**Первая тема**

1. **Переменные.**

Для создания переменной в JavaScript, используйте ключевое слово let:

let message;

message = 'Hello'; // сохранить строку

Можно объявить несколько переменных в одной строке, но это не рекомендуется, так как ухудшает читаемость кода:

let user = 'John', age = 25, message = 'Hello';

Многострочный вариант немного длиннее, но легче для чтения:

let user = 'John';

let age = 25;

let message = 'Hello';

Можно объявлять переменные на нескольких строках и даже с запятой в начале строки:

let user = 'John',

age = 25,

message = 'Hello';

let user = 'John'

, age = 25

, message = 'Hello';

В старых скриптах можно найти ключевое слово: var вместо let:

var message = 'Hello';

Ключевое слово var – почти то же самое, что и let. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.

**[Имена переменных](https://learn.javascript.ru/variables" \l "variable-naming)**

Название переменной должно иметь ясный и понятный смысл, говорить о том, какие данные в ней хранятся.

Именование переменных – это один из самых важных и сложных навыков в программировании. Быстрый взгляд на имена переменных может показать, какой код был написан новичком, а какой опытным разработчиком.

В реальном проекте большая часть времени тратится на изменение и расширение существующей кодовой базы, а не на написание чего-то совершенно нового с нуля. Когда мы возвращаемся к коду после какого-то промежутка времени, гораздо легче найти информацию, которая хорошо размечена. Или, другими словами, когда переменные имеют хорошие имена.

Несколько советов:

* Используйте легко читаемые имена, такие как userName или shoppingCart.
* Избегайте использования аббревиатур или коротких имён, таких как a, b, c, за исключением тех случаев, когда вы точно знаете, что так нужно.
* Делайте имена максимально описательными и лаконичными. Примеры плохих имён: data и value. Такие имена ничего не говорят. Их можно использовать только в том случае, если из контекста кода очевидно, какие данные хранит переменная.
* Договоритесь с вашей командой о используемых терминах. Если посетитель сайта называется «user» тогда мы должны назвать связанные с ним переменные currentUser или newUser вместо того, чтобы называть их currentVisitorили newManInTown.

В JavaScript есть два ограничения, касающиеся имён переменных:

1. Имя переменной должно содержать только буквы, цифры или символы $ и \_.
2. Первый символ не должен быть цифрой.

Если имя содержит несколько слов, обычно используется [верблюжья нотация](https://ru.wikipedia.org/wiki/CamelCase), например,  myVeryLongName.

Примеры допустимых имён:

let userName;

let test123;

let $ = 1; // объявили переменную с именем "$"

let \_ = 2; // а теперь переменную с именем "\_"

Примеры неправильных имён переменных:

let 1a; // не может начинаться с цифры

let my-name; // дефис '-' не разрешён в имени

Регистр имеет значение. Переменные с именами apple and AppLE – это две разные переменные.

Нелатинские буквы разрешены, но не рекомендуются. Можно использовать любой язык, включая кириллицу или даже иероглифы, например:

let имя = '...';

let 我 = '...';

Технически здесь нет ошибки, такие имена разрешены, но есть международная традиция использовать английский язык в именах переменных.

Существует [список зарезервированных слов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Lexical_grammar" \l "%D0%9A%D0%BB%D1%8E%D1%87%D0%B5%D0%B2%D1%8B%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%B0), которые нельзя использовать в качестве имён переменных, потому что они используются самим языком. Например: let, class, return и function зарезервированы. Приведённый ниже код даёт синтаксическую ошибку:

let let = 5; // нельзя назвать переменную "let", ошибка!

let return = 5; // также нельзя назвать переменную "return", ошибка!

Обычно нужно определить переменную перед её использованием. Но старые стандарты позволяли создавать переменную простым присвоением значения без использования let. Это все ещё работает, если не включить use strict, для поддержания совместимости со старыми скриптами.

// заметка: "use strict" в этом примере не используется

num = 5; // если переменная "num" не существовала, она создаётся

alert(num); // 5

Это плохая практика и приведёт к ошибке в строгом режиме:

"use strict";

num = 5; // error: num is not defined

**[Константы](https://learn.javascript.ru/variables" \l "konstanty)**

Чтобы объявить константную переменную необходимо использовать const вместо let:

const myBirthday = '18.04.1982';

Изменение константы приведет к ошибке:

const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // ошибка, константу нельзя перезаписать!

Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта. Названия таких констант пишутся с использованием заглавных букв и подчёркивания, например, для различных цветов в шестнадцатиричном формате:

const COLOR\_RED = "#F00";

const COLOR\_GREEN = "#0F0";

const COLOR\_BLUE = "#00F";

const COLOR\_ORANGE = "#FF7F00";

let color = COLOR\_ORANGE;

alert(color); // #FF7F00

Преимущества такого подхода:

* COLOR\_ORANGE гораздо легче запомнить, чем "#FF7F00".
* Гораздо легче допустить ошибку при вводе "#FF7F00", чем при вводе COLOR\_ORANGE.
* При чтении кода, COLOR\_ORANGE намного понятнее, чем #FF7F00.

Когда мы должны использовать для констант заглавные буквы, а когда называть их нормально? Давайте разберёмся и с этим.

1. **Типы данных.**

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:

// Не будет ошибкой

let message = "hello";

message = 123456;

Языки программирования, в которых такое возможно, называются динамически типизированными. Это значит, что типы данных есть, но переменные не привязаны ни к одному из них. Есть семь основных типов данных в JavaScript, пять из них называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка или число, или что-то ещё).

**[Число](https://learn.javascript.ru/types" \l "chislo)**

let n = 123;

n = 12.345;

Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой. Существует множество операций для чисел, например, умножение \*, деление /, сложение +, вычитание - и так далее.

Помимо обычных чисел существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity, -Infinity и NaN.

*Infinity* представляет собой математическую [бесконечность](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D1%81%D0%BA%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C" \l "%D0%92_%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B5) ∞. Это особое значение, которое больше любого числа. Мы можем получить его в результате деления на ноль или задать его явно:

alert( 1 / 0 ); // Infinity

alert( Infinity ); // Infinity

*NaN* означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:

alert( "не число" / 2 ); // NaN, такое деление является ошибкой

Любая операция с NaN возвращает NaN:

alert( "не число" / 2 + 5 ); // NaN

Математические операции в JavaScript позволяют делать что угодно: делить на ноль, обращаться со строками как с числами и т.д. Скрипт никогда не остановится с фатальной ошибкой. В худшем случае просто в результате выполнения будет возвращен NaN .

Тип *BigInt* –  представляет собой встроенный объект, который предоставляет способ представлять целые числа больше pow(2, 53) - 1, наибольшего числа, которое JavaScript может надежно представить с Number примитивом.

Чтобы обозначить, что число относится к типу BigInt нужно добавить n в конце числа. n означает, что это BigInt.

let oldMax = Number.MAX\_SAFE\_INTEGER; // 9007199254740991

++oldMax; // 9007199254740992

++oldMax; // 9007199254740992 <- такое же значение

let newMax = 9007199254740992n;

++newMax; // 9007199254740993n

++newMax; // 9007199254740994n

const a = 9007199254740991n; // 9007199254740991n

const b = BigInt(9007199254740991n); // 9007199254740991n

a === b; // true

typeof 10; // "number";

typeof 10n; // "bigint";

**[Строка](https://learn.javascript.ru/types" \l "stroka)**

Строка (string) в JavaScript должна быть заключена в кавычки.

let str = "Привет";

let str2 = 'Одинарные кавычки тоже подойдут';

let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;

В JavaScript существует три типа кавычек.

* Двойные кавычки: "Привет".
* Одинарные кавычки: 'Привет'.
* Обратные кавычки: `Привет`.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript. Обратные кавычки же имеют расширенный функционал. Они позволяют встраивать выражения в строку, заключая их в ${…}. Например:

let name = "Иван";

// Вставим переменную

alert( `Привет, ${name}!` ); // Привет, Иван!

// Вставим выражение

alert( `результат: ${1 + 2}` ); // результат: 3

Выражение внутри ${…} вычисляется, и его результат становится частью строки. Можно записать туда всё, что угодно: переменную name или выражение 1 + 2, или что-то более сложное.

В некоторых языках для одного символа существует специальный «символьный» тип. Например, в C и Java это char. В JavaScript подобного типа нет, есть только тип string. Строка может содержать один символ или множество.

**[Булевый (логический) тип](https://learn.javascript.ru/types" \l "bulevyy-logicheskiy-tip)**

Булевый тип (boolean) может принимать только два значения: true (истина) и false (ложь). Такой тип, как правило, используется для хранения значений да/нет: true значит «да, правильно», а false значит «нет, не правильно». Например:

let nameFieldChecked = true; // да, поле отмечено

let ageFieldChecked = false; // нет, поле не отмечено

Булевы значения также могут быть результатом сравнений:

let isGreater = 4 > 1;

alert( isGreater ); // true (результатом сравнения будет "да")

**[Значение «null»](https://learn.javascript.ru/types" \l "znachenie-null)**

Специальное значение null не относится ни к одному из типов, описанных выше. Оно формирует отдельный тип, который содержит только значение null:

let age = null;

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно». В приведённом выше коде указано, что переменная age неизвестна или не имеет значения по какой-то причине.

**[Значение «undefined»](https://learn.javascript.ru/types" \l "znachenie-undefined)**

Специальное значение undefined также стоит особняком. Оно формирует тип из самого себя так же, как и null. Оно означает, что «значение не было присвоено». Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined:

let x;

alert(x); // выведет "undefined"

Технически можно присвоить значение undefined любой переменной:

let x = 123;

x = undefined;

alert(x); // "undefined"

Но так делать не рекомендуется. Обычно null используется для присвоения переменной «пустого» или «неизвестного» значения, а undefined для проверок, была ли переменная назначена.

**[Объекты и символы](https://learn.javascript.ru/types" \l "obekty-i-simvoly)**

Тип object (объект) используется для хранения коллекций данных или более сложных объектов.

Тип symbol (символ) используется для создания уникальных идентификаторов объектов.

Рассмотрим объекты и символы позднее после того, как изучим подробнее примитивы.

**[Оператор typeof](https://learn.javascript.ru/types" \l "type-typeof)**

Оператор typeof возвращает тип аргумента. Это полезно, когда необходимо обрабатывать значения различных типов по-разному или просто сделать проверку.

У него есть два синтаксиса, результат одинаковый:

* Синтаксис оператора: typeof x.
* Синтаксис функции: typeof(x).

Вызов typeof x возвращает строку с именем типа:

typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

typeof null // "object" (1)

typeof alert // "function" (2)

Результатом вызова typeof null в строке (1) является "object". Это неверно. Это официально признанная ошибка в typeof, сохранённая для совместимости. null не является объектом, это специальное значение с отдельным типом.

Вызов typeof alert в строке (2) возвращает "function", потому что alert является функцией. В JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но typeof обрабатывает их особым образом, возвращая "function". Формально это неверно, но очень удобно на практике.

1. **Преобразование типов**

Чаще всего, операторы и функции автоматически приводят переданные им значения к нужному типу. Например, alert автоматически преобразует любое значение к строке, а математические операторы преобразуют значения к числам. Есть также случаи, когда нужно явно преобразовать значение в ожидаемый тип.

**[Строковое преобразование](https://learn.javascript.ru/type-conversions" \l "strokovoe-preobrazovanie)**

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. Например, alert(value) преобразует значение к строке.

Также можно использовать функцию String(value) чтобы преобразовать значение к строке:

let value = true;

alert(typeof value); // boolean

value = String(value); // теперь value это строка "true"

alert(typeof value); // string

Преобразование происходит очевидным способом. False становится "false", null становится "null" и т.п.

Почти все математические операторы выполняют численное преобразование. Исключение составляет +. Если одно из слагаемых является строкой, тогда и все остальные приводятся к строкам и они конкатенируются (присоединяются) друг к другу:

alert( 1 + '2' ); // '12' (строка справа)

alert( '1' + 2 ); // '12' (строка слева)

Так происходит, только если хотя бы один из аргументов является строкой. Во всех остальных случаях, значения складываются как числа.

**[Численное преобразование](https://learn.javascript.ru/type-conversions" \l "chislennoe-preobrazovanie)**

Численное преобразование происходит в математических функциях и выражениях. Например, когда операция деления / применяется не к числу:

alert( "6" / "2" ); // 3, Строки преобразуются в числа

Можно использовать функцию Number(value) чтобы явно преобразовать value к числу:

let str = "123";

alert(typeof str); // string

let num = Number(str); // становится числом 123

alert(typeof num); // number

Явноe преобразование часто применяется, когда необходимо получить число из строковых источников, вроде форм текстового ввода. Если строка не может быть явно приведена к числу, то результатом преобразования будет NaN. Например:

let age = Number("Любая строка вместо числа");

alert(age); // NaN, преобразование не удалось

Правила численного преобразования:

| **Значение** | **Результат** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| string | Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки «считывается» число. При ошибке результат NaN. |

Примеры:

alert( Number(" 123 ") ); // 123

alert( Number("123z") ); // NaN (ошибка чтения числа в "z")

alert( Number(true) ); // 1

alert( Number(false) ); // 0

null и undefined ведут себя по-разному: null становится нулём, undefined приводится к NaN.

**[Логическое преобразование](https://learn.javascript.ru/type-conversions" \l "logicheskoe-preobrazovanie)**

Логическое преобразование происходит в логических операторах но так же может быть выполнено явно с помощью функции Boolean(value).

Правила преобразования:

* Значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined, и NaN, становятся false.
* Все остальные значения становятся true.

Например:

alert( Boolean(1) ); // true

alert( Boolean(0) ); // false

alert( Boolean("Привет!") ); // true

alert( Boolean("") ); // false

Строка с нулём "0" преобразуется в true. Некоторые языки воспринимают строку "0" как false. Но в JavaScript, если строка не пустая, то она всегда true.

alert( Boolean("0") ); // true

alert( Boolean(" ") ); // пробел это тоже true

1. **Операторы.**

*Операнд* – то, к чему применяется оператор. Например, в умножении 5 \* 2 есть два операнда: левый операнд равен 5, а правый операнд равен 2. Иногда их называют аргументами.

*Унарным* называется оператор, который применяется к одному операнду. Например, оператор унарный минус "-"меняет знак числа на противоположный:

let x = 1;

x = -x;

alert( x ); // -1, применили унарный минус

*Бинарным* называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:

let x = 1, y = 3;

alert( y - x ); // 2, бинарный минус

Формально мы говорим о двух разных операторах: унарное отрицание (один операнд: меняет знак) и бинарное вычитание (два операнда: вычитает).

**[Сложение строк, бинарный +](https://learn.javascript.ru/operators" \l "slozhenie-strok-binarnyy)**

Обычно при помощи оператора плюс '+' складывают числа. Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну. Если хотя бы один операнд является строкой, то второй будет также преобразован к строке.

let s = "моя" + "строка";

alert(s); // моястрока

Например:

alert( '1' + 2 ); // "12"

alert( 2 + '1' ); // "21"

Важно то, что операции выполняются слева направо. Если перед строкой идут два числа, то числа будут сложены перед преобразованием в строку:

alert(2 + 2 + '1' ); // будет "41", а не "221"

Сложение и преобразование строк – это особенность бинарного плюса +. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа. Например, вычитание и деление:

alert( 2 - '1' ); // 1

alert( '6' / '2' ); // 3

**[Преобразование к числу, унарный плюс +](https://learn.javascript.ru/operators" \l "preobrazovanie-k-chislu-unarnyy-plyus)**

Плюс + существует в двух формах: бинарной, которая рассматривалась выше, и унарной. Унарный плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число. Это то же самое, что и Number(...), только короче. Например:

// Не влияет на числа

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

// Преобразует нечисла в число

alert( +true ); // 1

alert( +"" ); // 0

Необходимость преобразовывать строки в числа возникает очень часто. Например, обычно значения полей HTML-формы – это строки. Если их надо сложить, то сначала необходимо преобразовать к числам. Бинарный плюс сложит их как строки:

let apples = "2";

let oranges = "3";

alert( apples + oranges ); // "23"

Поэтому использутся унарный плюс, чтобы преобразовать к числу:

let apples = "2";

let oranges = "3";

// оба операнда предварительно преобразованы в числа

alert( +apples + +oranges ); // 5

// более длинный вариант

// alert( Number(apples) + Number(oranges) ); // 5

Сначала выполнятся унарные плюсы, приведут строки к числам, а затем – бинарный '+' их сложит. Этот порядок определяет приеоритет оператора.

**[Приоритет операторов](https://learn.javascript.ru/operators" \l "prioritet-operatorov)**

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется приоритетом, или, другими словами, существует определённый порядок выполнения операторов.

Известно, что умножение в выражении 2 \* 2 + 1 выполнится раньше сложения. Это потому, что умножение имеет более высокий приоритет, чем сложение.

Скобки важнее, чем приоритет, так что, если необходимо изменить порядок по умолчанию, можно использовать их, чтобы изменить приоритет. Например, написать (1 + 2) \* 2.

В JavaScript много операторов. Каждый оператор имеет соответствующий номер приоритета. Тот, у кого это число больше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.

Ниже представлена часть [таблицы приоритетов](https://developer.mozilla.org/ru/JavaScript/Reference/operators/operator_precedence). Обратите внимание, что у унарных операторов приоритет выше, чем у соответствующих бинарных:

| **Приоритет** | **Название** | **Обозначение** |
| --- | --- | --- |
| … | … | … |
| 16 | унарный плюс | + |
| 16 | унарный минус | - |
| 14 | умножение | \* |
| 14 | деление | / |
| 13 | сложение | + |
| 13 | вычитание | - |
| … | … | … |
| 3 | присваивание | = |
| … | … | … |

Так как «унарный плюс» имеет приоритет 16, который выше, чем 13 у «сложения» (бинарный плюс), то в выражении "+apples + +oranges" сначала выполнятся унарные плюсы, а затем сложение.

**[Присваивание](https://learn.javascript.ru/operators" \l "prisvaivanie)**

В таблице приоритетов также есть оператор присваивания =. У него один из самых низких приоритетов – 3. Именно поэтому, когда переменной что-либо присваивают, например, x = 2 \* 2 + 1, то сначала выполнится арифметика, а уже затем произойдёт присваивание =.

let x = 2 \* 2 + 1;

alert( x ); // 5

Возможно присваивание по цепочке:

let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4

alert( b ); // 4

alert( c ); // 4

Такое присваивание работает справа-налево. Сначала вычисляется самое правое выражение 2 + 2, и затем оно присвоится переменным слева: c, b и a. В конце у всех переменных будет одно значение.

**Оператор "=" возвращает значение**

Все операторы возвращают значение. Для некоторых это очевидно, например, сложение + или умножение \*. Но и оператор присваивания не является исключением. Вызов x = value записывает value в x и возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения:

let a = 1;

let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3

alert( c ); // 0

В примере выше результатом (a = b + 1) будет значение, которое присваивается в a, то есть 3. Потом оно используется для дальнейших вычислений. Писать в таком стиле не рекомендуется, так как это делает ваш код менее понятным и читабельным.

**[Остаток от деления %](https://learn.javascript.ru/operators" \l "ostatok-ot-deleniya)**

Оператор взятия остатка в выражении a % b возвращает остаток от деления a на b. Например:

alert( 5 % 2 ); // 1, остаток от деления 5 на 2

alert( 8 % 3 ); // 2, остаток от деления 8 на 3

alert( 6 % 3 ); // 0, остаток от деления 6 на 3

**[Возведение в степень \*\*](https://learn.javascript.ru/operators" \l "vozvedenie-v-stepen)**

Оператор возведения в степень \*\* недавно добавили в язык. Для натурального числа b результат a \*\* b равен a, умноженному на само себя b раз. Например:

alert( 2 \*\* 2 ); // 4 (2 \* 2)

alert( 2 \*\* 3 ); // 8 (2 \* 2 \* 2)

alert( 2 \*\* 4 ); // 16 (2 \* 2 \* 2 \* 2)

Оператор работает и для нецелых чисел. Например:

alert( 4 \*\* (1/2) ); // 2

alert( 8 \*\* (1/3) ); // 2

**[Инкремент/декремент](https://learn.javascript.ru/operators" \l "inkrement-dekrement)**

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу. Для этого существуют даже специальные операторы:

*Инкремент* ++ увеличивает на 1:

let counter = 2;

counter++; // работает как counter = counter + 1, просто запись короче

alert( counter ); // 3

*Декремент* -- уменьшает на 1:

let counter = 2;

counter--; // работает как counter = counter - 1, просто запись короче

alert( counter ); // 1

Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа 5++, вернёт ошибку.

Операторы ++ и -- могут быть расположены не только после, но и до переменной. Когда оператор идёт после переменной – это *постфиксная форма*: counter++. *Префиксная форма* – это когда оператор идёт перед переменной: ++counter. Обе эти формы записи делают одно и то же: увеличивают counter на 1. Разница в том, что префиксная форма возвращает новое значение, а постфиксная форма возвращает старое значение (до увеличения/уменьшения числа). Чтобы увидеть разницу, вот небольшой пример:

let counter = 1;

let a = ++counter; // (\*)

alert(a); // 2

В строке (\*) префиксная форма увеличения counter, она возвращает новое значение 2.

Пример использования постфиксной формы:

let counter = 1;

let a = counter++; // (\*)

alert(a); // 1

В строке (\*) постфиксная форма counter++ также увеличивает counter, но возвращает старое значение, которое было до увеличения. Так что alert покажет 1.

Операторы ++/-- могут также использоваться внутри выражений. Их приоритет выше, чем у арифметических операций. Например:

let counter = 1;

alert( 2 \* ++counter ); // 4

Сравните с:

let counter = 1;

alert( 2 \* counter++ ); // 2

Такая запись делает код менее читабельным. При беглом чтении кода можно с лёгкостью пропустить counter++, и будет неочевидно, что переменая увеличивается. Лучше использовать стиль «одна строка – одно действие»:

let counter = 1;

alert( 2 \* counter );

counter++;

**[Побитовые операторы](https://learn.javascript.ru/operators" \l "pobitovye-operatory)**

Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), на уровне их внутреннего двоичного представления. Эти операторы не являются чем-то специфичным для JavaScript, они поддерживаются в большинстве языков программирования. Поддерживаются следующие побитовые операторы:

* AND (и) ( & )
* OR (или) ( | )
* XOR (побитовое исключающее или) ( ^ )
* NOT (не) ( ~ )
* LEFT SHIFT (левый сдвиг) ( << )
* RIGHT SHIFT (правый сдвиг) ( >> )
* ZERO-FILL RIGHT SHIFT (правый сдвиг с заполнением нулями) ( >>> )

Они используются редко. Чтобы понять их, нужно углубиться в низкоуровневое представление чисел.

**[Сокращённая арифметика с присваиванием](https://learn.javascript.ru/operators" \l "sokraschyonnaya-arifmetika-s-prisvaivaniem)**

Часто нужно применить оператор к переменной и сохранить результат в ней же. Например:

let n = 2;

n = n + 5;

n = n \* 2;

Эту запись можно укоротить при помощи совмещённых операторов += и \*=:

let n = 2;

n += 5; // теперь n=7 (работает как n = n + 5)

n \*= 2; // теперь n=14 (работает как n = n \* 2)

alert( n ); // 14

Подобные краткие формы записи существуют для всех арифметических и побитовых операторов: /=, -= и так далее. Вызов с присваиванием имеет в точности такой же приоритет, как обычное присваивание, то есть выполнится после большинства других операций:

let n = 2;

n \*= 3 + 5;

alert( n ); // 16

**[Оператор запятая](https://learn.javascript.ru/operators" \l "operator-zapyataya)**

Оператор «запятая», редко используется и является одним из самых необычных. Иногда он используется для написания более короткого кода, поэтому нужно знать его, чтобы понимать, что при этом происходит.

Оператор запятая предоставляет возможность вычислять несколько выражений, разделяя их запятой. Каждое выражение выполняется, но возвращается результат только последнего. Например:

let a = (1 + 2, 3 + 4);

alert( a ); // 7

Первое выражение 1 + 2 выполняется, а результат отбрасывается. Затем идёт 3 + 4, выражение выполняется и возвращается результат.

Оператор запятая имеет очень низкий приоритет, приоритет которого ниже =, поэтому скобки важны в приведённом примере выше. Без них в a = 1 + 2, 3 + 4 сначала выполнится +, суммируя числа в a = 3, 7, затем оператор присваивания = присвоит a = 3, а то что идёт дальше, будет игнорировано. Всё так же, как в (a = 1 + 2), 3+4.

Этот оператор иногда используют в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:

// три операции в одной строке

for (a = 1, b = 3, c = a \* b; a < 10; a++) {

...

}

Такое написание кода используется во многих JavaScript-фреймворках и о нем стоит знать. Но обычно это не улучшает читабельность кода, поэтому этот опреатор не рекомендуется использовать.

1. **Операторы сравнения.**

Операторы сравнения известные из математики:

* Больше/меньше: a > b, a < b.
* Больше/меньше или равно: a >= b, a <= b.
* Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства =. Один знак равенства a = b означал бы присваивание.
* Не равно. В математике обозначается символом ≠. В JavaScript записывается как знак равенства с предшествующим ему восклицательным знаком: a != b.

Операторы сравнения, как и другие операторы, возвращают значение. Это значение имеет логический тип:

* true – означает «да», «верно», «истина».
* false – означает «нет», «неверно», «ложь».

Например:

alert( 2 > 1 ); // true (верно)

alert( 2 == 1 ); // false (неверно)

alert( 2 != 1 ); // true (верно)

Результат сравнения можно присвоить переменной, как и любое значение:

let result = 5 > 4; // результат сравнения присваивается переменной result

alert( result ); // true

**[Сравнение строк](https://learn.javascript.ru/comparison" \l "sravnenie-strok)**

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок. Другими словами, строки сравниваются посимвольно. Например:

alert( 'Я' > 'А' ); // true

alert( 'Кот' > 'Код' ); // true

alert( 'Сонный' > 'Сон' ); // true

Алгоритм сравнения двух строк довольно прост:

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
4. Сравнение продолжается, пока не закончится одна из строк.
5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

В примерах выше сравнение 'Я' > 'А' завершится на первом шаге, тогда как строки "Кот" и "Код" будут сравниваться посимвольно:

1. К равна К.
2. о равна о.
3. т больше чем д. На этом сравнение заканчивается. Первая строка больше.

Строчная "a" больше заглавной буквы "A". Потому что строчные буквы имеют больший код во внутренней таблице кодирования (Unicode), которую использует JavaScript.

**[Сравнение разных типов](https://learn.javascript.ru/comparison" \l "sravnenie-raznyh-tipov)**

При сравнении значений разных типов, JavaScript приводит каждое из них к числу. Например:

alert( '2' > 1 ); // true, строка '2' становится числом 2

alert( '01' == 1 ); // true, строка '01' становится числом 1

Логическое значение true становится 1, а false – 0. Например:

alert( true == 1 ); // true

alert( false == 0 ); // true

Возможна следующая ситуация. Два значения равны. Одно из них true как логическое значение, другое – false. Например:

let a = 0;

alert( Boolean(a) ); // false

let b = "0";

alert( Boolean(b) ); // true

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

С точки зрения JavaScript, результат ожидаем. Равенство преобразует значения, используя числовое преобразование, поэтому "0" становится 0. В то время как явное преобразование с помощью Boolean использует другой набор правил.

**[Строгое сравнение](https://learn.javascript.ru/comparison" \l "strogoe-sravnenie)**

Использование обычного сравнения == может вызывать проблемы. Например, оно не отличает 0 от false:

alert( 0 == false ); // true

Та же проблема с пустой строкой:

alert( '' == false ); // true

Это происходит из-за того, что операнды разных типов преобразуются оператором == к числу. В итоге, и пустая строка, и false становятся нулём.

Оператор строгого равенства === проверяет равенство без приведения типов. Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования:

alert( 0 === false ); // false, так как сравниваются разные типы

Оператор строгого равенства делает код более очевидным и оставляет меньше мест для ошибок.

Ещё есть оператор строгого неравенства !==, аналогичный !=.

**[Сравнение с null и undefined](https://learn.javascript.ru/comparison" \l "sravnenie-s-null-i-undefined)**

Сравнение null и undefined между собой и с другими значениями возвращает неожииданные результаты:

* При строгом равенстве === эти значения различны, так как различны их типы.

alert( null === undefined ); // false

* При нестрогом равенстве == эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

alert( null == undefined ); // true

* При использовании математических операторов и других операторов сравнения <, >, <=, >= значения null/undefined преобразуются к числам: null становится 0, а undefined – NaN.

Сравнение null с нулём:

alert( null > 0 ); // (1) false

alert( null == 0 ); // (2) false

alert( null >= 0 ); // (3) true

Результат последнего сравнения говорит о том, что "null больше или равно нулю", тогда результат одного из сравнений выше должен быть true, но они оба ложны. Причина в том, что нестрогое равенство и сравнения >, <, >=, <= работают по-разному. Сравнения преобразуют null в число, рассматривая его как 0. Поэтому выражение (3) null >= 0 истинно, а null > 0 ложно.

С другой стороны, для нестрогого равенства == значений undefined и null действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому (2) null == 0 ложно.

Значение undefined несравнимо с другими значениями:

alert( undefined > 0 ); // false (1)

alert( undefined < 0 ); // false (2)

alert( undefined == 0 ); // false (3)

Сравнение undefined с нулём всегда ложно, по следующим причинам:

* Сравнения (1) и (2) возвращают false, потому что undefined преобразуется в NaN, а NaN – это специальное числовое значение, которое возвращает false при любых сравнениях.
* Нестрогое равенство (3) возвращает false, потому что undefined равно только null и ни чему больше.

Относитесь к любому сравнению с undefined/null, кроме строгого равенства ===, с осторожностью. Не используйте сравнения >=, >, <, <= с переменными, которые могут принимать значения null/undefined, если вы не уверены в том, что делаете. Если переменная может принимать эти значения, то добавьте для них отдельные проверки.

1. **Логические операторы.**

В JavaScript есть три логических оператора: || (ИЛИ), && (И) и ! (НЕ). Данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.

**[|| (ИЛИ)](https://learn.javascript.ru/logical-operators" \l "ili)**

Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:

result = a || b;

Традиционно в программировании ИЛИ предназначено только для манипулирования булевыми значениями: в случае, если какой-либо из аргументов true, он вернёт true, в противной ситуации возвращается false. В JavaScript этот оператор работает несколько иным образом.

Существует всего четыре возможные логические комбинации:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

Как видно, результат операций всегда равен true, за исключением случая, когда оба аргумента false. Если значение не логического типа, то оно к нему приводится в целях вычислений. Например, число 1 будет воспринято как true, а 0 – как false:

if (1 || 0) {

alert( 'truthy!' );

}

Обычно оператор || используется в if для проверки истинности любого из заданных условий. К примеру:

let hour = 9;

if (hour < 10 || hour > 18) {

alert( 'Офис закрыт.' );

}

Можно передать и больше условий:

let hour = 12;

let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {

alert( 'Офис закрыт.' ); // это выходной

}

[ИЛИ «||» находит первое истинное значение](https://learn.javascript.ru/logical-operators" \l "ili-nahodit-pervoe-istinnoeznachenie).

При выполнении ИЛИ || с несколькими значениями result = value1 || value2 || value3; оператор || выполняет следующие действия:

* Вычисляет операнды слева направо.
* Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.
* Если все операнды являются ложными (false), возвращает последний из них.

Значение возвращается в исходном виде, без преобразования.

Другими словами, цепочка ИЛИ "||" возвращает первое истинное значение или последнее, если такое значение не найдено. Например:

alert( 1 || 0 ); // 1

alert( true || 'no matter what' ); // true

alert( null || 1 ); // 1 (первое истинное значение)

alert( null || 0 || 1 ); // 1 (первое истинное значение)

alert( undefined || null || 0 ); // 0 (т.к. все ложно, возвращается последнее значение)

Такой принцип действия позволяет применять оператор нетрадиционным способом для решения некоторых задач. Например, чтобы получить первое истинное значение из списка переменных или выражений. Допустим имеется ряд переменных, которые могут содержать данные или быть null/undefined. С помощью || можно найти первую переменную с данными:

let currentUser = null;

let defaultUser = "John";

let name = currentUser || defaultUser || "unnamed";

alert( name ); // выбирается "John" – первое истинное значение

Еще один пример использования ИЛИ для сокращённого вычисления. Операндами могут быть как отдельные значения, так и произвольные выражения. ИЛИ вычисляет их слева направо. Вычисление останавливается при достижении первого истинного значения. Этот процесс называется «сокращённым вычислением», поскольку второй операнд вычисляется только в том случае, если первого не достаточно для вычисления всего выражения. Это хорошо заметно, когда выражение, указанное в качестве второго аргумента, имеет побочный эффект, например, изменение переменной. В приведённом ниже примере x не изменяется:

let x;

true || (x = 1);

alert(x); // undefined, потому что (x = 1) не вычисляется

Если бы первый аргумент имел значение false, то || приступил бы к вычислению второго и выполнил операцию присваивания:

let x;

false || (x = 1);

alert(x); // 1

Этот вариант использования || является "аналогом if". Первый операнд преобразуется в логический. Если он оказывается ложным, начинается вычисление второго. В большинстве случаев лучше использовать «обычный» if, чтобы облегчить понимание кода.

**[&& (И)](https://learn.javascript.ru/logical-operators" \l "i)**

Оператор И пишется как два амперсанда &&:

result = a && b;

В традиционном программировании И возвращает true, если оба аргумента истинны, а иначе – false:

alert( true && true ); // true

alert( false && true ); // false

alert( true && false ); // false

alert( false && false ); // false

Пример с if:

let hour = 12;

let minute = 30;

if (hour == 12 && minute == 30) {

alert( 'The time is 12:30' );

}

Как и в случае с ИЛИ, любое значение допускается в качестве операнда И:

if (1 && 0) { // вычисляется как true && false

alert( "won't work, because the result is falsy" );

}

[И «&&» находит первое ложное значение](https://learn.javascript.ru/logical-operators" \l "i-nahodit-pervoe-lozhnoeznachenie). При нескольких подряд операторах И result = value1 && value2 && value3; оператор && выполняет следующие действия:

* Вычисляет операнды слева направо.
* Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.
* Если все операнды были истинными, возвращается последний.

Другими словами, И возвращает первое ложное значение или последнее, если ничего не найдено.

Вышеуказанные правила схожи с поведением ИЛИ. Разница в том, что И возвращает первое ложное значение, а ИЛИ –  первое истинное. Примеры:

// Если первый операнд истинный,

// И возвращает второй:

alert( 1 && 0 ); // 0

alert( 1 && 5 ); // 5

// Если первый операнд ложный,

// И возвращает его. Второй операнд игнорируется

alert( null && 5 ); // null

alert( 0 && "no matter what" ); // 0

Можно передать несколько значений подряд. В таком случае возвратится первое «ложное» значение, на котором остановились вычисления.

alert( 1 && 2 && null && 3 ); // null

Когда все значения верны, возвращается последнее

alert( 1 && 2 && 3 ); // 3

Приоритет оператора И && больше, чем ИЛИ ||, плэтому он выполняется раньше. Таким образом, код a && b || c && d по существу такой же, как если бы выражения && были в круглых скобках: (a && b) || (c && d).

Как и оператор ИЛИ, И && иногда может заменять if. Например:

let x = 1;

(x > 0) && alert( 'Greater than zero!' );

Действие в правой части && выполнится только в том случае, если до него дойдут вычисления. То есть, alert сработает, если в левой части (x > 0) будет true:

let x = 1;

if (x > 0) {

alert( 'Greater than zero!' );

}

Как правило, вариант с if лучше читается и воспринимается. Он более очевиден, поэтому лучше использовать его.

**[! (НЕ)](https://learn.javascript.ru/logical-operators" \l "ne)**

Оператор НЕ представлен восклицательным знаком !. Синтаксис:

result = !value;

Оператор принимает один аргумент и выполняет следующие действия:

* Сначала приводит аргумент к логическому типу true/false.
* Затем возвращает противоположное значение.

Например:

alert( !true ); // false

alert( !0 ); // true

Двойное НЕ используют для преобразования значений к логическому типу:

alert( !!"non-empty string" ); // true

alert( !!null ); // false

Первое НЕ преобразует значение в логическому типу и возвращает обратное, а второе НЕ снова инвертирует его. В результате получится простое преобразование значения в логическое. С помощью встроенной функции Boolean можно сделать то же самое:

alert( Boolean("non-empty string") ); // true

alert( Boolean(null) ); // false

Приоритет НЕ ! является наивысшим из всех логических операторов, поэтому он всегда выполняется первым, перед && или ||.

**?? (нуллевое слияние)**

Оператор нуллевого слияния ?? это логический оператор, который возвращает значение правого операнда когда значение левого операнда равно null или undefined, в противном случае будет возвращено значение левого операнда.

В отличие от логического ИЛИ (||), левая часть оператора вычисляется и возвращается даже если его результат после приведения к логическому типу оказывается ложным, но не является null или undefined. Другими словами, если вы используете || чтобы установить значение по умолчанию, вы можете столкнуться с неожиданным поведением если считаете некоторые ложные значения пригодными для использования (например, "" или 0). Примеры:

const foo = null ?? 'default string';

console.log(foo); // "default string"

const baz = 0 ?? 42;

console.log(baz); // 0

1. **Функции.**

Зачастую нам надо повторять одно и то же действие во многих частях программы. Чтобы не повторять один и тот же код во многих местах, придуманы функции. Примеры встроенных функций: alert(message), prompt(message, default) и confirm(question). Можно создавать и свои.

**[Объявление функции](https://learn.javascript.ru/function-basics" \l "obyavlenie-funktsii)** **(Function Declaration)**

Для создания функций можно использовать объявление функции. Такой синтаксис называется *Function Declaration*. Пример объявления функции:

function showMessage() {

alert( 'Всем привет!' );

}

Вначале идёт ключевое слово function, после него имя функции, затем список параметров в круглых скобках через запятую (в примере выше он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

function имя(параметры) {

...тело...

}

Функция может быть вызвана по её имени: showMessage(). Например:

function showMessage() {

alert( 'Всем привет!' );

}

showMessage();

showMessage();

Вызов showMessage() выполняет код функции. В результате появится сообщение дважды. Этот пример явно демонстрирует одно из главных предназначений функций: избавление от дублирования кода. Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

Переменные объявленные внутри функции, видны только внутри этой функции и являются *локальными*. Например:

function showMessage() {

let message = "Привет, я JavaScript!"; // локальная переменная

alert( message );

}

showMessage(); // Привет, я JavaScript!

alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции

Функция обладает полным доступом к *внешним* переменным и может изменять их значение. Например:

let userName = 'Вася';

function showMessage() {

userName = "Петя"; // (1) изменяем значение внешней переменной

let message = 'Привет, ' + userName;

alert(message);

}

alert( userName ); // Вася

showMessage();

alert( userName ); // Петя

Внешняя переменная используется только если внутри функции нет такой локальной. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде ниже функция использует локальную переменную userName. Внешняя будет проигнорирована:

let userName = 'Вася';

function showMessage() {

let userName = "Петя"; // объявляем локальную переменную

let message = 'Привет, ' + userName; // Петя

alert(message);

}

// функция создаст и будет использовать свою собственную локальную переменную userName

showMessage();

alert( userName ); // Вася, не изменилась, функция не трогала внешнюю переменную

Переменные, объявленные снаружи всех функций, такие как внешняя переменная userName в коде выше – называются *глобальными*. Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).

Желательно сводить использование глобальных переменных к минимуму. В современном коде их нет или они используются редко.

**[Параметры](https://learn.javascript.ru/function-basics" \l "parametry)**

Можно передать внутрь функции любую информацию, используя параметры (также называемые аргументы функции). В примере ниже функции передаются два параметра: from и text.

function showMessage(from, text) { // аргументы: from, text

alert(from + ': ' + text);

}

showMessage('Аня', 'Привет!'); // Аня: Привет! (\*)

showMessage('Аня', "Как дела?"); // Аня: Как дела? (\*\*)

Когда функция вызывается в строках (\*) и (\*\*), переданные значения копируются в локальные переменные from и text. Затем они используются в теле функции.

В примере ниже, есть переменная from, и она передаётся функции. Функция изменяет значение from, но это изменение не видно снаружи, так она получает только копию значения:

function showMessage(from, text) {

from = '\*' + from + '\*';

alert( from + ': ' + text );

}

let from = "Аня";

showMessage(from, "Привет"); // \*Аня\*: Привет

alert( from ); // Аня

**[Параметры по умолчанию](https://learn.javascript.ru/function-basics" \l "parametry-po-umolchaniyu)**

Если параметр не указан, то его значением становится undefined. Например, вышеупомянутая функция showMessage(from, text) может быть вызвана с одним аргументом:

showMessage("Аня");

Это не приведёт к ошибке. Такой вызов выведет "Аня: undefined". В вызове не указан параметр text, поэтому предполагается, что text === undefined. Если необходимо задать параметру text значение по умолчанию, то надо указать его после =:

function showMessage(from, text = "текст не добавлен") {

alert( from + ": " + text );

}

showMessage("Аня"); // Аня: текст не добавлен

Теперь, если параметр text не указан, его значением будет "текст не добавлен". В данном случае "текст не добавлен" это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра. Например:

function showMessage(from, text = anotherFunction()) {

// anotherFunction() выполнится только если не передан text

// результатом будет значение text

}

**Вычисление параметров по умолчанию**

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра. В примере выше anotherFunction() будет вызываться каждый раз, когда showMessage()вызывается без параметра text.

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах. Например, явная проверка на undefined или с помощью оператора ||:

function showMessage(from, text) {

if (text === undefined) {

text = 'текст не добавлен';

}

alert( from + ": " + text );

}

function showMessage(from, text) {

// Если значение text ложно, тогда присвоить параметру text значение по умолчанию

text = text || 'текст не добавлен';

...

}

**[Возврат значения](https://learn.javascript.ru/function-basics" \l "vozvrat-znacheniya)**

Функция может возвратить результат, который будет передан в вызвавший её код. Простейшим примером может служить функция сложения двух чисел:

function sum(a, b) {

return a + b;

}

let result = sum(1, 2);

alert( result ); // 3

Директива return может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной result выше). Вызовов return может быть несколько, например:

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('А родители разрешили?');

}

}

let age = prompt('Сколько вам лет?', 18);

if ( checkAge(age) ) {

alert( 'Доступ получен' );

} else {

alert( 'Доступ закрыт' );

}

Возможно использовать return и без значения. Это приведёт к немедленному выходу из функции. Например:

function showMovie(age) {

if ( !checkAge(age) ) {

return;

}

alert( "Вам показывается кино" ); // (\*)

// ...

}

В коде выше, если checkAge(age) вернёт false, showMovie не выполнит alert. Результат функции с пустым return или без него – undefined. Если функция не возвращает значения, это то же самое, что она возвращает undefined:

function doNothing() { /\* пусто \*/ }

alert( doNothing() === undefined ); // true

Пустой return аналогичен return undefined:

function doNothing() {

return;

}

alert( doNothing() === undefined ); // true

Для длинного выражения в return не стоит добавлять перевод строки между return и его значением, 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))

Таким образом, это фактически стало пустым return. Если необходимо, чтобы возвращаемое выражение занимало несколько строк, то нужно начать его на той же строке, что и return. Или, хотя бы, поставить там открывающую скобку:

return (

some + long + expression

+ or +

whatever \* f(a) + f(b)

)

**[Выбор имени функции](https://learn.javascript.ru/function-basics" \l "function-naming)**

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть простым, точным и описывать действие функции. Чтобы программист, который будет читать код, получил верное представление о том, что делает функция. Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов. Например, функции, начинающиеся с "show" обычно что-то показывают. Примеры префиксов:

* "get…" – возвращают значение,
* "calc…" – что-то вычисляют,
* "create…" – что-то создают,
* "check…" – что-то проверяют и возвращают логическое значение, и т.д.

Примеры имён функций с префиксами:

showMessage(..) // показывает сообщение

getAge(..) // возвращает возраст

calcSum(..) // вычисляет сумму и возвращает результат

createForm(..) // создаёт форму и обычно возвращает её

checkPermission(..) // проверяет доступ, возвращая true/false

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием. Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе. Например, функция getAge должна только возвращать возвраст, а не выводить alert с возрастом; createForm – должна только создавать форму и возвращать её, а не изменять документ, добавляя форму в него; checkPermission – должна только выполнять проверку и возвращать её результат, а не отображать сообщение с текстом доступ разрешён/запрещён и т.д.

Имена функций, которые используются очень часто, иногда делают сверхкороткими. Например, во фреймворке [jQuery](http://jquery.com/) есть функция с именем $. В библиотеке [Lodash](http://lodash.com/) основная функция представлена именем \_. Это исключения. В основном имена функций должны быть в меру краткими и описательными.

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Небольшие функции не только облегчает тестирование и отладку, но и являются хорошим комментарием. Например, сравним ниже две функции showPrimes(n). Каждая из них выводит [простое число](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE) до *n*.

Первый вариант использует метку nextPrime:

function showPrimes(n) {

nextPrime: for (let i = 2; i < n; i++) {

for (let j = 2; j < i; j++) {

if (i % j == 0) continue nextPrime;

}

alert( i ); // простое

}

}

Второй вариант использует дополнительную функцию isPrime(n) для проверки на простое:

function showPrimes(n) {

for (let i = 2; i < n; i++) {

if (!isPrime(i)) continue;

alert(i); // простое

}

}

function isPrime(n) {

for (let i = 2; i < n; i++) {

if ( n % i == 0) return false;

}

return true;

}

Второй вариант легче для понимания. Не надо разбираться с кодом, сразу видно название действия (isPrime). Разработчики называют такой код самодокументируемым.

Таким образом, рекомендуется создавать функции даже если не планируется повторно использовать их. Такие функции структурируют код и делают его более понятным.

1. **Функциональные выражения и функции-стрелки**

Существует ещё один синтаксис создания функций, который называется Function Expression (Функциональное Выражение):

let sayHi = function() {

alert( "Привет" );

};

В коде выше функция создаётся и явно присваивается переменной, как любое другое значение. Не зивисимо от того, как определена функция (Function Expression или Function Declaration), это просто значение, хранимое в переменной sayHi. Можно даже вывести это значение с помощью alert:

function sayHi() {

alert( "Привет" );

}

alert( sayHi ); // выведет код функции

Обратите внимание, что последняя строка не вызывает функцию sayHi, так как после её имени нет круглых скобок. В JavaScript функции – это значения, поэтому и обращаться с ними, надо как со значениями. Код выше выведет строковое представление функции, которое является её исходным кодом.

С функцией можно делать то же самое, что и с любым другим значением. Например, скопировать функцию в другую переменную:

function sayHi() { // (1)

alert( "Привет" );

}

let func = sayHi; // (2)

func(); // Привет // (3)

sayHi(); // Привет

Рассмотрим пример подробнее:

* 1. Объявление Function Declaration (1) создало функцию и присвоило её значение переменной с именем sayHi.
  2. В строке (2) ее значение скопировано в переменную func. Обратите внимание: нет круглых скобок после sayHi. Если бы они были, то выражение func = sayHi() записало бы результат вызова sayHi() в переменную func, а не саму функцию sayHi.
  3. Теперь функция может быть вызвана с помощью обеих переменных sayHi() и func().

Можно использовать и Function Expression для того, чтобы создать sayHi в первой строке. Результат будем таким же:

let sayHi = function() {

alert( "Привет" );

};

let func = sayHi;

// ...

Заметьте, что в Function Expression ставится точка с запятой ; в конце, а в Function Declaration нет:

function sayHi() {

// ...

}

let sayHi = function() {

// ...

};

Это потому, что Function Expression использует внутри себя инструкции присваивания let sayHi = ...; как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой не относится непосредственно к Function Expression, она лишь завершает инструкцию.

**[Функции-«колбэки»](https://learn.javascript.ru/function-expressions-arrows" \l "funktsii-kolbeki)**

Рассмотрим 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». В браузерах такие функции обычно отображают красивые диалоговые окна. Функция задает вопрос question и, в зависимости от того, как ответит пользователь, вызвать yes() или no():

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

function showOk() {

alert( "Вы согласны." );

}

function showCancel() {

alert( "Вы отменили выполнение." );

}

ask("Вы согласны?", showOk, showCancel);

Аргументы функции ask ещё называют функциями-колбэками или просто колбэками (от англ. «call back» – обратный вызов). Т.е. функция передаётся в качестве агрумента и вызывается обратно тогда, когда это необходимо. В нашем случае, showOk становится колбэком для ответа «yes», а showCancel – для ответа «no».

Можно переписать пример короче, используя Function Expression:

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

ask(

"Вы согласны?",

function() { alert("Вы согласились."); },

function() { alert("Вы отменили выполнение."); }

);

Здесь функции объявляются прямо внутри вызова ask(...). У них нет имён, поэтому они называются анонимными. Такие функции недоступны снаружи ask (потому что они не присвоены переменным).

Одним из отличий Function Expression от Function Declaration, кроме синтаксиса, является то, что 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. ).

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. Обычно этот процесс называют «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. Например, так будет работать:

sayHi("Вася"); // Привет, Вася

function sayHi(name) {

alert( `Привет, ${name}` );

}

Функция sayHi была создана, когда движок JavaScript подготавливал скрипт к выполнению, и такая функция видна повсюду в этом скрипте. Если бы это было Function Expression, то такой код привел бы к ошибке:

sayHi("Вася");

let sayHi = function(name) {

alert( `Привет, ${name}` );

};

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. Для примера представим, что нужно создать функцию welcome() в зависимости от значения переменной age, которое будет получено во время выполнения кода. Такой код, использующий Function Declaration, работать не будет:

let age = prompt("Сколько Вам лет?", 18);

// в зависимости от условия объявляем функцию

if (age < 18) {

function welcome() {

alert("Привет!");

}

} else {

function welcome() {

alert("Здравствуйте!");

}

}

welcome(); // Error: welcome is not defined

Это произошло, так как объявление Function Declaration видимо только внутри блока кода, в котором располагается.

Верным подходом будет воспользоваться функцией, объявленной при помощи Function Expression, и присвоить значение welcome переменной, объявленной снаружи if, что обеспечит нужную видимость. Такой код работает как требуется:

let age = prompt("Сколько Вам лет?", 18);

let welcome;

if (age < 18) {

welcome = function() {

alert("Привет!");

};

} else {

welcome = function() {

alert("Здравствуйте!");

};

}

welcome(); // теперь всё в порядке

Можно упростить этот код ещё больше, используя условный оператор ?:

let age = prompt("Сколько Вам лет?", 18);

let welcome = (age < 18) ?

function() { alert("Привет!"); } :

function() { alert("Здравствуйте!"); };

welcome();

**[Функции-стрелки](https://learn.javascript.ru/function-expressions-arrows" \l "arrow-functions)**

Существует ещё более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression. Он называется функции-стрелки или стрелочные функции (arrow functions), т.к. выглядит следующим образом:

let func = (arg1, arg2, ...argN) => expression

Такой код создаёт функцию func с аргументами arg1..argN и вычисляет expression с правой стороны с их использованием, возвращая результат. Это то же самое, что и:

let func = function(arg1, arg2, ...argN) {

return expression;

};

Рассмотрим пример:

let sum = (a, b) => a + b;

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

Если у передается только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:

let double = n => n \* 2;

alert( double(3) ); // 6

Если нет аргументов, используются пустые круглые скобки (их указывать обязательно):

let sayHi = () => alert("Hello!");

sayHi();

В примерах выше аргументы использовались слева от =>, а справа вычислялось выражение с их значениями. Но если требуется вычислить несколько выражений или инструкций, то необходимо заключить такие выражения в фигурные скобки с использованием директивы return внутри них, как в обычной функции. Например:

let sum = (a, b) => {

let result = a + b;

return result;

};

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

Важной особенностью стрелочных функций является то, что у них нет переменной arguments.

**Вторая тема**

1. **Методы примитивов.**

JavaScript позволяет работать с примитивными типами данных – строками, числами, и т.д., так, как если бы они были объектами. У них есть и методы. Рассмотрим ключевые различия между примитивами и объектами.

Примитив – это значение «примитивного» типа. Есть 6 примитивных типов: string, number, boolean, symbol, null и undefined.

Объект может хранить множество значений как свойства. Объявляется при помощи фигурных скобок {}, например: {name: "Рома", age: 30}. В JavaScript есть и другие виды объектов: например, функции тоже являются объектами.

Одна из лучших особенностей объектов – это то, что можно хранить функцию как одно из свойств объекта:

let roma = {

name: "Рома",

sayHi: function() {

alert("Привет, дружище!");

}

};

roma.sayHi(); // Привет, дружище!

Здесь создан объект roma с методом sayHi.

Существует множество встроенных объектов. Например, те, которые работают с датами, ошибками, HTML-элементами и т.д. Они имеют различные свойства и методы. Поэтому объекты «тяжелее» примитивов, они нуждаются в дополнительных ресурсах для поддержания внутренней структуры.

Язык JavaScript позволяет осуществлять доступ к методам и свойствам строк, чисел, булевых значений и символов. Чтобы это работало, при таком доступе создаётся специальный «объект-обёртка», который предоставляет нужный функционал, а после удаляется. Каждый примитив имеет свой собственный «объект-обёртку», которые называются: String, Number, Boolean и Symbol. Таким образом, они имеют разный набор методов. К примеру, существует метод [str.toUpperCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase), который возвращает cтроку в верхнем регистре:

let str = "Привет";

alert( str.toUpperCase() );

Рассмотрим, как это работет:

1. Строка str – примитив. В момент обращения к его свойству, создаётся специальный объект, который содержит значение строки и метод toUpperCase().
2. Этот метод вызывается и возвращает новую строку (выводится в alert).
3. Специальный объект удаляется, оставляя только примитив str.

Таким образом, примитивы могут предоставлять методы, и в то же время оставаться «лёгкими». Движок JavaScript сильно оптимизирует этот процесс. Он даже может пропустить создание специального объекта.

Число имеет собственный набор методов. Например, [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) округляет число до *n* знаков после запятой.

let n = 1.23456;

alert( n.toFixed(2) ); // 1.23

В JavaScript также можно явное создавать «объекты-обёртки» с помощью конструкторов String/Number/Boolean (new Number(1) или new Boolean(false)), но это очень не рекомендуется, так как иногда приводит к необжиданным результатам:

alert( typeof 0 ); // "число"

alert( typeof new Number(0) ); // "object"

Объекты в if всегда дают true, поэтому в примере ниже будет показан alert:

let zero = new Number(0);

if (zero) {

// zero возвращает "true", так как является объектом

alert( "zero имеет «истинное» значение?!?" );

}

С другой стороны, использование функций String/Number/Boolean (без оператора new) полезно, так как они превращают значение в соответствующий примитивный тип: в строку, в число, в булевой тип. Например:

let num = Number("123"); // превращает строку в число

Особенные примитивы null and undefined являются исключениями – у них нет соответствующих «объектов-обёрток» и они не имеют никаких методов.

1. **Числа.**

Все числа в JavaScript хранятся в 64-битном формате [IEEE-754](http://en.wikipedia.org/wiki/IEEE_754-1985), который также называют «числа с плавающей точкой двойной точности» (double precision floating point numbers).

**[Способы записи числа](https://learn.javascript.ru/number" \l "sposoby-zapisi-chisla)**

Допустим надо записать число 1 миллиард:

let billion = 1000000000;

Но в реальной жизни обычно множество нулей опускается, а укороченная запись может выглядеть как "1млрд" или "7.3млрд".  Такой принцип работает для всех больших чисел. В JavaScript можно использовать букву "e", чтобы укоротить запись числа. Она добавляется к числу и заменяет указанное количество нулей:

let billion = 1e9; // 1 миллиард

alert( 7.3e9 ); // 7.3 миллиардов

Другими словами, "e" производит операцию умножения числа на 1 с указанным количеством нулей.

1e3 = 1 \* 1000

1.23e6 = 1.23 \* 1000000

Допустим необходимо записать что-нибудь очень маленькое: 1 микросекунду (одна миллионная секунды):

let ms = 0.000001;

Записать микросекунду в укороченном виде также можно с помощью "e".

let ms = 1e-6; // шесть нулей, слева от 1

Т.е., отрицательное число после "e" подразумевает деление на 1 с указанным количеством нулей:

// -3 делится на 1 с 3 нулями

1e-3 = 1 / 1000 (=0.001)

// -6 делится на 1 с 6 нулями

1.23e-6 = 1.23 / 1000000 (=0.00000123)

**[Шестнадцатеричные, двоичные и восьмеричные числа](https://learn.javascript.ru/number" \l "shestnadtsaterichnye-dvoichnye-i-vosmerichnye-chisla)**

[Шестнадцатеричные](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D1%81%D1%82%D0%BD%D0%B0%D0%B4%D1%86%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) числа широко используются в JavaScript для представления цветов, кодировки символов и многое другое. Естественно, есть короткий стиль записи: 0x, после которого указывается число. Например:

alert( 0xff ); // 255

alert( 0xFF ); // 255 (регистр не имеет значения)

Не так часто используются двоичные и восьмеричные числа, но они также поддерживаются 0b для двоичных и 0o для восьмеричных:

let a = 0b11111111; // бинарная форма записи числа 255

let b = 0o377; // восьмеричная форма записи числа 255

alert( a == b ); // true

Есть только 3 системы счисления с такой поддержкой. Для других систем счисления рекомендуется использовать функцию parseInt (рассмотрим позже).

**[toString(base)](https://learn.javascript.ru/number" \l "tostring-base)**

Метод num.toString(base) возвращает строковое представление числа *num* в системе счисления base. Например:

let num = 255;

alert( num.toString(16) ); // ff

alert( num.toString(2) ); // 11111111

Значение *base* может варьироваться от 2 до 36 (по умолчанию 10):

* base = 16 – для шестнадцатеричного представления цвета, кодировки символов и т.д., цифры могут быть 0..9 или A..F.
* base = 2 – обычно используется для отладки побитовых операций, цифры 0 или 1.
* base = 36 – максимальное основание, цифры могут быть 0..9 или A..Z. То есть, используется весь латинский алфавит для представления числа.

Две точки в 123456..toString(36) используется, если надо вызвать метод toString непосредственно на числе. Если поставить одну точку: 123456.toString(36), тогда это будет ошибкой, поскольку синтаксис JavaScript предполагает, что после первой точки начинается десятичная часть. А если поставить две точки, то JavaScript понимает, что десятичная часть отсутствует, и начинается метод. Также можно записать как (123456).toString(36).

**[Округление](https://learn.javascript.ru/number" \l "okruglenie)**

Одна из часто используемых операций при работе с числами – это округление. В JavaScript есть несколько встроенных функций для работы с округлением:

* Math.floor – округление в меньшую сторону: 3.1 становится 3, а -1.1 – -2.
* Math.ceil – округление в большую сторону: 3.1 становится 4, а -1.1 – -1.
* Math.round – округление до ближайшего целого: 3.1 становится 3, 3.6 – 4, а -1.1 – -1.
* Math.trunc (не поддерживается в Internet Explorer) – производит удаление дробной части без округления: 3.1 становится 3, а -1.1 – -1.

Таблица с различиями между функциями округления:

|  | **Math.floor** | **Math.ceil** | **Math.round** | **Math.trunc** |
| --- | --- | --- | --- | --- |
| 3.1 | 3 | 4 | 3 | 3 |
| 3.6 | 3 | 4 | 4 | 3 |
| -1.1 | -2 | -1 | -1 | -1 |
| -1.6 | -2 | -1 | -2 | -1 |

Эти функции охватывают все возможные способы обработки десятичной части. Если надо округлить число до *n*-ого количества цифр в дробной части, то это можно сделать одним из следующих способов:

1. Умножить и разделить.

Например, чтобы округлить число до второго знака после запятой, можно умножить число на 100, вызвать функцию округления и разделить обратно.

let num = 1.23456;

alert( Math.floor(num \* 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23

1. Метод [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) округляет число до *n* знаков после запятой и возвращает строковое представление результата.

Если десятичная часть короче, чем необходима, будут добавлены нули в конец строки:

let num = 12.34;

alert( num.toFixed(5) ); // "12.34000", добавлены нули, чтобы получить 5 знаков после запятой

Можно преобразовать полученное значение в число, используя унарный оператор + или Number(), пример с унарным оператором: +num.toFixed(5).

**[Неточные вычисления](https://learn.javascript.ru/number" \l "netochnye-vychisleniya)**

Внутри JavaScript число представлено в виде 64-битного формата [IEEE-754](https://ru.wikipedia.org/wiki/IEEE_754-1985). Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 из них для хранения положения десятичной точки (если число целое, то хранится 0), и один бит отведён на хранения знака. Если число слишком большое, оно переполнит 64-битное хранилище, JavaScript вернёт бесконечность:

alert( 1e500 ); // Infinity

Наиболее часто встречающаяся ошибка при работе с числами в JavaScript – это потеря точности. Например, сумма 0.1 и 0.2 не равна 0.3:

alert( 0.1 + 0.2 == 0.3 ); // false

alert( 0.1 + 0.2 ); // 0.30000000000000004

Так происходит потому, что число хранится в памяти в бинарной форме, как последовательность бит – единиц и нулей. Но дроби, такие как 0.1, 0.2, которые выглядят довольно просто в десятичной системе счисления, на самом деле являются бесконечной дробью в двоичной форме.

Число 0.1 – это единица, делённая на десять – 1/10, одна десятая. В десятичной системе счисления такие числа легко представимы, по сравнению с одной третьей: 1/3, которая становится бесконечной дробью 0.33333(3). Деление на 10 хорошо работает в десятичной системе, но деление на 3 – нет. По той же причине и в двоичной системе счисления, деление на 2 обязательно сработает, а 1/10 становится бесконечной дробью.

В JavaScript нет возможности для хранения точных значений 0.1 или 0.2, используя двоичную систему, точно также, как нет возможности хранить одну третью в десятичной системе счисления. Числовой формат IEEE-754 решает эту проблему путём округления до ближайшего возможного числа. Правила округления обычно не позволяют увидеть эту нехначительную потерю точности, но она существует:

alert( 0.1.toFixed(20) ); // 0.10000000000000000555

Когда складываются 2 числа, их неточности тоже суммируются, поэтому 0.1 + 0.2 – это не совсем 0.3.

Наиболее надёжный способ обойти эту проблему – это округлить результат используя метод [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed):

let sum = 0.1 + 0.2;

alert( sum.toFixed(2) ); // 0.30

Метод toFixed всегда возвращает строку. Это гарантирует, что результат будет с заданным количеством цифр в десятичной части. В других случаях можно использовать унарный оператор +, чтобы преобразовать строку в число:

let sum = 0.1 + 0.2;

alert( +sum.toFixed(2) ); // 0.3

Также можно умножить число на 100 (или на большее), чтобы привести его к целому, выполнить математические действия, а после разделить обратно. Суммируя целые числа, мы уменьшаем погрешность, но она все равно появляется при финальном делении:

alert( (0.1 \* 10 + 0.2 \* 10) / 10 ); // 0.3

alert( (0.28 \* 100 + 0.14 \* 100) / 100); // 0.4200000000000001

Таким образом, метод умножения/деления уменьшает погрешность, но полностью её не решает.

Еще одно следствие внутреннего представления чисел – наличие двух нулей: 0 и -0. Это возможно потому, что знак представлен отдельным битом, так что, любое число может быть положительным и отрицательным, включая нуль. В большинстве случаев это поведение незаметно, так как операторы в JavaScript воспринимают их одинаковыми.

**[Проверка: isFinite и isNaN](https://learn.javascript.ru/number" \l "proverka-isfinite-i-isnan)**

Специальные числовые значения Infinity ( -Infinity) и NaN  принадлежат типу number, но они не являются «обычными» числами, поэтому есть функции для их проверки:

* isNaN(value) преобразует значение в число и проверяет является ли оно NaN:

alert( isNaN(NaN) ); // true

alert( isNaN("str") ); // true

Нельзя просто сравнить === NaN, так как значение NaN уникально тем, что оно не является равным ни чему другому, даже самому себе:

alert( NaN === NaN ); // false

* isFinite(value) преобразует аргумент в число и возвращает true, если оно является обычным числом, т.е. не NaN/Infinity/-Infinity:

alert( isFinite("15") ); // true

alert( isFinite("str") ); // false

alert( isFinite(Infinity) ); // false

Иногда isFinite используется для проверки, содержится ли в строке число:

let num = +prompt("Enter a number", '');

alert( isFinite(num) );

Не стоит забывать, что пустая строка интерпретируется как 0 во всех числовых функциях, включая isFinite.

**Сравнение Object.is**

Существует специальный метод [Object.is](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/is), который сравнивает значения примерно как ===, но более надёжен в двух особых ситуациях:

1. Сравнивает NaN: Object.is(NaN, NaN) === true.
2. Значения 0 и -0 разные: Object.is(0, -0) === false, это редко используется, но технически эти значения разные.

Во всех других случаях Object.is(a, b) идентичен a === b.

Этот способ сравнения часто используется в спецификации JavaScript. Когда алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует Object.is.

**[parseInt и parseFloat](https://learn.javascript.ru/number" \l "parseint-i-parsefloat)**

Для явного преобразования к числу можно использовать + или Number(). Если строка не является в точности числом, то результат будет NaN:

alert( +"100px" ); // NaN

Единственное исключение – это пробелы в начале строки и в конце, они игнорируются. На практике часто встречаются значения, у которых есть единица измерения, например, "100px" или "12pt" в CSS, во множестве стран символ валюты записывается после номинала "19€".

Чтобы получить числовое значение из таких строк есть методы parseInt и parseFloat. Они считывают число из строки. Если в процессе чтения возникает ошибка, они возвращают полученное до ошибки число. Функция parseInt возвращает целое число, а parseFloat возвращает число с плавающей точкой:

alert( parseInt('100px') ); // 100

alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12

alert( parseFloat('12.3.4') ); // 12.3

Функции parseInt/parseFloat вернут NaN, если не смогли прочитать ни одну цифру:

alert( parseInt('a123') ); // NaN

Функция parseInt(str, radix) имеет необязательный второй параметр. Он определяет систему счисления, таким образом parseInt может также читать строки с шестнадцатеричными числами, двоичными числами и т.д.:

alert( parseInt('0xff', 16) ); // 255

alert( parseInt('ff', 16) ); // 255

alert( parseInt('2n9c', 36) ); // 123456

**[Другие математические функции](https://learn.javascript.ru/number" \l "drugie-matematicheskie-funktsii)**

В JavaScript встроен объект [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math), который содержит различные математические функции и константы. Несколько примеров:

* Math.random() – возвращает псевдослучайное число в диапазоне [0, 1)

alert( Math.random() ); // 0.1234567894322

alert( Math.random() ); // 0.5435252343232

alert( Math.random() ); // ...

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

alert( Math.max(3, 5, -10, 0, 1) ); // 5

alert( Math.min(1, 2) ); // 1

* Math.pow(n, power) – возвращает число n, возведённое в степень power

alert( Math.pow(2, 10) ); // 2 в степени 10 = 1024

В объекте Math есть множество других функций и констант, включая тригонометрические функции, с которыми подробнее можно ознакомиться в документации по объекту [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math).

1. **Строки.**

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков. Внутренний формат для строк – всегда [UTF-16](https://ru.wikipedia.org/wiki/UTF-16), вне зависимости от кодировки страницы.

**[Кавычки](https://learn.javascript.ru/string" \l "kavychki)**

В JavaScript есть разные типы кавычек. Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

let single = 'single-quoted';

let double = "double-quoted";

let backticks = `backticks`;

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку можно вставлять произвольные выражения, обернув их в ${…}:

function sum(a, b) {

return a + b;

}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.

Ещё одно преимущество обратных кавычек – они могут занимать более одной строки, вот так:

let guestList = `Guests:

\* John

\* Pete

\* Mary

`;

alert(guestList);

Если использовать точно так же одинарные или двойные кавычки, то будет ошибка:

let guestList = "Guests: // Error: Unexpected token ILLEGAL

\* John";

Обратные кавычки также позволяют задавать «шаблонную функцию» перед первой обратной кавычкой. Используемый синтаксис: func`string`. Автоматически вызываемая функция func получает строку и встроенные в неё выражения и может их обработать. Подробнее об этом можно прочитать в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals" \l "Tagged_template_literals). Если перед строкой есть выражение, то шаблонная строка называется «теговым шаблоном». Это позволяет использовать свою шаблонизацию для строк, но на практике теговые шаблоны применяются редко.

**[Спецсимволы](https://learn.javascript.ru/string" \l "spetssimvoly)**

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя так называемый «символ перевода строки», который записывается как \n:

let guestList = "Guests:\n \* John\n \* Pete\n \* Mary";

alert(guestList); // список гостей, состоящий из нескольких строк

В частности, эти две строки эквивалентны, просто записаны по-разному:

let str1 = "Hello\nWorld";

let str2 = `Hello

World`;

alert(str1 == str2); // true

Есть и другие, реже используемые спецсимволы:

| **Символ** | **Описание** |
| --- | --- |
| \n | Перевод строки |
| \r | Возврат каретки: самостоятельно не используется. В текстовых файлах Windows для перевода строки используется комбинация символов \r\n. |
| \', \" | Кавычки |
| \\ | Обратный слеш |
| \t | Знак табуляции |
| \b, \f, \v | Backspace, Form Feed и Vertical Tab – оставлены для обратной совместимости, сейчас не используются. |
| \xXX | Символ с шестнадцатеричным юникодным кодом XX, например, '\x7A' – то же самое, что 'z'. |
| \uXXXX | Символ в кодировке UTF-16 с шестнадцатеричным кодом XXXX, например, \u00A9 – юникодное представление знака копирайта, ©. Код должен состоять ровно из 4 шестнадцатеричных цифр. |
| \u{X…XXXXXX} (от 1 до 6 шестнадцатеричных цифр) | Символ в кодировке UTF-32 с шестнадцатеричным кодом от U+0000 до U+10FFFF. Некоторые редкие символы кодируются двумя 16-битными словами и занимают 4 байта. Так можно вставлять символы с длинным кодом. |

Примеры с Юникодом:

// ©

alert( "\u00A9" );

// 佫, редкий китайский иероглиф

alert( "\u{20331}" );

// 😍

alert( "\u{1F60D}" );

Все спецсимволы начинаются с обратного слеша, \ – так называемого «символа экранирования». Он также используется, если необходимо вставить в строку кавычку. Например:

alert( 'I\'m the Walrus!' ); // I'm the Walrus!

Здесь перед входящей в строку кавычкой необходимо добавить обратный слеш  \, иначе она бы обозначала окончание строки. Требование экранировать относится только к таким же кавычкам, как те, в которые заключена строка. Можно использовать для этой строки двойные или обратные кавычки:

alert( `I'm the Walrus!` ); // I'm the Walrus!

Заметим, что обратный слеш \ служит лишь для корректного прочтения строки интерпретатором, но он не записывается в строку после её прочтения. Когда строка сохраняется в оперативную память, в неё не добавляется символ \. Это можно увидеть в выводах alert в примерах выше. Но если надо добавить в строку сам обратный слеш \, то это можно сделать, добавив перед ним ещё один обратный слеш:

alert( `The backslash: \\` ); // The backslash: \

**[Длина строки](https://learn.javascript.ru/string" \l "dlina-stroki)**

Свойство length содержит длину строки:

alert( `My\n`.length ); // 3

Обратите внимание, \n – это один спецсимвол, поэтому длина строки 3.

Так как str.length – это числовое свойство, а не функция, добавлять скобки не нужно.

**[Доступ к символам](https://learn.javascript.ru/string" \l "dostup-k-simvolam)**

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод charAt: [str.charAt(pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/charAt). Первый символ занимает нулевую позицию:

let str = `Hello`;

// первый символ

alert( str[0] ); // H

alert( str.charAt(0) ); // H

// последний символ

alert( str[str.length - 1] ); // o

Квадратные скобки – современный способ получить символ, в то время как charAt существует в основном по историческим причинам. Разница только в том, что если символ с такой позицией отсутствует, тогда [] вернёт undefined, а charAt – пустую строку:

let str = `Hello`;

alert( str[1000] ); // undefined

alert( str.charAt(1000) ); // ''

Также можно перебрать строку посимвольно, используя for..of:

for (let char of "Hello") {

alert(char); // H,e,l,l,o

}

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана – она такая навсегда:

let str = 'Hi';

str[0] = 'h'; // ошибка

alert( str[0] ); // не работает

Можно создать новую строку и записать её в ту же самую переменную вместо старой. Например:

let str = 'Hi';

str = 'h' + str[1]; // заменяем строку

alert( str ); // hi

**[Изменение регистра](https://learn.javascript.ru/string" \l "izmenenie-registra)**

Методы [toLowerCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase) и [toUpperCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) меняют регистр символов:

alert( 'Interface'.toUpperCase() ); // INTERFACE

alert( 'Interface'.toLowerCase() ); // interface

Если необходимо перевести в нижний регистр какой-то конкретный символ:

alert( 'Interface'[0].toLowerCase() ); // 'i'

**[Поиск подстроки](https://learn.javascript.ru/string" \l "poisk-podstroki)**

Существует несколько способов поиска подстроки.

Первый метод – [str.indexOf(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf). Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений. Например:

let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, т.к. подстрока 'Widget' найдена в начале

alert( str.indexOf('widget') ); // -1, поиск чувствителен к регистру

alert( str.indexOf("id") ); // 1

Необязательный второй аргумент позволяет начать поиск с определённой позиции. Например, первое вхождение "id" – на позиции 1. Для того, чтобы найти следующее, необходимо начать поиск с позиции 2:

let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12

Чтобы найти все вхождения подстроки, нужно запустить indexOf в цикле. Каждый раз, получив очередную позицию, начинаем новый поиск со следующей:

let str = 'Ослик Иа-Иа посмотрел на виадук';

let target = 'Иа'; // цель поиска

let pos = 0;

while (true) {

let foundPos = str.indexOf(target, pos);

if (foundPos == -1) break;

alert( `Найдено тут: ${foundPos}` );

pos = foundPos + 1; // продолжаем со следующей позиции

}

Также есть похожий метод [str.lastIndexOf(substr, position)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf), который ищет с конца строки к её началу. Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.

**Методы [includes, startsWith, endsWith](https://learn.javascript.ru/string" \l "includes-startswith-endswith)**

Более современный метод [str.includes(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/includes) возвращает true, если в строке str есть подстрока substr, либо false, если нет. Стоит его использовать, если необходимо проверить, есть ли совпадение, но позиция не нужна:

alert( "Widget with id".includes("Widget") ); // true

alert( "Hello".includes("Bye") ); // false

Необязательный второй аргумент str.includes позволяет начать поиск с определённой позиции:

alert( "Midget".includes("id") ); // true

alert( "Midget".includes("id", 3) ); // false, поиск начат с позиции 3

Методы [str.startsWith](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/startsWith) и [str.endsWith](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/endsWith) проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

alert( "Widget".startsWith("Wid") ); // true, "Wid" – начало "Widget"

alert( "Widget".endsWith("get") ); // true, "get" – окончание "Widget"

**[Получение подстроки](https://learn.javascript.ru/string" \l "poluchenie-podstroki)**

В JavaScript есть 3 метода для получения подстроки: substring, substr и slice.

* str.slice(start [, end]) – возвращает часть строки от start до (не включая) end.

Например:

let str = "stringify";

// 'strin', символы от 0 до 5 (не включая 5)

alert( str.slice(0, 5) );

// 's', от 0 до 1, не включая 1

alert( str.slice(0, 1) );

Если аргумент end отсутствует, slice возвращает символы до конца строки:

let str = "stringify";

alert( str.slice(2) ); // ringify, с позиции 2 и до конца

Также для start/end можно задавать отрицательные значения. Это означает, что позиция определена как заданное количество символов с конца строки:

let str = "stringify";

// начинаем с позиции 4 справа, заканчиваем на позиции 1 справа

alert( str.slice(-4, -1) ); // gif

* str.substring(start [, end]) – возвращает часть строки между start и end.

Это почти то же, что и slice, но можно задавать start больше end. Например:

let str = "stringify";

alert( str.substring(2, 6) ); // "ring"

alert( str.substring(6, 2) ); // "ring"

alert( str.slice(2, 6) ); // "ring"

alert( str.slice(6, 2) ); // ""

Отрицательные значения substring, в отличие от slice, не поддерживает, они интерпретируются как 0.

* str.substr(start [, length]) – возвращает часть строки от start длины length.

В противоположность предыдущим методам, этот позволяет указать длину вместо конечной позиции:

let str = "stringify";

alert( str.substr(2, 4) ); // ring

Значение первого аргумента может быть отрицательным, тогда позиция определяется с конца:

let str = "stringify";

alert( str.substr(-4, 2) ); // gi

Сравнительная таблица рассмотренных выше методов:

| **Метод** | **Диапазон** | **Отрицательные значения** |
| --- | --- | --- |
| slice(start, end) | от start до end (не включая end) | можно передавать отрицательные значения |
| substring(start, end) | между start и end | отрицательные значения равнозначны 0 |
| substr(start, length) | length символов, начиная от start | значение start может быть отрицательным |

Метод slice более гибок, он поддерживает отрицательные аргументы, и его короче писать, поэтому его используют наиболее часто.

**[Сравнение строк](https://learn.javascript.ru/string" \l "sravnenie-strok)**

Строки сравниваются посимвольно в алфавитном порядке. Тем не менее, есть некоторые нюансы:

1. Строчные буквы больше заглавных:

alert( 'a' > 'Z' ); // true

1. Буквы, имеющие диакритические знаки, идут «не по порядку»:

alert( 'Österreich' > 'Zealand' ); // true

Это может привести к своеобразным результатам при сортировке названий стран: ожидается, что Zealand будет после Österreich в списке. Чтобы разобраться, что происходит на самом деле, необходимо ознакомимся с внутренним представлением строк в JavaScript.

Строки кодируются в [UTF-16](https://ru.wikipedia.org/wiki/UTF-16). Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот:

* str.codePointAt(pos) – возвращает код для символа, находящегося на позиции pos. Одна и та же буква в нижнем и верхнем регистре будет иметь разные коды:

alert( "z".codePointAt(0) ); // 122

alert( "Z".codePointAt(0) ); // 90

* String.fromCodePoint(code) – создаёт символ по его коду code

alert( String.fromCodePoint(90) ); // Z

Также можно добавлять юникодные символы по их кодам, используя \u с шестнадцатеричным кодом символа:

// 90 – 5a в шестнадцатеричной системе счисления

alert( '\u005a' ); // Z

Можно сформировать строку, содержащую символы с кодами от 65 до 220 – это латиница и ещё некоторые распространённые символы:

let str = '';

for (let i = 65; i <= 220; i++) {

str += String.fromCodePoint(i);

}

alert( str );

// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~

// ¡¢£¤¥¦§¨©ª«¬­®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ

Как видино, сначала идут заглавные буквы, затем несколько спецсимволов, затем строчные и Ö ближе к концу вывода. Это объясняет, почему a > Z. Символы сравниваются по их кодам. Больший код – больший символ. Код a (97) больше кода Z (90):

* Все строчные буквы идут после заглавных, так как их коды больше.
* Некоторые буквы, такие как Ö, находятся вне основного алфавита. У этой буквы код больше, чем у любой буквы от a до z.

**[Правильное сравнение](https://learn.javascript.ru/string" \l "pravilnoe-sravnenie)**

«Правильный» алгоритм сравнения строк сложнее, чем может показаться, так как разные языки используют разные алфавиты. Поэтому браузеру нужно знать, какой язык использовать для сравнения. Все современные браузеры (для IE10 нужна дополнительная библиотека [Intl.JS](https://github.com/andyearnshaw/Intl.js/)) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), обеспечивающий правильное сравние строк на разных языках с учётом их правил. Для этого есть соответствующий метод.

Вызов [str.localeCompare(str2)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare) возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

* Отрицательное число, если str меньше str2.
* Положительное число, если str больше str2.
* 0, если строки равны.

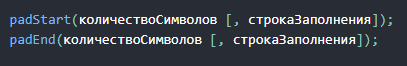
Например:

alert( 'Österreich'.localeCompare('Zealand') ); // -1

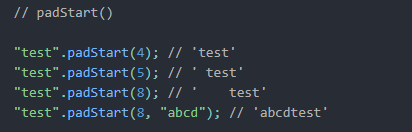
У этого метода есть два дополнительных аргумента, которые указаны в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare). Первый позволяет указать язык (по умолчанию берётся из окружения) – от него зависит порядок букв. Второй – определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между "a" и "á".

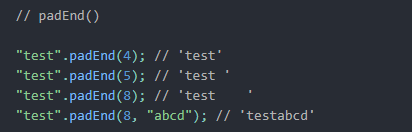
**Заполнение строки**

Целью заполнения строки является добавление символов в строку, чтобы она достигла определенной длины. ES2017 представляет два метода для строк padStart() и padEnd() - которые позволяют добавлять либо пустую строку, либо любую другую строку к началу или, концу исходной строки. Это оказывается удобным, если нужно выровнять текст, например, при выводе в консоль.



Пример использования:





1. **Массивы.**

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:

let arr = new Array();

let arr = [];

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

let fruits = ["Яблоко", "Апельсин", "Слива"];

Элементы массива нумеруются, начиная с нуля. Можно получить элемент, указав его номер в квадратных скобках:

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко

alert( fruits[1] ); // Апельсин

alert( fruits[2] ); // Слива

Можно заменить элемент:

fruits[2] = 'Груша'; // ["Яблоко", "Апельсин", "Груша"]

Или добавить новый элемент к существующему массиву:

fruits[3] = 'Лимон'; // ["Яблоко", "Апельсин", "Груша", "Лимон"]

Общее число элементов массива содержится в его свойстве length:

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits.length ); // 3

Вывести массив целиком можно при помощи alert:

let fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits ); // Яблоко, Апельсин, Слива

В массиве могут храниться элементы любого типа:

let arr = [ 'Яблоко', { name: 'Джон' }, true, function() { alert('привет'); } ];

alert( arr[1].name ); // Джон

arr[3](); // привет

**«Висячая» запятая**

Массив может оканчиваться запятой:

let fruits = [

"Яблоко",

"Апельсин",

"Слива",

];

«Висячая» запятая упрощает процесс добавления/удаления элементов, так как все строки становятся идентичными.

**[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. Можно 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:

let fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.pop() ); // удаляем "Груша" и выводим его

alert( fruits ); // Яблоко, Апельсин

* push – добавляет элемент в конец массива:

let fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");

alert( fruits ); // Яблоко, Апельсин, Груша

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

Методы, работающие с началом массива:

* shift – удаляет из массива первый элемент и возвращает его:

let fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.shift() ); // удаляем Яблоко и выводим его

alert( fruits ); // Апельсин, Груша

* unshift – добавляет элемент в начало массива:

let fruits = ["Апельсин", "Груша"];

fruits.unshift('Яблоко');

alert( fruits ); // Яблоко, Апельсин, Груша

Методы push и unshift могут добавлять сразу несколько элементов:

let fruits = ["Яблоко"];

fruits.push("Апельсин", "Груша");

fruits.unshift("Ананас", "Лимон");

alert( fruits ); // ["Ананас", "Лимон", "Яблоко", "Апельсин", "Груша"]

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

Массив – это особый подвид объектов. Квадратные скобки, используемые для того, чтобы получить доступ к свойству arr[0] – это обычный синтаксис доступа по ключу, как obj[key], где в роли obj выступает 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 все равно лежит объект.

Следует помнить, что в JavaScript существует всего 7 основных типов данных. Массив является объектом и, следовательно, ведёт себя как объект. Например, копируется по ссылке:

let fruits = ["Банан"]

let arr = fruits; // копируется по ссылке

alert( arr === fruits ); // true

arr.push("Груша"); // массив заполняется по ссылке

alert( fruits ); // Банан, Груша

Но то, что действительно делает массивы особенными – это их внутреннее представление. Движок JavaScript старается хранить элементы массива в непрерывной области памяти, один за другим. Существуют и другие способы оптимизации, благодаря которым массивы работают очень быстро. Но все они утратят эффективность, если перестать работать с массивом как с «упорядоченной коллекцией данных», и начать использовать его как обычный объект. Например, технически, можно сделать следующее:

let fruits = []; // создаём массив

fruits[99999] = 5; // создаём свойство с индексом, больше длины массива

fruits.age = 25; // создаём свойство с произвольным именем

Это возможно, потому что в основе массива лежит объект. Можно присвоить ему любые свойства. Но движок поймёт, что осуществляется работа с массивом, как с обычным объектом. Способы оптимизации, используемые для массивов, в этом случае не подходят и поэтому они будут отключены и никакой выгоды не принесут.

Варианты неправильного применения массива:

* Добавление нечислового свойства, например, arr.test = 5.
* Создание пустого пространства, 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, следует использовать их в таких случаях. Если нужны произвольные ключи, то лучше подойдёт обычный объект {}.

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

Методы push/pop выполняются быстро, а методы shift/unshift – медленно, т.е. у работать с концом массива быстрее, чем с его началом. Давайте посмотрим, что происходит во время выполнения:

fruits.shift(); // удаляем 1-ый элемент с начала

Просто взять и удалить элемент с номером 0 недостаточно. Нужно также заново пронумеровать остальные элементы. 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, нужно сначала сдвинуть существующие элементы вправо, увеличивая их индексы.

В процессе работы методам push/pop не нужно ничего перемещать. Чтобы удалить элемент в конце массива, метод pop очищает индекс и уменьшает значение length. Действия при операции pop:

fruits.pop(); // удаляем 1 элемент с конца

Метод pop не требует перемещения, потому что остальные элементы остаются на тех же индексах. 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:

let arr = ["Яблоко", "Апельсин", "Груша"];

for (let i = 0; i < arr.length; i++) {

alert( arr[i] );

}

Но для массивов возможен и другой вариант цикла – for..of:

let fruits = ["Яблоко", "Апельсин", "Слива"];

// проходит по значениям

for (let fruit of fruits) {

alert( fruit );

}

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

Технически, так как массив является объектом, можно использовать и цикл for..in:

let arr = ["Яблоко", "Апельсин", "Груша"];

for (let key in arr) {

alert( arr[key] ); // Яблоко, Апельсин, Груша

}

Но так делать не рекомендуется. Существуют скрытые недостатки этого способа:

1. Цикл for..in выполняет перебор всех свойств объекта, а не только цифровых.

В браузере и других программных средах также существуют так называемые «псевдомассивы» – объекты, которые выглядят, как массив. То есть, у них есть свойство length и индексы, но также они могут иметь дополнительные нечисловые свойства и методы, которые обычно не нужны. Тем не менее, цикл for..in выведет и их. Поэтому, если приходится иметь дело с объектами, похожими на массив, такие «лишние» свойства могут стать проблемой.

1. Цикл for..in оптимизирован под произвольные объекты, не массивы, и поэтому в 10-100 раз медленнее. Увеличение скорости выполнения может иметь значение только при возникновении узких мест.

Поэтому не следует использовать цикл for..in для массивов.

**[Свойство length](https://learn.javascript.ru/array" \l "nemnogo-o-length)**

Свойство length автоматически обновляется при изменении массива. Если быть точными, это не количество элементов массива, а наибольший цифровой индекс плюс один. Например, единственный элемент, имеющий большой индекс, даёт большую длину:

let fruits = [];

fruits[123] = "Яблоко";

alert( fruits.length ); // 124

Свойство length можно перезаписать. Если вручную увеличить его, то ничего особенного не произойдет. Но если уменьшить – массив станет короче. Этот процесс необратим:

let arr = [1, 2, 3, 4, 5];

arr.length = 2; // укорачиваем до 2 элементов

alert( arr ); // [1, 2]

arr.length = 5; // возвращаем length как было

alert( arr[3] ); // undefined: значения не восстановились

Таким образом, самый простой способ очистить массив – это arr.length = 0;.

**[new Array()](https://learn.javascript.ru/array" \l "new-array)**

Существует ещё один вариант синтаксиса для создания массива:

let arr = new Array("Яблоко", "Груша", "и тд");

Он редко применяется, так как квадратные скобки [] короче. Кроме того, у него есть одна особенность: если new Array вызывается с одним аргументом, который представляет собой число, он создаёт массив без элементов, но с заданной длиной:

let arr = new Array(2); // создаем массив [2]

alert( arr[0] ); // undefined! нет элементов.

alert( arr.length ); // length 2

Как видно из кода, представленного выше, в new Array(number) все элементы равны undefined. Чтобы избежать появления таких неожиданных ситуаций, обычно используются квадратные скобки, если конечно нет какой-то причины для использования именно Array.

**[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:

let matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

alert( matrix[1][1] ); // 5, центральный элемент

**Метод [toString](https://learn.javascript.ru/array" \l "tostring)**

Массивы по-своему реализуют метод toString, который возвращает список элементов, разделённых запятыми. Например:

let arr = [1, 2, 3];

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

alert( String(arr) === '1,2,3' ); // true

alert( [] + 1 ); // "1"

alert( [1] + 1 ); // "11"

alert( [1,2] + 1 ); // "1,21"

Массивы не имеют ни Symbol.toPrimitive, ни функционирующего valueOf, они реализуют только преобразование toString, таким образом здесь [] становится пустой строкой, [1] становится "1", а [1,2] становится "1,2". Когда бинарный оператор плюс "+" добавляет что-либо к строке, он тоже преобразует это в строку, таким образом:

alert( "" + 1 ); // "1"

alert( "1" + 1 ); // "11"

alert( "1,2" + 1 ); // "1,21"

1. **Методы массивов.**

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

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

* 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.

Есть и другие.

**Метод [splice](https://learn.javascript.ru/array-methods" \l "splice)**

Метод [arr.splice(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) – это универсальный метод для работы с массивами, который позволяет добавлять, удалять и заменять элементы.

Его синтаксис:

arr.splice(index[, deleteCount, elem1, ..., elemN])

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

Рассмотрим пример удаления элементов:

let arr = ["Я", "изучаю", "JavaScript"];

arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент

alert( arr ); // осталось ["Я", "JavaScript"]

Начиная с позиции 1, метод удалил 1 элемент.

Удалим 3 элемента и заменим их двумя другими:

let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];

arr.splice(0, 3, "Давай", "танцевать");

alert( arr ) // теперь ["Давай", "танцевать", "прямо", "сейчас"]

Здесь видно, что splice возвращает массив из удалённых элементов:

let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];

// удалить 2 первых элемента

let removed = arr.splice(0, 2);

alert( removed ); // "Я", "изучаю" – массив из удалённых элементов

Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0:

let arr = ["Я", "изучаю", "JavaScript"];

// с позиции 2 удалить 0 элементов и вставить "сложный", "язык"

arr.splice(2, 0, "сложный", "язык");

alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"

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

let arr = [1, 2, 5];

// начиная с индекса -1 (перед последним элементом) удалить 0 элементов,

// затем вставить числа 3 и 4

arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5

**Метод [slice](https://learn.javascript.ru/array-methods" \l "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:

arr.slice(start, end)

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. Метод похож на строковый метод str.slice, но вместо подстрок возвращает подмассивы. Например:

let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s

alert( arr.slice(-2) ); // s,t

**Метод  [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:

arr.concat(arg1, arg2...)

Он принимает любое количество аргументов, которые могут быть как массивами, так и простыми значениями. В результате будет сформирован новый массив, включающий в себя элементы из *arr*, а также *arg1*, *arg2* и так далее.

Если аргумент *argN* – массив, то все его элементы копируются, иначе скопируется сам аргумент. Например:

let arr = [1, 2];

alert( arr.concat([3, 4])); // 1,2,3,4

alert( arr.concat([3, 4], [5, 6])); // 1,2,3,4,5,6

alert( arr.concat([3, 4], 5, 6)); // 1,2,3,4,5,6

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

let arr = [1, 2];

let arrayLike = {

0: "что-то",

length: 1

};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]

//[1, 2, arrayLike]

Но если объект имеет специальное свойство Symbol.isConcatSpreadable, то он обрабатывается concat как массив: вместо него добавляются его элементы:

let arr = [1, 2];

let arrayLike = {

0: "что-то",

1: "ещё",

[Symbol.isConcatSpreadable]: true,

length: 2

};

alert( arr.concat(arrayLike) ); // 1,2,что-то,ещё

**[Перебор массива: метод 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:

arr.forEach(function(item, index, array) {

// вычисления с item

});

Например, этот код выведет на экран каждый элемент массива:

["Bilbo", "Gandalf", "Nazgul"].forEach(alert);

А этот выведет значение элемента и его позицию в массиве:

["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {

alert(`${item} имеет позицию ${index} в ${array}`);

});

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

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

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:

let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1

alert( arr.indexOf(false) ); // 2

alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true

Обратите внимание, что методы используют строгое сравнение ===. Таким образом, если осуществляется поиск false, то он находит именно false, а не ноль.

Если надо проверить наличие элемента и нет необходимости знать его точный индекс, тогда предпочтительным является arr.includes.

Метод [arr.find](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/find) позволяет найти элемент с определённым условием. Его синтаксис таков:

let result = arr.find(function(item, index, array) {

// если true - возвращается текущий элемент и перебор прерывается

// если все итерации оказались ложными возвращается undefined

});

Функция вызывается по очереди для каждого элемента массива:

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

Если функция вернёт true, поиск прерывается и возвращается item. Если ничего не найдено, возвращается undefined. Например, имеется массив пользователей, каждый из которых имеет поля id и name. Найдем того, кто с id == 1:

let users = [

{id: 1, name: "Вася"},

{id: 2, name: "Петя"},

{id: 3, name: "Маша"}

];

let user = users.find(item => item.id == 1);

alert(user.name); // Вася

Метод [arr.findIndex](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex) – в отличие от arr.find возвращает индекс, на котором был найден элемент, а не сам элемент, и -1, если ничего не найдено.

Метод find ищет один (первый попавшийся) элемент, на котором функция-колбэк вернёт true. Если найденных элементов может быть много, то стоит использовать 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:

let results = arr.filter(function(item, index, array) {

// если true - элемент добавляется к результату и перебор продолжается

// возвращается пустой массив в случае, если ничего не найдено

});

Например:

let users = [

{id: 1, name: "Вася"},

{id: 2, name: "Петя"},

{id: 3, name: "Маша"}

];

// возвращает массив, состоящий из двух первых пользователей

let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2

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

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. Синтаксис:

let result = arr.map(function(item, index, array) {

// возвращается новое значение вместо элемента

});

Например, в коде ниже осуществляется преобразование каждого элемента в его длину:

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);

alert(lengths); // 5,7,6

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. Например:

let arr = [ 1, 2, 15 ];

arr.sort();

alert( arr ); // 1, 15, 2

Порядок стал 1, 15, 2 так как элементы преобразуются в строки при сравнении и по умолчанию сортируются как строки. Для сортировки строк применяется лексикографический порядок где "2" > "15". Чтобы задать свой порядок сортировки, нужно предоставить функцию в качестве аргумента arr.sort(). Функция может возвращать следующие значения:

function compare(a, b) {

if (a > b) return 1; // если первое значение больше второго

if (a == b) return 0; // если равны

if (a < b) return -1; // если первое значение меньше второго

}

Например, для сортировки чисел:

function compareNumeric(a, b) {

if (a > b) return 1;

if (a == b) return 0;

if (a < b) return -1;

}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr); // 1, 2, 15

Функция сравнения может возвращать любое положительное число, если первый сравниваемый элемент больше второго и отрицательное – если меньше. Это позволяет писать более короткие функции:

let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

alert(arr); // 1, 2, 15

Можно использовать стрелочные функции, чтобы сортировка выглядела более компактной:

arr.sort( (a, b) => a - b );

Будет работать точно так же, как и более длинная версия выше.

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

let arr = [1, 2, 3, 4, 5];

arr.reverse();

alert( arr ); // 5,4,3,2,1

Он также возвращает массив arr с изменённым порядком элементов.

Метод [str.split(delim)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/split) разбивает строку на массив по заданному разделителю delim. В примере ниже таким разделителем является строка из запятой и пробела.

let names = 'Вася, Петя, Маша';

let arr = names.split(', ');

for (let name of arr) {

alert( `Сообщение получат: ${name}.` );

}

У метода split есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен. На практике это редко используется:

let arr = 'Вася, Петя, Маша, Саша'.split(', ', 2);

alert(arr); // Вася, Петя

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

let str = "тест";

alert( str.split('') ); // т,е,с,т

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

let arr = ['Вася', 'Петя', 'Маша'];

let str = arr.join(';'); // объединить массив в строку через ;

alert( str ); // Вася;Петя;Маша

Если надо перебрать массив – можно использовать forEach, for или for..of. Если надо перебрать массив и возвратить данные для каждого элемента – стоит используем map. Методы [arr.reduce](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce) и [arr.reduceRight](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) похожи на методы выше, но они немного сложнее. Они используются для вычисления какого-нибудь единого значения на основе всего массива. Синтаксис:

let value = arr.reduce(function(previousValue, item, index, array) {

// ...

}, [initial]);

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов. Аргументы:

* previousValue – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial),
* item – очередной элемент массива,
* index – индекс элемента,
* array – массив.

При вызове функции результат её вызова на предыдущем элементе массива передаётся как первый аргумент. Звучит сложновато, но всё становится проще, если думать о первом аргументе как «аккумулирующем» результат предыдущих вызовов функции. По окончании он становится результатом reduce. Этот метод проще всего понять на примере. Получим сумму всех элементов массива:

let arr = [1, 2, 3, 4, 5];

let result = arr.reduce((sum, current) => sum + current, 0);

alert(result); // 15

Здесь использован наиболее распространённый вариант reduce, который использует только 2 аргумента. Рассмотрим, как он работает:

1. При первом запуске, sum равен initial (последний аргумент reduce), то есть 0, а current – первый элемент массива, равнй 1. Таким образом, результат функции равен 1.
2. При втором запуске sum = 1, и к нему добавляется второй элемент массива (2).
3. На 3-м запуске sum = 3, к которому добавляется следующий элемент и так далее.

Ниже представлена таблица, где каждая строка – вызов функции на очередном элементе массива:

|  | **sum** | **current** | **result** |
| --- | --- | --- | --- |
| первый вызов | 0 | 1 | 1 |
| второй вызов | 1 | 2 | 3 |
| третий вызов | 3 | 3 | 6 |
| четвёртый вызов | 6 | 4 | 10 |
| пятый вызов | 10 | 5 | 15 |

Здесь отчётливо видно, как результат предыдущего вызова передаётся в первый аргумент следующего. Также можно опустить начальное значение:

let arr = [1, 2, 3, 4, 5];

let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15

Результат такой же потому, что при отсутствии initial в качестве первого значения берётся первый элемент массива, а перебор стартует со второго. Таблица вычислений будет такая же за вычетом первой строки. Но такое использование требует крайней осторожности. Если массив пуст, то вызов reduce без начального значения выдаст ошибку, поэтому рекомендуется всегда его указывать:

let arr = [];

// Error: Reduce of empty array with no initial value

arr.reduce((sum, current) => sum + current);

Метод [arr.reduceRight](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) работает аналогично, но проходит по массиву справа налево.

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

alert(typeof {}); // object

alert(typeof []); // object

Метод [Array.isArray(value)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray) возвращает true, если value массив, и false, если нет:

alert(Array.isArray({})); // false

alert(Array.isArray([])); // true

Почти все методы массива, которые вызывают функции – такие как find, filter, map, за исключением метода sort, принимают необязательный параметр thisArg. Этот параметр очень редко используется. Полный синтаксис этих методов:

arr.find(func, thisArg);

arr.filter(func, thisArg);

arr.map(func, thisArg);

Значение параметра thisArg становится this для func. Например, если необходимо использовать метод объекта как фильтр, то thisArg с этим поможет:

let user = {

age: 18,

younger(otherUser) {

return otherUser.age < this.age;

}

};

let users = [

{age: 12},

{age: 16},

{age: 32}

];

// найти число пользователей моложе, чем заданный

let youngerUsers = users.filter(user.younger, user);

alert(youngerUsers.length); // 2

В вызове выше используется user.younger как фильтр, а user – в качестве контекста для него. Если бы не предоставлялся контекст, users.filter(user.younger) вызвал бы user.younger как ни к чему не привязанную функцию с this=undefined. Это привело бы кошибке.

1. **Объекты.**

Как известно, в JavaScript существует семь типов данных. Шесть из них называются примитивными, так как содержат только одно значение. Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка. Поэтому стоит их досконально изучить.

Объект может быть создан с помощью фигурных скобок {…} с необязательным списком свойств. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

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

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

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

Обычно используют вариант с фигурными скобками {...}. Такое объявление называют литералом объекта или литеральной нотацией.

**[Литералы и свойства](https://learn.javascript.ru/object" \l "literaly-i-svoystva)**

При использовании литерального синтаксиса {...} сразу можно поместить в объект несколько свойств в виде пар «ключ: значение»:

let user = { // объект

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

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

};

Свойства объекта также иногда называют полями объекта. У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

1. Первое свойство с именем "name" и значением "John".
2. Второе свойство с именем "age" и значением 30.

Для обращения к свойствам используется запись «через точку»:

// получаем свойства объекта:

alert( user.name ); // John

alert( user.age ); // 30

Значение может быть любого типа. Добавим свойство с логическим значением:

user.isAdmin = true;

Для удаления свойства можно использовать оператор delete:

delete user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки. Последнее свойство объекта может заканчиваться висячей запятой:

let user = {

name: "John",

age: 30,

"likes birds": true,

};

**[Квадратные скобки](https://learn.javascript.ru/object" \l "kvadratnye-skobki)**

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

// приведет к синтаксической ошибкк

user.likes birds = true

Так происходит, потому что точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме $ и \_. Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

let user = {};

// присваивание значения свойству

user["likes birds"] = true;

// получение значения свойства

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

// удаление свойства

delete user["likes birds"];

Квадратные скобки также позволяют обратиться к свойству, имя которого может быть результатом выражения. Например, имя свойства может храниться в переменной:

let key = "likes birds";

user[key] = true;

Здесь переменная key может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого можно использовать её для доступа к свойству. Пример:

let user = {

name: "John",

age: 30

};

let key = prompt("Что вы хотите узнать о пользователе?", "name");

alert( user[key] ); // John (если ввели "name")

Запись «через точку» такого не позволяет:

let user = {

name: "John",

age: 30

};

let key = "name";

user.key // undefined

**[Вычисляемые свойства](https://learn.javascript.ru/object" \l "vychislyaemye-svoystva)**

Можно использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства. Пример:

let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {

[fruit]: 5, // имя свойства будет взято из переменной fruit

};

alert( bag.apple ); // 5, если fruit="apple"

Смысл вычисляемого свойства прост: запись [fruit] означает, что имя свойства необходимо взять из переменной fruit. И если посетитель введёт слово "apple", то в объекте bag теперь будет храниться свойство {apple: 5}.

Можно использовать и более сложные выражения в квадратных скобках:

let fruit = 'apple';

let bag = {

[fruit + 'Computers']: 5 // bag.appleComputers = 5

};

Таким образом, когда имена свойств известны и просты, используется запись через точку. Если же нужно что-то более сложное, то следует использовать квадратные скобки. Зарезервированные слова разрешено использовать как имена свойств:

let obj = {

for: 1,

let: 2,

return: 3

};

alert( obj.for + obj.let + obj.return ); // 6

**[Свойство из переменной](https://learn.javascript.ru/object" \l "svoystvo-iz-peremennoy)**

В реальном коде часто необходимо использовать существующие переменные как значения для свойств с тем же именем. Например:

function makeUser(name, age) {

return {

name: name,

age: age

// ...другие свойства

};

}

let user = makeUser("John", 30);

alert(user.name); // John

В примере выше название свойств name и age совпадают с названиями переменных, которые указываются в качестве значений этих свойств. Такой подход настолько распространен, что существуют специальные короткие свойства для упрощения этой записи. Вместо name:name можно написать просто name. Можно использовать как обычные свойства, так и короткие в одном и том же объекте:

function makeUser(name, age) {

return {

name,

age

height: 170// ...

};

}

**[Проверка существования свойства](https://learn.javascript.ru/object" \l "proverka-suschestvovaniya-svoystva)**

Особенность объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет. При обращении к свойству, которого нет, возвращается undefined. Это позволяет просто проверить существование свойства – сравнением его с undefined:

let user = {};

alert( user.noSuchProperty === undefined ); // true означает "свойства нет"

Также существует специальный оператор "in" для проверки существования свойства в объекте. Синтаксис оператора:

"key" in object

Пример:

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

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

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

Обратите внимание, что слева от оператора in должно быть имя свойства. Обычно это строка в кавычках. Если кавычки опускаются, это значит, что используется переменная, в которой находится имя свойства. Например:

let user = { age: 30 };

let key = "age";

alert( key in user ); // true

Обычно строгого сравнения "=== undefined" достаточно для проверки наличия свойства. Но есть особый случай, когда оно не подходит, и нужно использовать "in". Это когда свойство существует, но содержит значение undefined:

let obj = {

test: undefined

};

alert( obj.test ); // выведет undefined

alert( "test" in obj ); // true, свойство существует

В примере выше свойство obj.test технически существует в объекте. Оператор in сработал правильно. Подобные ситуации случаются очень редко, так как undefined обычно явно не присваивается. Для «неизвестных» или «пустых» свойств используется значение null.

**[Цикл «for…in»](https://learn.javascript.ru/object" \l "tsikl-for-in)**

Для перебора всех свойств объекта используется цикл for..in. Этот цикл отличается от изученного ранее цикла for(;;). Синтаксис:

for (key in object) {

// тело цикла выполняется для каждого свойства объекта

}

Например, выведем все свойства объекта user:

let user = {

name: "John",

age: 30,

isAdmin: true

};

for (let key in user) {

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

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

}

Обратите внимание, что все конструкции «for» позволяют объявлять переменную внутри цикла, как, например, let key здесь. Кроме того, можно использовать другое имя переменной. Например, часто используется вариант "for (let prop in obj)".

**[Упорядочение свойств объекта](https://learn.javascript.ru/object" \l "uporyadochenie-svoystv-obekta)**

Свойства объекта упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания. В качестве примера рассмотрим объект с телефонными кодами:

let codes = {

"49": "Германия",

"41": "Швейцария",

"44": "Великобритания",

// ..,

"1": "США"

};

for (let code in codes) {

alert(code); // 1, 41, 44, 49

}

Телефонные коды идут в порядке возрастания, потому что они являются целыми числами: 1, 41, 44, 49.

Термин «целочисленное свойство» означает строку, которая может быть преобразована в целое число и обратно без изменений. То есть, "49" – это целочисленное имя свойства, потому что если его преобразовать в целое число, а затем обратно в строку, то оно не изменится. А вот свойства "+49" или "1.2" таковыми не являются. Если ключи не целочисленные, то они перебираются в порядке создания, например:

let user = {

name: "John",

surname: "Smith"

};

user.age = 25;

for (let prop in user) {

alert( prop ); // name, surname, age

}

Таким образом, чтобы телефонные коды выводились, в том порядке в котром записаны, надо сделать коды не целочисленными свойствами. Для этого надо добавить знак "+" перед каждым кодом. Пример:

let codes = {

"+49": "Германия",

"+41": "Швейцария",

"+44": "Великобритания",

// ..,

"+1": "США"

};

for (let code in codes) {

alert( +code ); // 49, 41, 44, 1

}

**[Копирование по ссылке](https://learn.javascript.ru/object" \l "kopirovanie-po-ssylke)**

Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке». Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению». Например:

let message = "Hello!";

let phrase = message;

В результате имеются две независимые переменные, каждая из которых хранит строку "Hello!". Объекты ведут себя иначе. Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него. Например:

let user = {

name: "John"

};

Сам объект хранится где-то в памяти. А в переменной user лежит «ссылка» на эту область памяти. Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется:

let user = { name: "John" };

let admin = user; // копируется ссылка

В результате есть две переменные, каждая из которых содержит ссылку на один и тот же объект. Можно использовать любую из переменных для доступа к объекту и изменения его содержимого:

let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // изменено по ссылке из переменной "admin"

alert(user.name); // 'Pete', изменения видны по ссылке из переменной "user"

**[Сравнение объектов](https://learn.javascript.ru/object" \l "sravnenie-obektov)**

Операторы равенства == и строгого равенства === для объектов работают одинаково. Два объекта равны только в том случае, если это один и тот же объект. Например, две переменные ссылаются на один и тот же объект, они равны:

let a = {};

let b = a; // копирование по ссылке

alert( a == b ); // true

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

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

let a = {};

let b = {}; // два независимых объекта

alert( a == b ); // false

Для сравнений типа obj1 > obj2 или для сравнения с примитивом obj == 5 объекты преобразуются в примитивы. Такое сравнение используется очень редко и не рекомендуется.

**[Объекты-константы](https://learn.javascript.ru/object" \l "obekty-konstanty)**

Объект, объявленный через const, может быть изменен:

const user = {

name: "John"

};

user.age = 25; // (\*)

alert(user.age); // 25

Объявление const защищает от изменений только само значение user. В примере значение user – это ссылка на объект, и это значение не меняется. В строке (\*) вносятся изменения внутри объекта, а значение user не изменяется. Если же попытаться присвоить user другое значение, то const выдаст ошибку:

const user = {

name: "John"

};

// Ошибка (нельзя переопределять константу user)

user = {

name: "Pete"

};

Cделать константами свойства объекта тоже возможно с помощью флагов и дескрипторов свойств, кторые будут рассматриваться позже.

**[Клонирование и объединение объектов, Object.assign](https://learn.javascript.ru/object" \l "klonirovanie-i-obedinenie-obektov-object-assign)**

При копировании переменной объекта создаётся ещё одна ссылка на тот же самый объект. Если надо создать независимую копию (клон), то необходимо создать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их. Например так:

let user = {

name: "John",

age: 30

};

let clone = {};

for (let key in user) {

clone[key] = user[key];

}

clone.name = "Pete";

alert( user.name );

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

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

* Аргументы dest, и src1, ..., srcN (может быть столько, сколько нужно) являются объектами.
* Метод копирует свойства всех объектов src1, ..., srcN в объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект. После копирования метод возвращает объект dest.

Например, объединим несколько объектов в один:

let user = { name: "John" };

let permissions1 = { canView: true };

let permissions2 = { canEdit: true };

// user = { name: "John", canView: true, canEdit: true }

Object.assign(user, permissions1, permissions2);

Если принимающий объект (user) уже имеет свойство с таким именем, оно будет перезаписано:

let user = { name: "John" };

// user = { name: "Pete", isAdmin: true }

Object.assign(user, { name: "Pete", isAdmin: true });

Также можно использовать Object.assign для простого клонирования:

let user = {

name: "John",

age: 30

};

let clone = Object.assign({}, user);

Все свойства объекта user будут скопированы в пустой объект, и ссылка на этот объект будет в переменной clone. Такое клонирование работает так же, как и через цикл, но короче.

Если свойство не примитивно, а явлется ссылкой на другой объект, то при клонировании недостаточно просто скопировать clone.sizes = user.sizes, поскольку user.sizes – это объект, он будет скопирован по ссылке. А значит объекты clone и user в своих свойствах sizes будут ссылаться на один и тот же объект:

let user = {

name: "John",

sizes: {

height: 182,

width: 50

}

};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes );

user.sizes.width++;

alert(clone.sizes.width); // 51

Чтобы исправить это, необходимо в цикле клонирования делать проверку, не является ли значение user[key] объектом, и, если это так, – копировать и его структуру тоже. Это называется «глубокое клонирование». Существует стандартный алгоритм глубокого клонирования, [Structured cloning algorithm](http://w3c.github.io/html/infrastructure.html" \l "safe-passing-of-structured-data). Он решает описанную выше задачу, а также более сложные задачи.

1. **Set, Map, WeakSet и WeakMap**

Ранее рассматривались следующие сложные структуры данных: объекты – для хранения именованных коллекций, массивы – для хранения упорядоченных коллекций. Существуют и другие более сложные коллекции.

**[Map](https://learn.javascript.ru/map-set" \l "map)**

[Map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/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.

Например:

let map = new Map();

map.set("1", "str1"); // строка в качестве ключа

map.set(1, "num1"); // цифра как ключ

map.set(true, "bool1"); // булево значение как ключ

alert(map.get(1)); // "num1"

alert(map.get("1")); // "str1"

alert(map.size); // 3

Как мы видим, в отличие от объектов, ключи не были приведены к строкам. Можно использовать любые типы данных для ключей, даже объекты. Например:

let john = { name: "John" };

let visitsCountMap = new Map();

visitsCountMap.set(john, 123);

alert(visitsCountMap.get(john)); // 123

Использование объектов в качестве ключей – это одна из известных и часто применяемых возможностей объекта Map. При строковых ключах обычный объект Object может подойти, но для ключей-объектов – уже нет.

Чтобы сравнивать ключи, объект Map использует алгоритм [SameValueZero](https://tc39.github.io/ecma262/" \l "sec-samevaluezero). Это почти такое же сравнение, что и ===, с той лишь разницей, что NaN считается равным NaN. Так что NaN также может использоваться в качестве ключа. Этот алгоритм не может быть заменён или модифицирован.

Каждый вызов map.set возвращает объект map, так что можно объединить вызовы в цепочку:

map.set("1", "str1")

.set(1, "num1")

.set(true, "bool1");

Для перебора коллекциии Map есть 3 метода:

* 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.

Например:

let recipeMap = new Map([

["огурец", 500],

["помидор", 350],

["лук", 50]

]);

// перебор по ключам (овощи)

for (let vegetable of recipeMap.keys()) {

alert(vegetable); // огурец, помидор, лук

}

// перебор по значениям (числа)

for (let amount of recipeMap.values()) {

alert(amount); // 500, 350, 50

}

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

for (let entry of recipeMap) {

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

}

В отличие от обычных объектов Object, в Map перебор происходит в том же порядке, в каком происходило добавление элементов. Кроме этого, Map имеет встроенный метод forEach, схожий со встроенным методом массивов Array:

// выполняем функцию для каждой пары (ключ, значение)

recipeMap.forEach((value, key, map) => {

alert(`${key}: ${value}`); // огурец: 500 и так далее

});

**[Создание Map из Object](https://learn.javascript.ru/map-set" \l "object-entries-map-iz-object)**

Если есть обычный объект, и необходимо создать Map из него, то поможет встроенный метод [Object.entries(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/entries), который получает объект и возвращает массив пар ключ-значение для него, как раз в этом формате. Поэтому можно создать Map из обычного объекта следующим образом:

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.

**[Создание Object из Map](https://learn.javascript.ru/map-set" \l "object-fromentries-object-iz-map)**

Метод Object.fromEntries делает противоположное: получив массив пар вида [ключ, значение], он создаёт из них объект:

let prices = Object.fromEntries([

['banana', 1],

['orange', 2],

['meat', 4]

]);

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

alert(prices.orange); // 2

Можно использовать Object.fromEntries, чтобы получить обычный объект из Map. Например, есть данные в Map, но их нужно передать в сторонний код, который ожидает обычный объект. Вот как это сделать:

let map = new Map();

map.set('banana', 1);

map.set('orange', 2);

map.set('meat', 4);

let obj = Object.fromEntries(map);

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

alert(obj.orange); // 2

Вызов map.entries() возвращает массив пар ключ/значение, как раз в нужном формате для Object.fromEntries.

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

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

* 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.

Суть в том, что при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз. Например, список посетителей:

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)

}

Альтернативой множеству Set может выступать массив для хранения гостей и дополнительный код для проверки уже имеющегося элемента с помощью [arr.find](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/find). Но в этом случае будет хуже производительность, потому что arr.find проходит весь массив для проверки наличия элемента. Множество Set лучше оптимизировано для добавлений, оно автоматически проверяет на уникальность.

Можно перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach:

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

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

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

alert(value);

});

Функция в forEach у Set имеет 3 аргумента: значение value, потом снова то же самое значение valueAgain, и целевой объект. Значение появляется в списке аргументов дважды. Это сделано для совместимости с объектом Map, в котором колбэк forEach имеет 3 аргумента. Выглядит странно, но в некоторых случаях может помочь легко заменить Map на Set и наоборот. Set имеет те же встроенные методы, что и Map:

* set.keys() – возвращает перебираемый объект для значений,
* set.values() – то же самое, что и set.keys(), присутствует для обратной совместимости с Map,
* set.entries() – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

**WeakMap и WeakSet**

Как извстно, движок JavaScript хранит значения в памяти до тех пор, пока они достижимы (то есть, эти значения могут быть использованы). Например:

let john = { name: "John" };

// объект доступен, переменная john -- это ссылка на него

// перепишем ссылку

john = null;

// объект будет удалён из памяти

Обычно свойства объекта, элементы массива или другой структуры данных считаются достижимыми и сохраняются в памяти до тех пор, пока эта структура данных содержится в памяти. Например, если поместить объект в массив, то до тех пор, пока массив существует, объект также будет существовать в памяти, несмотря на то, что других ссылок на него нет. Например:

let john = { name: "John" };

let array = [ john ];

john = null; // перезаписываем ссылку на объект

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

Аналогично, если используется 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. Например:

let john = { name: "John" };

let map = new Map();

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

john = null; // перезаписываем ссылку на объект

// объект john сохранён внутри объекта `Map`

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");

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

Теперь, если использовать объект в качестве ключа и если больше нет ссылок на этот объект, то он будет удалён из памяти (и из объекта WeakMap) автоматически.

let john = { name: "John" };

let weakMap = new WeakMap();

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

john = null;

// объект john удалён из памяти

Теперь john существует только как ключ в WeakMap и может быть удалён оттуда автоматически. WeakMap не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё. В WeakMap присутствуют только следующие методы:

* weakMap.get(key)
* weakMap.set(key, value)
* weakMap.delete(key)
* weakMap.has(key)

Такие ограничения связаны с особенностью технической реализации. Если объект станет недостижим (как объект john в примере выше), то он будет автоматически удалён сборщиком мусора. Но нет информации, в какой момент произойдет эта очистка. Решение о том, когда делать сборку мусора, принимает движок JavaScript. Он может посчитать необходимым как удалить объект прямо сейчас, так и отложить эту операцию, чтобы удалить большее количество объектов за раз позже. Так что технически количество элементов в коллекции WeakMap неизвестно. Движок может произвести очистку сразу или потом, или сделать это частично. По этой причине методы для доступа ко всем сразу ключам/значениям недоступны.

В основном, WeakMap используется в качестве дополнительного хранилища данных или кеширования, 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. Для хранения результатов можно использовать Map:

// cache.js

let cache = new Map();

// вычисляем и запоминаем результат

function process(obj) {

if (!cache.has(obj)) {

let result = /\* какие-то вычисления \*/ obj;

cache.set(obj, result);

}

return cache.get(obj);

}

// main.js

let obj = {/\* какой-то объект \*/};

let result1 = process(obj);

let result2 = process(obj); // ранее вычисленный результат взят из кеша

obj = null;

alert(cache.size); // 1, объект всё ещё в кеше и занимает память

Многократные вызовы process(obj) с тем же самым объектом в качестве аргумента ведут к тому, что результат вычисляется только в первый раз, а затем последующие вызовы берут его из кеша. Недостатком является то, что необходимо вручную очищать cache от ставших ненужными объектов. Но если использовать WeakMap вместо Map, то эта проблема исчезнет: закешированные результаты будут автоматически удалены из памяти сборщиком мусора.

// cache.js

let cache = new WeakMap();

function process(obj) {

if (!cache.has(obj)) {

let result = /\* вычисляем результат для объекта \*/ obj;

cache.set(obj, result);

}

return cache.get(obj);

}

// main.js

let obj = {/\* какой-то объект \*/};

let result1 = process(obj);

let result2 = process(obj);

obj = null;

// Нет возможности получить cache.size, так как это WeakMap,

// но он равен 0 или скоро будет равен 0

// Когда сборщик мусора удаляет obj, связанные с ним данные из кеша тоже удаляются

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

Коллекция WeakSet ведёт себя похоже:

* Она аналогична Set, но можно 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.

WeakSet тоже служит в качестве дополнительного хранилища. Но не для произвольных данных, а скорее для значений типа «да/нет». Присутствие во множестве WeakSet может что-то сказать об объекте. Например, можно добавлять пользователей в WeakSet для учёта тех, кто посещал наш сайт:

let visitedSet = new WeakSet();

let john = { name: "John" };

let pete = { name: "Pete" };

let mary = { name: "Mary" };

visitedSet.add(john); // John

visitedSet.add(pete); // Pete

visitedSet.add(john); // John

// visitedSet сейчас содержит двух пользователей

alert(visitedSet.has(john)); // true

alert(visitedSet.has(mary)); // false

john = null;

// структура данных visitedSet будет очищена автоматически

Наиболее значительным ограничением WeakMap и WeakSet является то, что их нельзя перебрать или взять всё содержимое. Это может доставлять неудобства, но не мешает WeakMap/WeakSet выполнять их главную задачу – быть дополнительным хранилищем данных для объектов, управляемых из каких-то других мест в коде.

1. **Деструктурирующее присваивание**

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array. Объекты позволяют создавать одну сущность, которая хранит элементы данных по ключам, а массивы – хранить упорядоченные коллекции данных.

Но когда они передаются в функцию, то ей может понадобится не объект/массив целиком, а элементы по отдельности.

*Деструктурирующее присваивание* – это специальный синтаксис, который позволяет преобразовать массивы или объекты в кучу переменных, так как иногда они более удобны. Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию, и так далее.

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

Пример:

let arr = ["Steve", "Jobs"]

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

let [firstName, surname] = arr;

alert(firstName); // Steve

alert(surname); // Jobs

Теперь можно использовать переменные вместо элементов массива. Удобно использовать в сочетании со split или другими методами, возвращающими массив:

let [firstName, surname] = "Steve Jobs".split(' ');

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

Ненужные элементы массива также могут быть отброшены через запятую:

// второй элемент не нужен

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

alert( title ); // Consul

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

Можно использовать любой перебираемый объект, не только массивы:

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

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

Можно использовать что угодно с левой стороны. Например, можно присвоить свойству объекта:

let user = {};

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

alert(user.name); // Ilya

Ранее рассматривался метод [Object.entries(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/entries). Можем использовать его с деструктуризацией для цикличного перебора ключей и значений объекта:

let user = {

name: "John",

age: 30

};

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

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

}

Если надо не просто получить первые значения, но и собрать все остальные – мы можем добавить ещё один параметр, который получает остальные значения, используя троеточие "...":

let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius

alert(name2); // Caesar

alert(rest[0]); // Consul

alert(rest[1]); // of the Roman Republic

alert(rest.length); // 2

Переменная rest является массивом из оставшихся элементов. Вместо rest можно использовать любое другое название переменной, и она должна находится на последнем месте в деструктурирующем присваивании.

Если в массиве меньше значений, чем в присваивании, то ошибки не будет. Отсутствующие значения считаются неопределёнными:

let [firstName, surname] = [];

alert(firstName); // undefined

alert(surname); // undefined

Если необходимо указать значения по умолчанию, то можно использовать =:

let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name); // Julius (из массива)

alert(surname); // Anonymous (значение по умолчанию)

Значения по умолчанию могут быть гораздо более сложными выражениями или даже функциями. Они выполняются, только если значения отсутствуют. Например, в в примере ниже используется функция prompt для указания двух значений по умолчанию. Но она будет запущена только для отсутствующего значения:

let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name); // Julius (из массива)

alert(surname); // результат prompt

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

Деструктурирующее присваивание также работает с объектами. Имена переменных и ключи должны совпадать. Синтаксис:

let {var1, var2} = {var1:…, var2:…}

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

let options = {

title: "Menu",

width: 100,

height: 200

};

let {title, width, height} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

Свойства options.title, options.width и options.height присваиваются соответствующим переменным. Порядок не имеет значения. Вот так тоже работает:

// изменён порядок в let {...}

let {height, width, title} = { title: "Menu", height: 200, width: 100 }

Шаблон с левой стороны может быть более сложным и определять соответствие между свойствами и переменными.

Если нужно присвоить свойство объекта переменной с другим названием, например, свойство options.width присвоить переменной *w*, то можно использовать двоеточие:

let options = {

title: "Menu",

width: 100,

height: 200

};

// { sourceProperty: targetVariable }

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

// width -> w, height -> h, title -> title

alert(title); // Menu

alert(w); // 100

alert(h); // 200

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

Для потенциально отсутствующих свойств можно установить значения по умолчанию, используя "=":

let options = {

title: "Menu"

};

let {width = 100, height = 200, title} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

Как и в случае с массивами, значениями по умолчанию могут быть любые выражения или даже функции. Они выполнятся, если значения отсутствуют.

**[Остаток объекта «…»](https://learn.javascript.ru/destructuring-assignment" \l "ostatok-obekta)**

Если в объекте больше свойств, чем переменных, то можно использовать троеточие, так же как для массивов. В некоторых старых браузерах (IE) это не поддерживается, необходимо использовать полифилы. Например:

let options = {

title: "Menu",

height: 200,

width: 100

};

let {title, ...rest} = options;

// title="Menu", rest={height: 200, width: 100}

alert(rest.height); // 200

alert(rest.width); // 100

В примерах выше переменные были объявлены в присваивании: let {…} = {…}. Если использовать существующие переменные и не указывать let, то это не будет работать. Проблема в том, что JavaScript обрабатывает {...} в основном потоке кода (не внутри другого выражения) как блок кода. Такие блоки кода могут быть использованы для группировки операторов, например:

{

// блок кода

let message = "Hello";

// ...

alert( message );

}

Так что здесь JavaScript считает, что видит блок кода, отсюда и ошибка. На самом-то деле здесь деструктуризация. Чтобы показать JavaScript, что это не блок кода, можно заключить выражение в скобки (...):

let title, width, height;

({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu

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

Если объект или массив содержит другие вложенные объекты или массивы, то можно 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:

let options = {

size: {

width: 100,

height: 200

},

items: ["Cake", "Donut"],

extra: true

};

let {

size: {

width,

height

},

items: [item1, item2],

title = "Menu"

} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

alert(item1); // Cake

alert(item2); // Donut

Весь объект options, кроме свойства extra, присваивается в соответствующие переменные. В итоге есть width, height, item1, item2 и title со значением по умолчанию. Заметьте, что переменные для size и items отсутствуют, так как сразу использовали их содержимое.

**[Умные параметры функций](https://learn.javascript.ru/destructuring-assignment" \l "umnye-parametry-funktsiy)**

Есть ситуации, когда функция имеет много параметров, большинство из которых не обязательны. Это особенно верно для пользовательских интерфейсов. Представьте себе функцию, которая создаёт меню. Она может иметь ширину, высоту, заголовок, список элементов и так далее. Вот так – плохой способ писать подобные функции:

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

// ...

}

В реальной жизни проблема заключается в том, как запомнить порядок всех аргументов. Обычно IDE помогают в этом, особенно если код хорошо документирован. Но есть и другая проблема, которая заключается в том, как вызвать функцию, когда большинство параметров передавать не надо, и значения по умолчанию вполне подходят. Разве что вот так:

// undefined там, где подходят значения по умолчанию

showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])

Это выглядит плохо. И становится нечитаемым, когда имеется большое количество параметров. Решить эту проблему помогает деструктуризация. Можно передать параметры как объект, и функция немедленно деструктурирует его в переменные:

let options = {

title: "My menu",

items: ["Item1", "Item2"]

};

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

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

alert( items ); // Item1, Item2

}

showMenu(options);

Также можно использовать более сложное деструктурирование с вложенными объектами и двоеточием:

let options = {

title: "My menu",

items: ["Item1", "Item2"]

};

function showMenu({

title = "Untitled",

width: w = 100,

height: h = 200,

items: [item1, item2]

}) {

alert( `${title} ${w} ${h}` ); // My Menu 100 200

alert( item1 ); // Item1

alert( item2 ); // Item2

}

showMenu(options);

Полный синтаксис – такой же, как для деструктурирующего присваивания:

function({

incomingProperty: varName = defaultValue

...

})

Тогда для объекта с параметрами, будет создана переменная varName для свойства с именем incomingProperty, по умолчанию равная defaultValue. Обратите внимание, что такое деструктурирование подразумевает, что в showMenu() будет обязательно передан аргумент. Если нужны все значения по умолчанию, то следует определить пустой объект:

showMenu({}); // все значения по умолчанию

showMenu(); // ошибка

Можно исправить это, сделав {} значением по умолчанию для всего объекта параметров:

function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {

alert( `${title} ${width} ${height}` );

}

showMenu(); // Menu 100 200

В приведённом коде выше весь объект аргументов по умолчанию равен {}, поэтому всегда есть что-то, что можно деструктурировать.

**Третья тема**

1. **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. В некоторых браузерах (например, Edge) globalThis ещё не поддерживается, но легко реализуется с помощью полифила.

Прежде, решение было таким:

const getGlobal = function () {

if (typeof self !== undefined) {

return self;

}

if (typeof window !== undefined) {

return window;

}

if (typeof global !== undefined) {

return global;

}

throw new Error("unable to locate global object");

};

const globals = getGlobal();

// Сейчас есть `globalThis`

globalThis === window; // true

Рассмотрим подробнее глобальный объект window, так как наша среда – браузер. Ко всем свойствам глобального объекта можно обращаться напрямую:

alert("Привет");

// это то же самое, что и

window.alert("Привет");

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const), становятся свойствами глобального объекта:

var gVar = 5;

alert(window.gVar); // 5 (становится свойством глобального объекта)

Такое поведение поддерживается для совместимости. В современных проектах, использующих [JavaScript-модули](https://learn.javascript.ru/modules), такого не происходит.

Если объявить переменную при помощи let, то такого не произойдет:

let gLet = 5;

alert(window.gLet); // undefined (не становится свойством глобального объекта)

Если свойство настолько важное, что надо сделать его доступным для всей программы, то запишите его в глобальный объект напрямую:

window.currentUser = {

name: "John"

};

alert(currentUser.name); // John

alert(window.currentUser.name); // John

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.

Глобальный объект можно использовать, чтобы проверить поддержку современных возможностей языка. Например, проверить наличие встроенного объекта Promise (такая поддержка отсутствует в очень старых браузерах):

if (!window.Promise) {

alert("Ваш браузер очень старый!");

}

Если такой объект не поддерживается, то можно создать полифил: добавить функции, которые не поддерживаются окружением, но существуют в современном стандарте.

if (!window.Promise) {

window.Promise = ... // реализация современной возможности языка

}

1. **Замыкания, внутренняя работа функции.**

JavaScript – язык с сильным функционально-ориентированным уклоном. Функция может быть динамически создана, скопирована в другую переменную или передана как аргумент другой функции и позже вызвана из совершенно другого места. Функция может получить доступ к переменным из внешнего окружения, эта возможность используется очень часто.

Рассмотрим две ситуации:

1. Функция sayHi использует внешнюю переменную name. Какое значение будет использовать функция при выполнении?

let name = "John";

function sayHi() {

alert("Hi, " + name);

}

name = "Pete";

sayHi(); // что будет показано: "John" или "Pete"?

Такие ситуации распространены и в браузерной и в серверной разработке. Выполнение функции может быть запланировано позже, чем она была создана, например, после какого-нибудь пользовательского действия или сетевого запроса.

1. Функция makeWorker создаёт другую функцию и возвращает её. Новая функция может быть вызвана откуда-то ещё. Получит ли она доступ к внешним переменным из места своего создания или места выполнения или из обоих?

function makeWorker() {

let name = "Pete";

return function() {

alert(name);

};

}

let name = "John";

let work = makeWorker();

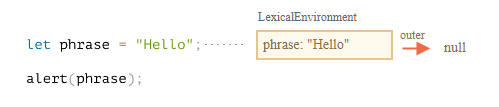
work(); // что будет показано? "Pete" (из места создания) или "John" (из места выполнения)

**[Лексическое окружение](https://learn.javascript.ru/closure" \l "leksicheskoe-okruzhenie)**

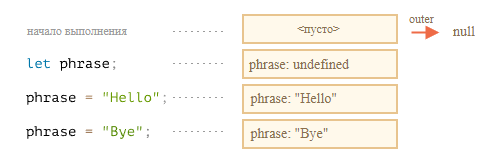
Чтобы понять, что происходит, давайте для начала, обсудим, что такое «переменная» на самом деле. В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением LexicalEnvironment. Объект лексического окружения состоит из двух частей:

1. Environment Record – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение this).
2. Ссылка на внешнее лексическое окружение – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок).

Переменная – это просто свойство специального внутреннего объекта: Environment Record. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта». Например, в этом простом коде только одно лексическое окружение:



Это, так называемое, глобальное лексическое окружение, связанное со всем скриптом. На картинке выше прямоугольник означает Environment Record (хранилище переменных), а стрелка означает ссылку на внешнее окружение. У глобального лексического окружения нет внешнего окружения, так что она указывает на null. А вот как оно изменяется при объявлении и присваивании переменной:



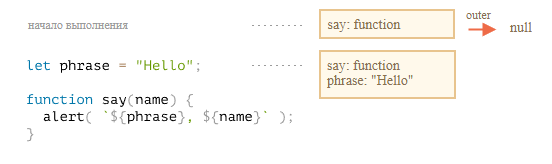
Прямоугольники с правой стороны демонстрируют, как глобальное лексическое окружение изменяется в процессе выполнения кода:

1. В начале скрипта лексическое окружение пустое.
2. Появляется определение переменной let phrase. У неё нет присвоенного значения, поэтому присваивается undefined.
3. Переменной phrase присваивается значение.
4. Переменная phrase меняет значение.

**[Function Declaration](https://learn.javascript.ru/closure" \l "function-declaration)**

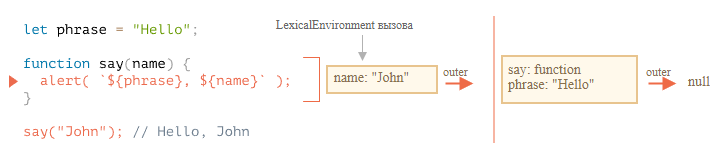
До сих поры рассматривались только переменные. Теперь рассмотрим Function Declaration. В отличие от переменных, объявленных с помощью let, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение. Для верхнеуровневых функций это означает момент, когда скрипт начинает выполнение. Вот почему можно вызвать функцию, объявленную через Function Declaration, до того, как она определена.

Следующий код демонстрирует, что уже с самого начала в лексическом окружении что-то есть. Там есть say, потому что это Function Declaration. И позже там появится phrase, объявленное через let:



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

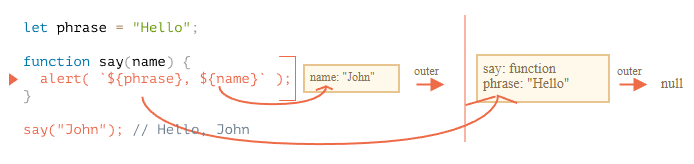
В течение вызова say() использует внешнюю переменную phrase. При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова. Например, для say("John") это выглядит так (выполнение находится на строке, отмеченной стрелкой):



Итак, в процессе вызова функции есть два лексических окружения: внутреннее (для вызываемой функции) и внешнее (глобальное). Внутреннее лексическое окружение соответствует текущему выполнению say. В нём находится одна переменная name, аргумент функции. При вызове say("John") значение переменной name равно "John". Внешнее лексическое окружение – это глобальное лексическое окружение. В нём находятся переменная phrase и сама функция. У внутреннего лексического окружения есть ссылка outer на внешнее. 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. Если переменная не была найдена, это будет ошибкой в strict mode. Без strict mode, для обратной совместимости, присваивание несуществующей переменной создаёт новую глобальную переменную с таким именем.

Рассмотрим, как происходит поиск в рассматриваемом примере:

* когда alert внутри say хочет получить доступ к name, он немедленно находит переменную в лексическом окружении функции;
* когда он хочет получить доступ к phrase, которой нет локально, он следует дальше по ссылке к внешнему лексическому окружению и находит переменную там.



Это ответ на первый вопрос из начала темы. Функция получает текущее значение внешних переменных, то есть, их последнее значение. Старые значения переменных нигде не сохраняются. Когда функция хочет получить доступ к переменной, она берёт её текущее значение из своего или внешнего лексического окружения.

Так что, ответ на первый вопрос – Pete:

let name = "John";

function sayHi() {

alert("Hi, " + name);

}

name = "Pete"; // (\*)

sayHi(); // Pete

Порядок выполнения кода, приведённого выше:

1. В глобальном лексическом окружении есть name: "John".
2. На строке (\*) глобальная переменная изменяется, теперь name: "Pete".
3. Выполняется функция sayHi() и берёт переменную name извне. Теперь из глобального лексического окружения, где переменная уже равна "Pete".

Обратите внимание, что новое лексическое окружение функции создаётся каждый раз, когда функция выполняется. Если функция вызывается несколько раз, то для каждого вызова будет своё лексическое окружение, со своими, специфичными для этого вызова, локальными переменными и параметрами.

Лексическое окружение – это специальный внутренний объект. Нельзя получить его в коде и изменять напрямую. Сам движок JavaScript может оптимизировать его, уничтожать неиспользуемые переменные для освобождения памяти и выполнять другие внутренние уловки, но видимое поведение объекта должно оставаться таким, как было описано.

**[Вложенные функции](https://learn.javascript.ru/closure" \l "vlozhennye-funktsii)**

Функция называется вложенной, когда она создаётся внутри другой функции. Можно использовать это для упорядочивания кода, например:

function sayHiBye(firstName, lastName) {

function getFullName() {

return firstName + " " + lastName;

}

alert( "Hello, " + getFullName() );

alert( "Bye, " + getFullName() );

}

Здесь вложенная функция getFullName() создана для удобства. Она может получить доступ к внешним переменным и, значит, вывести полное имя. В JavaScript вложенные функции используются очень часто.

Вложенная функция может быть возвращена: либо в качестве свойства нового объекта, либо сама по себе, и затем может быть использована в любом месте. Не важно где, она всё так же будет иметь доступ к тем же внешним переменным. Например, здесь, вложенная функция присваивается новому объекту в [конструкторе](https://learn.javascript.ru/constructor-new):

function User(name) {

this.sayHi = function() {

alert(name);

};

}

let user = new User("John");

user.sayHi();

Создаём и возвращаем функцию «счётчик»:

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter = makeCounter();

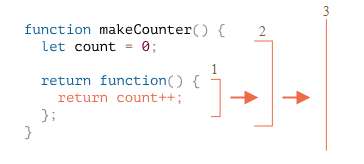
alert( counter() ); // 0

alert( counter() ); // 1

alert( counter() ); // 2

Рассмотрим далее makeCounter. Он создаёт функцию «counter», которая возвращает следующее число при каждом вызове. Несмотря на простоту, немного модифицированные варианты этого кода применяются на практике, например, в [генераторе псевдослучайных чисел](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D1%81%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D1%8B%D1%85_%D1%87%D0%B8%D1%81%D0%B5%D0%BB) и во многих других случаях.

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



1. Локальные переменные вложенной функции.
2. Переменные внешней функции.
3. И так далее, пока не будут достигнуты глобальные переменные.

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 было let count = 1.

Теперь рассмотрим два вопроса:

1. Можно ли каким-нибудь образом сбросить счётчик count из кода, который не принадлежит makeCounter? Например, после вызова alert в коде выше.
2. Если вызвать makeCounter несколько раз – возвращается много функций counter. Они независимы или разделяют одну и ту же переменную count?

Ответы на вопросы:

1. Такой возможности нет: count – локальная переменная функции, нельзя 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.

Вот демо:

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter1 = makeCounter();

let counter2 = makeCounter();

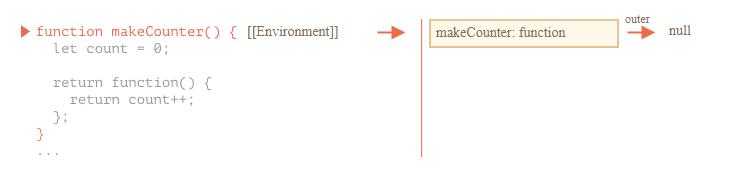
alert( counter1() ); // 0

alert( counter1() ); // 1

alert( counter2() ); // 0

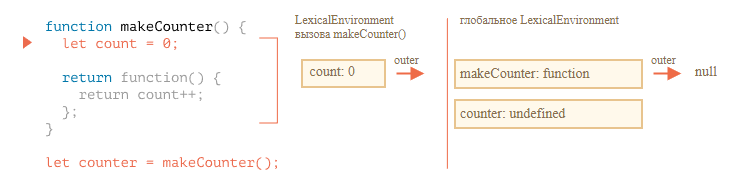
Рассмотрим, что происходит в примере с makeCounter шаг за шагом. Обратите внимание на дополнительное свойство [[Environment]].

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 создании 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. Другими словами, функция навсегда запоминает ссылку на лексическое окружение, где она была создана. И [[Environment]] – скрытое свойство функции, которое содержит эту ссылку.

1. Код продолжает выполняться, объявляется новая глобальная переменная counter, которой присваивается результат вызова makeCounter. На рисунке изображено состояние, когда интерпретатор находится на первой строке внутри makeCounter():



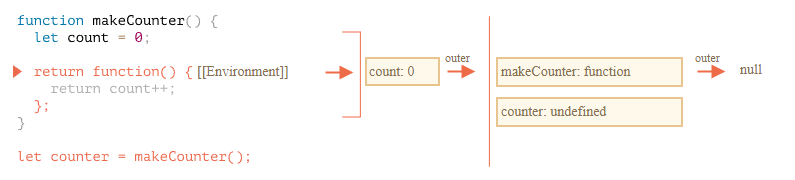
В момент вызова makeCounter() создаётся лексическое окружение, для хранения его переменных и аргументов. Как и все лексические окружения, оно содержит две вещи:

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

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

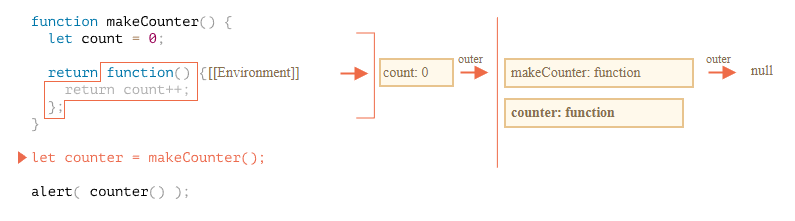
1. В процессе выполнения makeCounter() создаётся небольшая вложенная функция.

Не имеет значения, какой способ объявления функции используется: Function Declaration или Function Expression. Все функции получают свойство [[Environment]], которое ссылается на лексическое окружение, в которым они были созданы. Тоже происходит и с новой функцией. Для нее значением [[Environment]] будет текущее лексическое окружение makeCounter() (где она была создана):



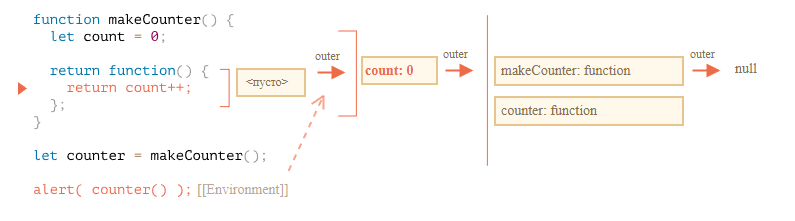
Обратите внимание, что на этом шаге внутренняя функция была создана, но ещё не вызвана. Код внутри function() { return count++ } не выполняется.

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



В этой функции есть только одна строчка: return count++, которая будет выполнена при вызове функции.

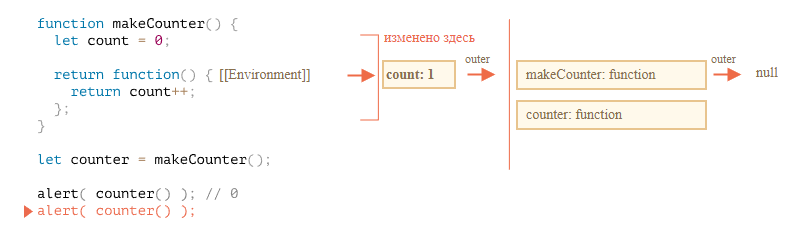
1. При вызове counter() для этого вызова создаётся новое лексическое окружение. Оно пустое, так как в самом counter локальных переменных нет. Но [[Environment]] counter используется, как ссылка на внешнее лексическое окружение outer, которое даёт доступ к переменным предшествующего вызова makeCounter,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

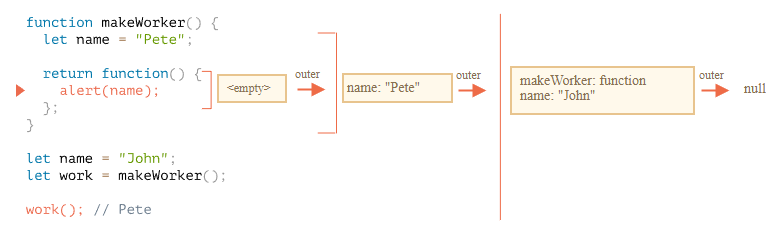
Обратите внимание, как здесь работает управление памятью. Хотя makeCounter() закончил выполнение некоторое время назад, его лексическое окружение остаётся в памяти, потому что есть вложенная функция с [[Environment]], который ссылается на него. В большинстве случаев, объект лексического окружения существует до того момента, пока есть функция, которая может его использовать. И только тогда, когда таких не остаётся, окружение уничтожается.

1. Вызов counter() не только возвращает значение count, но также увеличивает его. Модификация происходит «на месте». Значение count изменяется конкретно в том окружении, где оно было найдено.



1. Следующие вызовы counter() сделают то же самое.

Теперь ответ на второй вопрос из начала темы очевиден: функция work() в коде ниже получает name из того места, где была создана, через ссылку на внешнее лексическое окружение:



Так что, результатом будет "Pete". Но, если бы в makeWorker() не было let name, тогда бы поиск продолжился дальше и была бы взята глобальная переменная. В таком случае, результатом было бы "John".

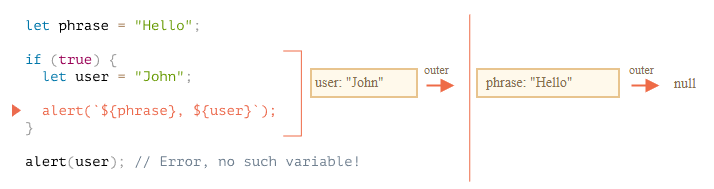
**Замыкания**

[Замыкание](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%D0%B5_(%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)) – это функция, которая запоминает свои внешние переменные и может получить к ним доступ. В некоторых языках это невозможно, или функция должна быть написана специальным образом, чтобы получилось замыкание. Но, как было описано выше, в JavaScript, все функции изначально являются замыканиями (есть только одно исключение, про которое будет рассказано в [Синтаксис "new Function"](https://learn.javascript.ru/new-function)). То есть, они автоматически запоминают, где были созданы, с помощью скрытого свойства [[Environment]] и все они могут получить доступ к внешним переменным.

**[Блоки кода и циклы](https://learn.javascript.ru/closure" \l "bloki-koda-i-tsikly-iife)**

Предыдущие примеры сосредоточены на функциях. Но лексическое окружение существует для любых блоков кода {...}. Лексическое окружение создаётся при выполнении блока кода и содержит локальные переменные для этого блока.

В следующем примере переменная user существует только в блоке if:



Когда выполнение попадает в блок if, для этого блока создаётся новое лексическое окружение. У него есть ссылка на внешнее окружение, так что phrase может быть найдена. Но все переменные и Function Expression, объявленные внутри if, остаются в его лексическом окружении и не видны снаружи. Например, после завершения if следующий alert не увидит user, что вызовет ошибку.

Для цикла у каждой итерации своё отдельное лексическое окружение. Если переменная объявлена в for(let ...), то она также в нём:

for (let i = 0; i < 10; i++) {

// {i: value}

}

alert(i); // Ошибка, нет такой переменной

Обявление *let i* визуально находится снаружи {...}, но конструкция for – особенная в этом смысле, у каждой итерации цикла своё собственное лексическое окружение с текущим *i* в нём. И так же, как и в if, ниже цикла *i* невидима.

Также можно использовать «простые» блоки кода {...}, чтобы изолировать переменные в «локальной области видимости». Например, в браузере все скрипты (кроме type="module") разделяют одну общую глобальную область. Так что, если создать глобальную переменную в одном скрипте, она станет доступна и в других. Но это становится источником конфликтов, если два скрипта используют одно и тоже имя переменной и перезаписывают друга друга. Это может произойти, если название переменной – широко распространённое слово, а авторы скрипта не знают друг о друге. Если надо этого избежать, то можно использовать блок кода для изоляции всего скрипта или какой-то его части:

{

let message = "Hello";

alert(message); // Hello

}

alert(message); // Ошибка: переменная message не определена

Из-за того, что у блока есть собственное лексическое окружение, код снаружи него (или в другом скрипте) не видит переменные этого блока.

Обычно лексическое окружение очищается и удаляется после того, как функция выполнилась. Например:

function f() {

let value1 = 123;

let value2 = 456;

}

f();

Здесь два значения, которые технически являются свойствами лексического окружения. Но после того, как f() завершится, это лексическое окружение станет недоступно, поэтому оно удалится из памяти. Но, если 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:

function f() {

let value = 123;

function g() { alert(value); }

return g;

}

let g = f();

Но на практике движки JavaScript пытаются это оптимизировать. Они анализирует использование переменных и, если легко по коду понять, что внешняя переменная не используется – она удаляется. Одним из важных побочных эффектов в V8 (Chrome, Opera) является то, что такая переменная становится недоступной при отладке. Это может привести к тому, что можно увидеть не ту внешнюю переменную при совпадающих названиях:

let value = "Сюрприз!";

function f() {

let value = "ближайшее значение";

function g() {

debugger; // в консоли: напишите alert(value);

}

return g;

}

let g = f();

g();

1. **Конструкция new Function.**

Существует ещё один вариант объявлять функции. Он используется крайне редко, но иногда другого решения не найти. Синтаксис для объявления функции:

let func = new Function([arg1, arg2, ...argN], functionBody);

Функция создается с заданными аргументами arg1...argN и телом functionBody. Это проще понять на конкретном примере. Здесь объявлена функция с двумя аргументами:

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.

Все предыдущие объявления требовали писать объявление функции в скрипте. Но new Function позволяет превратить любую строку в функцию. Например, можно получить новую функцию с сервера и затем выполнить ее:

let str = ... код, полученный с сервера динамически ...

let func = new Function(str);

func();

Это используется в очень специфических случаях, например, когда получаем код с сервера для динамической компиляции функции из шаблона, в сложных веб-приложениях.

Когда функция создаётся с использованием new Function, в её [[Environment]] записывается ссылка не на текущее лексическое окружение, а на глобальное. Поэтому такая функция не имеет доступа к внешним переменным, только к глобальным.

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

Новая функция должна взаимодействовать с основным скриптом. Если бы она имела доступ к внешним переменным это привело бы к проблеме. 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 достп. И внутри функции минификатор заменяет каждое её упоминание. Минификаторы анализируют структуру кода, и поэтому ничего не ломают.

Так что если бы new Function и имела доступ к внешним переменным, она не смогла бы найти переименованную userName. Кроме того, такой код был бы архитектурно хуже и более подвержен ошибкам. Чтобы передать что-то в функцию, созданную как new Function, можно использовать ее аргументы.

1. **Объект функции.**

Как известно, в JavaScript функция – это значение. Каждое значение в JavaScript имеет свой тип. В JavaScript, функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

**[Свойство name](https://learn.javascript.ru/function-object" \l "svoystvo-name)**

Объект функции содержит несколько полезных свойств. Например, имя функции доступно как свойство name:

function sayHi() {

alert("Hi");

}

alert(sayHi.name); // sayHi

Корректное имя присваивается даже в случае, если функция создается без имени и тут же присваивается:

let sayHi = function() {

alert("Hi");

};

alert(sayHi.name); // sayHi

Это даже работает в случае присваивания значения по умолчанию:

function f(sayHi = function() {}) {

alert(sayHi.name); // sayHi

}

f();

В спецификации это называется «контекстное имя»: если функция не имеет name, то JavaScript пытается понять его из контекста.

Также имена имеют и методы объекта:

let user = {

sayHi() {

// ...

},

sayBye: function() {

// ...

}

}

alert(user.sayHi.name); // sayHi

alert(user.sayBye.name); // sayBye

В этом нет никакой магии. Бывает, что корректное имя определить невозможно. В таких случаях свойство name имеет пустое значение. Например:

let arr = [function() {}];

alert( arr[0].name ); // <пустая строка>

На практике такое бывает редко, обычно функции имеют name.

**[Свойство length](https://learn.javascript.ru/function-object" \l "svoystvo-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

Как видно, троеточие, обозначающее «остальные параметры», здесь не учитываются.

Свойство length иногда используется для [интроспекций](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D1%80%D0%BE%D1%81%D0%BF%D0%B5%D0%BA%D1%86%D0%B8%D1%8F_(%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)) в функциях, которые работают с другими функциями. Например, в коде ниже функция ask принимает в качестве параметров вопрос question и произвольное количество функций-обработчиков ответа handler. Когда пользователь отвечает на вопрос, функция вызывает обработчики. Можно передать два типа обработчиков:

* Функцию без аргументов, которая будет вызываться только в случае положительного ответа.
* Функцию с аргументами, которая будет вызываться в обоих случаях и возвращать ответ.

Чтобы вызвать обработчик handler правильно, надо проверить свойство handler.length. Идея состоит в том, чтобы иметь простой синтаксис обработчика без аргументов для положительных ответов (наиболее распространённый случай), но также и возможность передавать универсальные обработчики:

function ask(question, ...handlers) {

let isYes = confirm(question);

for(let handler of handlers) {

if (handler.length == 0) {

if (isYes) handler();

} else {

handler(isYes);

}

}

}

ask("Вопрос?", () => alert('Вы ответили да'), result => alert(result));

**[Пользовательские свойства](https://learn.javascript.ru/function-object" \l "polzovatelskie-svoystva)**

Также можно добавить свои собственные свойства. Например, свойство counter для отслеживания общего количества вызовов:

function sayHi() {

alert("Hi");

sayHi.counter++;

}

sayHi.counter = 0;

sayHi(); // Hi

sayHi(); // Hi

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

**Свойство не есть переменная**

Свойство функции, назначенное как sayHi.counter = 0, не объявляет локальную переменную counter внутри неё. Другими словами, свойство counter и переменная let counter – это две независимые вещи. Можно использовать функцию как объект, хранить в ней свойства, но они никак не влияют на её выполнение. Переменные – это не свойства функции и наоборот.

Иногда свойства функции могут использоваться вместо замыканий. Например, можно переписать функцию-счетчик из вопроса про [замыкание](https://learn.javascript.ru/closure), используя её свойство:

function makeCounter() {

function counter() {

return counter.count++;

};

counter.count = 0;

return counter;

}

let counter = makeCounter();

alert( counter() ); // 0

alert( counter() ); // 1

Свойство count теперь хранится прямо в функции, а не в её внешнем лексическом окружении. Основное отличие такого подхода от замыкания в том, что если значение count живет во внешней переменной, то она не доступна для внешнего кода. Изменить её могут только вложенные функции. А если оно присвоено как свойство функции, то можно его получить:

function makeCounter() {

function counter() {

return counter.count++;

};

counter.count = 0;

return counter;

}

let counter = makeCounter();

counter.count = 10;

alert( counter() ); // 10

Поэтому выбор реализации зависит от целей разработчика.

**[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 func(who) {

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

};

Заметьте, что функция всё ещё задана как Function Expression. Добавление "func" после function не превращает объявление в Function Declaration, потому что оно все еще является частью выражения присваивания. Добавление такого имени ничего не ломает. Функция все еще доступна как sayHi():

let sayHi = function func(who) {

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

};

sayHi("John"); // Hello, John

Есть две важные особенности имени func, ради которого оно даётся:

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

Например, ниже функция sayHi вызывает себя с "Guest", если не передан параметр who:

let sayHi = function func(who) {

if (who) {

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

} else {

func("Guest");

}

};

sayHi(); // Hello, Guest

func(); // Ошибка

Не следует использовать имя sayHi для вложенного вызова, так как значение sayHi может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнет выдавать ошибки:

let sayHi = function(who) {

if (who) {

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

} else {

sayHi("Guest"); // Ошибка

}

};

let welcome = sayHi;

sayHi = null;

welcome(); // Ошибка

Так происходит, потому что функция берет 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. Все работает, потому что имя "func" локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию. Внешний код все еще содержит переменные sayHi и welcome, но теперь func – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Задать «внутреннее» имя можно только для Function Expression, и не нельзя для Function Declaration. Если нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

1. **Остаточные параметры и оператор расширения.**

Многие встроенные функции JavaScript поддерживают произвольное количество аргументов. Например: Math.max(arg1, arg2, ..., argN) – вычисляет максимальное число из переданных; Object.assign(dest, src1, ..., srcN) – копирует свойства из исходных объектов src1..N в целевой объект dest и др.

**[Остаточные параметры (...)](https://learn.javascript.ru/rest-parameters-spread-operator" \l "ostatochnye-parametry)**

Вызывать функцию можно с любым количеством аргументов, независимо от того, как она была определена. Например:

function sum(a, b) {

return a + b;

}

alert( sum(1, 2, 3, 4, 5) );

Лишние аргументы не вызовут ошибку, но приняты будут только первые два.

Остаточные параметры могут быть обозначены через три точки «...». Суть его в том, что оставшиеся параметры помещаются в массив. Например, соберём все аргументы в массив args:

function sumAll(...args) {

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

Можно положить первые несколько параметров в переменные и собрать в массив остальные. В примере ниже первые два аргумента функции станут именем и фамилией, а третий и последующие превратятся в массив titles:

function showName(firstName, lastName, ...titles) {

alert( firstName + ' ' + lastName ); // Юлий Цезарь

// titles = ["Консул", "Император"]

alert( titles[0] ); // Консул

alert( titles[1] ); // Император

alert( titles.length ); // 2

}

showName("Юлий", "Цезарь", "Консул", "Император");

Остаточные параметры собирают все остальные аргументы, поэтому бессмысленно писать что-либо после них. Это вызовет ошибку:

function f(arg1, ...rest, arg2) { // Ошибка

}

Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами. Например:

function showName() {

alert( arguments.length );

alert( arguments[0] );

alert( arguments[1] );

}

// Вывод: 2, Юлий, Цезарь

showName("Юлий", "Цезарь");

// Вывод: 1, Илья, undefined

showName("Илья");

Раньше в языке не было остаточных параметров, и получить все аргументы функции можно было только с помощью arguments. Этот способ всё ещё работает, его можно найти в старом коде. Но у него есть один недостаток. Хотя arguments похож на массив, и он тоже перебираемый, это всё же не массив. Он не поддерживает методы массивов, поэтому нельзя, например, вызвать arguments.map(...). К тому же, arguments всегда содержит все аргументы функции – нельзя получить их часть. А остаточные параметры позволяют это сделать.

Соответственно, для более удобной работы с аргументами лучше использовать остаточные параметры.

У стрелочных функций нет своего объекта arguments. Если обратиться к arguments из стрелочной функции, то получим аргументы внешней обычной функции. Пример:

function f() {

let showArg = () => alert(arguments[0]);

showArg(2);

}

f(1); // 1

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

Иногда нужно массив преобразовать в список параметров. 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

Если вызвать эту функцию для массива чисел [3, 5, 1], то она его не обработает, так как ожидает список параметров:

let arr = [3, 5, 1];

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

Чтобы преобразовать массив в список необходимо использовать оператор расширения. Он похож на остаточные параметры – тоже использует ..., но делает совершенно противоположное. Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов. Для Math.max:

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5

Этим же способом можно передать несколько итерируемых объектов и комбинировать оператор расширения с обычными значениями:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

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

Оператора расширения работает с любым перебираемым объектом. Например, оператор расширения подойдёт для того, чтобы превратить строку в массив символов:

let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т

Оператор расширения использует итераторы, чтобы собирать элементы. Так же, как это делает for..of. Цикл for..of перебирает строку как последовательность символов, поэтому из ...str получается "П", "р", "и", "в", "е", "т". Получившиеся символы собираются в массив при помощи стандартного объявления массива: [...str].

Для этой задачи можно использовать и Array.from. Он тоже преобразует перебираемый объект (такой как строка) в массив:

let str = "Привет";

alert( Array.from(str) ); // П,р,и,в,е,т

Результат аналогичен [...str]. Но между Array.from(obj) и [...obj] есть разница: Array.from работает как с псевдомассивами, так и с итерируемыми объектами; оператор расширения работает только с итерируемыми объектами. Таким образом, если нужно сделать из чего угодно массив, Array.from – более универсальный метод.

1. **Каррирование**

*[Каррирование](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)* или карринг (currying) в функциональном программирование – это преобразование функции с множеством аргументов в набор вложенных функций с одним аргументом. При вызове каррированной функции с передачей ей одного аргумента, она возвращает новую функцию, которая ожидает поступления следующего аргумента. Новые функции, ожидающие следующего аргумента, возвращаются при каждом вызове каррированной функции – до тех пор, пока функция не получит все необходимые ей аргументы. Ранее полученные аргументы, благодаря механизму замыканий, ждут того момента, когда функция получит всё, что ей нужно для выполнения вычислений. После получения последнего аргумента функция выполняет вычисления и возвращает результат.

Говоря о [каррировании](https://medium.com/@kbrainwave/currying-in-javascript-ce6da2d324fe), можно сказать, что это процесс превращения функции с несколькими аргументами в функцию с меньшей арностью.

*[Арность](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C)* – это количество аргументов функции. Например – вот объявление пары функций:

function fn(a, b) {

*//...*

}

function \_fn(a, b, c) {

*//...*

}

Функция fn принимает два аргумента (это бинарная или 2-арная функция), функция \_fn принимает три аргумента (тернарная, 3-арная функция).

В результате каррирования, функция с несколькими аргументами преобразуется в набор функций, каждая из которых принимает один аргумент. Рассмотрим пример:

function multiply(a, b, c) {

   return a \* b \* c;

}

Функция multiply принимает три аргумента и возвращает их произведение:  
  
multiply(1,2,3); *// 6*

Преобразовать её к набору функций, каждая из которых принимает один аргумент. Создадим каррированный вариант этой функции и посмотрим на то, как получить тот же результат в ходе вызова нескольких функций:

function multiply(a) {

   return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

}

log(multiply(1)(2)(3)) *// 6*

В примере вызов единственной функции с тремя аргументами multiply(1,2,3) преобразован к вызову трёх функций – multiply(1)(2)(3). При использовании новой конструкции каждая функция, кроме последней, возвращающей результат вычислений, принимает аргумент и возвращает другую функцию, также способную принять аргумент и возвратить другую функцию. Распишем конструкцию вида multiply(1)(2)(3), чтобы она была более понятной:

const mul1 = multiply(1)*;*

const mul2 = mul1(2)*;*

const result = mul2(3)*;*

log(result)*; // 6*

Подробнее разберём, что здесь происходит. Сначала передается аргумент 1 функции multiply:

const mul1 = multiply(1);

При работе этой функции срабатывает такая конструкция:

return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

Теперь в mul1 имеется ссылка на функцию, принимающую аргумент b. Вызовем функцию mul1, передав ей 2:

const mul2 = mul1(2);

В результате этого вызова выполнится следующий код:

return (c) => {

           return a \* b \* c

       }

Константа mul2 будет содержать ссылку на функцию, которая могла бы оказаться в ней, например, в результате выполнения следующей операции:

mul2 = (c) => {

           return a \* b \* c

       }

Если теперь вызвать функцию mul2, передав ей 3, то функция выполнит необходимые вычисления, воспользовавшись аргументами a и b:

const result = mul2(3);

Результатом этих вычислений будет 6:

log(result)*; // 6*

Функция mul2, обладающая самым большим уровнем вложенности, имеет доступ к областям видимости, к замыканиям, формируемым функциями multiply и mul1. Именно поэтому в функции mul2можно производить вычисления с переменными, объявленными в функциях, выполнение которых уже завершено, которые уже возвратили некие значения и обработаны сборщиком мусора.

**Каррирование и частичное применение функций**

Сейчас, возникает ощущение, что количество вложенных функций, при представлении функции в виде набора вложенных функций, зависит от количества аргументов функции. И, если речь идёт о каррировании, то это так. Особый вариант функции для вычисления объёма, можно сделать таким:

function volume(l) {

   return (w, h) => {

       return l \* w \* h

   }

}

Здесь применены идеи, очень похожие на рассмотренные выше. Пользоваться этой функцией можно так:

const hV = volume(70)*;*

hV(203,142)*;*

hV(220,122)*;*

hV(120,123)*;*

А можно и так:

volume(70)(90,30)*;*

volume(70)(390,320)*;*

volume(70)(940,340)*;*

Фактически, здесь командой volume(70), создана специализированная функция для вычисления объёма тел, одно из измерений которых (а именно – длина, l), зафиксировано. Функция volume ожидает 3 аргумента и содержит 2 вложенных функции, в отличие от предыдущей версии подобной функции, каррированный вариант которой содержал 3 вложенных функции.

Та функция, которая получилась после вызова volume(70) реализует концепцию частичного применения функции (partial function application). Каррирование и частичное применение функций очень похожи друг на друга, но концепции это разные.

При частичном применении функцию преобразуют в другую функцию, обладающую меньшим числом аргументов (меньшей арностью). Некоторые аргументы такой функции оказываются зафиксированными (для них задаются значения по умолчанию).

Каррирование и частичное применение функций может оказаться полезным в различных ситуациях. Например – при разработке небольших модулей, подходящих для повторного использования.

Частичное применение функций позволяет облегчить использование универсальных модулей. Например, есть интернет-магазин, в коде которого имеется функция, которая используется для вычисления суммы к оплате с учётом скидки.

function discount(price, discount) {

   return price \* discount

}

Есть определённая категория клиентов, которые получают скидку в 10%. Например, если такой клиент покупает что-то на $500, то он получает скидку размером $50:

const price = discount(500,0.10); // $50

// $500 - $50 = $450

Несложно заметить, что при таком подходе, постоянно придётся вызывать эту функцию с двумя аргументами:

const price = discount(1500,0.10); // $150

// $1,500 - $150 = $1,350

const price = discount(2000,0.10); // $200

// $2,000 - $200 = $1,800

const price = discount(50,0.10); // $5

// $50 - $5 = $45

const price = discount(5000,0.10); // $500

// $5,000 - $500 = $4,500

const price = discount(300,0.10); // $30

// $300 - $30 = $270

Исходную функцию можно привести к такому виду, который позволял бы получать новые функции с заранее заданным уровнем скидки, при вызове которых им достаточно передавать сумму покупки. Функция discount() в примере имеет два аргумента. Преобразуем ее следующим образом:

function discount(discount) {

   return (price) => {

       return price \* discount;

   }

}

const tenPercentDiscount = discount(0.1);

Функция tenPercentDiscount() представляет собой результат частичного применения функции discount(). При вызове tenPercentDiscount() этой функции достаточно передать цену, а скидка в 10%, то есть – аргумент discount, уже будет задана:

tenPercentDiscount(500); // $50

// $500 - $50 = $450

Если в магазине имеются покупатели, которым решено дать скидку размером в 20%, то получить соответствующую функцию для работы с ними можно так:

const twentyPercentDiscount = discount(0.2);

Теперь функцию twentyPercentDiscount() можно вызывать для расчёта стоимости товаров с учётом скидки в 20%:

twentyPercentDiscount(500); // 100

// $500 - $100 = $400

twentyPercentDiscount(5000); // 1000

// $5,000 - $1,000 = $4,000

twentyPercentDiscount(1000000); // 200000

// $1,000,000 - $200,000 = $600,000

1. **Генераторы**

Обычные функции возвращают только одно-единственное значение (или ничего). Генераторы могут порождать (yield) множество значений одно за другим, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

**[Функция-генератор](https://learn.javascript.ru/generators" \l "funktsiya-generator)**

Для объявления генератора используется специальная синтаксическая конструкция: function\* f(…) (или function \*f(…)), которая называется «функция-генератор». Выглядит она так:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, она не выполняет свой код. Вместо этого она возвращает специальный объект, так называемый «генератор» для управления её выполнением:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

alert(generator); // [object Generator]

Основным методом генератора является next(). При вызове он запускает выполнение кода до ближайшей инструкции yield <значение> (значение может отсутствовать, в этом случае оно полагается равным undefined). По достижении yield выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода next() всегда является объект с двумя свойствами:

* value: значение из yield.
* done: true, если выполнение функции завершено, иначе false.

Создадим генератор и получим первое из возвращаемых им значений:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

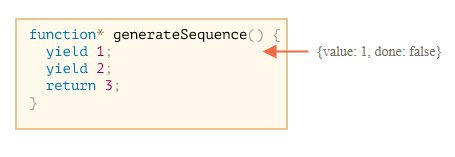
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}

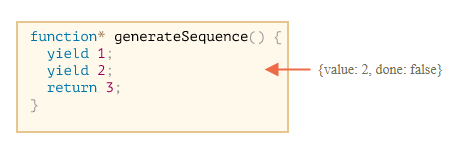
На данный момент получено только первое значение, выполнение функции остановлено на второй строке:



Повторный вызов generator.next() возобновит выполнение кода и вернёт результат следующего yield:

let two = generator.next();

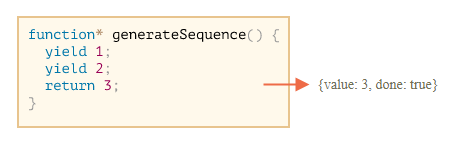
alert(JSON.stringify(two)); // {value: 2, done: false}



И, наконец, последний вызов завершит выполнение функции и вернёт результат return:

let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true}



Сейчас генератор полностью выполнен. Можно увидеть это по свойству done:true и обработать value:3 как окончательный результат. Новые вызовы generator.next() больше не имеют смысла. Впрочем, если они и будут, то не вызовут ошибки, но будут возвращать один и тот же объект: {done: true}.

**[Перебор генераторов](https://learn.javascript.ru/generators" \l "perebor-generatorov)**

Генераторы являются [перебираемыми](https://learn.javascript.ru/iterable) объектами. Возвращаемые ими значения можно перебирать через for..of:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, затем 2

}

Обратите внимание: пример выше выводит значение 1, затем 2. Значение 3 выведено не будет. Это потому что перебор через for..of игнорирует последнее значение, при котором done: true. Поэтому, если необходимо, чтобы были все значения при переборе через for..of, то надо возвращать их через yield:

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, затем 2, затем 3

}

Так как генераторы являются перебираемыми объектами, можно использовать всю связанную с ними функциональность, например, оператор расширения ...:

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let sequence = [0, ...generateSequence()];

alert(sequence); // 0, 1, 2, 3

В коде выше ...generateSequence() превращает перебираемый объект-генератор в массив элементов.

В примерах выше генерируются конечные последовательности, но также можно сделать генератор, который будет возвращать значения бесконечно. Например, бесконечная последовательность псевдослучайных чисел. Тут потребуется break (или return) в цикле for..of по такому генератору, иначе цикл будет продолжаться бесконечно и скрипт «зависнет».

**[Композиция генераторов](https://learn.javascript.ru/generators" \l "kompozitsiya-generatorov)**

Композиция генераторов – это особенная возможность генераторов, которая позволяет прозрачно «встраивать» генераторы друг в друга. Например, есть функция для генерации последовательности чисел:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

Надо использовать её при генерации более сложной последовательности: сначала цифры 0..9 (с кодами символов 48…57), за которыми следуют буквы алфавита a..z (коды символов 65…90), за которыми следуют буквы в верхнем регистре A..Z (коды символов 97…122). Можно использовать такую последовательность для генерации паролей, выбирать символы из неё (может быть, ещё добавить символы пунктуации), но сначала её нужно сгенерировать.

В обычной функции, чтобы объединить результаты из нескольких других функций, надо вызвать их, сохранить промежуточные результаты, а затем в конце их объединить. Для генераторов есть особый синтаксис yield\*, который позволяет «вкладывать» генераторы один в другой (осуществлять их композицию). Пример генератора с композицией:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

function\* generatePasswordCodes() {

// 0..9

yield\* generateSequence(48, 57);

// A..Z

yield\* generateSequence(65, 90);

// a..z

yield\* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {

str += String.fromCharCode(code);

}

alert(str); // 0..9A..Za..z

Директива yield\* делегирует выполнение другому генератору. Этот термин означает, что yield\* gen перебирает генератор gen и прозрачно направляет его вывод наружу. Как если бы значения были сгенерированы внешним генератором.

Результат – такой же, как если встроить код из вложенных генераторов:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

function\* generateAlphaNum() {

// yield\* generateSequence(48, 57);

for (let i = 48; i <= 57; i++) yield i;

// yield\* generateSequence(65, 90);

for (let i = 65; i <= 90; i++) yield i;

// yield\* generateSequence(97, 122);

for (let i = 97; i <= 122; i++) yield i;

}

let str = '';

for(let code of generateAlphaNum()) {

str += String.fromCharCode(code);

}

alert(str); // 0..9a..zA..Z

Композиция генераторов – естественный способ вставлять вывод одного генератора в поток другого. Она не использует дополнительную память для хранения промежуточных результатов.

Директива yield не только возвращает результат наружу, но и может передавать значение извне в генератор. Чтобы это сделать, нужно вызвать generator.next(arg) с аргументом. Этот аргумент становится результатом yield. Продемонстрируем это на примере:

function\* gen() {

// Передаём вопрос во внешний код и ожидаем ответа

let result = yield "2 + 2 = ?"; // (\*)

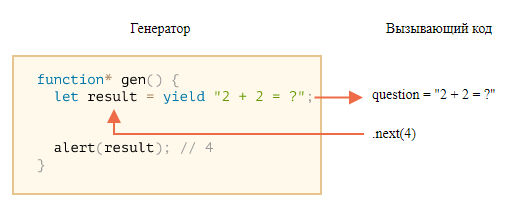
alert(result);

}

let generator = gen();

let question = generator.next().value; // <-- yield возвращает значение

generator.next(4); // --> передаём результат в генератор



1. Первый вызов generator.next() – всегда без аргумента, он начинает выполнение и возвращает результат первого yield "2+2=?". На этой точке генератор приостанавливает выполнение.
2. Затем, как показано на картинке выше, результат yield переходит во внешний код в переменную question.
3. При generator.next(4) выполнение генератора возобновляется, а 4 выходит из присваивания как результат: let result = 4.

Обратите внимание, что внешний код не обязан немедленно вызывать next(4). Ему может потребоваться время, генератор подождёт. Например:

// возобновить генератор через некоторое время

setTimeout(() => generator.next(4), 1000);

Как видно, в отличие от обычных функций, генератор может обмениваться результатами с вызывающим кодом, передавая значения в next/yield. Чтобы сделать происходящее более очевидным, вот ещё один пример с большим количеством вызовов:

function\* gen() {

let ask1 = yield "2 + 2 = ?";

alert(ask1); // 4

let ask2 = yield "3 \* 3 = ?"

alert(ask2); // 9

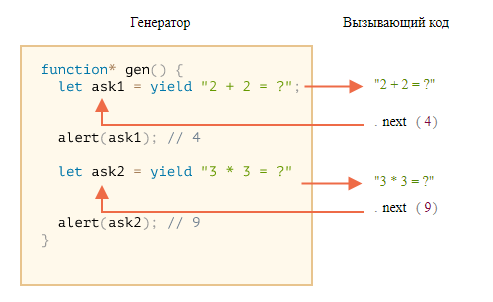
}

let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"

alert( generator.next(4).value ); // "3 \* 3 = ?"

alert( generator.next(9).done ); // true



1. Первый .next() начинает выполнение, доходит до первого yield.
2. Результат возвращается во внешний код.
3. Второй .next(4) передаёт 4 обратно в генератор как результат первого yield и возобновляет выполнение.
4. Выполнение доходит до второго yield, который станет результатом .next(4).
5. Третий next(9) передаёт 9 в генератор как результат второго yield и возобновляет выполнение, которое завершается окончанием функции, так что done: true.

Таким образом, каждый next(value) передаёт в генератор значение, которое становится результатом текущего yield, возобновляет выполнение и получает выражение из следующего yield.

**[generator.throw](https://learn.javascript.ru/generators" \l "generator-throw)**

Внешний код может передавать значение в генератор как результат yield. Но можно передать не только результат, но и инициировать (throw) ошибку. Это естественно, так как ошибка является своего рода результатом. Для того, чтобы передать ошибку в yield, нужно вызвать generator.throw(err). В этом случае, исключение err возникнет на строке с yield. Например, здесь yield "2 + 2 = ?" приведёт к ошибке:

function\* gen() {

try {

let result = yield "2 + 2 = ?"; // (1)

alert("Выполнение программы не дойдет до этой строки, потому что выше возникнет исключение");

} catch(e) {

alert(e); // покажет ошибку

}

}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("Ответ не найден в моей базе данных")); // (2)

Ошибка, которая проброшена в генератор на строке (2), приводит к исключению на строке (1) с yield. В примере выше try..catch перехватывает её и отображает. Если необходимо перехватывать её, то она, как и любое обычное исключение, будет передана из генератора во внешний код. Текущая строка вызывающего кода – это строка с generator.throw, отмечена (2). Таким образом, можно отловить её во внешнем коде, как здесь:

function\* generate() {

let result = yield "2 + 2 = ?"; // Ошибка в этой строке

}

let generator = generate();

let question = generator.next().value;

try {

generator.throw(new Error("Ответ не найден в моей базе данных"));

} catch(e) {

alert(e); // покажет ошибку

}

**Четвертая тема**

1. **Методы объектов, this.**

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

let user = {

name: "Джон",

age: 30

};

И так же, как и в реальном мире, пользователь может совершать действия: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п. Такие действия в JavaScript представлены свойствами-функциями объекта. Для начала добавим в объект user функцию приветствия:

let user = {

name: "Джон",

age: 30

};

user.sayHi = function() {

alert("Привет!");

};

user.sayHi(); // Привет!

Здесь просто использовано Function Expression, чтобы создать функцию для приветствия, и присвоить её свойству user.sayHi объекта user. Затем она вызывается.

Функцию, которая является свойством объекта, называют *методом* этого объекта. Таким образом создан метод sayHi объекта user. Конечно, можно было бы заранее объявить функцию и использовать её в качестве метода, например так:

let user = {

// ...

};

function sayHi() {

alert("Привет!");

};

user.sayHi = sayHi;

user.sayHi(); // Привет!

Существует более короткий синтаксис для методов в литерале объекта:

user = {

sayHi() {

alert("Привет");

}

};

Т.е., можно пропустить ключевое слово "function" и просто написать sayHi(). Нужно отметить, что эти две записи не полностью эквивалентны. Есть тонкие различия, связанные с наследованием объектов, но на данном этапе изучения это неважно. В большинстве случаев сокращённый синтаксис предпочтителен.

В JavaScript иногда необходимо сначала проверить, существует ли объект, а затем попытаться получить одно из его свойств, например, так:

const car = null;

const color = car && car.color;

Даже если car имеет значение null, у нас нет ошибок, а color присваивается значение null. Используя оператор &&, можно пройти несколько уровней вложенностей:

const car = {}

const colorName = car && car.color && car.color.name;

Оператор *опциональной последовательности* позволяет сделать код короче:

const car = {}

const color = car?.color;

const colorName = car?.color?.name;

Если car имеет значение null или undefined, результат будет undefined.

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода). Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user. Для доступа к информации внутри объекта метод может использовать ключевое слово this. Значение this – это объект «перед точкой», который использовался для вызова метода. Например:

let user = {

name: "Джон",

age: 30,

sayHi() {

// this - это "текущий объект"

alert(this.name);

}

};

user.sayHi(); // Джон

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user). Технически также возможно получить доступ к объекту без ключевого слова this, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект):

let user = {

name: "Джон",

age: 30,

sayHi() {

alert(user.name);

}

};

Но такой код будет ненадёжным. Если скопировать ссылку на объект user в другую переменную, например, admin = user, и перезаписать переменную user чем-то другим, тогда будет осуществлён доступ к неправильному объекту при вызове метода из admin. Это показано ниже:

let user = {

name: "Джон",

age: 30,

sayHi() {

alert( user.name );

}

};

let admin = user;

user = null;

admin.sayHi(); // Ошибка!

Если использовать this.name вместо user.name внутри alert, тогда этот код будет работать.

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции. 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 );

}

user.f = sayHi;

admin.f = sayHi;

user.f(); // Джон (this == user)

admin.f(); // Админ (this == admin)

admin['f'](); // Админ

Правило простое: при вызове obj.f() значение this внутри f равно obj. Так что, в приведённом примере это user или admin.

Можно 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 попытаться получить доступ к name, используя this.name – это вызовет ошибку.

В нестрогом режиме значением this в таком случае будет глобальный объект. Обычно подобный вызов является ошибкой программирования. Если внутри функции используется this, тогда ожидается, что она будет вызываться в контексте какого-либо объекта.

В других языках программирования this фиксировано – 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»).

Такая особенность 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.

Некоторые хитрые способы вызова метода приводят к потере значения this, например:

let user = {

name: "Джон",

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

bye() { alert("Пока"); }

};

user.hi();

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

В последней 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: при вызове будет ошибка, потому что значением "this" внутри функции становится undefined (полагаем, что у нас строгий режим). Так работает (доступ к методу объекта через точку):

user.hi();

Так уже не работает (вызываемый метод вычисляется):

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

Чтобы понять, почему так происходит, разберёмся, как работает вызов методов (obj.method()). В выражении obj.method() сначала оператор точка '.' возвращает свойство объекта – его метод (obj.method). Затем скобки () вызывают этот метод (исполняется код метода). Если поместить эти операции в отдельные строки, то значение this, естественно, будет потеряно:

let user = {

name: "Джон",

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

}

let hi = user.hi;

hi(); // Ошибка

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/" \l "sec-reference-specification-type). Этот ссылочный тип является внутренним типом. Нельзя явно использовать его, но он используется внутри языка. Значение ссылочного типа – это «триплет»: комбинация из трех 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).

Результатом доступа к свойству user.hi является не функция, а значение ссылочного типа. Для user.hi в строгом режиме оно будет таким:

// 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, 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).

Стрелочные функции особенные: у них нет своего «собственного» this. Если использовать this внутри стрелочной функции, то его значение берётся из внешней обычной функции. Например, здесь arrow() использует значение this из внешнего метода user.sayHi():

let user = {

firstName: "Вася",

sayHi() {

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

arrow();

}

};

user.sayHi(); // Вася

Это является особенностью стрелочных функций. Они полезны, когда нет необходимости иметь отдельное значение this, а надо брать его из внешнего контекста.

1. **Преобразование объектов.**

Если сложить два объекта obj1 + obj2, вычесть один из другого obj1 - obj2 или вывести их на экран, воспользовавшись alert(obj), то объекты сначала автоматически преобразуются в примитивы, а затем выполнится операция.

Правила преобразования объектов:

1. Все объекты в логическом контексте являются true. Существуют лишь их численные и строковые преобразования.
2. Численные преобразования происходят, когда осуществляется вычитание объектов или выполняются другие математические операции. Например, объекты Date могут вычитаться и результатом date1 - date2 будет временной отрезок между двумя датами.
3. Что касается строковых преобразований – они обычно происходят, при выводе объекта по типу alert(obj) и в подобных случаях.

**[Преобразование к примитивам](https://learn.javascript.ru/object-toprimitive" \l "preobrazovanie-k-primitivam)**

Можно настраивать строковые и численные преобразования, используя специальные методы объекта. Существуют три варианта преобразований («три хинта»), описанные в [спецификации](https://tc39.github.io/ecma262/" \l "sec-toprimitive):

* "string" – для преобразования объекта к строке, когда операция ожидает получить строку, например, alert:

alert(obj);

// объект в качества имени свойства

anotherObj[obj] = 123;

* "number" – для преобразования объекта к числу, в случае математических операций:

// явное преобразование

let num = Number(obj);

// математическое (исключая бинарный "+")

let n = +obj; // унарный плюс

let delta = date1 - date2;

// больше/меньше сравнения

let greater = user1 > user2;

* "default" – происходит редко, когда оператор «не уверен», какой тип ожидать.

Например, бинарный плюс + может работать с обоими типами: строками (конкатенировать их) и числами (складывать). Таким образом, и те, и другие будут вычисляться. Или, когда происходит сравнение объектов с помощью нестрогого равенства == со строкой, числом или символом, и неясно какое преобразование должно быть выполнено.

// бинарный плюс

let total = car1 + car2;

// obj == string/number/symbol

if (user == 1) { ... };

Оператор больше/меньше <> также может работать, как со строками, так и с числами. Однако, по историческим причинам он использует хинт «number», а не «default».

На практике все встроенные объекты, исключая Date, реализуют "default" преобразования тем же способом, что и "number".

Обратите внимание, что существуют лишь три варианта хинтов. Не существует хинта со значением «boolean» (все объекты являются true в логическом контексте) или каких-либо ещё.

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.

**Метод [Symbol.toPrimitive](https://learn.javascript.ru/object-toprimitive" \l "symbol-toprimitive)**

Метод используется для всех преобразований:

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

};

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;

}

};

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

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

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

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

**[Методы toString/valueOf](https://learn.javascript.ru/object-toprimitive" \l "metody-tostring-valueof)**

Методы toString и valueOf существуют давно. Они не символы, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода Symbol.toPrimitive, движок JavaScript пытается найти эти методы и вызвать следующим образом:

* toString – valueOf для хинта со значением «string».
* valueOf – toString – в ином случае.

Для примера, используем их в реализации всё того же объекта user. Воспроизведём его поведение комбинацией методов toString и valueOf:

let user = {

name: "John",

money: 1000,

// "string"

toString() {

return `{name: "${this.name}"}`;

},

// "number" или "default"

valueOf() {

return this.money;

}

};

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

alert(+user); // valueOf -> 1000

alert(user + 500); // valueOf -> 1500

Как видно, получилось то же поведение, что и у предыдущего примера с Symbol.toPrimitive.

Часто надо описать одно универсальное преобразование объекта к примитиву, для всех ситуаций. Для этого достаточно создать один toString:

let user = {

name: "John",

toString() {

return this.name;

}

};

alert(user); // toString -> John

alert(user + 500); // toString -> John500

В отсутствие Symbol.toPrimitive и valueOf, toString обработает все случаи преобразований к примитивам.

Важно понимать, что все описанные методы для преобразований объектов не обязаны возвращать именно «требуемый хинтом» тип примитива. Нет требований, чтобы toString() возвращал строго строку, или к тому, чтобы метод Symbol.toPrimitive возвращал число для хинта равного «number». Единственное обязательное требование: методы должны возвращать примитив, а не объект.

Если toString или valueOf вернёт объект, то ошибки не будет, но такое значение будет проигнорировано (как если бы метода вообще не существовало). Метод Symbol.toPrimitive, напротив, обязан возвращать примитив, иначе будет ошибка.

**[Последующие операции](https://learn.javascript.ru/object-toprimitive" \l "posleduyuschie-operatsii)**

Операция, инициализировавшая преобразование, получает примитив, и затем продолжает работу с ним, производя дальнейшие преобразования, если это необходимо. Например:

* Математические операции, исключая бинарный плюс, преобразуют примитив к числу:

let obj = {

toString() {

return "2";

}

};

alert(obj \* 2); // 4

* Бинарный плюс + в аналогичном случае сложит строки:

let obj = {

toString() {

return "2";

}

};

alert(obj + 2); // 22

1. **Создание объектов через "new".**

Обычный синтаксис {...} позволяет создать только один объект. Но часто нужно создать множество однотипных объектов, таких как пользователи, элементы меню и т.д. Это можно сделать при помощи функции-конструктора и оператора "new".

**[Функция-конструктор](https://learn.javascript.ru/constructor-new" \l "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 = {};

this.name = name;

this.isAdmin = false;

// return this;

}

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

let user = {

name: "Вася",

isAdmin: false

};

Теперь, когда необходимо будет создать других пользователей, можно использовать new User("Маша"), new User("Даша") и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Ещё раз заметим: технически, любая функция может быть использована как конструктор. То есть, каждая функция может быть вызвана при помощи оператора new и выполнит алгоритм, указанный выше в примере. Заглавная буква в названии функции является всеобщим соглашением по именованию, она как бы подсказывает разработчику, что данная функция является функцией-конструктором и её нужно вызывать через new.

**new function() { … }**

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

let user = new function() {

this.name = "Вася";

this.isAdmin = false;

// ...

};

Такой конструктор не может быть вызван дважды, так как он нигде не сохраняется, просто создаётся и тут же вызывается. Таким образом, такой метод создания позволяет инкапсулировать код, который создаёт отдельный объект, но без возможности его повторного использования. Данный метод используется очень редко.

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

Обычно конструкторы ничего не возвращают. 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";

// возвращает объект

return { name: "Godzilla" };

}

alert( new BigUser().name );

Пример с пустым return (можно поставить примитив после return, не важно):

function SmallUser() {

this.name = "Вася";

return; // возвращает this

// ...

}

alert( new SmallUser().name ); // Вася

Можно не ставить скобки после new, если вызов конструктора идёт без аргументов:

let user = new User;

// то же, что и

let user = new User();

Пропуск скобок считается плохой практикой, но синтаксис языка такое позволяет.

**[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 можно 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() { ... }

}

\*/

1. **Флаги и дескрипторы свойств.**

**[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.

Эти атрибуты обычно скрыты. Когда создается свойство «обычным способом», все эти атрибуты имеют значение true. Но можно 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:

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

* obj – объект, из которого получаем информацию.
* propertyName – имя свойства.

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

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

}

\*/

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

Object.defineProperty(obj, propertyName, descriptor)

* 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 создает 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:

let user = {};

Object.defineProperty(user, "name", {

value: "John"

});

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

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

/\*

{

"value": "John",

"writable": false,

"enumerable": false,

"configurable": false

}

\*/

Сравните этот способ с user.name, который создан выше «обычным способом»: в этот раз все флаги имеют значение false. Если это не то, что нужно, надо присвоить им значения true в параметре descriptor.

Рассмотрим на примерах, что 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:

let user = {

name: "John"

};

Object.defineProperty(user, "name", {

writable: false

});

user.name = "Pete"; // Ошибка

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, не видно 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.

Вот тот же пример, но свойство создано «с нуля»:

let user = { };

Object.defineProperty(user, "name", {

value: "John",

enumerable: true,

configurable: true

});

alert(user.name); // John

user.name = "Pete"; // Ошибка

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

Добавим собственный метод toString к объекту user. Встроенный метод toString в объектах – неперечисляемый, его не видно в цикле for..in. Но если написать свой собственный метод toString, цикл for..in будет выводить его по умолчанию:

let user = {

name: "John",

toString() {

return this.name;

}

};

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

Если в этом нет необходимости, можно установить для свойства enumerable: false. Тогда оно перестанет появляться в цикле for..in, аналогично встроенному toString:

let user = {

name: "John",

toString() {

return this.name;

}

};

Object.defineProperty(user, "toString", {

enumerable: false

});

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

Неперечисляемые свойства также не возвращаются Object.keys:

alert(Object.keys(user)); // name

**[Неконфигурируемое свойство](https://learn.javascript.ru/property-descriptors" \l "nekonfiguriruemoe-svoystvo)**

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

let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

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

/\*

{

"value": 3.141592653589793,

"writable": false,

"enumerable": false,

"configurable": false

}

\*/

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

Math.PI = 3; // Ошибка

Если свойство определено как неконфигурируемое, то нельзя поменять его обратно, потому что defineProperty не работает с неконфигурируемыми свойствами. В коде ниже свойство name определено как константа:

let user = { };

Object.defineProperty(user, "name", {

value: "John",

writable: false,

configurable: false

});

Object.defineProperty(user, "name", {writable: true}); // Ошибка

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:

Object.defineProperties(obj, {

prop1: descriptor1,

prop2: descriptor2

// ...

});

Например:

Object.defineProperties(user, {

name: { value: "John", writable: false },

surname: { value: "Smith", writable: false },

// ...

});

Таким образом, можно определить множество свойств одной операцией.

**Метод [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:

let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));

Обычно при клонировании объекта используется присваивание, чтобы скопировать его свойства:

for (let key in user) {

clone[key] = user[key]

}

Но это не копирует флаги. Поэтому если нужен клон с флагами, предпочтительнее использовать Object.defineProperties. Другое отличие в том, что for..in игнорирует символьные свойства, а Object.getOwnPropertyDescriptors возвращает дескрипторы всех свойств, включая свойства-символы.

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

Дескрипторы свойств работают на уровне конкретных свойств. Но еще 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.

1. **Геттеры и сеттеры.**

Есть два типа свойств объекта. Первый тип это свойства-данные (data properties). Все свойства, которые использовались до текущего момента были свойствами-данными. Второй тип свойств 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:

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 назначается. Например, есть объект user со свойствами name и surname:

let user = {

name: "John",

surname: "Smith"

};

Добавим свойство объекта fullName для полного имени – "John Smith". Реализуем его при помощи аксессора:

let user = {

name: "John",

surname: "Smith",

get fullName() {

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

}

};

alert(user.fullName); // John Smith

Снаружи свойство-аксессор выглядит как обычное свойство. В этом и заключается смысл свойств-аксессоров. user.fullName  не вызывается  как функция, а читается как обычное свойство: геттер сам вернет нужное значение.

На данный момент в примере fullName имеет только геттер. Если попытаться присвоить значение свойству user.fullName, то это вызовет ошибку:

let user = {

get fullName() {

return `...`;

}

};

user.fullName = "Тест"; // Ошибка

Давайте исправим это, добавив сеттер для user.fullName:

let user = {

name: "John",

surname: "Smith",

get fullName() {

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

},

set fullName(value) {

[this.name, this.surname] = value.split(" ");

}

};

user.fullName = "Alice Cooper";

alert(user.name); // Alice

alert(user.surname); // Cooper

В итоге получим «виртуальное» свойство fullName. Его можно прочитать и изменить, но по факту его не существует.

При попытке удалить свойство-аксессор оператором delete будет ошибка.

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 можно передать дескриптор с использованием get и set:

let user = {

name: "John",

surname: "Smith"

};

Object.defineProperty(user, 'fullName', {

get() {

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

},

set(value) {

[this.name, this.surname] = value.split(" ");

}

});

alert(user.fullName); // John Smith

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

Ещё раз заметим, что свойство объекта может быть только свойством-аксессором (с методами get/set) или свойством-данных (со значением value). При попытке указать и get и value в одном дескрипторе будет ошибка:

// Error: Invalid property descriptor.

Object.defineProperty({}, 'prop', {

get() {

return 1

},

value: 2

});

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

let user = {

get name() {

return this.\_name;

},

set name(value) {

if (value.length < 4) {

alert("Имя слишком короткое, должно быть более 4 символов");

return;

}

this.\_name = value;

}

};

user.name = "Pete";

alert(user.name); // Pete

user.name = ""; // Имя слишком короткое...

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 извне пределов объекта.

Аксессоры позволяют в любой момент взять «обычное» свойство и изменить его поведение, поменяв на геттер и сеттер. Например, представим, что реализован объект user, с использованием свойств-данных имя name и возраст age:

function User(name, age) {

this.name = name;

this.age = age;

}

let john = new User("John", 25);

alert( john.age ); // 25

Но со временем взамен возраста age можно хранить дату рождения birthday, потому что так более точно и удобно:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

}

let john = new User("John", new Date(1992, 6, 1));

Чтобы не менять весь старый код, который использует свойство age можно добавить геттер для age:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

Object.defineProperty(this, "age", {

get() {

let todayYear = new Date().getFullYear();

return todayYear - this.birthday.getFullYear();

}

});

}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday );

alert( john.age );

Теперь и старый код работает, и появилось полезное дополнительное свойство.

1. **Привязка контекста.**

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

**[Потеря «this»](https://learn.javascript.ru/bind" \l "poterya-this)**

Примеры потери this рассматривались ранее. Как только метод передается отдельно от объекта – this теряется. Вот как это может произойти с setTimeout:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

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

При запуске этого кода видно, что вызов this.firstName возвращает не «Вася», а undefined. Это произошло потому, что setTimeout получил функцию sayHi, отдельно от объекта user (именно здесь функция и потеряла контекст). То есть последняя строка может быть переписана как:

let f = user.sayHi;

setTimeout(f, 1000); // контекст user потеряли

Метод setTimeout в браузере имеет особенность: он устанавливает this=window для вызова функции (в Node.js thisстановится объектом таймера, но здесь это не имеет значения). Таким образом, для this.firstName он пытается получить window.firstName, которого не существует. В других подобных случаях обычно this просто становится undefined.

Задача состоит в том, что надо передать метод объекта куда-то ещё (в этом конкретном случае – в планировщик), где он будет вызван. Нужно, чтобы он был вызван в правильном контексте.

Есть несколько решений этой задачи. Одно из них – [сделать функцию-обёртку](https://learn.javascript.ru/bind" \l "reshenie-1-sdelat-funktsiyu-obyortku), обернуть вызов в анонимную функцию, создав замыкание:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

setTimeout(function() {

user.sayHi(); // Привет, Вася!

}, 1000);

Теперь код работает корректно, так как объект user достаётся из замыкания, а затем вызывается его метод sayHi. То же самое, только короче:

setTimeout(() => user.sayHi(), 1000); // Привет, Вася!

Теперь в коде появилась небольшая уязвимость: до момента срабатывания setTimeout (задержка составляет целую секунду) в переменную user может быть записано другое значение. Тогда вызов будет совсем не тот:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

setTimeout(() => user.sayHi(), 1000);

user = { sayHi() { alert("Другой пользователь в 'setTimeout'!"); } };

Второе решение описанной выше задачи гарантирует, что такого не случится. Оно заключается в привязке [контекста с помощью bind](https://learn.javascript.ru/bind" \l "reshenie-2-privyazat-kontekst-s-pomoschyu-bind). В современном JavaScript у функций есть встроенный метод [bind](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/bind), который позволяет зафиксировать this. Базовый синтаксис bind:

let boundFunc = func.bind(context);

Результатом вызова func.bind(context) является особый «экзотический объект» (термин взят из спецификации), который вызывается как функция и прозрачно передает вызов в func, при этом устанавливая this=context. Другими словами, вызов boundFunc подобен вызову func с фиксированным this. Например, в коде ниже funcUser передает вызов в func, фиксируя this=user:

let user = {

firstName: "Вася"

};

function func() {

alert(this.firstName);

}

let funcUser = func.bind(user);

funcUser(); // Вася

Здесь func.bind(user) – это «связанный вариант» func, с фиксированным this=user. Все аргументы передаются исходному методу func «как есть», например:

let user = {

firstName: "Вася"

};

function func(phrase) {

alert(phrase + ', ' + this.firstName);

}

// привязка this к user

let funcUser = func.bind(user);

funcUser("Привет"); // Привет, Вася

Теперь давайте попробуем с методом объекта:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

let sayHi = user.sayHi.bind(user); // (\*)

sayHi(); // Привет, Вася!

setTimeout(sayHi, 1000); // Привет, Вася!

В строке (\*) метод user.sayHi привязываем к user. Теперь SayHi – это «связанная» функция, которая может быть вызвана отдельно или передана в setTimeout (контекст всегда будет правильным). Здесь можно видеть, что bind исправляет только this, а аргументы передаются «как есть»:

let user = {

firstName: "Вася",

say(phrase) {

alert(`${phrase}, ${this.firstName}!`);

}

};

let say = user.say.bind(user);

say("Привет"); // Привет, Вася

say("Пока"); // Пока, Вася

До сих пор речь шла только о привязывании this. Можно 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], ...);

Это позволяет привязать контекст this и начальные аргументы функции. Например, у нас есть функция умножения mul(a, b):

function mul(a, b) {

return a \* b;

}

Воспользуемся bind, чтобы создать функцию double на её основе:

function mul(a, b) {

return a \* b;

}

let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6

alert( double(4) ); // = mul(2, 4) = 8

alert( double(5) ); // = mul(2, 5) = 10

Вызов mul.bind(null, 2) создаёт новую функцию double, которая передаёт вызов mul, фиксируя null как контекст и 2 – как первый аргумент. Следующие аргументы передаются «как есть». Это называется [частичное применение](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5) – создаётся новая функция, которая фиксирует некоторые из существующих параметров.

Обратите внимание, что в данном случае не используется this. Но для bind это обязательный параметр, так что надо передать туда что-нибудь вроде null. В следующем коде функция triple умножает значение на три:

function mul(a, b) {

return a \* b;

}

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9

alert( triple(4) ); // = mul(3, 4) = 12

alert( triple(5) ); // = mul(3, 5) = 15

Польза от частично применённой функции в том, что можно создать независимую функцию с понятным названием (double, triple). Можно использовать её и не передавать каждый раз первый аргумент, т.к. он зафиксирован с помощью bind.

В других случаях частичное применение полезно, когда есть очень общая функция и для удобства надо создать её частный вариант. Например, есть функция send(from, to, text) и позже может возникнуть необходимость внутри объекта user использовать её частный вариант: sendTo(to, text), который отправляет текст от имени текущего пользователя.

Возможна ситуация, когда надо зафиксировать некоторые аргументы, но не контекст this. Например, для метода объекта. Встроенный bind не позволяет этого. Нельзя просто опустить контекст и перейти к аргументам. Но можно создать вспомогательную функцию partial, которая привязывает только аргументы. Вот так:

function partial(func, ...argsBound) {

return function(...args) { // (\*)

return func.call(this, ...argsBound, ...args);

}

}

let user = {

firstName: "John",

say(time, phrase) {

alert(`[${time}] ${this.firstName}: ${phrase}!`);

}

};

user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello"); // [10:00] John: Hello!

Результатом вызова partial(func[, arg1, arg2...]) будет обёртка (\*), которая вызывает func с:

* Тем же this, который она получает (для вызова user.sayNow – это будет user)
* Затем передаёт ей ...argsBound – аргументы из вызова partial ("10:00")
* Затем передаёт ей ...args – аргументы, полученные обёрткой ("Hello")

**Пятая тема**

1. **[Прототипное наследование](https://learn.javascript.ru/prototype-inheritance)**

В программировании часто возникает необходимость что-то расширить. Например, есть объект user со своими свойствами и методами, надо создать объекты admin и guest как его слегка изменённые варианты. Хотелось 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.

**Свойство [[[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».

Если при чтении свойство из object отсутствует, JavaScript автоматически берет его из прототипа. В программировании такой механизм называется *прототипным наследованием*. Многие возможности языка и техники программирования основываются на нем.

Свойство [[Prototype]] является внутренним и скрытым, но есть много способов задать его. Одним из них является использование \_\_proto\_\_, например так:

let animal = {

eats: true

};

let rabbit = {

jumps: true

};

rabbit.\_\_proto\_\_ = animal;

 Свойство \_\_proto\_\_ – не то же самое, что [[Prototype]]. Это геттер/сеттер для него. Он существует по историческим причинам, в современном языке его заменяют функции Object.getPrototypeOf/Object.setPrototypeOf, которые также получают/устанавливают прототип. По спецификации \_\_proto\_\_ должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его.

В примере ниже осуществляется поиск свойства в rabbit, а оно отсутствует, и JavaScript автоматически берет его из animal:

let animal = {

eats: true

};

let rabbit = {

jumps: true

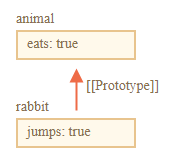
};

rabbit.\_\_proto\_\_ = animal; // (\*)

alert( rabbit.eats ); // true (\*\*)

alert( rabbit.jumps ); // true

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 её в animal (смотрите снизу вверх):



Здесь можно сказать, что animal является прототипом rabbit или rabbit прототипно наследует от animal. Так что если у animal много полезных свойств и методов, то они автоматически становятся доступными у rabbit. Такие свойства называются *унаследованными*. Например, есть метод в animal, он может быть вызван на rabbit:

let animal = {

eats: true,

walk() {

alert("Animal walk");

}

};

let rabbit = {

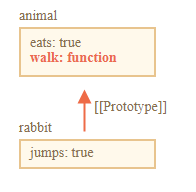
jumps: true,

\_\_proto\_\_: animal

};

rabbit.walk(); // Animal walk

Метод автоматически берётся из прототипа:



Цепочка прототипов может быть длиннее:

let animal = {

eats: true,

walk() {

alert("Animal walk");

}

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

let longEar = {

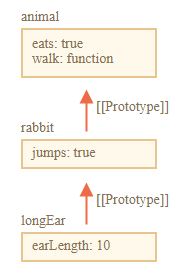
earLength: 10,

\_\_proto\_\_: rabbit

};

longEar.walk(); // Animal walk

alert(longEar.jumps); // true (для rabbit)



Есть только два ограничения:

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

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

Прототип используется только для чтения свойств. Операции записи/удаления работают напрямую с объектом. В приведённом ниже примере присваивается rabbit собственный метод walk:

let animal = {

eats: true,

walk() {

/\* ... \*/

}

};

let rabbit = {

\_\_proto\_\_: animal

};

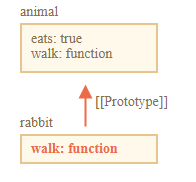
rabbit.walk = function() {

alert("Rabbit! Bounce-bounce!");

};

rabbit.walk(); // Rabbit! Bounce-bounce!

Теперь вызов rabbit.walk() находит метод непосредственно в объекте и выполняет его, не используя прототип:



Свойства-акссессоры – исключение, так как запись в него обрабатывается функцией-сеттером. То есть, это, фактически, вызов функции. По этой причине admin.fullName работает корректно в приведённом ниже коде:

let user = {

name: "John",

surname: "Smith",

set fullName(value) {

[this.name, this.surname] = value.split(" ");

},

get fullName() {

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

}

};

let admin = {

\_\_proto\_\_: user,

isAdmin: true

};

alert(admin.fullName); // John Smith (\*)

// срабатывает сеттер!

admin.fullName = "Alice Cooper"; // (\*\*)

Здесь в строке (\*) свойство admin.fullName имеет геттер в прототипе user, поэтому вызывается он. В строке (\*\*) свойство также имеет сеттер в прототипе, который и будет вызван.

**[Значение «this»](https://learn.javascript.ru/prototype-inheritance" \l "znachenie-this)**

Прототипы никак не влияют на this. Неважно, где находится метод: в объекте или его прототипе. При вызове метода this – всегда объект перед точкой. Таким образом, вызов сеттера admin.fullName в качестве this использует admin, а не user.

Это на самом деле очень важная деталь, потому что может быть большой объект со множеством методов, от которого можно наследовать. Затем наследущие объекты могут вызывать его методы, но они будут изменять состояние этих объектов, а не большого. Например, здесь animal представляет собой «хранилище методов», и rabbit использует его. Вызов rabbit.sleep() устанавливает this.isSleeping для объекта rabbit:

let animal = {

walk() {

if (!this.isSleeping) {

alert(`I walk`);

}

},

sleep() {

this.isSleeping = true;

}

};

let rabbit = {

name: "White Rabbit",

\_\_proto\_\_: animal

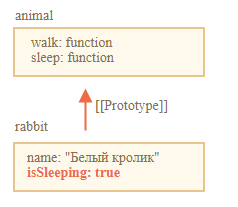
};

rabbit.sleep();

alert(rabbit.isSleeping); // true

alert(animal.isSleeping); // undefined (нет такого свойства в прототипе)

Картинка с результатом:



Если бы были другие объекты, такие как bird, snake и т.д., унаследованные от animal, они также получили бы доступ к методам animal. Но this при вызове каждого метода будет соответствовать объекту, на котором происходит вызов (перед точкой), а не animal. Поэтому, когда записываются данные в this, они сохраняются в этих объектах. В результате методы являются общими, а состояние объекта — нет.

**[Цикл for…in](https://learn.javascript.ru/prototype-inheritance" \l "tsikl-for-in)**

Цикл for..in проходит не только по собственным, но и по унаследованным свойствам объекта. Например:

let animal = {

eats: true

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

alert(Object.keys(rabbit)); // jumps

for(let prop in rabbit) alert(prop); // jumps, then eats

Если унаследованные свойства не нужны, то можно отфильтровать их при помощи встроенного метода [obj.hasOwnProperty(key)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty): он возвращает true, если у obj есть собственное, не унаследованное, свойство с именем key. Пример такой фильтрации:

let animal = {

eats: true

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

for(let prop in rabbit) {

let isOwn = rabbit.hasOwnProperty(prop);

if (isOwn) {

alert(`Our: ${prop}`); // Our: jumps

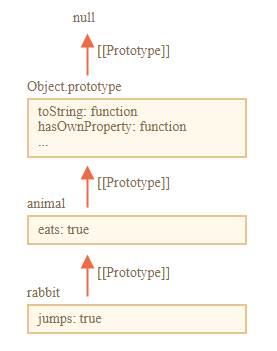
} else {

alert(`Inherited: ${prop}`); // Inherited: eats

}

}

В этом примере цепочка наследования выглядит так: rabbit наследует от animal, который наследует от Object.prototype (так как animal – литеральный объект {...}, это по умолчанию), а затем null на самом верху:



Стоит отметить следующее: метод rabbit.hasOwnProperty явно не определен. Если посмотреть на цепочку прототипов, то видно, что он берётся из Object.prototype.hasOwnProperty. То есть, он унаследован, но не появляется в цикле for..in, в отличие от eats и jumps. Дело в том, что это свойство не перечислимо. То есть, у него внутренний флаг enumerable стоит false, как и у других свойств Object.prototype. Поэтому оно и не появляется в цикле.

Почти все методы, получающие ключи/значения, такие как Object.keys, Object.values и другие – игнорируют унаследованные свойства. Они учитывают только свойства самого объекта, не его прототипа.

1. **[F.prototype](https://learn.javascript.ru/function-prototype)**

Как известно, новые объекты могут быть созданы с помощью функции-конструктора, new F(). Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.

JavaScript использовал прототипное наследование с момента своего появления. Это одна из основных особенностей языка.

Но раньше, прямого доступа к прототипу объекта не было. Надёжно работало только свойство "prototype" функции-конструктора. Поэтому оно используется во многих скриптах. Обратите внимание, что F.prototype означает обычное свойство с именем "prototype" для F. Это ещё не «прототип объекта», а обычное свойство F с таким именем. Приведём пример:

let animal = {

eats: true

};

function Rabbit(name) {

this.name = name;

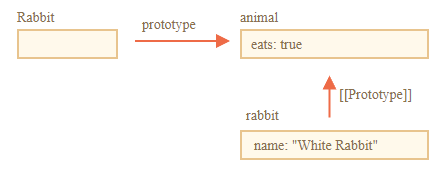
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); // rabbit.\_\_proto\_\_ == animal

alert( rabbit.eats ); // true

Установка Rabbit.prototype = animal буквально говорит интерпретатору следующее: "При создании объекта через new Rabbit() запиши ему animal в [[Prototype]]". Результат будет выглядеть так:



На изображении: "prototype" – горизонтальная стрелка, обозначающая обычное свойство для "F", а [[Prototype]]– вертикальная, обозначающая наследование rabbit от animal.

F.prototype используется только при вызове new F() и присваивается в качестве свойства [[Prototype]]нового объекта. После этого F.prototype и новый объект ничего не связывает. После создания F.prototype может измениться, и новые объекты, созданные с помощью new F(), будут иметь другой объект в качестве [[Prototype]], но уже существующие объекты сохранят старый.

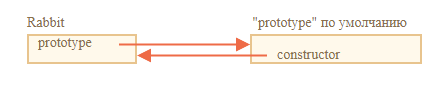
У каждой функции по умолчанию уже есть свойство "prototype". По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор. Вот такой:

function Rabbit() {}

/\* прототип по умолчанию

Rabbit.prototype = { constructor: Rabbit };

\*/



Проверим это:

function Rabbit() {}

// по умолчанию:

// Rabbit.prototype = { constructor: Rabbit }

alert( Rabbit.prototype.constructor == Rabbit ); // true

Соответственно, если ничего не меняется, то свойство constructor будет доступно всем кроликам через [[Prototype]]:

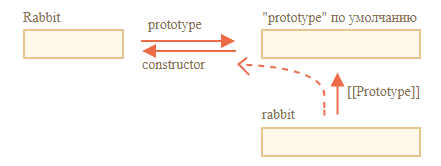
function Rabbit() {}

// по умолчанию:

// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // наследует от {constructor: Rabbit}

alert(rabbit.constructor == Rabbit); // true (свойство получено из прототипа)



Можно использовать свойство constructor существующего объекта для создания нового. Пример:

function Rabbit(name) {

this.name = name;

alert(name);

}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");

Это удобно, когда есть объект, но неизвестно какой конструктор использовался для его создания (например, он был взят из сторонней библиотеки), а необходимо создать ещё один такой объект.

Самое важное о свойстве "constructor" это то, что JavaScript сам по себе не гарантирует правильное значение свойства "constructor". Оно является свойством по умолчанию в "prototype" у функций, но что будет с ним позже – зависит только от разработчика. В частности, если заменить прототип по умолчанию на другой объект, свойства "constructor" в нём не будет. Например:

function Rabbit() {}

Rabbit.prototype = {

jumps: true

};

let rabbit = new Rabbit();

alert(rabbit.constructor === Rabbit); // false

Таким образом, чтобы сохранить верное свойство "constructor", надо добавлять/удалять/изменять свойства у прототипа по умолчанию вместо того, чтобы перезаписывать его целиком. В примере ниже Rabbit.prototype не перезаписывается полностью, а добавляется к нему свойство. Прототип по умолчанию сохраняется, и сохраняется доступ к Rabbit.prototype.constructor.

function Rabbit() {}

Rabbit.prototype.jumps = true

Или можно заново создать свойство constructor:

Rabbit.prototype = {

jumps: true,

constructor: Rabbit

};

1. **[Встроенные прототипы](https://learn.javascript.ru/native-prototypes)**

Свойство "prototype" широко используется в внутри самого языка JavaScript. Все встроенные функции-конструкторы используют его.

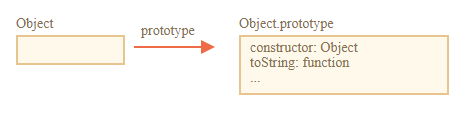
**[Object.prototype](https://learn.javascript.ru/native-prototypes" \l "object-prototype)**

Выведем пустой объект:

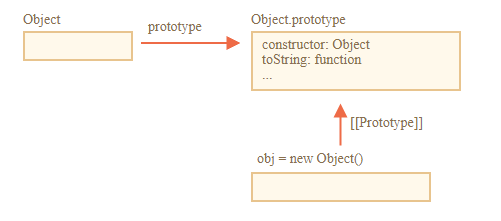
let obj = {};

alert( obj ); // "[object Object]" ?

В рассматриваемом примере нету кода, который генерирует строку "[object Object]". Понятно, что это встроенный метод toString, но явно не видно, где он объявлен,  ведь obj  пуст. Дело в том, что краткая нотация obj = {} это то же самое, что и obj = new Object(), где Object – встроенная функция-конструктор для объектов с собственным свойством prototype, который ссылается на огромный объект с методом toString и другими. Вот что происходит:



Когда вызывается new Object() (или создаётся объект с помощью литерала {...}), свойство [[Prototype]] этого объекта устанавливается на Object.prototype по правилам, которые рассматривались в предыдущем вопросе:



Таким образом, когда вызывается obj.toString(), метод берётся из Object.prototype. Можно проверить это так:

let obj = {};

alert(obj.\_\_proto\_\_ === Object.prototype); // true

// obj.toString === obj.\_\_proto\_\_.toString == Object.prototype.toString

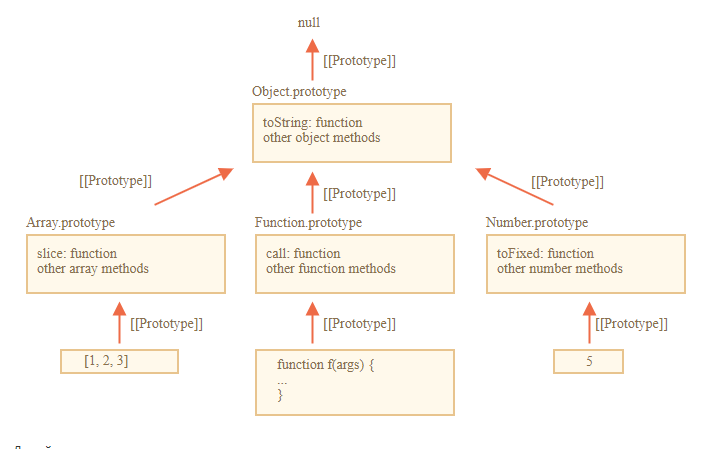
Обратим внимание, что выше Object.prototype по цепочке прототипов больше нет [[Prototype]]:

alert(Object.prototype.\_\_proto\_\_); // null

Другие встроенные объекты, такие как Array, Date, Function и другие, также хранят свои методы в прототипах. Например, при создании массива [1, 2, 3] внутренне используется конструктор массива Array. Поэтому прототипом массива становится Array.prototype, предоставляя ему свои методы. Это позволяет эффективно использовать память.

Согласно спецификации, наверху иерархии встроенных прототипов находится Object.prototype. Поэтому иногда говорят, что «всё наследует от объектов».

Вот более полная картина (для 3 встроенных объектов):



Проверим прототипы:

let arr = [1, 2, 3];

// наследует от Array.prototype?

alert( arr.\_\_proto\_\_ === Array.prototype ); // true

// затем от Object.prototype?

alert( arr.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype ); // true

// и null на вершине иерархии

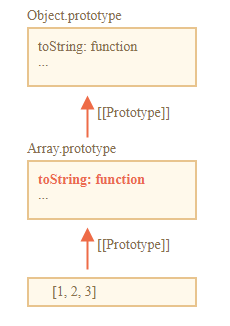
alert( arr.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ ); // null

Некоторые методы в прототипах могут пересекаться, например, у Array.prototype есть свой метод toString, который выводит элементы массива через запятую:

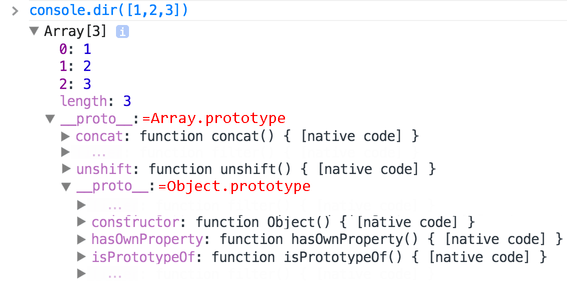
let arr = [1, 2, 3]

alert(arr); // 1,2,3 <-- результат Array.prototype.toString

Как известно, у Object.prototype есть свой метод toString, но так как Array.prototype ближе в цепочке прототипов, то берётся именно вариант для массивов:



В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования (возможно, потребуется использовать console.dir для встроенных объектов):



Другие встроенные объекты устроены аналогично. Даже функции – объекты встроенного конструктора Function, и все их методы (call/apply и другие) берутся из Function.prototype. Также у функций есть свой метод toString.

function f() {}

alert(f.\_\_proto\_\_ == Function.prototype); // true

alert(f.\_\_proto\_\_.\_\_proto\_\_ == Object.prototype); // true, наследует от Object

**[Примитивы](https://learn.javascript.ru/native-prototypes" \l "primitivy)**

Самое сложное происходит со строкам, числами и булевым типом. Как известно, они не объекты. Но если попытаться получить доступ к их свойствам, тогда будет создан временный объект-обёртка с использованием встроенных конструкторов String, Number, Boolean, который предоставит методы и после чего исчезнет. Эти объекты создаются невидимо для нас, и большая часть движков оптимизирует этот процесс, но спецификация описывает это именно таким образом. Методы этих объектов также находятся в прототипах, доступных как String.prototype, Number.prototype и Boolean.prototype.

Специальные значения null и undefined не имеют объектов-обёрток, так что методы и свойства им недоступны. Также у них нет соответствующих прототипов.

**[Изменение встроенных прототипов](https://learn.javascript.ru/native-prototypes" \l "native-prototype-change)**

Встроенные прототипы можно изменять. Например, если добавить метод к String.prototype, метод становится доступен для всех строк:

String.prototype.show = function() {

alert(this);

};

"BOOM!".show(); // BOOM!

В течение процесса разработки могут возникнуть идеи о новых встроенных методах, которые хотелось бы иметь и добавить их во встроенные прототипы. Это плохая идея. Прототипы глобальны, поэтому очень легко могут возникнуть конфликты. Если две библиотеки добавляют метод String.prototype.show, то одна из них перепишет метод другой.

В современном программировании есть только один случай, в котором одобряется изменение встроенных прототипов. Это создание полифилов. Полифил – это термин, который означает замену метода, который существует в спецификации JavaScript, но он ещё не поддерживается текущим движком JavaScript. Тогда можно реализовать его и добавить его во встроенный прототип. Например:

if (!String.prototype.repeat) { // Если такого метода нет

// добавляем его в прототип

String.prototype.repeat = function(n) {

// повторить строку n раз

return new Array(n + 1).join(this);

};

}

alert( "La".repeat(3) ); // LaLaLa

**[Заимствование у прототипов](https://learn.javascript.ru/native-prototypes" \l "zaimstvovanie-u-prototipov)**

Ранее рассматривалось заимствовании методов. Это когда метод из одного объекта и копируется в другой. Некоторые методы встроенных прототипов часто одалживают. Например, если создать объект, похожий на массив (псевдомассив), можно скопировать некоторые методы из Array в этот объект. Пример:

let obj = {

0: "Hello",

1: "world!",

length: 2,

};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hello,world!

Это работает, потому что для внутреннего алгоритма встроенного метода join важна только корректность индексов и свойства length, он не проверяет является ли объект на самом деле массивом. И многие встроенные методы работают так же. Альтернативная возможность –можно унаследовать от массива, установив obj.\_\_proto\_\_ как Array.prototype, таким образом все методы Array станут автоматически доступны в obj. Но это будет невозможно, если obj уже наследует от другого объекта, ведь можно наследовать только от одного объекта одновременно.

Заимствование методов – гибкий способ, позволяющий смешивать функциональность разных объектов по необходимости.

1. **[Методы прототипов](https://learn.javascript.ru/prototype-methods)**

Ранее упоминалось, что существуют современные методы работы с прототипами. Свойство \_\_proto\_\_ считается устаревшим, и по стандарту должно поддерживаться только браузерами. Современные методы это:

* [Object.create(proto, [descriptors])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/create) – создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.
* [Object.getPrototypeOf(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf) – возвращает свойство [[Prototype]] объекта obj.
* [Object.setPrototypeOf(obj, proto)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf) – устанавливает свойство [[Prototype]] объекта obj как proto.

Эти методы нужно использовать вместо \_\_proto\_\_. Например:

let animal = {

eats: true

};

let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal);

Object.setPrototypeOf(rabbit, {});

У Object.create есть необязательный второй аргумент: дескрипторы свойств. Можно добавить дополнительное свойство новому объекту таким образом:

let animal = {

eats: true

};

let rabbit = Object.create(animal, {

jumps: {

value: true

}

});

alert(rabbit.jumps); // true

Также можно использовать Object.create для глубокого клонирования объекта, более мощного, чем копирование свойств в цикле for..in:

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

Такой вызов создаёт точную копию объекта obj, включая все свойства: перечисляемые и не перечисляемые, свойства, геттеры/сеттеры для свойств – и все это с правильным свойством [[Prototype]].

Технически, можно установить/получить [[Prototype]] в любое время. Но обычно прототип устанавливается только раз во время создания объекта, а после не меняется: rabbit наследует от animal, и это не изменится.

JavaScript движки хорошо оптимизированы для этого. Изменение прототипа «на лету» с помощью Object.setPrototypeOf или obj.\_\_proto\_\_= – очень медленная операция, которая ломает внутренние оптимизации для операций доступа к свойствам объекта. Так что лучше избегайте этого, кроме тех случаев, когда знаете, что делаете, либо скорость JavaScript для вас не имеет никакого значения.

Как известно, объекты можно использовать как ассоциативные массивы для хранения пар ключ/значение. Но если попробовать хранить созданные пользователями ключи (например, словари с пользовательским вводом), можно заметить интересный сбой: все ключи работают как ожидается, за исключением "\_\_proto\_\_". Например:

let obj = {};

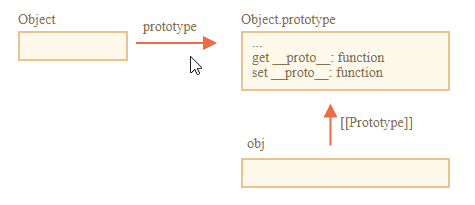
let key = prompt("What's the key?", "\_\_proto\_\_");

obj[key] = "some value";

alert(obj[key]); // [object Object], не "some value"

Если пользователь введёт \_\_proto\_\_, присвоение проигнорируется, так как свойство \_\_proto\_\_ должно быть либо объектом, либо null, а строка не может стать прототипом.

Свойство \_\_proto\_\_ – не обычное, а аксессор, заданный в Object.prototype:



Так что при чтении или установке obj.\_\_proto\_\_ вызывается соответствующий геттер/сеттер из прототипа obj, и именно он устанавливает/получает свойство [[Prototype]].

Поните, что \_\_proto\_\_ – это способ доступа к свойству [[Prototype]], это не само свойство [[Prototype]].

Теперь, если надо использовать объект как ассоциативный массив, можно сделать это следующим образом:

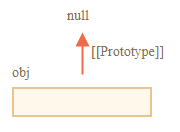
let obj = Object.create(null);

let key = prompt("What's the key?", "\_\_proto\_\_");

obj[key] = "some value";

alert(obj[key]); // "some value"

Object.create(null) создаёт пустой объект без прототипа ([[Prototype]] будет null):



Таким образом не будет унаследованного геттера/сеттера для \_\_proto\_\_. Теперь это свойство обрабатывается как обычное свойство, и приведённый выше пример работает правильно. Можно назвать такой объект «простейшим» или «чистым словарным объектом», потому что он ещё проще чем обычный объект {...}. Недостаток в том, что у таких объектов не будет встроенных методов объекта, таких как toString:

let obj = Object.create(null);

alert(obj); // Error (no toString)

Но обычно это нормально для ассоциативных массивов. Обратите внимание, что большая часть методов, связанных с объектами, имеют вид Object.something(...). К примеру, Object.keys(obj) не находятся в прототипе, так что они продолжат работать для таких объектов:

let chineseDictionary = Object.create(null);

chineseDictionary.hello = "你好";

chineseDictionary.bye = "再见";

alert(Object.keys(chineseDictionary)); // hello,bye

1. **[Классы](https://learn.javascript.ru/class)**

В объектно-ориентированном программировании класс – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

На практике часто надо создавать много объектов одного вида, например пользователей, товары или что-то еще. Как известно, 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:

class MyClass {

// методы класса

constructor() { ... }

method1() { ... }

method2() { ... }

method3() { ... }

...

}

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами. При этом автоматически вызывается метод constructor(), в нём можно инициализовать объект. Например:

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

let user = new User("Иван");

user.sayHi();

Когда вызывается new User("Иван"):

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

Затем можно вызывать методы объекта, такие как user.sayHi().

Методы в классе не разделяются запятой. Это приводит к синтаксической ошибке.

В JavaScript класс – это разновидность функции. Рассмотрим пример:

class User {

constructor(name) { this.name = name; }

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

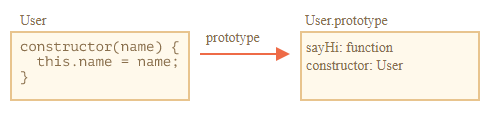
}

alert(typeof User); // function

Вот что на самом деле делает конструкция class User {...}:

1. Создает функцию с именем User, которая становится результатом объявления класса. Код функции берется из метода constructor (она будет пустой, если такого метода нет).
2. Сохраняет все методы, такие как sayHi, в User.prototype.

Затем, при вызове метода на новых объектах new User, он возьмётся из прототипа. Таким образом, объект new User имеет доступ к методам класса. На картинке показан результат объявления class User:



Как видно из кода ниже, класс – это функция или, если точнее, это метод constructor, методы находятся в User.prototype.

class User {

constructor(name) { this.name = name; }

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

}

alert(typeof User); // function

alert(User === User.prototype.constructor); // true

alert(User.prototype.sayHi); // alert(this.name);

alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi

Иногда говорят, что class – это просто «синтаксический сахар» в JavaScript (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что можно сделать все то же самое без конструкции class. Например:

// перепишем класс User с помощью функций

// 1. Создаём функцию constructor

function User(name) {

this.name = name;

}

// 2. Добавляем метод в прототип

User.prototype.sayHi = function() {

alert(this.name);

};

let user = new User("Иван");

user.sayHi();

Результат этого кода очень похож на предыдущий. Поэтому, class можно считать синтаксическим сахаром для определения конструктора вместе с методами прототипа. Однако есть важные отличия:

1. Во-первых, функция, созданная с помощью class, помечена специальным внутренним свойством [[FunctionKind]]:"classConstructor". Поэтому это не совсем то же самое, что создавать её вручную.

В отличие от обычных функций, конструктор класса не может быть вызван без new:

class User {

constructor() {}

}

alert(typeof User); // function

User(); // Error: Class constructor User cannot be invoked without 'new'

Кроме того, строковое представление конструктора класса в большинстве движков JavaScript начинается с «class …».

class User {

constructor() {}

}

alert(User); // class User { ... }

1. Методы класса являются неперечислимыми. Определение класса устанавливает флаг enumerable в false для всех методов в "prototype".
2. Классы всегда используют use strict. Весь код внутри класса автоматически находится в строгом режиме.

**[Class Expression](https://learn.javascript.ru/class" \l "class-expression)**

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д. Пример Class Expression (по аналогии с Function Expression):

let User = class {

sayHi() {

alert("Привет");

}

};

Как и Named Function Expressions, выражения классов могут иметь имя, которое видно только внутри класса. Если у Class Expression есть имя, то оно видно только внутри класса:

let User = class MyClass {

sayHi() {

alert(MyClass);

}

};

new User().sayHi(); // работает

alert(MyClass); // ошибка

Можно динамически создавать классы «по-запросу»:

function makeClass(phrase) {

// объявляем класс и возвращаем его

return class {

sayHi() {

alert(phrase);

};

};

}

// Создаем новый класс

let User = makeClass("Привет");

new User().sayHi(); // Привет

Как и в литеральных объектах, в классах можно объявлять генераторы, вычисляемые свойства, геттеры/сеттеры и т.д. Пример user.name, реализованного с использованием get/set:

class User {

constructor(name) {

// вызывает сеттер

this.name = name;

}

get name() {

return this.\_name;

}

set name(value) {

if (value.length < 4) {

alert("Имя слишком короткое.");

return;

}

this.\_name = value;

}

}

let user = new User("Иван");

alert(user.name); // Иван

user = new User(""); // Имя слишком короткое.

При объявлении класса геттеры/сеттеры создаются на User.prototype:

Object.defineProperties(User.prototype, {

name: {

get() {

return this.\_name

},

set(name) {

// ...

}

}

});

Пример с вычисляемым свойством в скобках [...]:

class User {

['say' + 'Hi']() {

alert("Привет");

}

}

new User().sayHi();

Для методов-генераторов добавьте перед именем \*.

**[Свойства классов](https://learn.javascript.ru/class" \l "svoystva-klassov)**

Свойства классов добавлены в язык недавно. Старым браузерам может понадобиться полифил. В приведённом выше примере у класса User были только методы. Добавим свойство:

class User {

name = "Аноним";

sayHi() {

alert(`Привет, ${this.name}!`);

}

}

new User().sayHi();

Свойство name не устанавливается в User.prototype. Вместо этого оно создаётся оператором new перед запуском конструктора, это именно свойство объекта.

1. **[Наследование классов](https://learn.javascript.ru/class-inheritance)**

Допустим, у нас есть два класса. Класс Animal:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed += speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

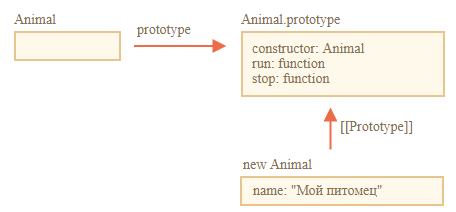
this.speed = 0;

alert(`${this.name} стоит.`);

}

}

let animal = new Animal("Мой питомец");



Класс Rabbit:

class Rabbit {

constructor(name) {

this.name = name;

}

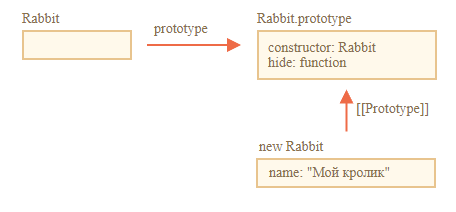
hide() {

alert(`${this.name} прячется!`);

}

}

let rabbit = new Rabbit("Мой кролик");



Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {..}. Ниже Rabbit наследует от Animal:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed += speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} стоит.`);

}

}

// Наследуем от Animal указывая "extends Animal"

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

}

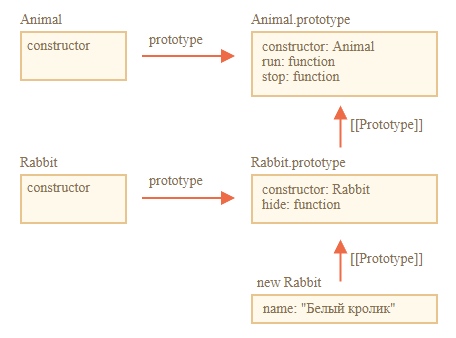
let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.hide(); // Белый кролик прячется!

Теперь код Rabbit стал короче, так как используется конструктор класса Animal по умолчанию и кролик может использовать метод run как и все животные.

На самом деле ключевое слово extends добавляет ссылку на [[Prototype]] из Rabbit.prototype в Animal.prototype:



Если метод не найден в Rabbit.prototype, JavaScript возьмёт его из Animal.prototype.

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

function f(phrase) {

return class {

sayHi() { alert(phrase) }

}

}

class User extends f("Привет") {}

new User().sayHi(); // Привет

Здесь class User наследует от результата вызова f("Привет"). Это может быть полезно для продвинутых приёмов проектирования, где можно 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)**

Сейчас Rabbit наследует от Animal метод stop, который устанавливает this.speed = 0. Если определить свой метод stop в классе Rabbit, то он будет использоваться взамен родительского:

class Rabbit extends Animal {

stop() {

// ...будет использован для rabbit.stop()

}

}

Обычно нет необходимости полностью заменять родительский метод, а только сделать новый на его основе, изменяя или расширяя его функциональность. Для этого надо определить новый метод, добавив нужный функционал, и вызывать родительский метод до/после или в процессе.

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

* super.method(...) вызывает родительский метод.
* super(...) вызывает родительский конструктор (работает только внутри нашего конструктора).

Пусть наш кролик автоматически прячется при остановке:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed += speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} стоит.`);

}

}

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

stop() {

super.stop(); // вызываем родительский метод stop

this.hide(); // и затем hide

}

}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.stop(); // Белый кролик стоит. Белый кролик прячется!

Теперь у класса Rabbit есть метод stop, который вызывает родительский super.stop() в процессе выполнения.

У стрелочных функций нет super. При обращении к super стрелочной функции он берётся из внешней функции:

class Rabbit extends Animal {

stop() {

setTimeout(() => super.stop(), 1000); // вызывает родительский stop после 1 секунды

}

}

В примере super в стрелочной функции тот же самый, что и в stop(), поэтому метод отрабатывает как и ожидается. Если указать здесь «обычную» функцию, была бы ошибка:

// Unexpected super

setTimeout(function() { super.stop() }, 1000);

Согласно [спецификации](https://tc39.github.io/ecma262/" \l "sec-runtime-semantics-classdefinitionevaluation), если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор. Например, у Rabbit нет своего конструктора.

class Rabbit extends Animal {

constructor(...args) {

super(...args);

}

}

Как видно, он просто вызывает конструктор родительского класса. Так будет происходить, пока не будет создан собственный конструктор. Добавим конструктор для Rabbit. Он будет устанавливать earLength в дополнение к name:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

this.speed = 0;

this.name = name;

this.earLength = earLength;

}

// ...

}

let rabbit = new Rabbit("Белый кролик", 10); // Error: this is not defined.

Ошибка возникла из-за того, что в классах-потомках конструктор обязан вызывать super(...) и делать это перед использованием this. Дело в том, что в JavaScript существует различие между «функцией-конструктором наследующего класса» и всеми остальными. В наследующем классе соответствующая функция-конструктор помечена специальным внутренним свойством [[ConstructorKind]]:"derived". Разница в следующем:

* Когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его this.
* Когда запускается конструктор унаследованного класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.

Поэтому, если создать собственный конструктор, то надо вызвать super, в противном случае объект для this не будет создан, и возникнет ошибка. Чтобы конструктор Rabbit работал, он должен вызвать super() до того, как использовать this, чтобы не было ошибки:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

super(name);

this.earLength = earLength;

}

// ...

}

let rabbit = new Rabbit("Белый кролик", 10);

alert(rabbit.name); // Белый кролик

alert(rabbit.earLength); // 10

**[[[HomeObject]]](https://learn.javascript.ru/class-inheritance" \l "homeobject)**

В JavaScript для функций добавлено специальное внутреннее свойство: [[HomeObject]]. Когда функция объявлена как метод внутри класса или объекта, её свойство [[HomeObject]] становится равно этому объекту. Затем super использует его, чтобы получить прототип родителя и его методы. Посмотрим, как это работает – опять же, используя простые объекты:

let animal = {

name: "Животное",

eat() { // animal.eat.[[HomeObject]] == animal

alert(`${this.name} eats.`);

}

};

let rabbit = {

\_\_proto\_\_: animal,

name: "Кролик",

eat() { // rabbit.eat.[[HomeObject]] == rabbit

super.eat();

}

};

let longEar = {

\_\_proto\_\_: rabbit,

name: "Длинноух",

eat() { // longEar.eat.[[HomeObject]] == longEar

super.eat();

}

};

longEar.eat(); // Длинноух ест.

Это работает как задумано благодаря [[HomeObject]]. Метод, такой как longEar.eat, знает свой [[HomeObject]] и получает метод родителя из его прототипа. Вообще без использования this.

**[Методы не «свободны»](https://learn.javascript.ru/class-inheritance" \l "metody-ne-svobodny)**

Как известно, функции в JavaScript «свободны», не привязаны к объектам. Их можно копировать между объектами и вызывать с любым this. Но само существование [[HomeObject]] нарушает этот принцип, так как методы запоминают свои объекты. [[HomeObject]] нельзя изменить, эта связь – навсегда. Единственное место в языке, где используется [[HomeObject]] – это super. Поэтому если метод не использует super, то все ещё можно считать его свободным и копировать между объектами. А вот если super в коде есть, то возможны побочные эффекты. Вот пример неверного результата super после копирования:

let animal = {

sayHi() {

console.log(`Я животное`);

}

};

// rabbit наследует от animal

let rabbit = {

\_\_proto\_\_: animal,

sayHi() {

super.sayHi();

}

};

let plant = {

sayHi() {

console.log("Я растение");

}

};

// tree наследует от plant

let tree = {

\_\_proto\_\_: plant,

sayHi: rabbit.sayHi // (\*)

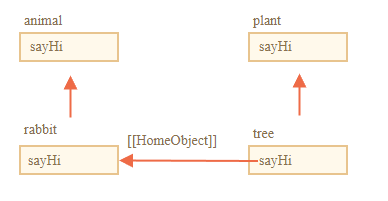
};

tree.sayHi(); // Я животное

Вызов tree.sayHi() показывает «Я животное». Определённо неверно. Причина проста:

* В строке (\*), метод tree.sayHi скопирован из rabbit.
* Его [[HomeObject]] – это rabbit, ведь он был создан в rabbit. Свойство [[HomeObject]] никогда не меняется.
* В коде tree.sayHi() есть вызов super.sayHi(). Он идёт вверх от rabbit и берёт метод из animal.

Вот диаграмма происходящего:



Свойство [[HomeObject]] определено для методов как классов, так и обычных объектов. Но для объектов методы должны быть объявлены именно как method(), а не "method: function()". В приведённом ниже примере используется синтаксис не метода, свойства-функции. Поэтому у него нет [[HomeObject]], и наследование не работает:

let animal = {

eat: function() { // должен быть короткий синтаксис: eat() {...}

// ...

}

};

let rabbit = {

\_\_proto\_\_: animal,

eat: function() {

super.eat();

}

};

rabbit.eat(); // Ошибка вызова super

1. **[Статические свойства и методы](https://learn.javascript.ru/static-properties-methods)**

Можно присвоить метод самой функции-классу, а не её "prototype". Такие методы называются статическими. В классе такие методы обозначаются ключевым словом static, например:

class User {

static staticMethod() {

alert(this === User);

}

}

User.staticMethod(); // true

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

class User() { }

User.staticMethod = function() {

alert(this === User);

};

Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»).

Обычно статические методы используются для реализации функций, принадлежащих классу, но не к каким-то конкретным его объектам. Например, есть объекты статей Article, и нужна функция для их сравнения. Естественное решение – сделать для этого метод Article.compare:

class Article {

constructor(title, date) {

this.title = title;

this.date = date;

}

static compare(articleA, articleB) {

return articleA.date - articleB.date;

}

}

let articles = [

new Article("HTML", new Date(2019, 1, 1)),

new Article("CSS", new Date(2019, 0, 1)),

new Article("JavaScript", new Date(2019, 11, 1))

];

articles.sort(Article.compare);

alert( articles[0].title ); // CSS

Здесь метод Article.compare стоит «над» статьями, как способ их сравнения. Это метод не отдельной статьи, а всего класса. Другим примером может быть так называемый «фабричный» метод. Представим, что нужно создавать статьи различными способами:

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

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

class Article {

constructor(title, date) {

this.title = title;

this.date = date;

}

static createTodays() {

// this = Article

return new this("Сегодняшний дайджест", new Date());

}

}

let article = Article.createTodays();

alert( article.title ); // Сегодняшний дайджест

Теперь каждый раз, когда 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, предположим, что Article - это специальный класс для управления статьями статический метод для удаления статьи:

Article.remove({id: 12345});

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

Эта возможность была добавлена в язык недавно. Примеры работают в последнем Chrome. Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:

class Article {

static publisher = "Иван Иванов";

}

alert( Article.publisher ); // Иван Иванов

Это то же самое, что и прямое присваивание Article:

Article.publisher = "Иван Иванов";

Статические свойства и методы наследуются. Например, метод Animal.compare в коде ниже наследуется и доступен как Rabbit.compare:

class Animal {

constructor(name, speed) {

this.speed = speed;

this.name = name;

}

run(speed = 0) {

this.speed += speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

static compare(animalA, animalB) {

return animalA.speed - animalB.speed;

}

}

// Наследует от Animal

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

}

let rabbits = [

new Rabbit("Белый кролик", 10),

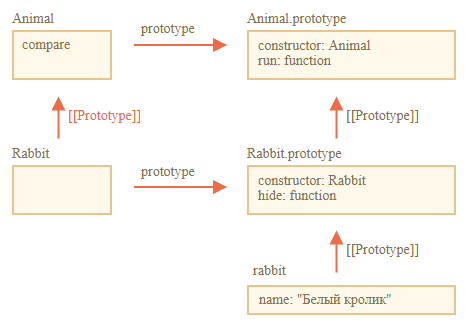
new Rabbit("Чёрный кролик", 5)

];

rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Чёрный кролик бежит со скоростью 5.

Можно вызвать Rabbit.compare, при этом будет вызван унаследованный Animal.compare. Это работает с использованием прототипов. Extends даёт Rabbit ссылку [[Prototype]] на Animal.



Так что Rabbit extends Animal создаёт две ссылки на прототип:

1. Функция Rabbit прототипно наследует от Animal function.
2. Rabbit.prototype прототипно наследует от Animal.prototype.

В результате наследование работает как для обычных, так и для статических методов:

class Animal {}

class Rabbit extends Animal {}

// для статики

alert(Rabbit.\_\_proto\_\_ === Animal); // true

// для обычных методов

alert(Rabbit.prototype.\_\_proto\_\_ === Animal.prototype);

1. **[Приватные и защищённые методы и свойства](https://learn.javascript.ru/private-protected-properties-methods)**

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

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

Внутренний интерфейс используется для работы объекта, его методы, свойства используют друг друга. Всё, что нужно для использования объекта, это знать его внешний интерфейс. Совершенно не обязательно знать, его внутреннюю структуру и логику.

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

* *Публичные*: доступны отовсюду. Они составляют внешний интерфейс. До этого момента в рассматриваемых примерах использовались 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 разрёшен доступ для наследующих классов) и также полезны для внутреннего интерфейса. В некотором смысле они более распространены, чем приватные, потому что обычно надо, 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.

Создадим простой класс для описания работы кофеварки:

class CoffeeMachine {

waterAmount = 0; // количество воды внутри

constructor(power) {

this.power = power;

alert( `Создана кофеварка, мощность: ${power}` );

}

}

// создаём кофеварку

let coffeeMachine = new CoffeeMachine(100);

// добавляем воды

coffeeMachine.waterAmount = 200;

Сейчас свойства waterAmount и power публичные. Можно легко получать и устанавливать им любое значение извне. Изменим свойство waterAmount на защищённое, чтобы иметь больше контроля над ним. Например, не надо, чтобы кто-либо устанавливал его ниже нуля.

Защищённые свойства обычно начинаются с префикса \_. Это не синтаксис языка: есть хорошо известное соглашение между программистами, что такие свойства и методы не должны быть доступны извне. Большинство программистов следуют этому соглашению. Пожтому свойство будет называться \_waterAmount:

class CoffeeMachine {

\_waterAmount = 0;

set waterAmount(value) {

if (value < 0) throw new Error("Отрицательное количество воды");

this.\_waterAmount = value;

}

get waterAmount() {

return this.\_waterAmount;

}

constructor(power) {

this.\_power = power;

}

}

// создаём новую кофеварку

let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды

coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды

Теперь доступ под контролем, поэтому указать воду ниже нуля не удалось.

Сделаем свойство power доступным только для чтения. Иногда нужно, чтобы свойство устанавливалось только при создании объекта и после этого никогда не изменялось. Это как раз требуется для кофеварки: мощность никогда не меняется. Для этого нужно создать только геттер, но не сеттер:

class CoffeeMachine {

// ...

constructor(power) {

this.\_power = power;

}

get power() {

return this.\_power;

}

}

let coffeeMachine = new CoffeeMachine(100);

alert(`Мощность: ${coffeeMachine.power}W`); // Мощность: 100W

coffeeMachine.power = 25; // Error (no setter)

Здесь использовался синтаксис геттеров/сеттеров. Но в большинстве случаев использование функций get.../set... предпочтительнее:

class CoffeeMachine {

\_waterAmount = 0;

setWaterAmount(value) {

if (value < 0) throw new Error("Отрицательное количество воды");

this.\_waterAmount = value;

}

getWaterAmount() {

return this.\_waterAmount;

}

}

new CoffeeMachine().setWaterAmount(100);

Это выглядит немного длиннее, но функции более гибкие. Они могут принимать несколько аргументов.

Если унаследовать class MegaMachine extends CoffeeMachine, ничто не помешает нам обращаться к this.\_waterAmount или this.\_power из методов нового класса. Таким образом защищённые методы, конечно же, наследуются. В отличие от приватных полей.

**[Приватное свойство «#waterLimit»](https://learn.javascript.ru/private-protected-properties-methods" \l "privatnoe-svoystvo-waterlimit)**

Поддержка приватных свойств и методов была добавлена в язык недавно. В движках JavaScript пока не поддерживается или поддерживается частично, нужен полифилл. Приватные свойства и методы должны начинаться со знака #. Они доступны только внутри класса. Например, в классе ниже есть приватное свойство #waterLimit и приватный метод #checkWater для проверки количества воды:

class CoffeeMachine {

#waterLimit = 200;

#checkWater(value) {

if (value < 0) throw new Error("Отрицательный уровень воды");

if (value > this.#waterLimit) throw new Error("Слишком много воды");

}

}

let coffeeMachine = new CoffeeMachine();

// снаружи нет доступа к приватным методам класса

coffeeMachine.#checkWater(); // Error

coffeeMachine.#waterLimit = 1000; // Error

На уровне языка # является специальным символом, который означает, что поле приватное. Нельзя получить к нему доступ извне или из наследуемых классов. Приватные поля не конфликтуют с публичными. Может быть два поля одновременно – приватное #waterAmount и публичное waterAmount.

Например, давайте сделаем аксессор waterAmount для #waterAmount:

class CoffeeMachine {

#waterAmount = 0;

get waterAmount() {

return this.#waterAmount;

}

set waterAmount(value) {

if (value < 0) throw new Error("Отрицательный уровень воды");

this.#waterAmount = value;

}

}

let machine = new CoffeeMachine();

machine.waterAmount = 100;

alert(machine.#waterAmount); // Error

В отличие от защищённых, функционал приватных полей обеспечивается самим языком. Но если унаследовать от CoffeeMachine, то не будет прямого доступа к #waterAmount, только через геттер/сеттер waterAmount:

class MegaCoffeeMachine extends CoffeeMachine() {

method() {

alert( this.#waterAmount ); // Error: can only access from CoffeeMachine

}

}

Во многих случаях такое ограничение неудобно. Если расширяется CoffeeMachine, то должен быть доступ к внутренним методам и свойствам. Поэтому защищённые свойства используются чаще, хоть они и не поддерживаются синтаксисом языка.

Как известно, обычно можно получить доступ к полям объекта с помощью this[name]:

class User {

...

sayHi() {

let fieldName = "name";

alert(`Hello, ${this[fieldName]}`);

}

}

С приватными свойствами такое невозможно: this['#name'] не работает. Это ограничение синтаксиса сделано для обеспечения приватности.

1. **[Расширение встроенных классов](https://learn.javascript.ru/extend-natives)**

От встроенных классов, таких как Array, Map и других, тоже можно наследовать. Например, в этом примере PowerArray наследуется от встроенного Array:

class PowerArray extends Array {

isEmpty() {

return this.length === 0;

}

}

let arr = new PowerArray(1, 2, 5, 10, 50);

alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);

alert(filteredArr); // 10, 50

alert(filteredArr.isEmpty()); // false

Обратите внимание: встроенные методы, такие как filter, map и другие возвращают новые объекты унаследованного класса PowerArray. Их внутренняя реализация такова, что для этого они используют свойство объекта constructor.

В примере выше arr.constructor === PowerArray. Поэтому при вызове метода arr.filter() он внутри создаёт массив результатов, именно используя arr.constructor, а не обычный массив. Это удобно, поскольку можно продолжать использовать методы PowerArray далее на результатах.

Более того, можно настроить это поведение. При помощи специального статического геттера Symbol.species можно вернуть конструктор, который JavaScript будет использовать в filter, map и других методах для создания новых объектов. Если надо, чтобы методы map, filter и т. д. возвращали обычные массивы, то нужно вернуть Array в Symbol.species, вот так:

class PowerArray extends Array {

isEmpty() {

return this.length === 0;

}

// встроенные методы массива будут использовать этот метод как конструктор

static get [Symbol.species]() {

return Array;

}

}

let arr = new PowerArray(1, 2, 5, 10, 50);

alert(arr.isEmpty()); // false

// filter создаст новый массив, используя arr.constructor[Symbol.species] как конструктор

let filteredArr = arr.filter(item => item >= 10);

// filteredArr не является PowerArray, это Array

alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function

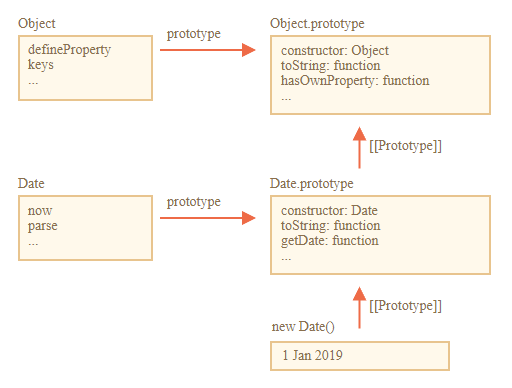
Теперь .filter возвращает Array. Расширенная функциональность не будет передаваться далее.

Другие коллекции, такие как Map, Set, работают аналогично. Они также исполуют Symbol.species.

**[Отсутствие статического наследования встроенных классов](https://learn.javascript.ru/extend-natives" \l "otsutstvie-staticheskogo-nasledovaniya-vstroennyh-klassov)**

У встроенных объектов есть собственные статические методы, например, Object.keys, Array.isArray и т. д. Как известно, встроенные классы расширяют друг друга. Обычно, когда один класс наследует другому, то наследуются и статические методы. Но встроенные классы – исключение. Они не наследуют статические методы друг друуга. Например, и Array и Date наследуют от Object, так что в их экземплярах доступны методы из Object.prototype. Но Array.[[Prototype]] не ссылается на Object, поэтому нет методов Array.keys() или Date.keys().

Ниже представлена структура Date и Object:



Как видно, нет связи между Date и Object. Они независимы, только Date.prototype наследует от Object.prototype. В этом важное отличие наследования встроенных объектов от тех, которые получаются с использованием extends.

1. **[Проверка класса: "instanceof"](https://learn.javascript.ru/instanceof)**

Оператор instanceof позволяет проверить, к какому классу принадлежит объект, с учётом наследования. Такая проверка может потребоваться во многих случаях. В рассматриваемых примерах она используется для создания полиморфной функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

**[Оператор instanceof](https://learn.javascript.ru/instanceof" \l "ref-instanceof)**

Синтаксис:

obj instanceof Class

Оператор вернёт true, если obj принадлежит классу Class или наследующему от него. Например:

class Rabbit {}

let rabbit = new Rabbit();

alert( rabbit instanceof Rabbit ); // true

Также это работает с функциями-конструкторами:

function Rabbit() {}

alert( new Rabbit() instanceof Rabbit ); // true

И для встроенных классов, таких как Array:

let arr = [1, 2, 3];

alert( arr instanceof Array ); // true

alert( arr instanceof Object ); // true

Обратите внимание, что arr также принадлежит классу Object, потому что Array наследует от Object.

Обычно оператор instanceof просматривает для проверки цепочку прототипов. Но это поведение может быть изменено при помощи статического метода Symbol.hasInstance.

Алгоритм работы obj instanceof Class работает примерно так:

1. Если имеется статический метод Symbol.hasInstance, тогда вызвать его: Class[Symbol.hasInstance](obj). Он должен вернуть либо true, либо false. Это как раз и есть возможность ручной настройки instanceof.

Пример:

class Animal {

static [Symbol.hasInstance](obj) {

if (obj.canEat) return true;

}

}

let obj = { canEat: true };

alert(obj instanceof Animal); // true: вызван Animal[Symbol.hasInstance](obj)

1. Большая часть классов не имеет метода Symbol.hasInstance. В этом случае используется стандартная логика: проверяется, равен ли Class.prototype одному из прототипов в прототипной цепочке obj.

Другими словами, сравнивается:

obj.\_\_proto\_\_ === Class.prototype?

obj.\_\_proto\_\_.\_\_proto\_\_ === Class.prototype?

obj.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ === Class.prototype?

...

// если какой-то из ответов true - возвратить true

// если дошли до конца цепочки - false

В примере выше rabbit.\_\_proto\_\_ === Rabbit.prototype, так что результат будет получен немедленно. В случае с наследованием, совпадение будет на втором шаге:

class Animal {}

class Rabbit extends Animal {}

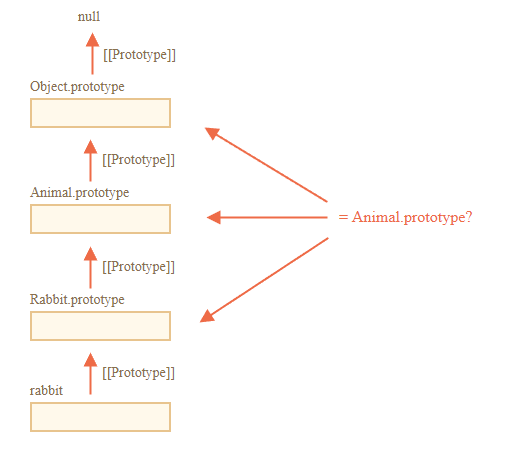
let rabbit = new Rabbit();

alert(rabbit instanceof Animal); // true

// rabbit.\_\_proto\_\_ === Rabbit.prototype

// rabbit.\_\_proto\_\_.\_\_proto\_\_ === Animal.prototype

Вот иллюстрация того как rabbit instanceof Animal сравнивается с Animal.prototype:



Есть метод [objA.isPrototypeOf(objB)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/object/isPrototypeOf), который возвращает true, если объект objA есть где-то в прототипной цепочке объекта objB. Так что obj instanceof Class можно перефразировать как Class.prototype.isPrototypeOf(obj). Сам конструктор Class не участвует в процессе проверки. Важна только цепочка прототипов Class.prototype. Это может приводить к интересным последствиям при изменении свойства prototype после создания объекта. Как, например, тут:

function Rabbit() {}

let rabbit = new Rabbit();

// заменяем прототип

Rabbit.prototype = {};

alert( rabbit instanceof Rabbit ); // false

Известно, что обычные объекты преобразуется к строке как [object Object]:

let obj = {};

alert(obj); // [object Object]

alert(obj.toString()); // то же самое

Так работает реализация метода toString. Но у toString имеются скрытые возможности, которые делают метод гораздо более мощным. Можно использовать его как расширенную версию typeof и как альтернативу instanceof. Согласно [спецификации](https://tc39.github.io/ecma262/" \l "sec-object.prototype.tostring) встроенный метод toString может бы позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения.

* для числа это будет [object Number];
* для булева типа это будет [object Boolean];
* для null: [object Null];
* для undefined: [object Undefined];
* для массивов: [object Array] и т.д.

Например:

let objectToString = Object.prototype.toString;

let arr = [];

alert( objectToString.call(arr) ); // [object Array]

В примере использовался [call](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/function/call), чтобы выполнить функцию objectToString в контексте this=arr. Внутри, алгоритм метода toString анализирует контекст вызова this и возвращает соответствующий результат. Пример:

let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]

alert( s.call(null) ); // [object Null]

alert( s.call(alert) ); // [object Function]

**Свойство [Symbol.toStringTag](https://learn.javascript.ru/instanceof" \l "symbol-tostringtag)**

Поведение метода объектов toString можно настраивать, используя специальное свойство объекта Symbol.toStringTag. Например:

let user = {

[Symbol.toStringTag]: "User"

};

alert( {}.toString.call(user) ); // [object User]

Такое свойство есть у большей части объектов, специфичных для определённых окружений. Вот несколько примеров для браузера:

alert( window[Symbol.toStringTag]); // window

alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]

alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]

Как видно, результат – это значение Symbol.toStringTag (если он имеется) обёрнутое в [object ...]. В итоге получили typeof, который не только работает с примитивными типами данных, но также и со встроенными объектами, и даже может быть настроен.

Можно использовать {}.toString.call вместо instanceof для встроенных объектов, когда надо получить тип в виде строки, а не просто сделать проверку.