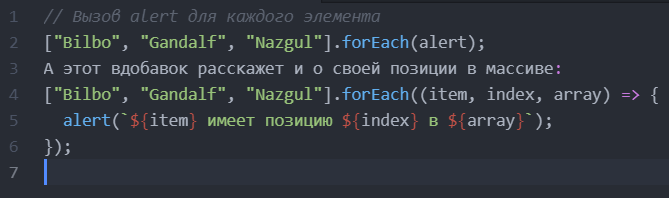
## [Перебор: forEach](https://learn.javascript.ru/array-methods" \l "perebor-foreach)

Метод [arr.forEach](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) позволяет запускать функцию для каждого элемента массива.

Его синтаксис:



Например, этот код выведет на экран каждый элемент массива:



Результат функции (если она вообще что-то возвращает) отбрасывается и игнорируется.

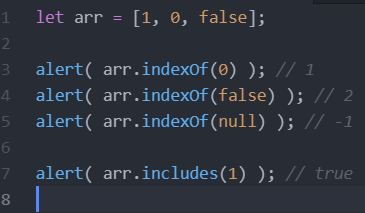
## [Поиск в массиве](https://learn.javascript.ru/array-methods" \l "poisk-v-massive)

## [indexOf/lastIndexOf и includes](https://learn.javascript.ru/array-methods#indexof-lastindexof-i-includes)

Методы [arr.indexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf), [arr.lastIndexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/lastIndexOf) и [arr.includes](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/includes) имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

* arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
* arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.
* arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

Например:



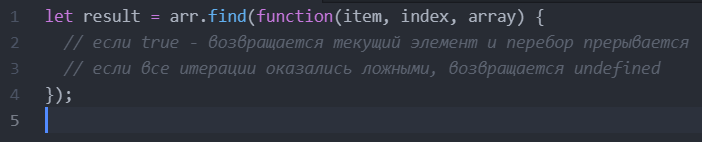
Если мы хотим проверить наличие элемента, и нет необходимости знать его точный индекс, тогда предпочтительным является arr.includes. Кроме того, очень незначительным отличием includes является то, что он правильно обрабатывает NaN в отличие от indexOf/lastIndexOf:

### [find и findIndex](https://learn.javascript.ru/array-methods#find-i-findindex)

Представьте, что у нас есть массив объектов. Как нам найти объект с определённым условием?

Здесь пригодится метод [arr.find](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/find).

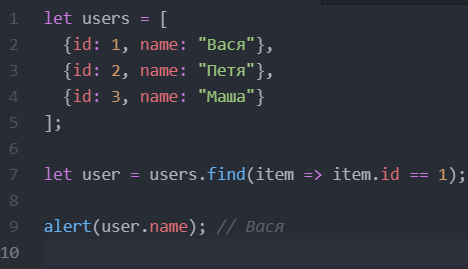
Его синтаксис таков:

Функция вызывается по очереди для каждого элемента массива:

* item – очередной элемент.
* index – его индекс.
* array – сам массив.

Если функция возвращает true, поиск прерывается и возвращается item. Если ничего не найдено, возвращается undefined.

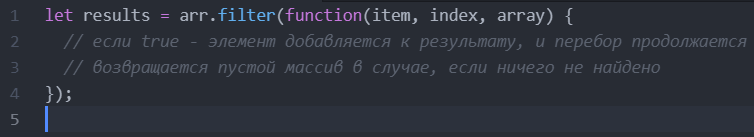
Например, у нас есть массив пользователей, каждый из которых имеет поля id и name. Попробуем найти того, кто с id == 1:



Обратите внимание, что в данном примере мы передаём find функцию item => item.id == 1, с одним аргументом. Это типично, дополнительные аргументы этой функции используются редко. Метод [arr.findIndex](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex) – по сути, то же самое, но возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

### [filter](https://learn.javascript.ru/array-methods" \l "filter)

Метод find ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт true. На тот случай, если найденных элементов может быть много, предусмотрен метод [arr.filter(fn)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/filter). Синтаксис этого метода схож с find, но filter возвращает массив из всех подходящих элементов:



## [Преобразование массива](https://learn.javascript.ru/array-methods" \l "preobrazovanie-massiva)

### [map](https://learn.javascript.ru/array-methods" \l "map)

Метод [arr.map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/map) является одним из наиболее полезных и часто используемых. Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции. Например, здесь мы преобразуем каждый элемент в его длину:

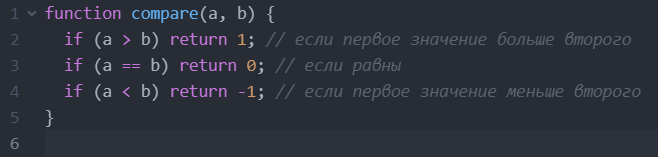
### [sort(fn)](https://learn.javascript.ru/array-methods#sort-fn)

Вызов [arr.sort()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) сортирует массив на месте, меняя в нём порядок элементов.

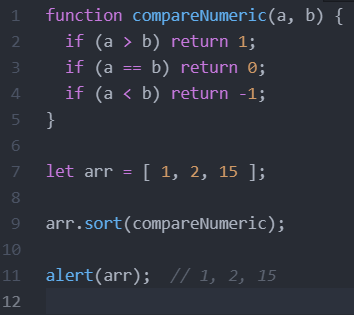
Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr.

**По умолчанию элементы сортируются как строки.**

Буквально, элементы преобразуются в строки при сравнении. Для строк применяется лексикографический порядок. Чтобы использовать наш собственный порядок сортировки, нам нужно предоставить функцию в качестве аргумента arr.sort(). Функция должна для пары значений возвращать:



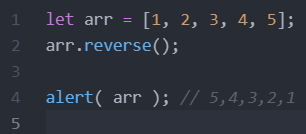
Например, для сортировки чисел:



### [reverse](https://learn.javascript.ru/array-methods" \l "reverse)

Метод [arr.reverse](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse) меняет порядок элементов в arr на обратный.

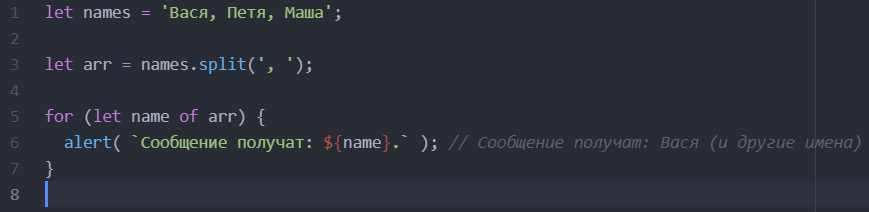
Например:



Он также возвращает массив arr с изменённым порядком элементов.

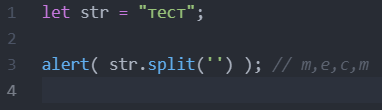
### [split и join](https://learn.javascript.ru/array-methods" \l "split-i-join)

Ситуация из реальной жизни. Мы пишем приложение для обмена сообщениями, и посетитель вводит имена тех, кому его отправить, через запятую: Вася, Петя, Маша. Но нам-то гораздо удобнее работать с массивом имён, чем с одной строкой. Метод [str.split(delim)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/split) именно это и делает. Он разбивает строку на массив по заданному разделителю delim.



**Разбивка по буквам**

Вызов split(s) с пустым аргументом s разбил бы строку на массив букв:



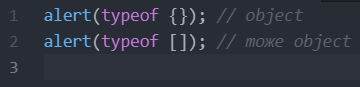
Вызов [arr.join(glue)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/join) делает в точности противоположное split. Он создаёт строку из элементов arr, вставляя glue между ними.

Например:

### 

## [Array.isArray](https://learn.javascript.ru/array-methods" \l "array-isarray)

Массивы не образуют отдельный тип языка. Они основаны на объектах. Поэтому typeof не может отличить простой объект от массива:



Однако массивы используются настолько часто, что для этого придумали специальный метод: [Array.isArray(value)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray). Он возвращает true, если value массив, и false, если нет.

## 

Небольшая шпаргалка по методам массива:

* Для добавления/удаления элементов:
  + push (...items) – добавляет элементы в конец,
  + pop() – извлекает элемент с конца,
  + shift() – извлекает элемент с начала,
  + unshift(...items) – добавляет элементы в начало.
  + splice(pos, deleteCount, ...items) – начиная с индекса pos, удаляет deleteCount элементов и вставляет items.
  + slice(start, end) – создаёт новый массив, копируя в него элементы с позиции start до end (не включая end).
  + concat(...items) – возвращает новый массив: копирует все члены текущего массива и добавляет к нему items. Если какой-то из items является массивом, тогда берутся его элементы.
* Для поиска среди элементов:
  + indexOf/lastIndexOf(item, pos) – ищет item, начиная с позиции pos, и возвращает его индекс или -1, если ничего не найдено.
  + includes(value) – возвращает true, если в массиве имеется элемент value, в противном случае false.
  + find/filter(func) – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается true.
  + findIndex похож на find, но возвращает индекс вместо значения.
* Для перебора элементов:
  + forEach(func) – вызывает func для каждого элемента. Ничего не возвращает.
* Для преобразования массива:
  + map(func) – создаёт новый массив из результатов вызова func для каждого элемента.
  + sort(func) – сортирует массив «на месте», а потом возвращает его.
  + reverse() – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
  + split/join – преобразует строку в массив и обратно.
  + reduce/reduceRight(func, initial) – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.
* Дополнительно:
  + Array.isArray(arr) проверяет, является ли arr массивом.

Обратите внимание, что методы sort, reverse и splice изменяют исходный массив.

Полный список есть в [справочнике MDN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array).

# Деструктурирующее присваивание

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array.

* Объекты позволяют нам создавать одну сущность, которая хранит элементы данных по ключам.
* Массивы позволяют нам собирать элементы данных в упорядоченный список.

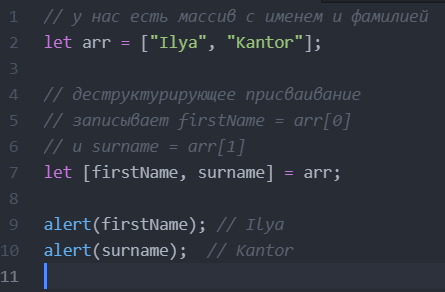
Но когда мы передаём их в функцию, то ей может понадобиться не объект/массив целиком, а элементы по отдельности.

Деструктурирующее присваивание – это специальный синтаксис, который позволяет нам «распаковать» массивы или объекты в несколько переменных, так как иногда они более удобны.

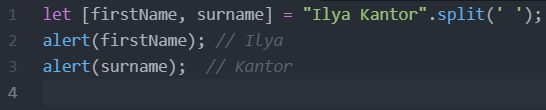
Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию и так далее. Скоро мы увидим это.

## **[Деструктуризация массива](https://learn.javascript.ru/destructuring-assignment" \l "destrukturizatsiya-massiva)**

Вот пример деструктуризации массива на переменные:



Теперь мы можем использовать переменные вместо элементов массива:



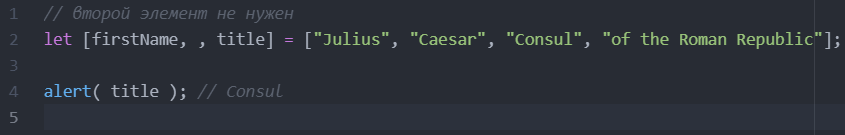
Как вы можете видеть, синтаксис прост. Однако есть несколько странных моментов. Давайте посмотрим больше примеров, чтобы лучше понять это.

**«Деструктуризация» не означает «разрушение».**

«Деструктурирующее присваивание» не уничтожает массив. Оно вообще ничего не делает с правой частью присваивания, его задача – только скопировать нужные значения в переменные.

**Пропускайте элементы, используя запятые**

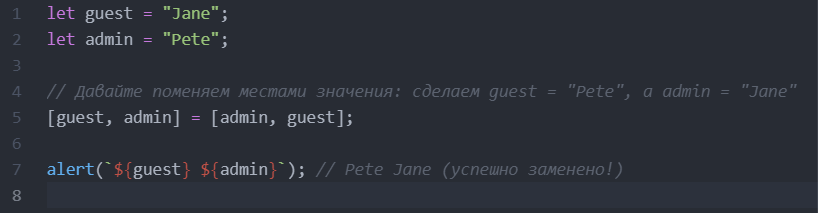
Нежелательные элементы массива также могут быть отброшены с помощью дополнительной запятой:

В примере выше второй элемент массива пропускается, а третий присваивается переменной title, оставшиеся элементы массива также пропускаются (так как для них нет переменных).

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

**Трюк обмена переменных**

Существует хорошо известный трюк для обмена значений двух переменных с использованием деструктурирующего присваивания:

Здесь мы создаём временный массив из двух переменных и немедленно деструктурируем его в порядке замены. Таким образом, мы можем поменять местами даже более двух переменных.

## **[Деструктуризация объекта](https://learn.javascript.ru/destructuring-assignment" \l "destrukturizatsiya-obekta)**

Деструктурирующее присваивание также работает с объектами.

Синтаксис:

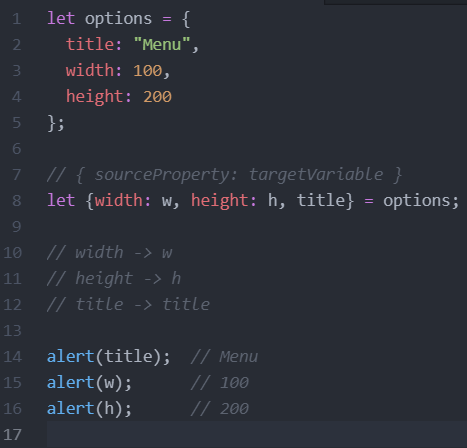


У нас есть существующий объект с правой стороны, который мы хотим разделить на переменные. Левая сторона содержит «шаблон» для соответствующих свойств. В простом случае это список названий переменных в {...}.

Например:

Свойства options.title, options.width и options.height присваиваются соответствующим переменным. Порядок не имеет значения.

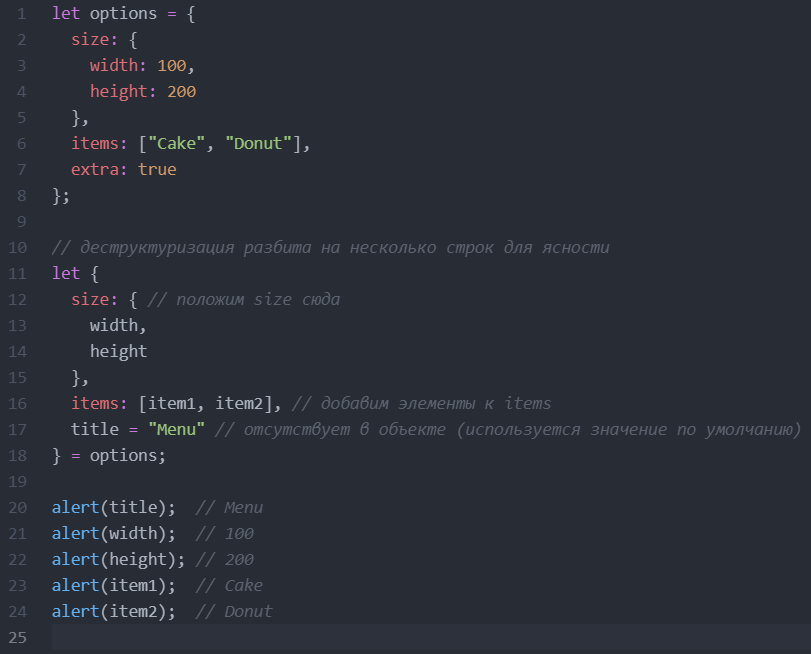
Если мы хотим присвоить свойство объекта переменной с другим названием, например, свойство options.width присвоить переменной w, то мы можем использовать двоеточие:



Двоеточие показывает «что : куда идёт». В примере выше свойство width сохраняется в переменную w, свойство height сохраняется в h, а title присваивается одноимённой переменной.

## **[Вложенная деструктуризация](https://learn.javascript.ru/destructuring-assignment" \l "vlozhennaya-destrukturizatsiya)**

Если объект или массив содержит другие вложенные объекты или массивы, то мы можем использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства. В приведённом ниже коде options хранит другой объект в свойстве size и массив в свойстве items. Шаблон в левой части присваивания имеет такую же структуру, чтобы извлечь данные из них:

Весь объект options, кроме свойства extra, которое в левой части отсутствует, присваивается в соответствующие переменные:

* В итоге у нас есть width, height, item1, item2 и title со значением по умолчанию.
* Заметим, что переменные для size и items отсутствуют, так как мы взяли сразу их содержимое.

## [Итого](https://learn.javascript.ru/destructuring-assignment#itogo)

* Деструктуризация позволяет разбивать объект или массив на переменные при присвоении.
* Полный синтаксис для объекта:

let {prop : varName = default, ...rest} = object

Cвойство prop объекта object здесь должно быть присвоено переменной varName. Если в объекте отсутствует такое свойство, переменной varName присваивается значение по умолчанию.

Свойства, которые не были упомянуты, копируются в объект rest.

* Полный синтаксис для массива:

let [item1 = default, item2, ...rest] = array

Первый элемент отправляется в item1; второй отправляется в item2, все остальные элементы попадают в массив rest.

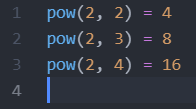
* Можно извлекать данные из вложенных объектов и массивов, для этого левая сторона должна иметь ту же структуру, что и правая.

# Рекурсия и стек

Рекурсия – это приём программирования, полезный в ситуациях, когда задача может быть естественно разделена на несколько аналогичных, но более простых задач. В процессе выполнения задачи в теле функции могут быть вызваны другие функции для выполнения подзадач. Частный случай подвызова – когда функция вызывает сама себя. Это как раз и называется рекурсией.

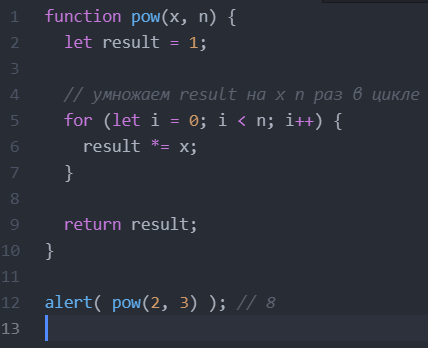
## [Два способа мышления](https://learn.javascript.ru/recursion" \l "dva-sposoba-myshleniya)

В качестве первого примера напишем функцию pow(x, n), которая возводит x в натуральную степень n. Иначе говоря, умножает x на само себя n раз.

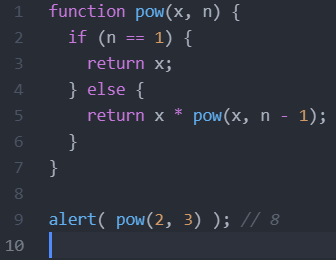


Рассмотрим два способа её реализации.

1. Итеративный способ: цикл for:



1. Рекурсивный способ: упрощение задачи и вызов функцией самой себя:



Обратите внимание, что рекурсивный вариант отличается принципиально.

Когда функция pow(x, n) вызывается, исполнение делится на две ветви:

if n==1 = x

/

pow(x, n) =

\

else = x \* pow(x, n - 1)

1. Если n == 1, тогда всё просто. Эта ветвь называется базой рекурсии, потому что сразу же приводит к очевидному результату: pow(x, 1) равно x.
2. Мы можем представить pow(x, n) в виде: x \* pow(x, n - 1). Что в математике записывается как: xn = x \* xn-1. Эта ветвь – шаг рекурсии: мы сводим задачу к более простому действию (умножение на x) и более простой аналогичной задаче (pow с меньшим n). Последующие шаги упрощают задачу всё больше и больше, пока n не достигает 1.

Итак, рекурсию используют, когда вычисление функции можно свести к её более простому вызову, а его – к ещё более простому и так далее, пока значение не станет очевидно. Рекурсивное решение задачи обычно короче, чем итеративное.

Максимальная глубина рекурсии ограничена движком JavaScript. Точно можно рассчитывать на 10000 вложенных вызовов, некоторые интерпретаторы допускают и больше, но для большинства из них 100000 вызовов – за пределами возможностей. Существуют автоматические оптимизации, помогающие избежать переполнения стека вызовов («оптимизация хвостовой рекурсии»), но они ещё не поддерживаются везде и работают только для простых случаев.

Это ограничивает применение рекурсии, но она всё равно широко распространена: для решения большого числа задач рекурсивный способ решения даёт более простой код, который легче поддерживать.

## [Контекст выполнения, стек](https://learn.javascript.ru/recursion" \l "kontekst-vypolneniya-stek)

Теперь мы посмотрим, как работают рекурсивные вызовы. Для этого заглянем «под капот» функций.

Информация о процессе выполнения запущенной функции хранится в её контексте выполнения (execution context).

[Контекст выполнения](https://tc39.github.io/ecma262/#sec-execution-contexts) – специальная внутренняя структура данных, которая содержит информацию о вызове функции. Она включает в себя конкретное место в коде, на котором находится интерпретатор, локальные переменные функции, значение this (мы не используем его в данном примере) и прочую служебную информацию. Один вызов функции имеет ровно один контекст выполнения, связанный с ним.

Когда функция производит вложенный вызов, происходит следующее:

* Выполнение текущей функции приостанавливается.
* Контекст выполнения, связанный с ней, запоминается в специальной структуре данных – стеке контекстов выполнения.
* Выполняются вложенные вызовы, для каждого из которых создаётся свой контекст выполнения.
* После их завершения старый контекст достаётся из стека, и выполнение внешней функции возобновляется с того места, где она была остановлена.

Для выполнения вложенного вызова JavaScript запоминает текущий контекст выполнения в стеке контекстов выполнения. Для любых функций процесс одинаков:

1. Текущий контекст «запоминается» на вершине стека.
2. Создаётся новый контекст для вложенного вызова.
3. Когда выполнение вложенного вызова заканчивается – контекст предыдущего вызова восстанавливается, и выполнение соответствующей функции продолжается.

**Любая рекурсия может быть переделана в цикл. Как правило, вариант с циклом будет эффективнее.**

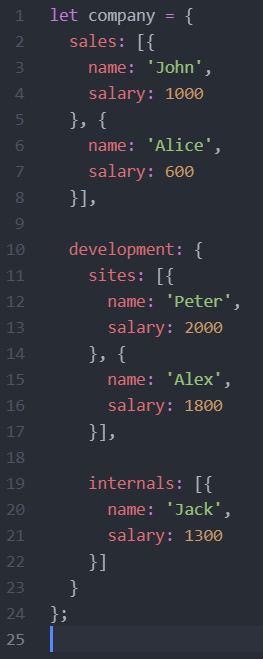
Но переделка рекурсии в цикл может быть нетривиальной, особенно когда в функции в зависимости от условий используются различные рекурсивные подвызовы, результаты которых объединяются, или когда ветвление более сложное. Оптимизация может быть ненужной и совершенно нестоящей усилий.

Часто код с использованием рекурсии более короткий, лёгкий для понимания и поддержки. Оптимизация требуется не везде, как правило, нам важен хороший код, поэтому она и используется.

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

Другим отличным применением рекурсии является рекурсивный обход.

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



Другими словами, в компании есть отделы.

* Отдел может состоять из массива работников. Например, в отделе sales работают 2 сотрудника: Джон и Алиса.
* Или отдел может быть разделён на подотделы, например, отдел development состоит из подотделов: sites и internals. В каждом подотделе есть свой персонал.
* Также возможно, что при росте подотдела он делится на подразделения (или команды).

Например, подотдел sites в будущем может быть разделён на команды siteA и siteB. И потенциально они могут быть разделены ещё. Этого нет на картинке, просто нужно иметь это в виду.

Теперь, допустим, нам нужна функция для получения суммы всех зарплат. Итеративный подход не прост, потому что структура довольно сложная. Первая идея заключается в том, чтобы сделать цикл for поверх объекта company с вложенным циклом над отделами 1-го уровня вложенности. Но затем нам нужно больше вложенных циклов для итераций над сотрудниками отделов второго уровня, таких как sites… А затем ещё один цикл по отделам 3-го уровня, которые могут появиться в будущем? Если мы поместим в код 3-4 вложенных цикла для обхода одного объекта, то это будет довольно некрасиво. Давайте попробуем рекурсию.

Как мы видим, когда наша функция получает отдел для подсчёта суммы зарплат, есть два возможных случая:

1. Либо это «простой» отдел с массивом – тогда мы сможем суммировать зарплаты в простом цикле.
2. Или это объект с N подотделами – тогда мы можем сделать N рекурсивных вызовов, чтобы получить сумму для каждого из подотделов, и объединить результаты.

Случай (1), когда мы получили массив, является базой рекурсии, тривиальным случаем.

Случай (2), при получении объекта, является шагом рекурсии. Сложная задача разделяется на подзадачи для подотделов. Они могут, в свою очередь, снова разделиться на подотделы, но рано или поздно это разделение закончится, и решение сведётся к случаю (1).

Алгоритм даже проще читается в виде кода:

Код краток и прост для понимания. В этом сила рекурсии. Она работает на любом уровне вложенности отделов.

Схема вызовов:

Принцип прост: для объекта {...} используются рекурсивные вызовы, а массивы [...] являются «листьями» дерева рекурсии, они сразу дают результат.

Обратите внимание, что в коде используются возможности, о которых мы говорили ранее:

* Метод arr.reduce из главы [Методы массивов](https://learn.javascript.ru/array-methods) для получения суммы элементов массива.
* Цикл for(val of Object.values(obj)) для итерации по значениям объекта: Object.values возвращает массив значений.

### [Связанный список](https://learn.javascript.ru/recursion" \l "svyazannyy-spisok)

Представьте себе, что мы хотим хранить упорядоченный список объектов.

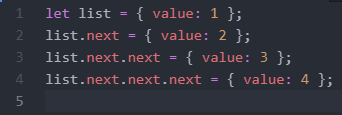
Естественным выбором будет массив, но у массивов есть недостатки. Операции «удалить элемент» и «вставить элемент» являются дорогостоящими. Например, операция arr.unshift(obj) должна переиндексировать все элементы, чтобы освободить место для нового obj, и, если массив большой, на это потребуется время. То же самое с arr.shift().

Единственные структурные изменения, не требующие массовой переиндексации – это изменения, которые выполняются с конца массива: arr.push/pop. Таким образом, массив может быть довольно медленным для больших очередей, когда нам приходится работать с его началом.

Или же, если нам действительно нужны быстрые вставка/удаление, мы можем выбрать другую структуру данных, называемую [связанный список](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA).

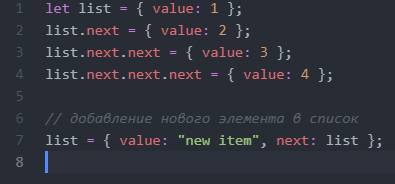
Элемент связанного списка определяется рекурсивно как объект с:

* value,
* next – свойство, ссылающееся на следующий элемент связанного списка или null, если это последний элемент.



Здесь мы можем ещё лучше увидеть, что есть несколько объектов, каждый из которых имеет value и next, указывающий на соседа. Переменная list является первым объектом в цепочке, поэтому, следуя по указателям next из неё, мы можем попасть в любой элемент.

Список можно легко разделить на несколько частей и впоследствии объединить обратно. И, конечно, мы можем вставить или удалить элементы из любого места. Например, для добавления нового элемента нам нужно обновить первый элемент списка:



Чтобы удалить элемент из середины списка, нужно изменить значение next предыдущего элемента:



list.next перепрыгнуло с 1 на значение 2. Значение 1 теперь исключено из цепочки. Если оно не хранится где-нибудь ещё, оно будет автоматически удалено из памяти. В отличие от массивов, нет перенумерации, элементы легко переставляются. Естественно, списки не всегда лучше массивов. В противном случае все пользовались бы только списками.

Главным недостатком является то, что мы не можем легко получить доступ к элементу по его индексу. В простом массиве: arr[n] является прямой ссылкой. Но в списке мы должны начать с первого элемента и перейти в next N раз, чтобы получить N-й элемент.

Списки могут быть улучшены:

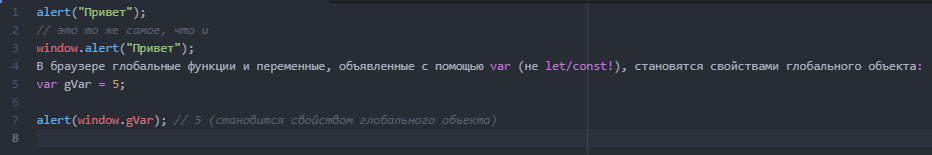
* Можно добавить свойство prev в дополнение к next для ссылки на предыдущий элемент, чтобы легко двигаться по списку назад.
* Можно также добавить переменную tail, которая будет ссылаться на последний элемент списка (и обновлять её при добавлении/удалении элементов с конца).
* Возможны другие изменения: главное, чтобы структура данных соответствовала нашим задачам с точки зрения производительности и удобства.

# Глобальный объект

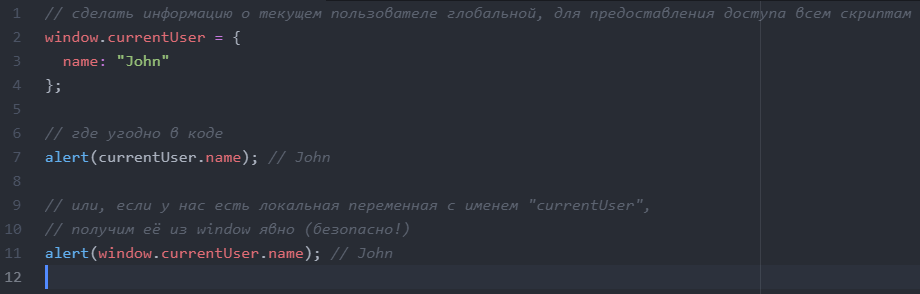
Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения. В браузере он называется window, в Node.js — global, в другой среде исполнения может называться иначе.

Недавно globalThis был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении. Он поддерживается во всех основных браузерах.

Ко всем свойствам глобального объекта можно обращаться напрямую:

То же самое касается функций, объявленных с помощью синтаксиса Function Declaration (выражения с ключевым словом function в основном потоке кода, не Function Expression)

Однако, в современных проектах, использующих [JavaScript-модули](https://learn.javascript.ru/modules), такого не происходит. Если свойство настолько важное, что вы хотите сделать его доступным для всей программы, запишите его в глобальный объект напрямую:



При этом обычно не рекомендуется использовать глобальные переменные. Следует применять их как можно реже. Дизайн кода, при котором функция получает входные параметры и выдаёт определённый результат, чище, надёжнее и удобнее для тестирования, чем когда используются внешние, а тем более глобальные переменные.

# Планирование: setTimeout и setInterval

Мы можем вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова».

Для этого существуют два метода:

* setTimeout позволяет вызвать функцию **один раз** через определённый интервал времени.
* setInterval позволяет вызывать функцию **регулярно**, повторяя вызов через определённый интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

## [setTimeout](https://learn.javascript.ru/settimeout-setinterval" \l "settimeout)

Синтаксис:



Параметры:

**func|code**

Функция или строка кода для выполнения. Обычно это функция. По историческим причинам можно передать и строку кода, но это не рекомендуется.

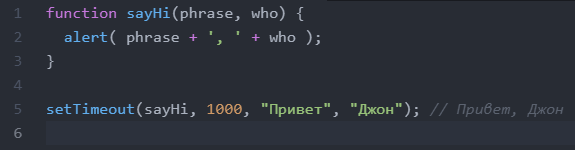
**delay**

Задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

**arg1, arg2…**

Аргументы, передаваемые в функцию (не поддерживается в IE9-)

Например, данный код вызывает sayHi() спустя одну секунду:



Если первый аргумент является строкой, то JavaScript создаст из неё функцию.

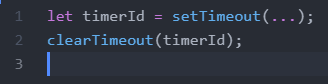
**Передавайте функцию, но не запускайте её**

Начинающие разработчики иногда ошибаются, добавляя скобки () после функции. Это не работает, потому что setTimeout ожидает ссылку на функцию. Здесь sayHi() запускает выполнение функции, и результат выполнения отправляется в setTimeout.

### [Отмена через clearTimeout](https://learn.javascript.ru/settimeout-setinterval" \l "otmena-cherez-cleartimeout)

Вызов setTimeout возвращает «идентификатор таймера» timerId, который можно использовать для отмены дальнейшего выполнения.

Синтаксис для отмены:

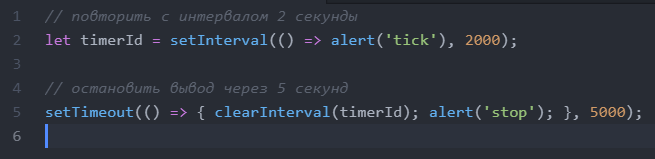


В коде ниже планируем вызов функции и затем отменяем его. В результате ничего не происходит.

## [setInterval](https://learn.javascript.ru/settimeout-setinterval" \l "setinterval)

Метод setInterval имеет такой же синтаксис как setTimeout. Все аргументы имеют такое же значение. Но отличие этого метода от setTimeout в том, что функция запускается не один раз, а периодически через указанный интервал времени. Чтобы остановить дальнейшее выполнение функции, необходимо вызвать clearInterval(timerId).

Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:



**Во время показа alert время тоже идёт**

* Методы setInterval(func, delay, ...args) и setTimeout(func, delay, ...args) позволяют выполнять func регулярно или только один раз после задержки delay, заданной в мс.
* Для отмены выполнения необходимо вызвать clearInterval/clearTimeout со значением, которое возвращают методы setInterval/setTimeout.
* Вложенный вызов setTimeout является более гибкой альтернативой setInterval. Также он позволяет более точно задать интервал между выполнениями.
* Планирование с нулевой задержкой setTimeout(func,0) или, что то же самое, setTimeout(func) используется для вызовов, которые должны быть исполнены как можно скорее, после завершения исполнения текущего кода.
* Браузер ограничивает 4-мя мс минимальную задержку между пятью и более вложенными вызовами setTimeout, а также для setInterval, начиная с 5-го вызова.

Обратим внимание, что все методы планирования не гарантируют точную задержку.

Например, таймер в браузере может замедляться по многим причинам:

* Перегружен процессор.
* Вкладка браузера в фоновом режиме.
* Работа ноутбука от аккумулятора.

Всё это может увеличивать минимальный интервал срабатывания таймера (и минимальную задержку) до 300 или даже 1000 мс в зависимости от браузера и настроек производительности ОС.

# Декораторы и переадресация вызова, call/apply

JavaScript предоставляет исключительно гибкие возможности по работе с функциями: они могут быть переданы в другие функции, использованы как объекты, и сейчас мы рассмотрим, как перенаправлять вызовы между ними и как их декорировать.

Декоратор – это обёртка вокруг функции, которая изменяет поведение последней. Основная работа по-прежнему выполняется функцией.

Обычно безопасно заменить функцию или метод декорированным, за исключением одной мелочи. Если исходная функция предоставляет свойства, такие как func.calledCount или типа того, то декорированная функция их не предоставит. Потому что это обёртка. Так что нужно быть осторожным в их использовании. Некоторые декораторы предоставляют свои собственные свойства.

Декораторы можно рассматривать как «дополнительные возможности» или «аспекты», которые можно добавить в функцию. Мы можем добавить один или несколько декораторов. И всё это без изменения кода оригинальной функции!

## [Прозрачное кеширование](https://learn.javascript.ru/call-apply-decorators" \l "prozrachnoe-keshirovanie)

Представим, что у нас есть функция slow(x), выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Другими словами, для одного и того же x она всегда возвращает один и тот же результат. Если функция вызывается часто, то, вероятно, мы захотим кешировать (запоминать) возвращаемые ею результаты, чтобы сэкономить время на повторных вычислениях.

Вместо того, чтобы усложнять slow(x) дополнительной функциональностью, мы заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование. Далее мы увидим, что в таком подходе масса преимуществ.

Вот код с объяснениями:



В коде выше cachingDecorator – это декоратор, специальная функция, которая принимает другую функцию и изменяет её поведение. Идея состоит в том, что мы можем вызвать cachingDecorator с любой функцией, в результате чего мы получим кеширующую обёртку. Это здорово, т.к. у нас может быть множество функций, использующих такую функциональность, и всё, что нам нужно сделать – это применить к ним cachingDecorator. Отделяя кеширующий код от основного кода, мы также сохраняем чистоту и простоту последнего. Результат вызова cachingDecorator(func) является «обёрткой», т.е. function(x) «оборачивает» вызов func(x) в кеширующую логику:

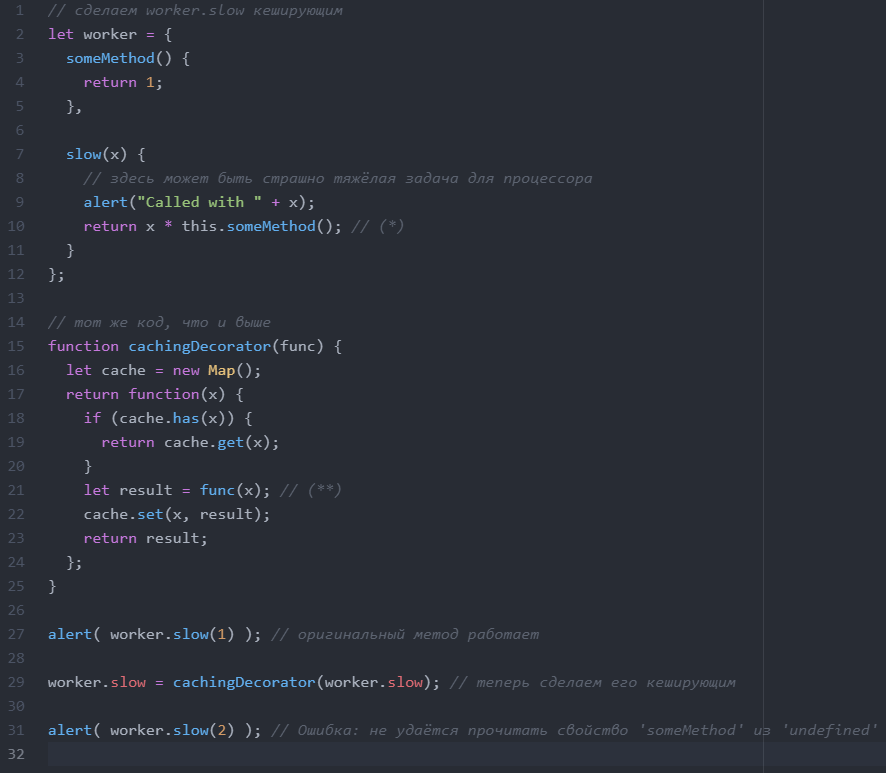
С точки зрения внешнего кода, обёрнутая функция slow по-прежнему делает то же самое. Обёртка всего лишь добавляет к её поведению аспект кеширования.

Подводя итог, можно выделить несколько преимуществ использования отдельной cachingDecorator вместо изменения кода самой slow:

* Функцию cachingDecorator можно использовать повторно. Мы можем применить её к другой функции.
* Логика кеширования является отдельной, она не увеличивает сложность самой slow (если таковая была).
* При необходимости мы можем объединить несколько декораторов (речь об этом пойдёт позже).

## [Применение «func.call» для передачи контекста.](https://learn.javascript.ru/call-apply-decorators" \l "primenenie-func-call-dlya-peredachi-konteksta)

Упомянутый выше кеширующий декоратор не подходит для работы с методами объектов. Например, в приведённом ниже коде worker.slow() перестаёт работать после применения декоратора:

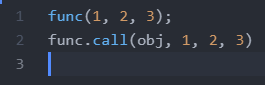
Ошибка возникает в строке (\*). Функция пытается получить доступ к this.someMethod и завершается с ошибкой. Причина в том, что в строке (\*\*) декоратор вызывает оригинальную функцию как func(x), и она в данном случае получает this = undefined. Декоратор передаёт вызов оригинальному методу, но без контекста. Следовательно – ошибка.

Существует специальный встроенный метод функции [func.call(context, …args)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/call), который позволяет вызывать функцию, явно устанавливая this.

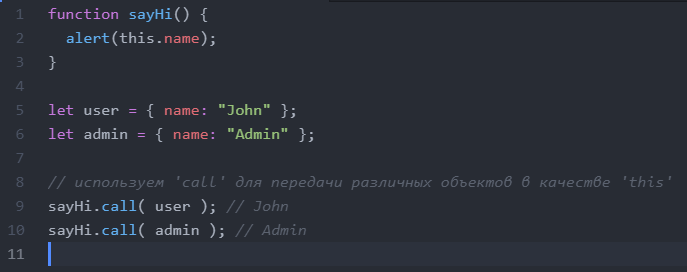
Синтаксис:



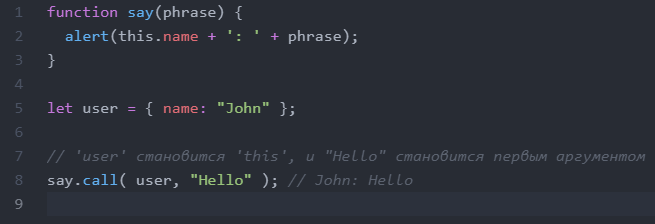
Он запускает функцию func, используя первый аргумент как её контекст this, а последующие – как её аргументы. Проще говоря, эти два вызова делают почти то же самое:



Они оба вызывают func с аргументами 1, 2 и 3. Единственное отличие состоит в том, что func.call ещё и устанавливает this равным obj. Например, в приведённом ниже коде мы вызываем sayHi в контексте различных объектов: sayHi.call(user) запускает sayHi, передавая this=user, а следующая строка устанавливает this=admin:



Здесь мы используем call для вызова say с заданным контекстом и фразой:



В нашем случае мы можем использовать call в обёртке для передачи контекста в исходную функцию:

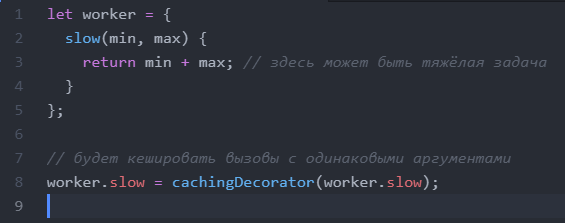
Теперь всё в порядке.

Чтобы всё было понятно, давайте посмотрим глубже, как передаётся this:

1. После декорации worker.slow становится обёрткой function (x) { ... }.
2. Так что при выполнении worker.slow(2) обёртка получает 2 в качестве аргумента и this=worker (так как это объект перед точкой).
3. Внутри обёртки, если результат ещё не кеширован, func.call(this, x) передаёт текущий this (=worker) и текущий аргумент (=2) в оригинальную функцию.

## [Переходим к нескольким аргументам с «func.apply»](https://learn.javascript.ru/call-apply-decorators" \l "perehodim-k-neskolkim-argumentam-s-func-apply)

Теперь давайте сделаем cachingDecorator ещё более универсальным. До сих пор он работал только с функциями с одним аргументом.



Здесь у нас есть две задачи для решения.

Во-первых, как использовать оба аргумента min и max для ключа в коллекции cache? Ранее для одного аргумента x мы могли просто сохранить результат cache.set(x, result) и вызвать cache.get(x), чтобы получить его позже. Но теперь нам нужно запомнить результат для комбинации аргументов (min,max). Встроенный Map принимает только одно значение как ключ.

Есть много возможных решений:

1. Реализовать новую (или использовать стороннюю) структуру данных для коллекции, которая более универсальна, чем встроенный Map, и поддерживает множественные ключи.
2. Использовать вложенные коллекции: cache.set(min) будет Map, которая хранит пару (max, result). Тогда получить result мы сможем, вызвав cache.get(min).get(max).
3. Соединить два значения в одно. В нашем конкретном случае мы можем просто использовать строку "min,max" как ключ к Map. Для гибкости, мы можем позволить передавать хеширующую функцию в декоратор, которая знает, как сделать одно значение из многих.

Для многих практических применений третий вариант достаточно хорош, поэтому мы будем придерживаться его.

Также нам понадобится заменить func.call(this, x) на func.call(this, ...arguments), чтобы передавать все аргументы обёрнутой функции, а не только первый.

Вот более мощный cachingDecorator:



Теперь он работает с любым количеством аргументов.

Есть два изменения:

* В строке (\*) вызываем hash для создания одного ключа из arguments. Здесь мы используем простую функцию «объединения», которая превращает аргументы (3, 5) в ключ "3,5". В более сложных случаях могут потребоваться другие функции хеширования.
* Затем в строке (\*\*) используем func.call(this, ...arguments) для передачи как контекста, так и всех аргументов, полученных обёрткой (независимо от их количества), в исходную функцию.

Вместо func.call(this, ...arguments) мы могли бы написать func.apply(this, arguments).

Синтаксис встроенного метода [func.apply](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/apply):



Он выполняет func, устанавливая this=context и принимая в качестве списка аргументов псевдомассив args. Единственная разница в синтаксисе между call и apply состоит в том, что call ожидает список аргументов, в то время как apply принимает псевдомассив.

# Флаги и дескрипторы свойств

Как мы знаем, объекты могут содержать свойства. До этого момента мы рассматривали свойство только как пару «ключ-значение». Но на самом деле свойство объекта гораздо мощнее и гибче.

## [Флаги свойств](https://learn.javascript.ru/property-descriptors" \l "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.

Мы ещё не встречали эти атрибуты, потому что обычно они скрыты. Когда мы создаём свойство «обычным способом», все они имеют значение true. Но мы можем изменить их в любое время.

Метод [Object.getOwnPropertyDescriptor](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor) позволяет получить полную информацию о свойстве.

Его синтаксис:



**obj**

Объект, из которого мы получаем информацию.

**propertyName**

Имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги.

Например:



Чтобы изменить флаги, мы можем использовать метод [Object.defineProperty](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty).

Его синтаксис:



**obj, propertyName**

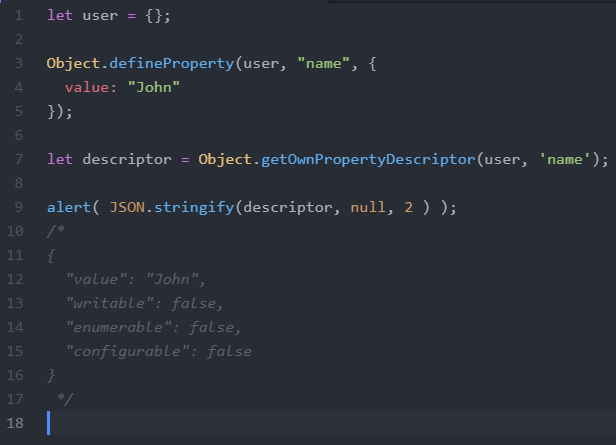
Объект и его свойство, для которого нужно применить дескриптор.

**descriptor**

Применяемый дескриптор.

Если свойство существует, defineProperty обновит его флаги. В противном случае метод создаёт новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение false.

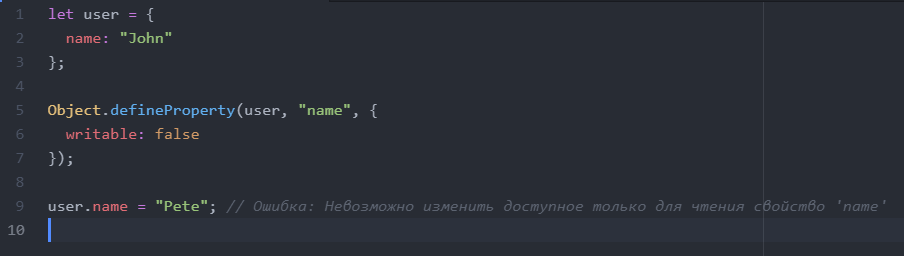
Например, здесь создаётся свойство name, все флаги которого имеют значение false:



Сравните это с предыдущим примером, в котором мы создали свойство user.name «обычным способом»: в этот раз все флаги имеют значение false. Если это не то, что нам нужно, надо присвоить им значения true в параметре descriptor. Теперь давайте рассмотрим на примерах, что нам даёт использование флагов.

## [Только для чтения](https://learn.javascript.ru/property-descriptors" \l "tolko-dlya-chteniya)

Сделаем свойство user.name доступным только для чтения. Для этого изменим флаг writable:

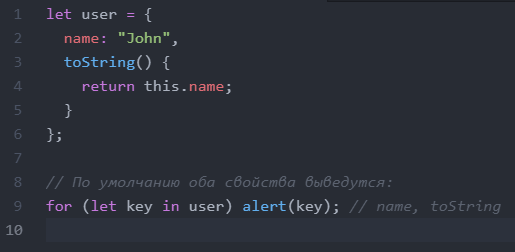
Теперь никто не сможет изменить имя пользователя, если только не обновит соответствующий флаг новым вызовом defineProperty.

**Ошибки появляются только в строгом режиме**

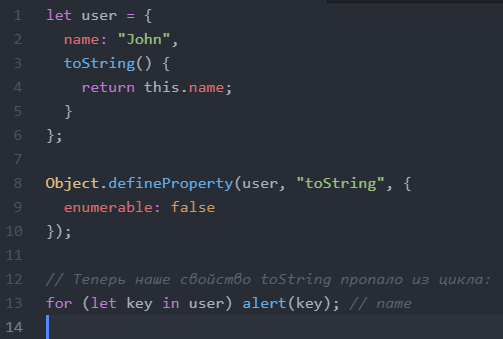
В нестрогом режиме, без use strict, мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Но эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

## [Неперечислимое свойство](https://learn.javascript.ru/property-descriptors" \l "neperechislimoe-svoystvo)

Теперь добавим собственный метод toString к объекту user. Встроенный метод toString в объектах – неперечислимый, его не видно в цикле for..in. Но если мы напишем свой собственный метод toString, цикл for..in будет выводить его по умолчанию:



Если мы этого не хотим, можно установить для свойства enumerable:false. Тогда оно перестанет появляться в цикле for..in аналогично встроенному toString:

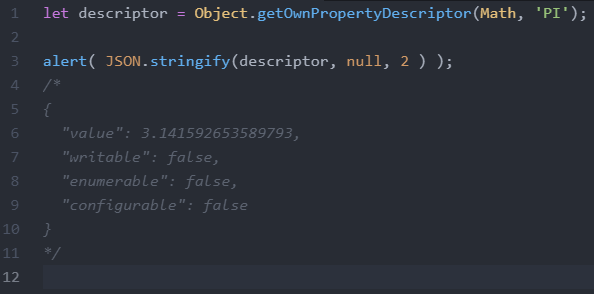


Неперечислимые свойства также не возвращаются Object.keys:

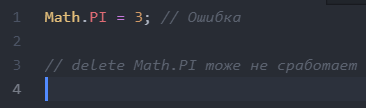
## 

## [Неконфигурируемое свойство](https://learn.javascript.ru/property-descriptors#nekonfiguriruemoe-svoystvo)

Флаг неконфигурируемого свойства (configurable:false) иногда предустановлен для некоторых встроенных объектов и свойств. Неконфигурируемое свойство не может быть удалено. Например, свойство Math.PI – только для чтения, неперечислимое и неконфигурируемое:



То есть программист не сможет изменить значение Math.PI или перезаписать его.



Определение свойства как неконфигурируемого – это дорога в один конец. Мы не сможем отменить это действие, потому что defineProperty не работает с неконфигурируемыми свойствами.

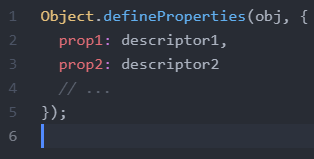
**Ошибки отображаются только в строгом режиме**

В нестрогом режиме мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

## [Метод Object.defineProperties](https://learn.javascript.ru/property-descriptors" \l "metod-object-defineproperties)

Существует метод [Object.defineProperties(obj, descriptors)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties), который позволяет определять множество свойств сразу.

Его синтаксис:



Таким образом, мы можем определить множество свойств одной операцией.

## [Object.getOwnPropertyDescriptors](https://learn.javascript.ru/property-descriptors" \l "object-getownpropertydescriptors)

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться методом [Object.getOwnPropertyDescriptors(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptors). Вместе с Object.defineProperties этот метод можно использовать для клонирования объекта вместе с его флагами:



Обычно при клонировании объекта мы используем присваивание, чтобы скопировать его свойства. Но это не копирует флаги. Так что, если нам нужен клон «получше», предпочтительнее использовать Object.defineProperties.

## [Глобальное запечатывание объекта](https://learn.javascript.ru/property-descriptors" \l "globalnoe-zapechatyvanie-obekta)

Дескрипторы свойств работают на уровне конкретных свойств.

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.

На практике эти методы используются редко.

# Свойства - геттеры и сеттеры

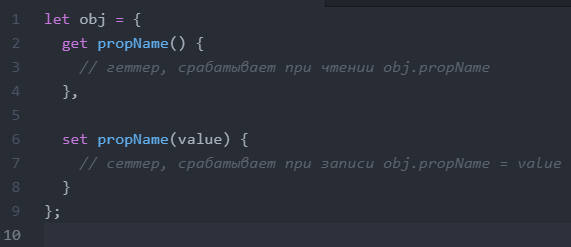
Есть два типа свойств объекта.

Первый тип это свойства-данные (data properties). Мы уже знаем, как работать с ними. Все свойства, которые мы использовали до текущего момента, были свойствами-данными.

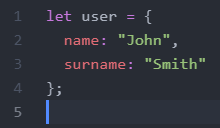
Второй тип свойств мы ещё не рассматривали. Это свойства-аксессоры (accessor properties). По своей сути это функции, которые используются для присвоения и получения значения, но во внешнем коде они выглядят как обычные свойства объекта.

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

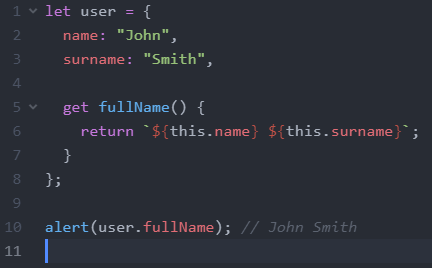
Свойства-аксессоры представлены методами: «геттер» – для чтения и «сеттер» – для записи. При литеральном объявлении объекта они обозначаются get и set:



Геттер срабатывает, когда obj.propName читается, сеттер – когда значение присваивается. Например, у нас есть объект user со свойствами name и surname:



А теперь добавим свойство объекта fullName для полного имени, которое в нашем случае "John Smith". Само собой, мы не хотим дублировать уже имеющуюся информацию, так что реализуем его при помощи аксессора:



Снаружи свойство-аксессор выглядит как обычное свойство. В этом и заключается смысл свойств-аксессоров. Мы не вызываем user.fullName как функцию, а читаем как обычное свойство: геттер выполнит всю работу за кулисами. На данный момент у fullName есть только геттер. Если мы попытаемся назначить user.fullName=, произойдёт ошибка.

## [Дескрипторы свойств доступа](https://learn.javascript.ru/property-accessors" \l "deskriptory-svoystv-dostupa)

Дескрипторы свойств-аксессоров отличаются от «обычных» свойств-данных.

Свойства-аксессоры не имеют value и writable, но взамен предлагают функции get и set.

То есть, дескриптор аксессора может иметь:

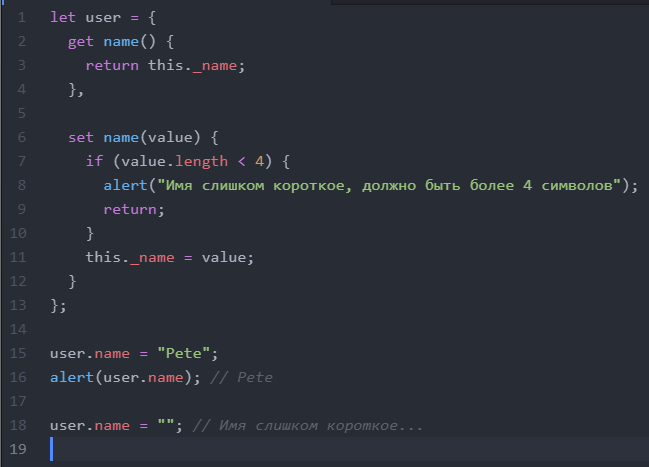
* **get** – функция без аргументов, которая сработает при чтении свойства,
* **set** – функция, принимающая один аргумент, вызываемая при присвоении свойства,
* **enumerable** – то же самое, что и для свойств-данных,
* **configurable** – то же самое, что и для свойств-данных.

Например, для создания аксессора fullName при помощи defineProperty мы можем передать дескриптор с использованием get и set.

**При попытке указать и get, и value в одном дескрипторе будет ошибка**

## [Умные геттеры/сеттеры](https://learn.javascript.ru/property-accessors" \l "umnye-gettery-settery)

Геттеры/сеттеры можно использовать как обёртки над «реальными» значениями свойств, чтобы получить больше контроля над операциями с ними. Например, если мы хотим запретить устанавливать короткое имя для user, мы можем использовать сеттер name для проверки, а само значение хранить в отдельном свойстве \_name:



Таким образом, само имя хранится в \_name, доступ к которому производится через геттер и сеттер. Технически, внешний код всё ещё может получить доступ к имени напрямую с помощью user.\_name, но существует широко известное соглашение о том, что свойства, которые начинаются с символа "\_", являются внутренними, и к ним не следует обращаться из-за пределов объекта.

# Прототипное наследование

В программировании мы часто хотим взять что-то и расширить. Например, у нас есть объект user со своими свойствами и методами, и мы хотим создать объекты admin и guest как его слегка изменённые варианты. Мы хотели бы повторно использовать то, что есть у объекта user, не копировать/переопределять его методы, а просто создать новый объект на его основе.

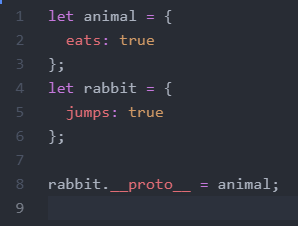
Прототипное наследование — это возможность языка, которая помогает в этом.

## [[[Prototype]]](https://learn.javascript.ru/prototype-inheritance" \l "prototype)

В JavaScript объекты имеют специальное скрытое свойство [[Prototype]] (так оно названо в спецификации), которое либо равно null, либо ссылается на другой объект. Этот объект называется «прототип»:

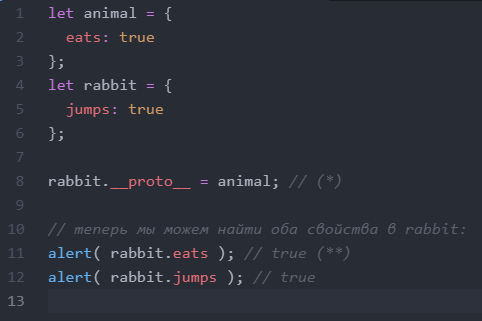
* Когда мы хотим прочитать свойство из object, а оно отсутствует, JavaScript автоматически берёт его из прототипа. В программировании такой механизм называется «прототипным наследованием». Многие интересные возможности языка и техники программирования основываются на нём.
* Свойство [[Prototype]] является внутренним и скрытым, но есть много способов задать его.

Одним из них является использование \_\_proto\_\_, например так:



Если мы ищем свойство в rabbit, а оно отсутствует, JavaScript автоматически берёт его из animal.

Например:



Здесь строка (\*) устанавливает animal как прототип для rabbit. Затем, когда alert пытается прочитать свойство rabbit.eats (\*\*), его нет в rabbit, поэтому JavaScript следует по ссылке [[Prototype]] и находит его в animal. Так что если у animal много полезных свойств и методов, то они автоматически становятся доступными у rabbit. Такие свойства называются «унаследованными».

Есть только два ограничения:

1. Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если мы попытаемся назначить \_\_proto\_\_ по кругу.
2. Значение \_\_proto\_\_ может быть объектом или null. Другие типы игнорируются.

Это вполне очевидно, но всё же: может быть только один [[Prototype]]. Объект не может наследоваться от двух других объектов.

**Свойство \_\_proto\_\_ — исторически обусловленный геттер/сеттер для [[Prototype]]**

Это распространённая ошибка начинающих разработчиков – не знать разницы между этими двумя понятиями. Обратите внимание, что \_\_proto\_\_ — не то же самое, что внутреннее свойство [[Prototype]]. Это геттер/сеттер для [[Prototype]]. Позже мы увидим ситуации, когда это имеет значение, а пока давайте просто будем иметь это в виду, поскольку мы строим наше понимание языка JavaScript.

Свойство \_\_proto\_\_ немного устарело, оно существует по историческим причинам. Современный JavaScript предполагает, что мы должны использовать функции Object.getPrototypeOf/Object.setPrototypeOf вместо того, чтобы получать/устанавливать прототип. Мы также рассмотрим эти функции позже. По спецификации \_\_proto\_\_ должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его.

## [В](https://learn.javascript.ru/prototype-inheritance#itogo) итоге про прототипы:

* В JavaScript все объекты имеют скрытое свойство [[Prototype]], которое является либо другим объектом, либо null.
* Мы можем использовать obj.\_\_proto\_\_ для доступа к нему (исторически обусловленный геттер/сеттер, есть другие способы, которые скоро будут рассмотрены).
* Объект, на который ссылается [[Prototype]], называется «прототипом».
* Если мы хотим прочитать свойство obj или вызвать метод, которого не существует у obj, тогда JavaScript попытается найти его в прототипе.
* Операции записи/удаления работают непосредственно с объектом, они не используют прототип (если это обычное свойство, а не сеттер).
* Если мы вызываем obj.method(), а метод при этом взят из прототипа, то this всё равно ссылается на obj. Таким образом, методы всегда работают с текущим объектом, даже если они наследуются.
* Цикл for..in перебирает как свои, так и унаследованные свойства. Остальные методы получения ключей/значений работают только с собственными свойствами объекта.

# Класс: базовый синтаксис

На практике нам часто надо создавать много объектов одного вида, например пользователей, товары или что-то ещё. Как мы уже знаем из главы [Конструктор, оператор "new"](https://learn.javascript.ru/constructor-new), с этим может помочь new function.

Но в современном JavaScript есть и более продвинутая конструкция «class», которая предоставляет новые возможности, полезные для объектно-ориентированного программирования.

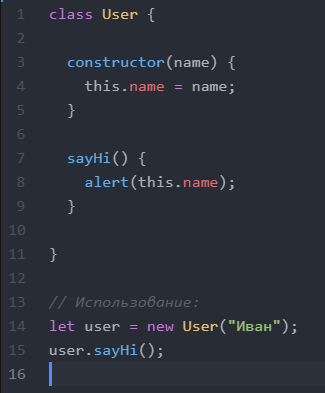
## [Синтаксис «class»](https://learn.javascript.ru/class" \l "sintaksis-class)

Базовый синтаксис выглядит так:



Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами. При этом автоматически вызывается метод constructor(), в нём мы можем инициализировать объект.

Например:



Когда вызывается new User("Иван"):

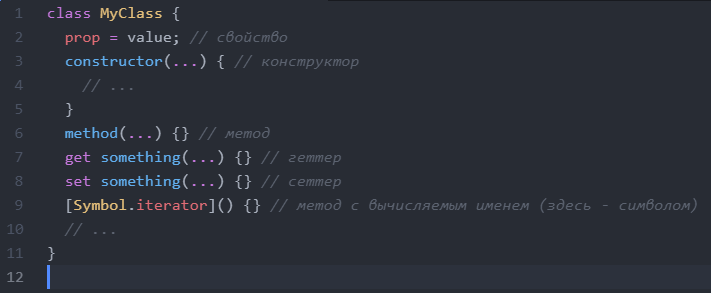
1. Создаётся новый объект.
2. constructor запускается с заданным аргументом и сохраняет его в this.name.

**Методы в классе не разделяются запятой**

Частая ошибка начинающих разработчиков – ставить запятую между методами класса, что приводит к синтаксической ошибке. Синтаксис классов отличается от литералов объектов, не путайте их. Внутри классов запятые не требуются.

## [new MyClass()](https://learn.javascript.ru/class#itogo).

Базовый синтаксис для классов выглядит так:



MyClass технически является функцией (той, которую мы определяем как constructor), в то время как методы, геттеры и сеттеры записываются в MyClass.prototype.

# Наследование классов

Наследование классов – это способ расширения одного класса другим классом. Таким образом, мы можем добавить новый функционал к уже существующему.

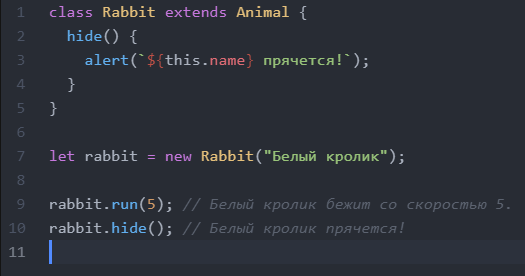
## [Ключевое слово «extends»](https://learn.javascript.ru/class-inheritance" \l "klyuchevoe-slovo-extends)

Допустим, у нас есть класс Animal:



Поскольку кролики – это животные, класс Rabbit должен быть основан на Animal, и иметь доступ к методам животных, так чтобы кролики могли делать то, что могут делать «общие» животные. Синтаксис для расширения другого класса следующий: class Child extends Parent.

Создадим class Rabbit, который наследуется от Animal:



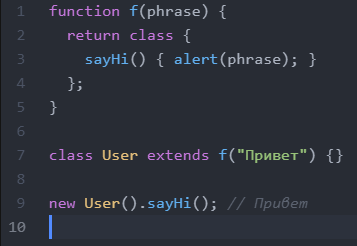
Объект класса Rabbit имеет доступ как к методам Rabbit, таким как rabbit.hide(), так и к методам Animal, таким как rabbit.run(). Внутри ключевое слово extends работает по старой доброй механике прототипов. Оно устанавливает Rabbit.prototype.[[Prototype]] в Animal.prototype. Таким образом, если метода не оказалось в Rabbit.prototype, JavaScript берет его из Animal.prototype.

Например, чтобы найти метод rabbit.run, движок проверяет (снизу вверх на картинке):

1. Объект rabbit (не имеет run).
2. Его прототип, то есть Rabbit.prototype (имеет hide, но не имеет run).
3. Его прототип, то есть (вследствие extends) Animal.prototype, в котором, наконец, есть метод run.

**После extends разрешены любые выражения**

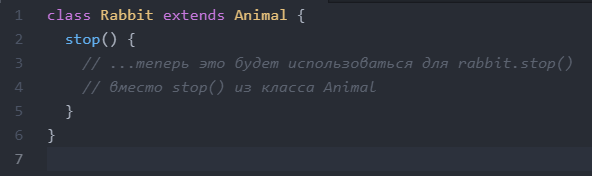
Синтаксис создания класса допускает указывать после extends не только класс, но и любое выражение. Пример вызова функции, которая генерирует родительский класс:



Здесь class User наследует от результата вызова f("Привет"). Это может быть полезно для продвинутых приёмов проектирования, где мы можем использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.

## [Переопределение методов](https://learn.javascript.ru/class-inheritance" \l "pereopredelenie-metodov)

Теперь давайте продвинемся дальше и переопределим метод. По умолчанию все методы, не указанные в классе Rabbit, берутся непосредственно «как есть» из класса Animal. Но если мы укажем в Rabbit собственный метод, например stop(), то он будет использован вместо него:



Впрочем, обычно мы не хотим полностью заменить родительский метод, а скорее хотим сделать новый на его основе, изменяя или расширяя его функциональность. Мы делаем что-то в нашем методе и вызываем родительский метод до/после или в процессе.

У классов есть ключевое слово "super" для таких случаев.

* super.method(...) вызывает родительский метод.
* super(...) для вызова родительского конструктора (работает только внутри нашего конструктора).

## [Краткая](https://learn.javascript.ru/class-inheritance#itogo) справка для самостоятельного изучения.

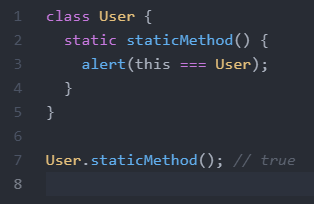
1. Чтобы унаследовать от класса: class Child extends Parent:
   * При этом Child.prototype.\_\_proto\_\_ будет равен Parent.prototype, так что методы будут унаследованы.
2. При переопределении конструктора:
   * Обязателен вызов конструктора родителя super() в конструкторе Child до обращения к this.
3. При переопределении другого метода:
   * Мы можем вызвать super.method() в методе Child для обращения к методу родителя Parent.
4. Внутренние детали:
   * Методы запоминают свой объект во внутреннем свойстве [[HomeObject]]. Благодаря этому работает super, он в его прототипе ищет родительские методы.
   * Поэтому копировать метод, использующий super, между разными объектами небезопасно.

Также:

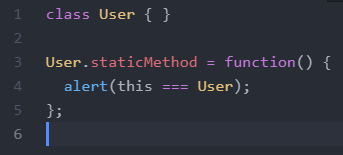
* У функций-стрелок нет своего this и super, поэтому они «прозрачно» встраиваются во внешний контекст.

# Статические свойства и методы

Мы также можем присвоить метод самому классу. Такие методы называются статическими. В объявление класса они добавляются с помощью ключевого слова static, например:



Это фактически то же самое, что присвоить метод напрямую как свойство функции:



Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»). Обычно статические методы используются для реализации функций, принадлежащих классу целиком, вообще, и при этом не относящимся к каким-то отдельным объектам. Например, есть объекты статей Article, и нужна функция для их сравнения. Естественное решение – сделать для этого статический метод Article.compare:



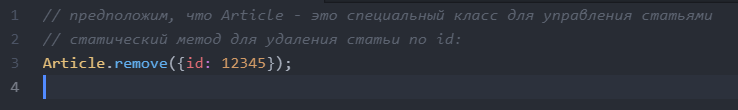
Здесь метод Article.compare стоит «над» статьями, как средство для их сравнения. Это метод не отдельной статьи, а всего класса. Другим примером может быть так называемый «фабричный» метод. Скажем, нам нужно несколько способов создания статьи:

1. Создание через заданные параметры (title, date и т. д.).
2. Создание пустой статьи с сегодняшней датой.
3. …

Первый способ может быть реализован через конструктора. А для второго можно использовать статический метод класса. Такой как Article.createTodays() в следующем примере:



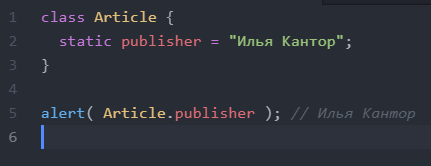
Теперь каждый раз, когда нам нужно создать сегодняшний дайджест, нужно вызывать Article.createTodays(). Ещё раз, это не метод одной статьи, а метод всего класса. Статические методы также используются в классах, относящихся к базам данных, для поиска/сохранения/удаления вхождений в базу данных, например:

**Статические методы недоступны для отдельных объектов**

Статические методы могут вызываться для классов, но не для отдельных объектов.

## [Статические свойства](https://learn.javascript.ru/static-properties-methods" \l "staticheskie-svoystva)

Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:



Это то же самое, что и прямое присваивание Article.

## Что в итоге?

* Статические методы используются для функциональности, принадлежат классу «в целом», а не относятся к конкретному объекту класса. Например, метод для сравнения двух статей Article.compare(article1, article2) или фабричный метод Article.createTodays().
* В объявлении класса они помечаются ключевым словом static.
* Статические свойства используются в тех случаях, когда мы хотели бы сохранить данные на уровне класса, а не какого-то одного объекта.
* Технически, статическое объявление – это то же самое, что и присвоение классу.
* Статические свойства и методы наследуются.
* Для class B extends A прототип класса B указывает на A: B.[[Prototype]] = A. Таким образом, если поле не найдено в B, поиск продолжается в A.

# Приватные и защищённые методы и свойства

Один из важнейших принципов объектно-ориентированного программирования – разделение внутреннего и внешнего интерфейсов. Это обязательная практика в разработке чего-либо сложнее, чем «hello world».

## [Внутренний и внешний интерфейсы](https://learn.javascript.ru/private-protected-properties-methods" \l "vnutrenniy-i-vneshniy-interfeysy)

В объектно-ориентированном программировании свойства и методы разделены на 2 группы:

* Внутренний интерфейс – методы и свойства, доступные из других методов класса, но не снаружи класса.
* Внешний интерфейс – методы и свойства, доступные снаружи класса.

Всё, что нам нужно для использования объекта, это знать его внешний интерфейс. Мы можем совершенно не знать, как это работает внутри, и это здорово.

В JavaScript есть два типа полей (свойств и методов) объекта:

* Публичные: доступны отовсюду. Они составляют внешний интерфейс. До этого момента мы использовали только публичные свойства и методы.
* Приватные: доступны только внутри класса. Они для внутреннего интерфейса.

Во многих других языках также существуют «защищённые» поля, доступные только внутри класса или для дочерних классов (то есть, как приватные, но разрешён доступ для наследующих классов) и также полезны для внутреннего интерфейса. В некотором смысле они более распространены, чем приватные, потому что мы обычно хотим, чтобы наследующие классы получали доступ к внутренним полям. Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их эмулируют.

**Практическое задание**

* 1. Создайте массив с товарами/элементами сайта (например, для корзины). По нажатию кнопки один из элементов массива должен меняться на другой, по нажатию на другую кнопку первый элемент удаляется.
  2. Создайте фильтр, который принимает массив, ищет элементы со значениями больше или равными a и меньше или равными b и возвращает результат в виде массива. (Функция должна возвращать новый массив и не изменять исходный.)
  3. Создайте сортировку на сайте элементов в порядке возрастания/убывания.
  4. В элементе уведомлений, созданном ранее, добавьте появление новых уведомлений каждые 3 секунды (счетчик новых уведомлений должен обновляться, содержание элемента на ваше усмотрение).
  5. При нажатии на кнопку уведомлений, с использованием задерживающего декоратора, остановите счетчик новых уведомлений на 10 секунд.