1. **Переменные.** **Имена переменных. Константы.**

**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#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#%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#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.** **Оператор typeof.**

**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#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#%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#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#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#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#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#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#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#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#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#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.** **Бинарные и унарные операторы. Приоритет операторов. Совмещение операторов.**

**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#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#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#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#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#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#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#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#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#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. **Условные операторы: if, '?'. Конструкция switch.**

**Условные операторы.**

Чтобы выполнить различные действия в зависимости от условий, нам нужно использовать оператор if и условный оператор ?, который также называют «оператор вопросительный знак».

[**Оператор «if»**](https://learn.javascript.ru/ifelse#operator-if)

Оператор if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода. Например:

let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');

if (year == 2015) alert( 'Вы правы!' );

В примере выше, условие – это простая проверка на равенство (year == 2015), но оно может быть и гораздо более сложным. Если надо выполнить более одной инструкции, то нужно заключить блок кода в фигурные скобки:

if (year == 2015) {

alert( "Правильно!" );

alert( "Вы такой умный!" );

}

Рекомендуется использовать фигурные скобки {} всегда, когда используется оператор if, даже если выполняется только одна команда. Это улучшает читабельность кода.

[**Преобразование к логическому типу**](https://learn.javascript.ru/ifelse#preobrazovanie-k-logicheskomu-tipu)

Оператор if (…) вычисляет выражение в скобках и преобразует результат к логическому типу. Вспомним правила преобразования типов:

* Число 0, пустая строка "", null, undefined и NaN становятся false. Из-за этого их называют «ложными» значениями.
* Остальные значения становятся true, поэтому их называют «правдивыми».

Таким образом, код при таком условии никогда не выполнится:

if (0) {

...

}

А при таком – выполнится всегда:

if (1) {

...

}

Также можно передать заранее вычисленное в переменной логическое значение в if, например, так:

let cond = (year == 2015); // преобразуется к true или false

if (cond) {

...

}

[**Блок «else»**](https://learn.javascript.ru/ifelse#blok-else)

Оператор if может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно. Например:

let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');

if (year == 2015) {

alert( 'Да вы знаток!' );

} else {

alert( 'А вот и неправильно!' ); // любое значение, кроме 2015

}

[**Несколько условий: «else if»**](https://learn.javascript.ru/ifelse#neskolko-usloviy-else-if)

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок else if. Например:

let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');

if (year < 2015) {

alert( 'Это слишком рано...' );

} else if (year > 2015) {

alert( 'Это поздновато' );

} else {

alert( 'Верно!' );

}

В приведённом выше коде, JavaScript сначала проверит year < 2015. Если это неверно, он переходит к следующему условию year > 2015. Если оно тоже ложно, тогда сработает последний alert. Блоков else if может быть и больше. Присутствие блока else не является обязательным.

[**Условный оператор „?“**](https://learn.javascript.ru/ifelse#uslovnyy-operator)

Иногда нужно назначить переменную в зависимости от условия. Например:

let accessAllowed;

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

if (age > 18) {

accessAllowed = true;

} else {

accessAllowed = false;

}

alert(accessAllowed);

Так называемый «условный» оператор «вопросительный знак» позволяет сделать это более коротким и простым способом. Оператор представлен знаком вопроса ?. Его также называют «*тернарный*», так как этот оператор, единственный в своём роде, имеет три аргумента. Синтаксис:

let result = условие ? значение1 : значение2;

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение 2. Например:

let accessAllowed = (age > 18) ? true : false;

Технически, мы можем опустить круглые скобки вокруг age > 18. Оператор вопросительного знака имеет низкий приоритет, поэтому он выполняется после сравнения >. Этот пример будет делать то же самое, что и предыдущий:

// оператор сравнения "age > 18" выполняется первым в любом случае

// (нет необходимости заключать его в скобки)

let accessAllowed = age > 18 ? true : false;

Но скобки делают код более читабельным, поэтому рекомендуется их использовать.

В примере выше, можно избежать использования оператора вопросительного знака ?, т.к. сравнение само по себе уже возвращает true/false:

// то же самое

let accessAllowed = age > 18;

[**Несколько операторов „?“**](https://learn.javascript.ru/ifelse#neskolko-operatorov)

Последовательность операторов вопросительного знака ? позволяет вернуть значение, которое зависит от более чем одного условия. Например:

let age = prompt('Возраст?', 18);

let message = (age < 3) ? 'Здравствуй, малыш!' :

(age < 18) ? 'Привет!' :

(age < 100) ? 'Здравствуйте!' :

'Какой необычный возраст!';

alert( message );

Поначалу может быть сложно понять, что происходит. Но при ближайшем рассмотрении видно, что это обычная последовательная проверка:

1. Первый знак вопроса проверяет age < 3.
2. Если верно – возвращает 'Здравствуй, малыш!'. В противном случае, проверяет выражение после двоеточия „":"“, вычисляет age < 18.
3. Если это верно – возвращает 'Привет!'. В противном случае, проверяет выражение после следующего двоеточия „":"“, вычисляет age < 100.
4. Если это верно – возвращает 'Здравствуйте!'. В противном случае, возвращает выражение после последнего двоеточия – 'Какой необычный возраст!'.

Те же проверки при использовании if..else:

if (age < 3) {

message = 'Здравствуй, малыш!';

} else if (age < 18) {

message = 'Привет!';

} else if (age < 100) {

message = 'Здравствуйте!';

} else {

message = 'Какой необычный возраст!';

}

Иногда оператор вопросительный знак ? используется в качестве замены if:

let company = prompt('Какая компания создала JavaScript?', '');

(company == 'Netscape') ?

alert('Верно!') : alert('Неправильно.');

В зависимости от условия company == 'Netscape', будет выполнена либо первая, либо вторая часть после ?. Здесь не присваивается результат переменной. Вместо этого выполняется различный код в зависимости от условия. Не рекомендуется использовать оператор вопросительного знака таким образом, так как она менее читабельна.

Тот же код, использующий if:

let company = prompt('Какая компания создала JavaScript?', '');

if (company == 'Netscape') {

alert('Верно!');

} else {

alert('Неправильно.');

}

**Конструкция switch.**

Конструкция switch заменяет собой сразу несколько if. Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Конструкция switch имеет один или более блок *case* и необязательный блок *default*. Синтаксис:

switch(x) {

case 'value1': // if (x === 'value1')

...

[break]

case 'value2': // if (x === 'value2')

...

[break]

default:

...

[break]

}

* Переменная *x* проверяется на строгое равенство первому значению *value1*, затем второму *value2* и так далее.
* Если соответствие установлено – switch начинает выполняться от соответствующей директивы case и далее, до ближайшего break (или до конца switch).
* Если ни один case не совпал – выполняется (если есть) вариант default.

Пример использования switch:

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Маловато' );

break;

case 4:

alert( 'В точку!' );

break;

case 5:

alert( 'Перебор' );

break;

default:

alert( "Нет таких значений" );

}

Здесь оператор switch последовательно сравнит *a* со всеми вариантами из case. Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert( 'В точку!' ) и далее, до ближайшего break, который прервёт выполнение. Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются. Пример без break:

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Маловато' );

case 4:

alert( 'В точку!' );

case 5:

alert( 'Перебор' );

default:

alert( "Нет таких значений" );

}

В примере выше последовательно выполнятся три alert:

alert( 'В точку!' );

alert( 'Перебор' );

alert( "Нет таких значений" );

И switch и case допускают любое выражение в качестве аргумента. Например:

let a = "1";

let b = 0;

switch (+a) {

case b + 1:

alert("Выполнится, т.к. значением +a будет 1, что в точности равно b+1");

break;

default:

alert("Это не выполнится");

}

В этом примере результатом выражения +a будет 1, что совпадает с выражением b + 1 в case, и, следовательно, код в этом блоке будет выполнен.

Несколько вариантов case, использующих один код, можно группировать. Для примера, выполним один и тот же код для case 3 и case 5, сгруппировав их:

let a = 2 + 2;

switch (a) {

case 4:

alert('Правильно!');

break;

case 3: // (\*) группируем оба case

case 5:

alert('Неправильно!');

alert("Может вам посетить урок математики?");

break;

default:

alert('Результат выглядит странновато. Честно.');

}

Теперь оба варианта 3 и 5 выводят одно сообщение. Возможность группировать case – это побочный эффект того, как switch/case работает без break. Здесь выполнение case 3 начинается со строки (\*) и продолжается в case 5, потому что отсутствует break.

Стоит отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство. Для примера, рассмотрим следующий код:

let arg = prompt("Введите число?");

switch (arg) {

case '0':

case '1':

alert( 'Один или ноль' );

break;

case '2':

alert( 'Два' );

break;

case 3:

alert( 'Никогда не выполнится!' );

break;

default:

alert( 'Неизвестное значение' );

}

1. Для '0' и '1' выполнится первый alert.
2. Для '2' – второй alert.
3. Но для 3, результат выполнения prompt будет строка "3", которая не соответствует строгому равенству === с числом 3. Таким образом, код в case 3 не выполнится. Выполнится вариант default.
4. **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#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#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#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#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. **Циклы while, for.**

При написании скриптов зачастую встаёт задача сделать однотипное действие много раз. Например, вывести товары из списка один за другим. Или просто перебрать все числа от 1 до 10 и для каждого выполнить одинаковый код. Для многократного повторения одного участка кода предусмотрены *циклы*.

[**Цикл «while»**](https://learn.javascript.ru/while-for#tsikl-while)

Цикл while имеет следующий синтаксис:

while (condition) {

// код – тело цикла

}

Код из тела цикла выполняется, пока условие condition истинно. Например, цикл ниже выводит i, пока i < 3:

let i = 0;

while (i < 3) { // выводит 0, затем 1, затем 2

alert( i );

i++;

}

Одно выполнение тела цикла называется *итерация*. Цикл в примере выше совершает три итерации. Если бы строка i++ отсутствовала, то цикл бы повторялся (в теории) вечно. Но в действительности браузер предоставит пользователю возможность остановить «подвисший» скрипт, а JavaScript на стороне сервера придётся «убить» процесс.

Любое выражение или переменная может быть условием цикла, а не только сравнение: условие while вычисляется и преобразуется в логическое значение. Например, while (i) – более краткий вариант while (i != 0):

let i = 3;

while (i) { // когда i будет равно 0, условие станет ложным, и цикл остановится

alert( i );

i--;

}

Если тело цикла состоит лишь из одной инструкции, мы можем опустить фигурные скобки {…}:

let i = 3;

while (i) alert(i--);

[**Цикл «do…while»**](https://learn.javascript.ru/while-for#tsikl-dowhile)

Проверку условия можно разместить под телом цикла, используя специальный синтаксис do..while:

do {

// тело цикла

} while (condition);

Цикл сначала выполнит тело, а затем проверит условие condition, и пока его значение равно true, он будет выполняться снова и снова. Например:

let i = 0;

do {

alert( i );

i++;

} while (i < 3);

Цикл do..while стоит использовать, если необходимо, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным. На практике чаще используется форма с предусловием: while(…) {…}.

[**Цикл «for»**](https://learn.javascript.ru/while-for#tsikl-for)

Более сложный, но при этом самый распространённый цикл – цикл for. Синтаксис:

for (начало; условие; шаг) {

// ... тело цикла ...

}

Рассмотрим пример. Цикл ниже выполняет alert(i) для i от 0 до (но не включая) 3:

for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2

alert(i);

}

То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке *условия*, после которой выполняется *тело* и *шаг*.

В примере переменная счётчика *i* была объявлена прямо в цикле. Это так называемое встроенное объявление переменной. Такие переменные видны только внутри цикла.

for (let i = 0; i < 3; i++) {

alert(i); // 0, 1, 2

}

alert(i); // ошибка, нет такой переменной

Вместо объявления новой переменной можно использовать уже существующую:

let i = 0;

for (i = 0; i < 3; i++) {

alert(i); // 0, 1, 2

}

alert(i); // 3, переменная доступна

Любая часть for может быть пропущена. Например, можно пропустить начало если ничего не нужно делать перед стартом цикла:

let i = 0;

for (; i < 3; i++) {

alert( i ); // 0, 1, 2

}

Можно убрать шаг, это сделает цикл аналогичным while (i < 3):

let i = 0;

for (; i < 3;) {

alert( i++ );

}

Можно убрать всё, получив бесконечный цикл:

for (;;) {

// будет выполняться вечно

}

При этом сами точки с запятой ; обязательно должны присутствовать, иначе будет ошибка синтаксиса.

[**Прерывание цикла: «break»**](https://learn.javascript.ru/while-for#preryvanie-tsikla-break)

Обычно цикл завершается при вычислении *условия* в false. Но можно выйти из цикла в любой момент с помощью специальной директивы break. Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

let sum = 0;

while (true) {

let value = +prompt("Введите число", '');

if (!value) break; // (\*)

sum += value;

}

alert( 'Сумма: ' + sum );

Директива break в строке (\*) полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на alert. Использование директивы break в бесконечном цикле удобно в том случае, если условие, по которому нужно прерваться, находится не в начале или конце цикла, а посередине.

**[Переход к следующей итерации: continue](https://learn.javascript.ru/while-for" \l "continue)**

При выполнении директивы continue цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true). Её используют, если известно, что в текущей итерации цикла нет необходимости. Например, цикл ниже использует continue, чтобы выводить только нечётные значения:

for (let i = 0; i < 10; i++) {

// если true, пропустить оставшуюся часть тела цикла

if (i % 2 == 0) continue;

alert(i); // 1, затем 3, 5, 7, 9

}

Для чётных значений *i*, директива continue прекращает выполнение тела цикла и передаёт управление на следующую итерацию for (со следующим числом). Таким образом alert вызывается только для нечётных значений.

Директива continue позволяет избегать вложенности. Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

for (let i = 0; i < 10; i++) {

if (i % 2) {

alert( i );

}

}

Но здесь появился дополнительный уровень вложенности. Если код внутри if более длинный, то это ухудшает читаемость, в отличие от варианта с continue.

Эти синтаксические конструкции break/continue не являются выражениями и не могут быть использованы с тернарным оператором ?. Это приведет к синтаксической ошибке. Например:

(i > 5) ? alert(i) : continue; // continue здесь приведёт к ошибке

1. **Функции (Function Declaration). Параметры по умолчанию.**

[**Объявление функции**](https://learn.javascript.ru/function-basics#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#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#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#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#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 (Function Expression) 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-стрелки**](https://learn.javascript.ru/function-expressions-arrows#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.

1. **Числа. Способы записи числа. Системы счисления. Методы типа Number: преобразование к числу, округление, проверка специальных числовых значений.**

**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#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#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#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#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#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#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#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#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. Методы типа String: изменение регистра, поиск подстроки. Сравнение строк.**
2. **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#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#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#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#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#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#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#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#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#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#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#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.

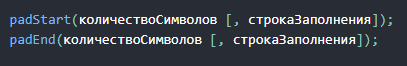
Например:

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

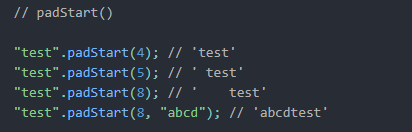
У этого метода есть два дополнительных аргумента, которые указаны в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare). Первый позволяет указать язык (по умолчанию берётся из окружения) – от него зависит порядок букв. Второй – определить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

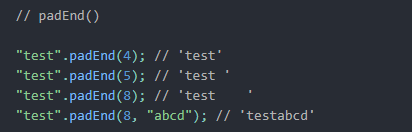
**Заполнение строки**

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#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#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#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#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#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#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#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#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#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 и замена элементов, объединение массивов, поиск в массиве.**

[**Добавление/удаление элементов**](https://learn.javascript.ru/array-methods#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#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#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#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

**Методы** [**поиска**](https://learn.javascript.ru/array-methods#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

1. **Методы перебора и преобразование массива.**

[**Перебор массива: метод forEach**](https://learn.javascript.ru/array-methods#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.

[**Преобразование массива**](https://learn.javascript.ru/array-methods#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. Литералы и свойства. Вычисляемые и короткие свойства. Проверка существования свойства. Перебор и упорядочение свойств объекта.**

**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#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,

};

[**Вычисляемые свойства**](https://learn.javascript.ru/object#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#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#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#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#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

}

1. **Копирование, клонирование, сравнение, объединение объектов. Объекты-константы.**

[**Копирование по ссылке**](https://learn.javascript.ru/object#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#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#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#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. **Коллекции Set, WeakSet.**

**[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.

[**WeakSet**](https://learn.javascript.ru/weakmap-weakset#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. **Коллекции Map, WeakMap.**

[**Map**](https://learn.javascript.ru/map-set#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#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#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.

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

1. **Деструктурирующее присваивание. Вложенная деструктуризация.**

**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#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#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#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#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#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.

1. **Глобальны объект. Создание функции с помощью конструктора (new Function).**

**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

}

**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.

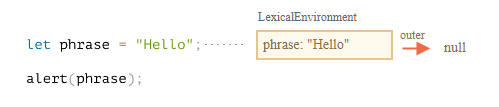
1. **[Лексическое](https://learn.javascript.ru/closure" \l "leksicheskoe-okruzhenie)** **окружение (LexicalEnvironment). Замыкание.**

**[Лексическое окружение](https://learn.javascript.ru/closure" \l "leksicheskoe-okruzhenie)**

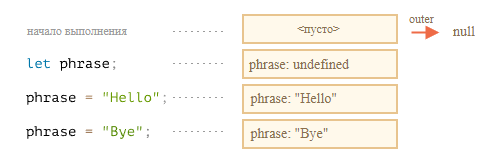
Чтобы понять, что происходит, давайте для начала, обсудим, что такое «переменная» на самом деле. В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением LexicalEnvironment. Объект лексического окружения состоит 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:



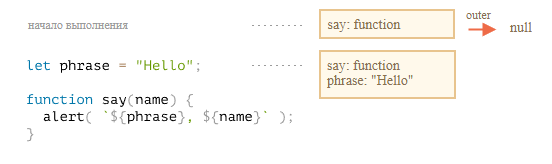
Прямоугольники с правой стороны демонстрируют, как глобальное лексическое окружение изменяется в процессе выполнения кода:

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#function-declaration)

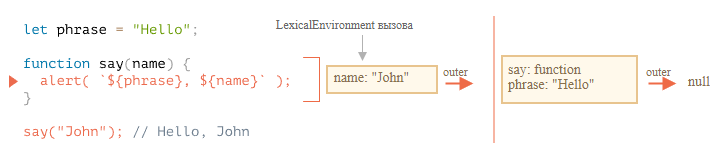
До сих поры рассматривались только переменные. Теперь рассмотрим Function Declaration. В отличие от переменных, объявленных с помощью let, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение. Для верхнеуровневых функций это означает 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#vnutrennee-i-vneshnee-leksicheskoe-okruzhenie)

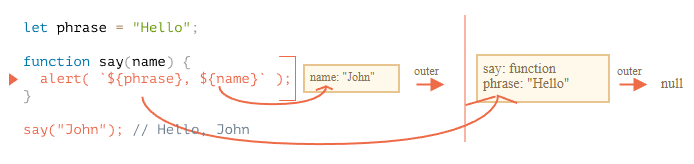
В течение вызова say() использует внешнюю переменную phrase. При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова. Например, для say("John") это выглядит так (выполнение находится на строке, отмеченной стрелкой):



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#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();

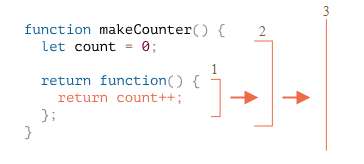
alert( counter() ); // 0

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();

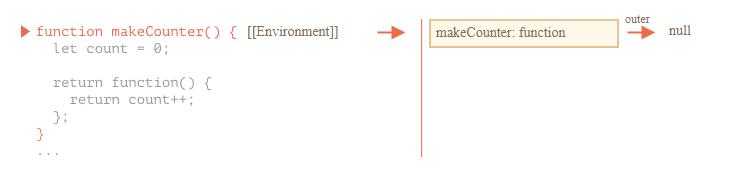
alert( counter1() ); // 0

alert( counter1() ); // 1

alert( counter2() ); // 0

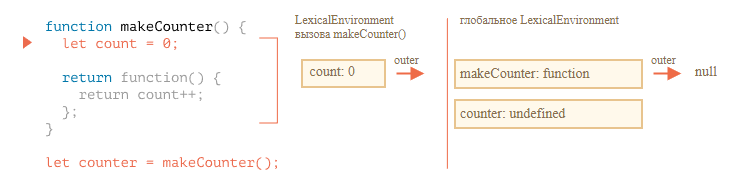
Рассмотрим, что происходит в примере с makeCounter шаг за шагом. Обратите внимание на дополнительное свойство [[Environment]].

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():



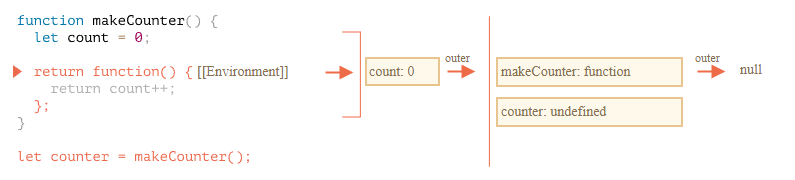
В момент вызова makeCounter() создаётся лексическое окружение, для хранения его переменных и аргументов. Как и все лексические окружения, оно содержит две вещи:

* 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.

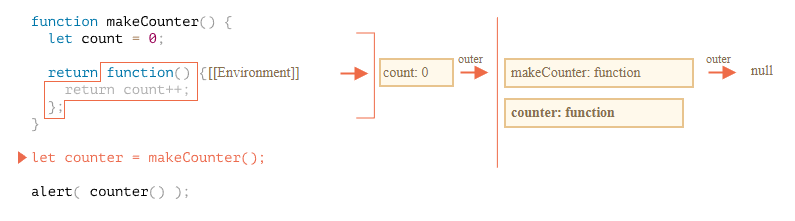
1. В процессе выполнения makeCounter() создаётся небольшая вложенная функция.

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):



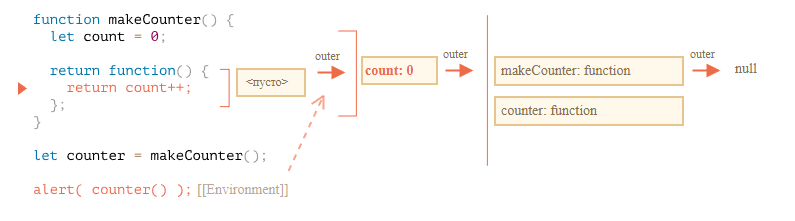
Обратите внимание, что на этом шаге внутренняя функция была создана, но ещё не вызвана. Код внутри function() { return count++ } не выполняется.

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.

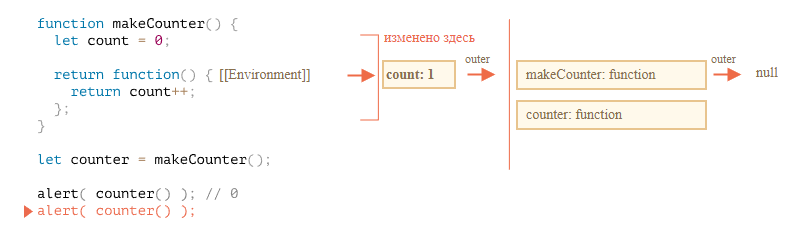
1. При вызове counter() для этого вызова создаётся новое лексическое окружение. Оно пустое, так как в самом counter локальных переменных нет. Но [[Environment]] counter используется, как ссылка на внешнее лексическое окружение outer, которое даёт доступ к переменным предшествующего вызова makeCounter,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

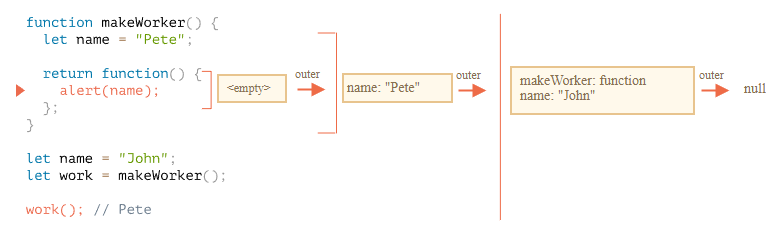
Обратите внимание, как здесь работает управление памятью. Хотя makeCounter() закончил выполнение некоторое время назад, его лексическое окружение остаётся в памяти, потому что есть вложенная функция с [[Environment]], который ссылается на него. В большинстве случаев, объект лексического окружения существует 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".

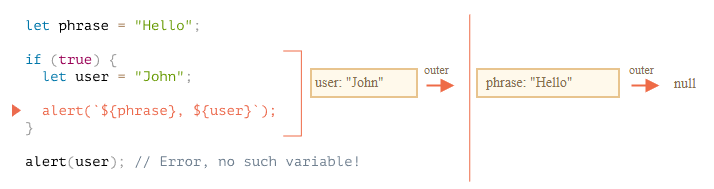
**Замыкания**

[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#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. **Объект функции. Именованное функциональное выражение (Named Function Expression).**

**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#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#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#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#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.**

**[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#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 и частичное применение функции.**

*[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. Функции-генераторы. Перебор объектов-генераторов.**

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#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;

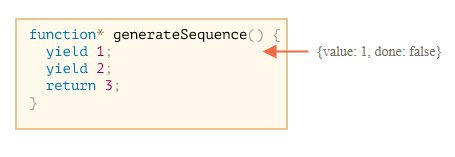
}

let generator = generateSequence();

let one = generator.next();

alert(JSON.stringify(one)); // {value: 1, done: false}

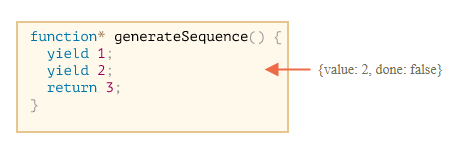
На данный момент получено только первое значение, выполнение функции остановлено на второй строке:



Повторный вызов generator.next() возобновит выполнение кода и вернёт результат следующего yield:

let two = generator.next();

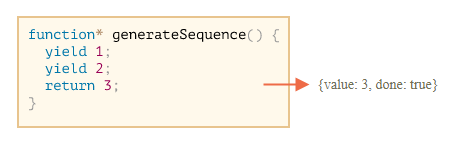
alert(JSON.stringify(two)); // {value: 2, done: false}



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#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#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 = ?"; // (\*)

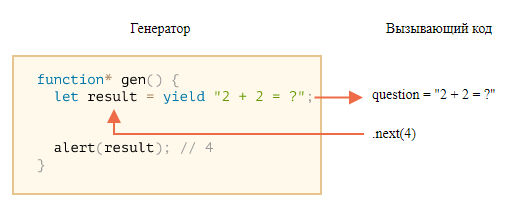
alert(result);

}

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

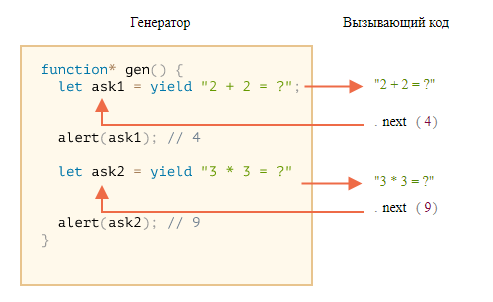
}

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#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

}

1. **Методы объектов, this. Оператор опциональной последовательности.**

**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#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/#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#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#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#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#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#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#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#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#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#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#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#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#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. **Декораторы. Методы call(), apply(), bind().**

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

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

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

Вместо того, чтобы усложнять slow(x) дополнительной функциональностью заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование. Вот код с объяснениями:

function slow(x) {

// здесь могут быть CPU ресурсоёмкие вычисления

alert(`Called with ${x}`);

return x;

}

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) { // если кэш содержит такой x

return cache.get(x); // читаем из него результат

}

let result = func(x); // иначе, вызываем функцию

cache.set(x, result); // и кешируем (запоминаем) результат

return result;

};

}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) кешируем

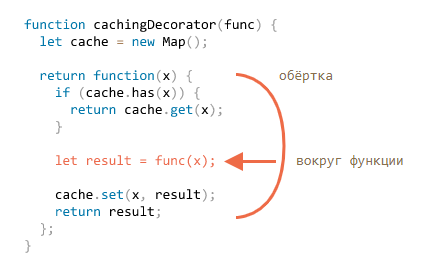
alert( "Again: " + slow(1) ); // возвращаем кеш

alert( slow(2) ); // slow(2) кешируем

alert( "Again: " + slow(2) ); // возвращаем кеш

В коде выше cachingDecorator – это декоратор, специальная функция, которая принимает другую функцию и изменяет её поведение. Идея состоит в том, что можно вызвать cachingDecorator с любой функцией, в результате чего получим кеширующую обёртку. Это удобно, т.к. может быть множество функций, использующих такой функционал, и все, что нужно сделать – это применить к ним cachingDecorator. Отделяя кеширующий код от основного кода, также сохраняем чистоту и простоту последнего.

Результат вызова cachingDecorator(func) является «обёрткой», т.е. function(x) «оборачивает» вызов func(x) в кеширующую логику:



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

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

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

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

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

let worker = {

someMethod() {

return 1;

},

slow(x) {

alert("Called with " + x);

return x \* this.someMethod(); // (\*)

}

};

// тот же код, что и выше

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) {

return cache.get(x);

}

let result = func(x); // (\*\*)

cache.set(x, result);

return result;

};

}

alert( worker.slow(1) ); // оригинальный метод работает

worker.slow = cachingDecorator(worker.slow); // сделаем его кеширующим

alert( worker.slow(2) ); // Ошибка

Ошибка возникает в строке (\*). Функция пытается получить доступ к this.someMethod и завершается с ошибкой. Причина в том, что в строке (\*\*) декоратор вызывает оригинальную функцию как func(x), а в этом случае функция получает this = undefined. Т.е. декоратор передаёт вызов оригинальному методу, но без контекста.

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

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

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

func(1, 2, 3);

func.call(obj, 1, 2, 3)

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

function sayHi() {

alert(this.name);

}

let user = { name: "John" };

let admin = { name: "Admin" };

sayHi.call( user ); // this = John

sayHi.call( admin ); // this = Admin

Здесь используется call для вызова say с заданным контекстом и фразой:

function say(phrase) {

alert(this.name + ': ' + phrase);

}

let user = { name: "John" };

say.call( user, "Hello" ); // John: Hello

В рассматриваемом примере можно использовать call в обёртке для передачи контекста в исходную функцию:

let worker = {

someMethod() {

return 1;

},

slow(x) {

alert("Called with " + x);

return x \* this.someMethod(); // (\*)

}

};

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) {

return cache.get(x);

}

let result = func.call(this, x); // 'this' передаётся правильно

cache.set(x, result);

return result;

};

}

worker.slow = cachingDecorator(worker.slow); // сделаем её кеширующей

alert( worker.slow(2) ); // работает

alert( worker.slow(2) ); // работает

Рассмотрим подробнее, как передаётся this:

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

[**Функция с несколькими аргументами – func.apply**](https://learn.javascript.ru/call-apply-decorators#perehodim-k-neskolkim-argumentam-s-func-apply)

Сделаем cachingDecorator ещё более универсальным. До сих пор он работал только с функциями с одним аргументом. Кешируем метод с несколькими аргументами worker.slow:

let worker = {

slow(min, max) {

return min + max; // здесь может быть тяжёлая задача

}

};

// should remember same-argument calls

worker.slow = cachingDecorator(worker.slow);

Ранее для одного аргумента *x* можно было просто сохранить результат cache.set(x, result) и вызвать cache.get(x) чтобы получить его. Но теперь нужно запомнить результат для комбинации аргументов (min,max). Встроенный Map принимает только одно значение как ключ. Есть много возможных решений:

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

Для многих практических применений третий вариант наиболее предпочтителен, поэтому реализуем его.

Нужно заменить func.call(this, x) на func.call(this, ...arguments), чтобы передавать все аргументы обёрнутой функции, а не только первый. Вот более мощный cachingDecorator:

let worker = {

slow(min, max) {

alert(`Called with ${min},${max}`);

return min + max;

}

};

function cachingDecorator(func, hash) {

let cache = new Map();

return function() {

let key = hash(arguments); // (\*)

if (cache.has(key)) {

return cache.get(key);

}

let result = func.call(this, ...arguments); // (\*\*)

cache.set(key, result);

return result;

};

}

function hash(args) {

return args[0] + ',' + args[1];

}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // работает

alert( "Again " + worker.slow(3, 5) ); // аналогично (из кеша)

Теперь он работает с любым количеством аргументов. Есть два изменения:

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

Вместо func.call(this, ...arguments) можно написать func.apply(this, arguments). Синтаксис встроенного метода [func.apply](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/apply):

func.apply(context, args)

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

func.call(context, ...args);

func.apply(context, args);

Есть только одна небольшая разница: оператор расширения ... позволяет передавать перебираемый объект args в виде списка в call, а apply принимает только псевдомассив args. Так что эти вызовы дополняют друг друга. Для перебираемых объектов сработает call, а там, где ожидается псевдомассив – apply.

Если есть, например, реальный массив, то технически можно использовать любой, но apply, вероятно, будет быстрее, потому что большинство движков JavaScript внутренне оптимизируют его лучше. Передача всех аргументов вместе с контекстом другой функции называется «перенаправлением вызова» (call forwarding). Простейший вид такого перенаправления:

let wrapper = function() {

return func.apply(this, arguments);

};

При вызове wrapper из внешнего кода его не отличить от вызова исходной функции.

[**Заимствование метода**](https://learn.javascript.ru/call-apply-decorators#method-borrowing)

Сделаем ещё одно небольшое улучшение функции хеширования:

function hash(args) {

return args[0] + ',' + args[1];

}

На данный момент он работает только для двух аргументов. Было бы лучше, если бы она могла склеить любое количество args. Естественным решением было бы использовать метод [arr.join](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/join):

function hash(args) {

return args.join();

}

Но это не сработает, потому что вызывается hash(arguments), а объект arguments является перебираемым и псевдомассивом, но не реальным массивом. Таким образом, вызов join для него не сработает:

function hash() {

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

}

hash(1, 2);

Тем не менее, есть простой способ использовать соединение массива:

function hash() {

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

}

hash(1, 2);

Этот способ называется заимствование метода. Т.е. заимствуется метод join из обычного массива [].join и используется [].join.call, чтобы выполнить его в контексте arguments. Это связано с тем, что внутренний алгоритм встроенного метода arr.join(glue) очень прост:

1. Пускай первым аргументом будет glue или, в случае отсутствия аргументов, им будет запятая ",".
2. Пускай result будет пустой строкой "".
3. Добавить this[0] к result.
4. Добавить glue и this[1].
5. Добавить glue и this[2].
6. Выполнять до тех пор, пока this.length элементов не будет склеено.
7. Вернуть result.

Таким образом, технически он принимает this и объединяет this[0], this[1]… и т.д. вместе. Он намеренно написан так, что допускает любой псевдомассив this (не случайно, многие методы следуют этой практике). Вот почему он также работает с this=arguments.

1. **Функции setTimeout and setInterval.**

Можно вызвать функцию не в данный момент, а позже, через заданный интервал времени. Это называется «планирование вызова». Для этого существует два метода:

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

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

**Метод** [**setTimeout**](https://learn.javascript.ru/settimeout-setinterval#settimeout)

Синтаксис:

let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)

Параметры:

func|code – функция или строка кода для выполнения. Обычно, это функция. Можно передать и строку кода, но это не рекомендуется.

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

arg1, arg2… – аргументы, передаваемые в функцию (не поддерживается в IE9-)

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

function sayHi() {

alert('Привет');

}

setTimeout(sayHi, 1000);

С аргументами:

function sayHi(phrase, who) {

alert( phrase + ', ' + who );

}

setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон

Если первый аргумент является строкой, то JavaScript создаст из неё функцию. Это также будет работать:

setTimeout("alert('Привет')", 1000);

Но использование строк не рекомендуется. Вместо этого используйте функции. Например, так:

setTimeout(() => alert('Привет'), 1000);

Передавайте функцию, но не запускайте её. Начинающие разработчики иногда ошибаются, добавляя скобки () после функции:

// не правильно!

setTimeout(sayHi(), 1000);

Это не работает, потому что setTimeout ожидает ссылку на функцию. Здесь sayHi() запускает выполнение функции и результат выполнения отправляется в setTimeout. В нашем случае результатом выполнения sayHi() является undefined (так как функция ничего не возвращает), поэтому ничего не планируется.

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

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

let timerId = setTimeout(...);

clearTimeout(timerId);

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

let timerId = setTimeout(() => alert("ничего не происходит"), 1000);

alert(timerId); // идентификатор таймера

clearTimeout(timerId);

alert(timerId); // тот же идентификатор

Как видно из вывода alert, в браузере идентификатором таймера является число. В других средах это может быть что-то ещё. Например, Node.js возвращает объект таймера с дополнительными методами.

**Метод** [**setInterval**](https://learn.javascript.ru/settimeout-setinterval#setinterval)

Метод setInterval имеет такой же синтаксис как setTimeout:

let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)

Все аргументы имеют такое же значение. Но отличие этого метода от setTimeout в том, что функция запускается не один раз, а периодически через указанный интервал времени. Чтобы остановить дальнейшее выполнение функции, необходимо вызвать clearInterval(timerId). Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:

// повторить с интервалом 2 секунды

let timerId = setInterval(() => alert('tick'), 2000);

// остановить вывод через 5 секунд

setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);

В большинстве браузеров, включая Chrome и Firefox внутренний счётчик продолжает тикать во время показа alert/confirm/prompt. Так что если запустить код выше и подождать с закрытием alert несколько секунд, то следующий alert будет показан сразу, как вы его закроете. Интервал времени между сообщениями alert будет короче, чем 2 секунды.

[**Рекурсивный setTimeout**](https://learn.javascript.ru/settimeout-setinterval#rekursivnyy-settimeout)

Есть два способа запускать что-то регулярно. Один из них setInterval. Другим является рекурсивный setTimeout. Например:

/\* вместо:

let timerId = setInterval(() => alert('tick'), 2000);

\*/

let timerId = setTimeout(function tick() {

alert('tick');

timerId = setTimeout(tick, 2000); // (\*)

}, 2000);

Метод setTimeout выше планирует следующий вызов прямо после окончания, текущего (\*). Рекурсивный setTimeout – более гибкий метод, чем setInterval. С его помощью, последующий вызов может быть задан по-разному, в зависимости от результатов предыдущего. Например, необходимо написать сервис, который отправляет запрос для получения данных на сервер каждые 5 секунд, но если сервер перегружен, то необходимо увеличить интервал запросов до 10, 20, 40 секунд. Например:

let delay = 5000;

let timerId = setTimeout(function request() {

//...отправить запрос...

if (/\* ошибка запроса из-за перегрузки сервера \*/) {

// увеличить интервал для следующего запроса

delay \*= 2;

}

timerId = setTimeout(request, delay);

}, delay);

Рекурсивный setTimeout позволяет задать задержку между выполнениями более точно, чем setInterval. Сравним два фрагмента кода. Первый использует setInterval:

let i = 1;

setInterval(function() {

func(i);

}, 100);

Второй использует рекурсивный setTimeout:

let i = 1;

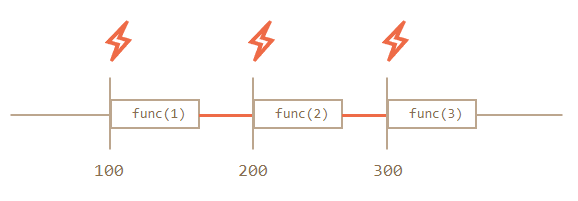
setTimeout(function run() {

func(i);

setTimeout(run, 100);

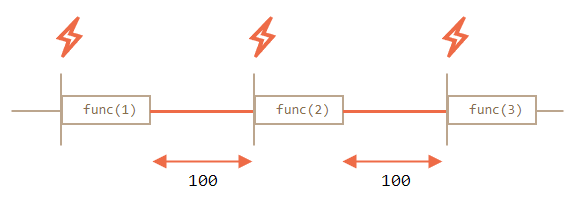
}, 100);

Для setInterval внутренний планировщик выполнит func(i) каждые 100 мс:



Реальная задержка между func для setInterval меньше, чем видно из кода. Это нормально, потому что время, затраченное на выполнение func, «потребляет» часть заданного интервала времени. Вполне возможно, что выполнение func будет дольше, чем ожидается, и займёт более 100 мс. В данном случае движок ждёт окончания выполнения func и затем проверяет планировщик и, если время истекло, немедленно запускает его снова. В крайнем случае, если функция всегда выполняется дольше, чем задержка delay, то вызовы будут выполняться без задержек вовсе.

Ниже представлено изображение, показывающее процесс работы рекурсивного setTimeout:



Рекурсивный setTimeout гарантирует фиксированную задержку (здесь 100 мс). Это потому, что новый вызов планируется в конце предыдущего.

Когда функция передаётся в setInterval/setTimeout, на неё создаётся внутренняя ссылка и сохраняется в планировщике. Это предотвращает попадание функции в сборщик муссора, даже если на неё нет других ссылок.

// функция остаётся в памяти до тех пор, пока планировщик обращается к ней

setTimeout(function() {...}, 100);

Для setInterval функция остаётся в памяти до тех, пока не будет вызван clearInterval. Есть и побочный эффект. Функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные существуют тоже. Они могут занимать больше памяти, чем сама функция. Поэтому, если регулярный вызов функции больше не нужен, то лучше отменить его, даже если функция очень маленькая.

Особый вариант использования: setTimeout(func, 0) или просто setTimeout(func). Это планирует вызов func настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода. Так вызов функции будет запланирован сразу после выполнения текущего кода.

Например, этот код выводит «Привет» и затем сразу «Мир»:

setTimeout(() => alert("Мир"));

alert("Привет");

Первая строка «помещает вызов в календарь через 0 мс». Но планировщик «проверит календарь» после того, как текущий код завершится. Поэтому "Привет" выводится первым, а "Мир" после него.

В браузере есть ограничение, как часто внутренние счётчики могут выполняться. В [стандарте HTML5](https://www.w3.org/TR/html5/webappapis.html#timers) говорится: «после пяти вложенных таймеров интервал должен составлять не менее четырёх миллисекунд.».

Продемонстрируем в примере ниже, что это означает. Вызов setTimeout повторно вызывает себя через 0 мс. Каждый вызов запоминает реальное время от предыдущего вызова в массиве times. Посмотрим какова реальная задержка:

let start = Date.now();

let times = [];

setTimeout(function run() {

times.push(Date.now() - start);

if (start + 100 < Date.now()) alert(times);

else setTimeout(run);

});

Первый таймер запускается сразу (как и указано в спецификации) и затем начинается задержка и вывод 9, 15, 20, 24. Аналогичное происходит при использовании setInterval вместо setTimeout: setInterval(f) запускает f несколько раз с нулевой задержкой, а затем с задержкой 4+ мс. Это ограничение существует давно, многие скрипты полагаются на него, поэтому оно сохраняется по историческим причинам. Этого ограничения нет в серверном JavaScript. Там есть и другие способы планирования асинхронных задач. Например, [setImmediate](https://nodejs.org/api/timers.html) для Node.js. Так что это ограничение относится только к браузерам.

1. **Прототипное наследование. Собственные и унаследованные свойства. Свойство F.prototype.**

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#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

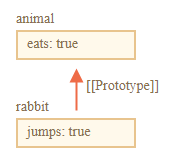
};

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 = {

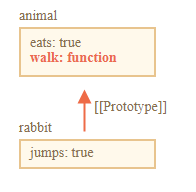
jumps: true,

\_\_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 = {

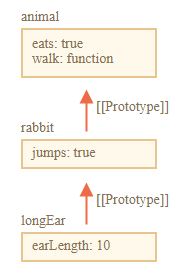
earLength: 10,

\_\_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

};

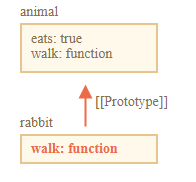
rabbit.walk = function() {

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#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

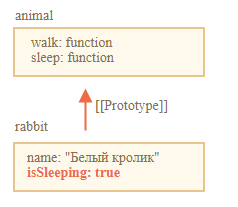
};

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#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

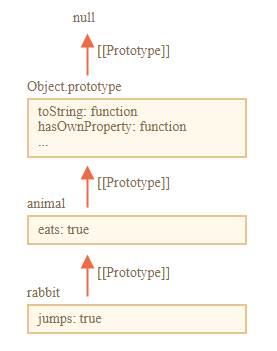
} else {

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.

[**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;

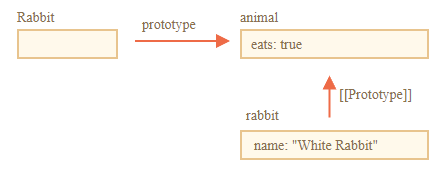
}

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.

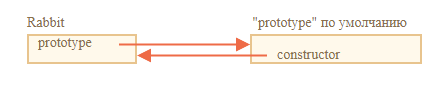
У каждой функции по умолчанию уже есть свойство "prototype". По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор. Вот такой:

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]]:

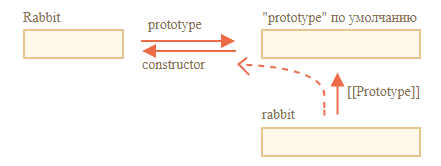
function Rabbit() {}

// 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. **Классы. Class Expression. Приватные и защищённые методы и свойства.**

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#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); }

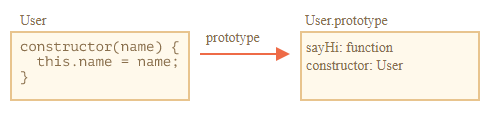
}

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#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#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.

[**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#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. **Наследование классов. Переопределение методов. Статические свойства и методы.** **Оператор instanceof**

**[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() {

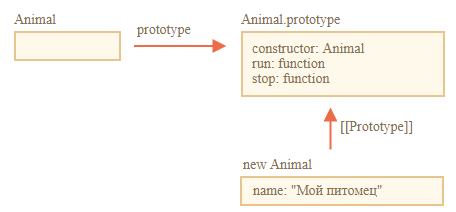
this.speed = 0;

alert(`${this.name} стоит.`);

}

}

let animal = new Animal("Мой питомец");



Класс Rabbit:

class Rabbit {

constructor(name) {

this.name = name;

}

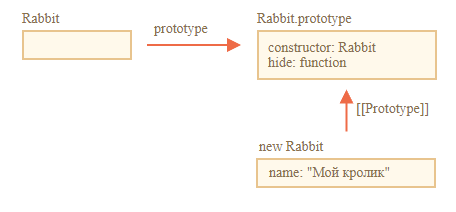
hide() {

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!`);

}

}

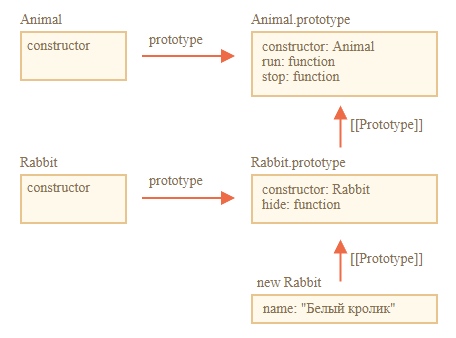
let rabbit = new Rabbit("Белый кролик");

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#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/#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#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 // (\*)

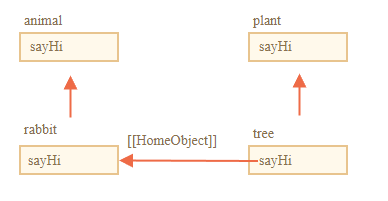
};

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

[**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#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),

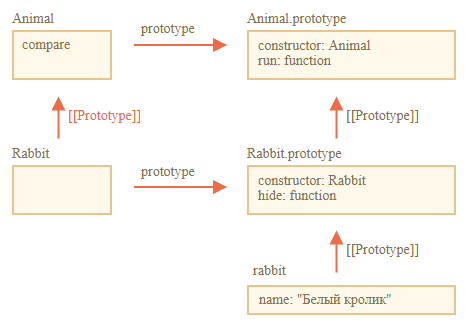
new Rabbit("Чёрный кролик", 5)

];

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);

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#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 {}

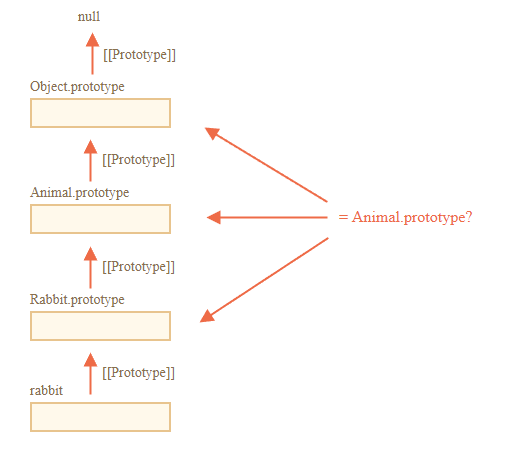
let rabbit = new Rabbit();

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/#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#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.

1. **Модули. Основные возможности модулей.**

По мере роста приложения, обычно возникает необходимость разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями. Долгое время в JavaScript отсутствовал синтаксис модулей на уровне языка. Это не было проблемой, потому что первые скрипты были маленькими и простыми. В модулях не было необходимости. Но со временем скрипты становились всё более и более сложными, поэтому сообщество придумало несколько вариантов организации кода в модули. Появились библиотеки для динамической подгрузки модулей. Например:

* [AMD](https://ru.wikipedia.org/wiki/Asynchronous_module_definition) – одна из самых старых модульных систем, изначально реализована библиотекой [require.js](http://requirejs.org/).
* [CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1) – модульная система, созданная для сервера Node.js.
* [UMD](https://github.com/umdjs/umd) – ещё одна модульная система, предлагается как универсальная, совместима с AMD и CommonJS.

Теперь все они постепенно становятся частью истории, хотя их и можно найти в старых скриптах.

Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году и постепенно эволюционировала. На данный момент она поддерживается большинством браузеров и Node.js.

Модуль – это файл с кодом. Один скрипт – это один модуль. Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

* export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
* import позволяет импортировать функциональность из других модулей.

Например, если есть файл sayHi.js, который экспортирует функцию:

//sayHi.js

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

Тогда другой файл может импортировать её и использовать:

// main.js

import {sayHi} from './sayHi.js';

alert(sayHi); // function...

sayHi('John'); // Hello, John!

Директива import загружает модуль по пути ./sayHi.js относительно текущего файла и записывает эксортированную функцию sayHi в соответствующую переменную. Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута <script type="module">. Вот так:

// say.js

export function sayHi(user) {

return `Hello, ${user}!`;

}

// index.html

<!doctype html>

<script type="module">

import {sayHi} from './say.js';

document.body.innerHTML = sayHi('John');

</script>

Браузер автоматически загрузит и запустит импортированный модуль (и те, которые он импортирует, если надо), а затем запустит скрипт.



[**Основные возможности модулей**](https://learn.javascript.ru/modules-intro#osnovnye-vozmozhnosti-moduley)

Есть основные возможности и особенности, работающие как в браузере, так и в серверном JavaScript.

В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

<script type="module">

a = 5; // ошибка

</script>

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

В следующем примере импортированы 2 скрипта, и hello.js пытается использовать переменную user, объявленную в user.js. В итоге ошибка:



// hello.js

alert(user);

// user.js

let user = "John";

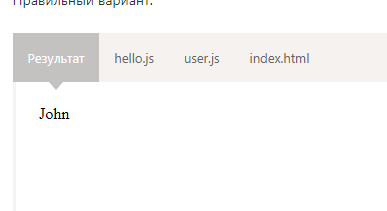
// index.html

<!doctype html>

<script type="module" src="user.js"></script>

<script type="module" src="hello.js"></script>

Модули должны экспортировать функционал, предназначенный для использования извне. А другие модули могут его импортировать. Так что надо импортировать user.js в hello.js и взять из него нужный функционал, вместо того чтобы полагаться на глобальные переменные. Правильный вариант:



// hello.js

import {user} from './user.js';

document.body.innerHTML = user; // John

// user.js

export let user = "John";

// index.html

import {user} from './user.js';

document.body.innerHTML = user; // John

В браузере также существует независимая область видимости для каждого скрипта <script type="module">:

<script type="module">

// Переменная доступна только в этом модуле

let user = "John";

</script>

<script type="module">

alert(user); // Error: user is not defined

</script>

Если нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту window, тогда получить значение переменной можно обратившись к window.user. Но это должно быть исключением, требующим веской причины.

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам. Это очень важно для понимания работы модулей. Рассмотрим примеры.

Во-первых, если при запуске модуля возникают побочные эффекты, например, выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте:

// alert.js

alert("Модуль выполнен!");

// Импорт одного и того же модуля в разных файлах

// 1.js

import `./alert.js`; // Модуль выполнен!

// 2.js

import `./alert.js`; // (ничего не покажет)

На практике, задача кода модуля – это обычно инициализация, создание внутренних структур данных, а если надо, чтобы что-то можно было использовать много раз, то экспортируем это.

Рассмотрим более сложный пример. Представим, что модуль экспортирует объект:

// admin.js

export let admin = {

name: "John"

};

Если модуль импортируется в нескольких файлах, то код модуля будет выполнен только один раз, объект admin будет создан и в дальнейшем будет передан всем импортёрам. Все импортёры получат один-единственный объект admin:

// 1.js

import {admin} from './admin.js';

admin.name = "Pete";

// 2.js

import {admin} from './admin.js';

alert(admin.name); // Pete

В примере оба файла, 1.js и 2.js, импортируют один и тот же объект. Изменения, сделанные в 1.js, будут видны в 2.js. Если что-то изменится в объекте admin, то другие модули тоже увидят эти изменения. Такое поведение позволяет конфигурировать модули при первом импорте. Можно установить его свойства один раз, и в дальнейших импортах он будет уже настроенным.

Например, модуль admin.js предоставляет определённую функциональность, но ожидает передачи учётных данных в объект admin извне:

// admin.js

export let admin = { };

export function sayHi() {

alert(`Ready to serve, ${admin.name}!`);

}

В init.js, первом скрипте рассматриваемого приложения, установим admin.name. Тогда все это увидят, включая вызовы, сделанные из самого admin.js:

// init.js

import {admin} from './admin.js';

admin.name = "Pete";

Другой модуль тоже увидит admin.name:

// other.js

import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete

sayHi(); // Ready to serve, Pete!

Объект import.meta содержит информацию о текущем модуле. Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

<script type="module">

alert(import.meta.url); // ссылка на html страницу для встроенного скрипта

</script>

В модуле на верхнем уровне this не определён (undefined). Это незначительная особенность, но для полноты картины нужно упомянуть об этом. Сравним с немодульными скриптами, там this – глобальный объект:

<script>

alert(this); // window

</script>

<script type="module">

alert(this); // undefined

</script>

Есть и несколько других, именно браузерных особенностей скриптов с type="module" по сравнению с обычными скриптами.

Модули всегда выполняются в отложенном (deferred) режиме, точно так же, как скрипты с атрибутом defer. Это верно и для внешних и встроенных скриптов-модулей. Другими словами:

* загрузка внешних модулей, таких как <script type="module" src="...">, не блокирует обработку HTML.
* модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
* сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

Как побочный эффект, модули всегда видят полностью загруженную HTML-страницу, включая элементы под ними. Например:

<script type="module">

alert(typeof button); // object: скрипт может 'видеть' кнопку под ним

</script>

Сравним с обычным скриптом ниже:

<script>

alert(typeof button); // Ошибка: кнопка не определена, скрипт не видит элементы под ним

</script>

<button id="button">Кнопка</button>

Второй скрипт выполнится раньше, чем первый. Поэтому сначала будет выведено undefined, а потом object. Это потому, что модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта видно первым.

При использовании модулей стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполнятся модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать. Следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

[Атрибут async работает во встроенных скриптах](https://learn.javascript.ru/modules-intro#atribut-async-rabotaet-vo-vstroennyh-skriptah). Для немодульных скриптов атрибут async работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ.

Для модулей атрибут async работает на любых скриптах. Например, в скрипте ниже есть async, поэтому он выполнится сразу после загрузки, не ожидая других скриптов. Скрипт выполнит импорт (загрузит ./analytics.js) и сразу запустится, когда будет готов, даже если HTML документ ещё не будет загружен, или если другие скрипты ещё загружаются. Это очень полезно, когда модуль ни с чем не связан, например, для счётчиков, рекламы, обработчиков событий.

<script async type="module">

import {counter} from './analytics.js';

counter.count();

</script>

[**Внешние скрипты**](https://learn.javascript.ru/modules-intro#vneshnie-skripty)

Внешние скрипты с атрибутом type="module" имеют два отличия:

1. Внешние скрипты с одинаковым атрибутом src запускаются только один раз. В примере ниже скрипт my.js загрузится и будет выполнен только один раз:

<script type="module" src="my.js"></script>

<script type="module" src="my.js"></script>

1. Внешний скрипт, который загружается с другого домена, требует указания заголовков [CORS](https://developer.mozilla.org/ru/docs/Web/HTTP/CORS). Другими словами, если модульный скрипт загружается с другого домена, то удалённый сервер должен установить заголовок Access-Control-Allow-Origin означающий, что загрузка скрипта разрешена.

<script type="module" src="http://another-site.com/their.js"></script>

Это обеспечивает лучшую безопасность по умолчанию.

В браузере import должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (bare). Они не разрешены в import. Например, этот import неправильный:

import {sayHi} from 'sayHi';

Другие окружения, например, Node.js, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать. Но браузеры пока не поддерживают «голые» модули.

Старые браузеры не понимают атрибут type="module". Скрипты с неизвестным атрибутом type просто игнорируются. Можно сделать для них «резервный» скрипт при помощи атрибута nomodule:

<script type="module">

alert("Работает в современных браузерах");

</script>

<script nomodule>

alert("Современные браузеры понимают оба атрибута - и type=module, и nomodule, поэтому пропускают этот тег script")

alert("Старые браузеры игнорируют скрипты с неизвестным атрибутом type=module, но выполняют этот.");

</script>

[Инструменты сборки](https://learn.javascript.ru/modules-intro#instrumenty-sborki)

В реальной жизни модули в браузерах редко используются как есть. Обычно, они объединяются вместе, специальными инструментами, например [Webpack](https://webpack.js.org/) и после выкладывается код на рабочий сервер.

Одно из преимуществ использования сборщика – он предоставляет больший контроль над тем, как модули ищутся, позволяет использовать «голые» модули и многое другое «своё», например, CSS/HTML-модули. Сборщик делает следующее:

1. Берёт «основной» модуль, который необходимо поместить в <script type="module"> в HTML.
2. Анализирует зависимости (импорты, импорты импортов и так далее).
3. Собирает один файл со всеми модулями (или несколько файлов, это можно настроить), перезаписывает встроенный import функцией импорта от сборщика, чтобы всё работало. «Специальные» типы модулей, такие как HTML/CSS тоже поддерживаются.
4. В процессе могут происходить и другие трансформации, и оптимизации кода:

* недоступный код удаляется;
* неиспользуемые экспорты удаляются («tree-shaking»);
* специфические операторы для разработки, такие как console и debugger, удаляются;
* современный синтаксис JavaScript также может быть трансформирован в предыдущий стандарт, с похожей функциональностью;
* полученный файл можно минимизировать (удалить пробелы, заменить названия переменных на более короткие и т.д.).

Если используются инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют import/export на свои вызовы. Поэтому итоговую сборку (в примере ниже – bundle.js) можно подключать и без атрибута type="module", как обычный скрипт:

<script src="bundle.js"></script>

Хотя и «как есть» модули тоже можно использовать, а сборщик настроить позже при необходимости.

1. **Модули: экспорт и импорт.**

Директивы экспорт и импорт имеют несколько вариантов вызова.

[**Экспорт до объявления**](https://learn.javascript.ru/import-export#eksport-do-obyavleniya)

Можно пометить любое объявление как экспортируемое, разместив export перед ним, будь то переменная, функция или класс. Например, все следующие экспорты допустимы:

// экспорт массива

export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// экспорт константы

export const MODULES\_BECAME\_STANDARD\_YEAR = 2015;

// экспорт класса

export class User {

constructor(name) {

this.name = name;

}

}

Обратите внимание, что export перед классом или функцией не делает их [функциональным выражением](https://learn.javascript.ru/function-expressions-arrows). Это всё также объявление функции, хотя и экспортируемое. Большинство руководств по стилю кода в JavaScript не рекомендуют ставить точку с запятой после объявлений функций или классов. Поэтому в конце export class и export function не нужна точка с запятой:

export function sayHi(user) {

alert(`Hello, ${user}!`);

} // без ; в конце

[**Экспорт отдельно от объявления**](https://learn.javascript.ru/import-export#eksport-otdelno-ot-obyavleniya)

Также можно написать export отдельно. Здесь сначала объявляются функции, а затем экспортируются:

// say.js

function sayHi(user) {

alert(`Hello, ${user}!`);

}

function sayBye(user) {

alert(`Bye, ${user}!`);

}

export {sayHi, sayBye}; // список экспортируемых переменных

Технически также можно расположить export выше функций.

[**Импорт \***](https://learn.javascript.ru/import-export#import)

Обычно список того, что надо импортировать располагается, в фигурных скобках import {...}, например, вот так:

// main.js

import {sayHi, sayBye} from './say.js';

sayHi('John'); // Hello, John!

sayBye('John'); // Bye, John!

Но если импортировать нужно много чего, то можно импортировать всё сразу в виде объекта, используя import \* as <obj>. Например:

// main.js

import \* as say from './say.js';

say.sayHi('John');

say.sayBye('John');

На первый взгляд «импортировать всё» выглядит очень удобно, не надо писать лишнего. Но иногда надо явно перечислять список того, что нужно импортировать , тому есть несколько причин:

1. Современные инструменты сборки ([webpack](http://webpack.github.io/) и другие) собирают модули вместе и оптимизируют их, ускоряя загрузку и удаляя неиспользуемый код.

Предположим, в проект была добавлена сторонняя библиотеку say.js с множеством функций:

// say.js

export function sayHi() { ... }

export function sayBye() { ... }

export function becomeSilent() { ... }

Теперь, если из этой библиотеки в проекте использовать только одну функцию:

// main.js

import {sayHi} from './say.js';

тогда оптимизатор увидит, что другие функции не используются, и удалит остальные из собранного кода, тем самым делая код меньше. Это называется «tree-shaking».

1. Явно перечисляя то, что надо импортировать, можно получить более короткие имена функций: sayHi() вместо say.sayHi().
2. Явное перечисление импортов делает код более понятным, позволяет увидеть, что именно и где используется. Это упрощает поддержку и рефакторинг кода.

[**Импорт «как»**](https://learn.javascript.ru/import-export#import-kak)

Также мжно использовать as, чтобы импортировать под другими именами. Например, для краткости импортируем sayHi в локальную переменную hi, а sayBye импортируем как bye:

// main.js

import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!

bye('John'); // Bye, John!

[**Экспортировать «как»**](https://learn.javascript.ru/import-export#eksportirovat-kak)

Аналогичный синтаксис существует и для export. Давайте экспортируем функции, как hi и bye:

// say.js

export {sayHi as hi, sayBye as bye};

Теперь hi и bye – официальные имена для внешнего кода, их нужно использовать при импорте:

// main.js

import \* as say from './say.js';

say.hi('John'); // Hello, John!

say.bye('John'); // Bye, John!

[**Экспорт по умолчанию**](https://learn.javascript.ru/import-export#eksport-po-umolchaniyu)

На практике модули встречаются в основном одного из двух типов:

1. Модуль, содержащий библиотеку или набор функций, как say.js выше.
2. Модуль, который объявляет что-то одно, например, модуль user.js экспортирует только class User.

По большей части, удобнее второй подход, когда каждая «вещь» находится в своём собственном модуле. Естественно, требуется много файлов, если для всего делать отдельный модуль, но это не проблема. Так удобнее: навигация по проекту становится проще, особенно, если у файлов хорошие имена, и они структурированы по папкам.

Модули предоставляют специальный синтаксис export default («эспорт по умолчанию») для второго подхода. Указываем export default перед тем, что нужно экспортировать:

// user.js

export default class User {

constructor(name) {

this.name = name;

}

}

В файле может быть не более одного export default, и потом импорт без фигурных скобок:

// main.js

import User from './user.js'; // не {User}, просто User

new User('John');

Импорты без фигурных скобок выглядят красивее. Обычная ошибка начинающих: забывать про фигурные скобки. Запомним: фигурные скобки необходимы в случае именованных экспортов, для export default они не нужны.

| **Именованный экспорт** | **Экспорт по умолчанию** |
| --- | --- |
| export class User {...} | export default class User {...} |
| import {User} from ... | import User from ... |

Технически в одном модуле может быть, как экспорт по умолчанию, так и именованные экспорты, но на практике обычно их не смешивают. То есть, в модуле находятся либо именованные экспорты, либо один экспорт по умолчанию.

Так как в файле может быть максимум один export default, то экспортируемая сущность не обязана иметь имя. Например, всё это – полностью корректные экспорты по умолчанию:

export default class { // у класса нет имени

constructor() { ... }

}

export default function(user) { // у функции нет имени

alert(`Hello, ${user}!`);

}

// экспортируем значение, не создавая переменную

export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

Это нормально, потому что может быть только один export default в файле, так что import без фигурных скобок всегда знает, что импортировать. Без default такой экспорт выдал бы ошибку, так как необходимо имя класса, если это не экспорт по умолчанию:

export class { // Ошибка

constructor() {}

}

[**Имя «default»**](https://learn.javascript.ru/import-export#imya-default)

В некоторых ситуациях для обозначения экспорта по умолчанию в качестве имени используется default. Например, чтобы экспортировать функцию отдельно от её объявления:

function sayHi(user) {

alert(`Hello, ${user}!`);

}

export {sayHi as default};

Представим следующую ситуацию: модуль user.js экспортирует одну сущность «по умолчанию» и несколько именованных:

// user.js

export default class User {

constructor(name) {

this.name = name;

}

}

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

Вот как импортировать экспорт по умолчанию вместе с именованным экспортом:

// main.js

import {default as User, sayHi} from './user.js';

new User('John');

Если импортируется всё как объект import \*, тогда его свойство default – как раз и будет экспортом по умолчанию:

// main.js

import \* as user from './user.js';

let User = user.default; // экспорт по умолчанию

new User('John');

[Однако, у экспорта по умолчанию](https://learn.javascript.ru/import-export#dovod-protiv-eksportov-po-umolchaniyu) есть недостатки. Именованные экспорты «включают в себя» своё имя. Эта информация является частью модуля, она сообщает, что именно экспортируется. Именованные экспорты вынуждают использовать правильное имя при импорте:

// import {MyUser} не сработает, должно быть именно имя {User}

import {User} from './user.js';

В то время как для экспорта по умолчанию можно выбрать любое имя при импорте:

import User from './user.js'; // сработает

import MyUser from './user.js'; // сработает

Так что члены команды разработчиков могут использовать разные имена для импорта одной и той же вещи, и это не очень хорошо. Обычно, чтобы избежать этого и соблюсти единообразие кода, есть правило: имена импортируемых переменных должны соответствовать именам файлов. Вот так:

import User from './user.js';

import LoginForm from './loginForm.js';

import func from '/path/to/func.js';

Тем не менее, в некоторых командах это считают серьёзным доводом против экспортов по умолчанию и предпочитают использовать именованные экспорты везде. Даже если экспортируется только одна вещь, она всё равно экспортируется с именем, без использования default. Это также немного упрощает реэкспорт.

[**Реэкспорт**](https://learn.javascript.ru/import-export#reeksport)

Синтаксис «реэкспорта» export ... from ... позволяет импортировать что-то и тут же экспортировать, даже под другим именем:

export {sayHi} from './say.js'; // реэкспортировать sayHi

export {default as User} from './user.js'; // реэкспортировать default

Рассмотрим практический пример использования. Представим, что разрабатывается «пакет»: папка со множеством модулей, из которой часть функционала экспортируется наружу (инструменты вроде NPM позволяют публиковать и распространять такие пакеты), а многие модули – просто вспомогательные, для внутреннего использования в других модулях пакета. Структура файлов может быть такой:

auth/

index.js

user.js

helpers.js

tests/

login.js

providers/

github.js

facebook.js

...

Надо сделать функционал пакета доступным через единую точку входа: «главный файл» auth/index.js. Чтобы можно было использовать его следующим образом:

import {login, logout} from 'auth/index.js'

Идея в том, что внешние разработчики, которые будут использовать пакет, не должны разбираться с его внутренней структурой. Всё, что нужно, надо экспортировать в auth/index.js, а остальное скрыть от разработчиков. Так как нужный функционал может быть разбросан по модулям пакета, то надо импортировать их в auth/index.js и тут же экспортировать наружу.

// auth/index.js

// импортировать login/logout и тут же экспортировать

import {login, logout} from './helpers.js';

export {login, logout};

// импортировать экспорт по умолчанию как User и тут же экспортировать

import User from './user.js';

export {User};

Теперь пользователи пакета могут писать import {login} from "auth/index.js". Запись export ... from ...– это просто более короткий вариант такого импорта-экспорта:

// auth/index.js

// импортировать login/logout и тут же экспортировать

export {login, logout} from './helpers.js';

// импортировать экспорт по умолчанию как User и тут же экспортировать

export {default as User} from './user.js';

[**Реэкспорт экспорта по умолчанию**](https://learn.javascript.ru/import-export#reeksport-eksporta-po-umolchaniyu)

При реэкспорте экспорт по умолчанию нужно обрабатывать особым образом. Например, есть user.js, из которого надо реэкспортировать класс User:

// user.js

export default class User {

// ...

}

1. export User from './user.js' не будет работать, возникнет синтаксическая ошибка.

Чтобы реэкспортировать экспорт по умолчанию, надо написать export {default as User}, как в примере выше.

1. export \* from './user.js' реэкспортирует только именованные экспорты, исключая экспорт по умолчанию.

Если надо реэкспортировать и именованные экспорты, и экспорт по умолчанию, то понадобятся две инструкции:

export \* from './user.js'; // для реэкспорта именованных экспортов

export {default} from './user.js'; // для реэкспорта по умолчанию

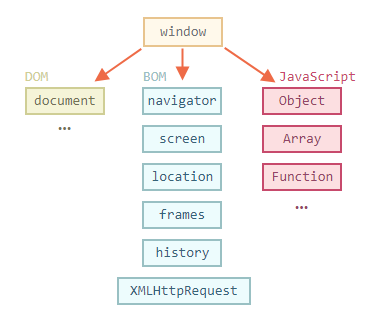
Такое особое поведение реэкспорта с экспортом по умолчанию – одна из причин того, почему их неудобно использовать.

1. **Окружение: DOM, BOM. Дерево DOM.**

**Окружение: DOM, BOM.**

Язык JavaScript изначально был создан для веб-браузеров. Но с тех пор он значительно эволюционировал и превратился в кроссплатформенный язык программирования для решения широкого круга задач. Сегодня JavaScript может использоваться в браузере, на веб-сервере или в какой-то другой среде, даже в кофеварке. Каждая среда предоставляет свой функционал, который спецификация JavaScript называет *окружением*.

Окружение предоставляет свои объекты и дополнительные функции, в дополнение базовым языковым. Браузеры, например, дают средства для управления веб-страницами. Node.js делает доступными какие-то серверные возможности и так далее. На изображении ниже в общих чертах показано, что доступно для JavaScript в браузерном окружении:



Как видно, имеется корневой объект window, который выступает в 2 ролях:

1. Во-первых, это глобальный объект для JavaScript-кода.
2. Во-вторых, он также представляет собой окно браузера и располагает методами для управления им.

Например, здесь используется window как глобальный объект:

function sayHi() {

alert("Hello");

}

window.sayHi();

А здесь используется window как объект окна браузера, чтобы узнать его высоту:

alert(window.innerHeight); // внутренняя высота окна браузера

Существует гораздо больше свойств и методов для управления окном браузера. Они будут рассмотрены позднее.

[**DOM (Document Object Model)**](https://learn.javascript.ru/browser-environment#dom-document-object-model)

*Document Object Model*, сокращенно DOM – объектная модель документа, которая представляет все содержимое страницы в виде объектов, которые можно менять.

*Объект document* – основная «входная точка». С его помощью можно что-то создавать или менять на странице. Например:

// заменим цвет фона на красный,

document.body.style.background = "red";

// а через секунду вернём как было

setTimeout(() => document.body.style.background = "", 1000);

В примере использован только document.body.style, но на самом деле возможности по управлению страницей намного шире. Различные свойства и методы описаны в спецификации: DOM Living Standard на [https://dom.spec.whatwg.org](https://dom.spec.whatwg.org/).

Спецификация DOM описывает структуру документа и предоставляет объекты для манипуляций со страницей. Существует и другие, отличные от браузеров, инструменты, использующие DOM. Например, серверные скрипты, которые загружают и обрабатывают HTML-страницы, также могут использовать DOM. При этом они могут поддерживать спецификацию не полностью.

Правила стилей CSS структурированы иначе чем HTML. Для них есть отдельная спецификация [CSSOM](https://www.w3.org/TR/cssom-1/), которая объясняет, как стили должны представляться в виде объектов, как их читать и писать. CSSOM используется вместе с DOM при изменении стилей документа. В реальности CSSOM требуется редко, обычно правила CSS статичны. Стили из JavaScript редко добавляются/удаляются, но и это возможно.

[**BOM (Browser Object Model)**](https://learn.javascript.ru/browser-environment#bom-browser-object-model)

*Объектная модель браузера (Browser Object Model, BOM)* – это дополнительные объекты, предоставляемые браузером (окружением), чтобы работать со всем, кроме документа. Например:

* Объект [navigator](https://developer.mozilla.org/ru/docs/Web/API/Window/navigator) даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).
* Объект [location](https://developer.mozilla.org/ru/docs/Web/API/Window/location) позволяет получить текущий URL и перенаправить браузер по новому адресу.

Вот как можно использовать объект location:

alert(location.href); // показывает текущий URL

if (confirm("Перейти на Wikipedia?")) {

location.href = "https://wikipedia.org"; // перенаправляет браузер на другой URL

}

Функции alert/confirm/prompt тоже являются частью BOM: они не относятся непосредственно к странице, но представляют собой методы объекта окна браузера для коммуникации с пользователем.

BOM является частью общей [спецификации HTML](https://html.spec.whatwg.org/). Спецификация HTML по адресу [https://html.spec.whatwg.org](https://html.spec.whatwg.org/) не только про «язык HTML» (теги, атрибуты), она также покрывает целое множество объектов, методов и специфичных для каждого браузера расширений DOM. Это всё «HTML в широком смысле». Для некоторых вещей есть отдельные спецификации, перечисленные на [https://spec.whatwg.org](https://spec.whatwg.org/).

**Дерево DOM.**

Основой HTML-документа являются теги. В соответствии с объектной моделью документа («Document Object Model», коротко DOM), каждый HTML-тег является объектом. Вложенные теги являются «детьми» родительского элемента. Текст, который находится внутри тега, также является объектом. Все эти объекты доступны при помощи JavaScript, можно использовать их для изменения страницы. Например, document.body – объект для тега <body>. Если запустить этот код, то <body> станет красным на 3 секунды:

document.body.style.background = 'red'; // сделать фон красным

setTimeout(() => document.body.style.background = '', 3000); // вернуть назад

Это был лишь небольшой пример того, что может DOM.

[**Структура DOM**](https://learn.javascript.ru/dom-nodes#primer-dom)

Разберемся со структурой DOM. Рассмотрим простой документ:

<!DOCTYPE HTML>

<html>

<head>

<title>О лосях</title>

</head>

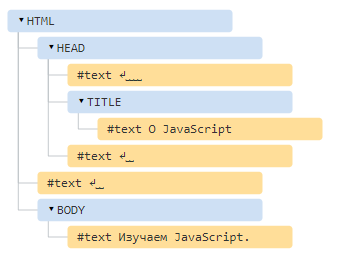
<body>

Правда о лосях.

</body>

</html>

DOM – это представление HTML-документа в виде дерева тегов. Вот как оно выглядит:



Каждый узел этого дерева – это объект. Теги являются узлами-элементами (или просто элементами). Они образуют структуру дерева: <html> – это корневой узел, <head> и <body> его дочерние узлы, и т.д. Текст внутри элементов образует текстовые узлы, обозначенные как #text. Текстовый узел содержит в себе только строку текста. У него не может быть потомков, т.е. он находится всегда на самом нижнем уровне. Например, в теге <title> есть текстовый узел "О лосях".

Обратите внимание на специальные символы в текстовых узлах: перевод строки: ↵ (в JavaScript он обозначается как \n) и пробел: ␣. Пробелы и переводы строки – это полноправные символы, как буквы и цифры. Они образуют текстовые узлы и становятся частью дерева DOM. Так, в примере выше в теге <head> есть несколько пробелов перед <title>, которые образуют текстовый узел #text (он содержит в себе только перенос строки и несколько пробелов).

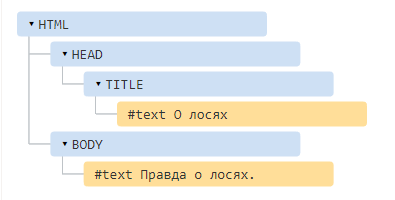
Существует всего два исключения из этого правила:

1. По историческим причинам пробелы и перевод строки перед тегом <head> игнорируются.
2. Если записать что-либо после закрывающего тега </body>, браузер автоматически перемещает эту запись в конец body, поскольку спецификация HTML требует чтобы все содержимое было внутри <body>. Поэтому после закрывающего тега </body> не может быть никаких пробелов.

В остальных если в документе есть пробелы (или любые другие символы), они становятся текстовыми узлами дерева DOM, и если их надо удалим, то в DOM их тоже не будет. Здесь пробельных текстовых узлов нет:

<!DOCTYPE HTML>

<html><head><title>О лосях</title></head><body>Правда о лосях.</body></html>



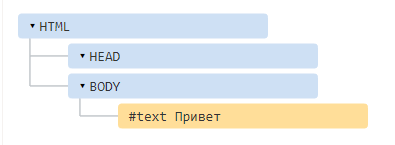
Пробелы по краям строк и пробельные текстовые узлы скрыты в инструментах разработки и обычно не отображаются. Таким образом инструменты разработки экономят место на экране.

В дальнейших иллюстрациях DOM также будут для краткости пропущены пробельные текстовые узлы там, где они не имеют значения. Обычно они не влияют на то, как отображается документ.

**[Автоисправление](https://learn.javascript.ru/dom-nodes" \l "avtoispravlenie)**

Если браузер сталкивается с некорректно написанным HTML-кодом, он автоматически корректирует его при построении DOM. Например, в начале документа всегда должен быть тег <html>. Даже если его нет в документе – он будет в дереве DOM, браузер его создаст. То же самое касается и тега <body>.

Например, если HTML-файл состоит из единственного слова "Привет", браузер обернёт его в теги <html> и <body>, добавит необходимый тег <head>, и DOM будет выглядеть так:



При генерации DOM браузер самостоятельно обрабатывает ошибки в документе, закрывает теги и так далее. Такой документ с незакрытыми тегами:

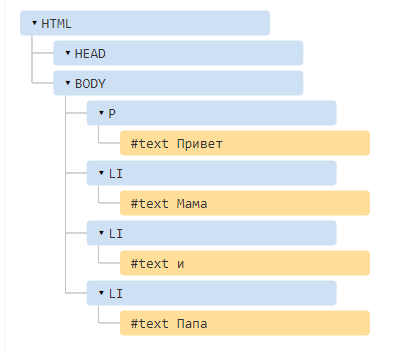
<p>Привет

<li>Мама

<li>и

<li>Папа

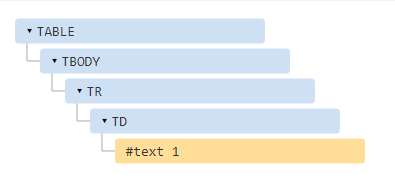
Но DOM будет нормальным, потому что браузер сам закроет теги и восстановит отсутствующие детали:



Важный «особый случай» – работа с таблицами. По стандарту DOM у них должен быть <tbody>, но в HTML их можно написать (официально) без него. В этом случае браузер добавляет <tbody> в DOM самостоятельно. Для такого HTML:

<table id="table"><tr><td>1</td></tr></table>

DOM-структура будет такой:



[**Другие типы узлов**](https://learn.javascript.ru/dom-nodes#drugie-tipy-uzlov)

Есть и некоторые другие типы узлов, кроме элементов и текстовых узлов. Например, узел-комментарий:

<!DOCTYPE HTML>

<html>

<body>

Правда о лосях.

<ol>

<li>Лось -- животное хитрое</li>

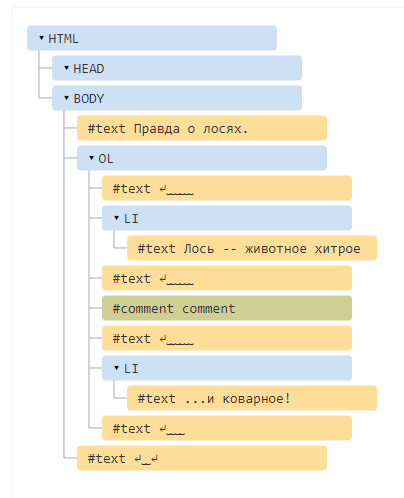
<!-- комментарий -->

<li>...и коварное!</li>

</ol>

</body>

</html>

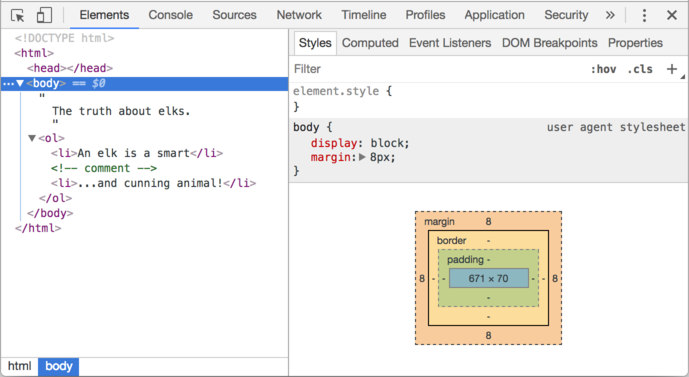


Здесь видно, что новый узел нового типа – комментарий, обозначенный как #comment, между двумя текстовыми узлами. Есть важное правило: если что-то есть в HTML, то оно должно быть в DOM-дереве. Поэтому все, что есть в HTML, даже комментарии, является частью DOM. Даже директива <!DOCTYPE...>, которая располагается в начале HTML, тоже является DOM-узлом. Она находится в дереве DOM прямо перед <html>. Даже объект document, представляющий весь документ, формально, является DOM узлом.

Существует [12 типов узлов](https://dom.spec.whatwg.org/#node). Но на практике в основном работают с четырьмя из них:

1. document – «входная точка» в DOM.
2. Узлы-элементы – HTML-теги, основные строительные блоки.
3. Текстовые узлы – содержат текст.
4. Комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

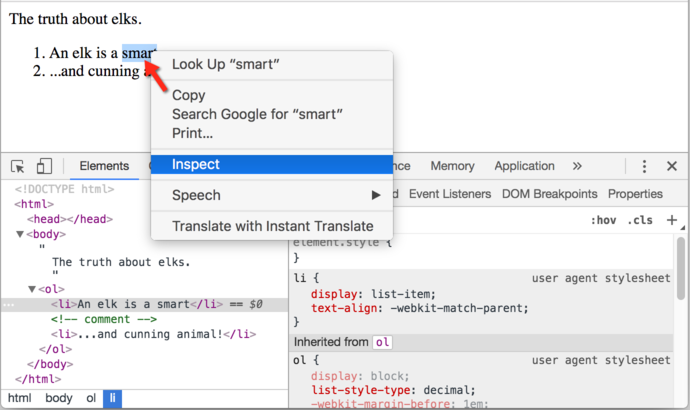
Чтобы посмотреть структуру DOM в реальном времени, можно использовать инструменты разработчика браузера. Для этого откройте страницу [\*.html](https://learn.javascript.ru/article/dom-nodes/elks.html), включите инструменты разработчика и перейдите на вкладку Elements. Выглядит примерно так:



Можно увидеть DOM, раскрыть/свернуть элементы, детально рассмотреть их и так далее. Обратите внимание, что структура DOM в инструментах разработчика отображается в упрощённом виде. Текстовые узлы показаны как простой текст. И кроме пробелов нет никаких «пустых» текстовых узлов, что очень удобно.

Клик по этой  кнопке в левом верхнем углу инспектора позволяет при помощи мыши (или другого устройства ввода) выбрать элемент на веб-странице и «проинспектировать» его (браузер сам найдёт и отметит его во вкладке Elements). Этот способ отлично подходит, когда есть огромная HTML-страница (и соответствующий ей огромный DOM), и надо увидеть, где находится интересующий нас элемент.

Есть и другой способ сделать это, можно кликнуть на странице по элементу правой кнопкой мыши и в контекстном меню выбрать «Inspect».



В правой части инструментов разработчика находятся следующие подразделы:

* Styles – здесь показан CSS, применённый к текущему элементу: правило за правилом, включая встроенные стили (выделены серым). Почти все можно отредактировать на месте, включая размеры/внешние и внутренние отступы.
* Computed – здесь видны итоговые CSS-свойства элемента, которые он приобрёл в результате применения всего каскада стилей (в том числе унаследованные свойства и т.д.).
* Event Listeners – в этом разделе видны обработчики событий, привязанные к DOM-элементам.

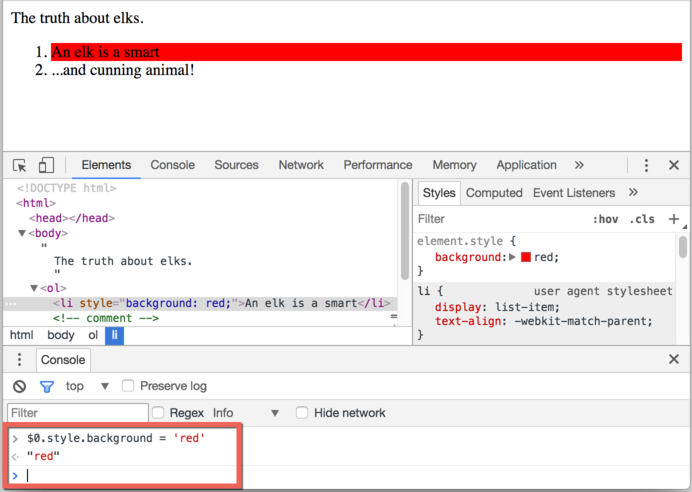
[**Взаимодействие с консолью**](https://learn.javascript.ru/dom-nodes#vzaimodeystvie-s-konsolyu)

При работе с DOM, часто требуется применить к нему JavaScript. Например: получить узел и запустить какой-нибудь код для его изменения, чтобы посмотреть результат. Вот несколько подсказок по тому, как перемещаться между вкладками Elements и Console. Для начала:

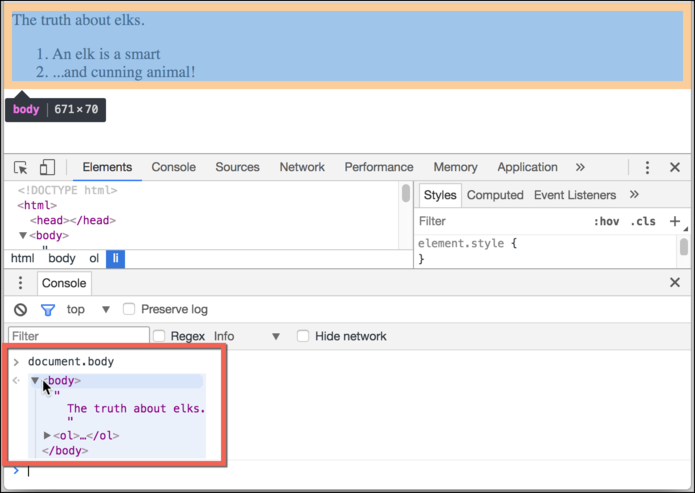
* На вкладке Elements выберите первый элемент <li>.
* Нажмите Esc – прямо под вкладкой Elements откроется Console.

Последний элемент, выбранный во вкладке Elements, доступен в консоли как $0, предыдущий, выбранный до него, как $1 и т.д.

Теперь можно запускать на них команды. Например $0.style.background = 'red' сделает выбранный элемент красным, как здесь:



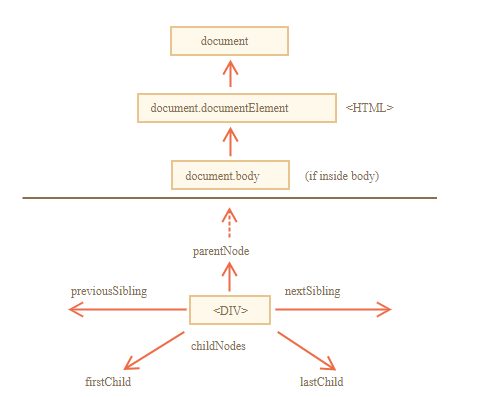
Так можно получить узел из Elements в Console. Есть и обратный путь: если есть переменная node, ссылающаяся на DOM-узел, можно использовать в консоли команду inspect(node), чтобы увидеть этот элемент во вкладке Elements. Или можно просто вывести DOM-узел в консоль и исследовать «на месте», как document.body ниже:



Это может быть полезно для отладки. Инструменты разработчика браузера отлично помогают в разработке: можно исследовать DOM, пробовать с ним что-то делать и смотреть, что идёт не так.

1. **Навигация и методы поиска DOM-элементов.**

DOM позволяет делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект. Все операции с DOM начинаются с объекта document. Это главная «точка входа» в DOM. Из него можно получить доступ к любому узлу. Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



Самые верхние элементы дерева доступны как свойства объекта document: <html> = document.documentElement. Самый верхний узел документа: document.documentElement. В DOM он соответствует тегу <html>.

<body> = document.body – другой часто используемый DOM-узел – узел тега <body>: document.body.

<head> = document.head – тег <head> доступен как document.head.

Есть одна тонкость: document.body может быть равен null. Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта. В частности, если скрипт находится в <head>, document.body в нём недоступен, потому что браузер его еще не прочитал. Поэтому, в примере ниже первый alert выведет null:

<html>

<head>

<script>

alert( "Из HEAD: " + document.body ); // null, <body> еще нет

</script>

</head>

<body>

<script>

alert( "Из BODY: " + document.body ); // HTMLBodyElement, теперь он есть

</script>

</body>

</html>

В DOM значение null значит «не существует» или «нет такого узла».

[**Дети: childNodes, firstChild, lastChild**](https://learn.javascript.ru/dom-navigation#deti-childnodes-firstchild-lastchild)

Здесь и далее будут использоваться два принципиально разных термина:

* Дочерние узлы (или дети) – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, <head> и <body> являются детьми элемента <html>.
* Потомки – все элементы, которые лежат внутри данного, включая детей, их детей и т.д.

В примере ниже детьми тега <body> являются теги <div> и <ul> (и несколько пустых текстовых узлов):

<html>

<body>

<div>Начало</div>

<ul>

<li>

<b>Информация</b>

</li>

</ul>

</body>

</html>

А потомки <body>– это и прямые дети <div>, <ul> и вложенные в них: <li> (потомок <ul>) и <b> (потомок <li>) – в общем, все элементы поддерева.

Коллекция childNodes содержит список всех детей, включая текстовые узлы. Пример ниже последовательно выведет детей document.body:

<html>

<body>

<div>Начало</div>

<ul>

<li>Информация</li>

</ul>

<div>Конец</div>

<script>

for (let i = 0; i < document.body.childNodes.length; i++) {

alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT

}

</script>

...какой-то HTML-код...

</body>

</html>

Обратим внимание на одну особенность. Если запустить пример выше, то последним будет выведен элемент <script>. На самом деле, в документе есть ещё «какой-то HTML-код», но на момент выполнения скрипта браузер ещё до него не дошёл, поэтому скрипт не видит его.

Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему дочернему элементу. Они, по сути, являются всего лишь сокращениями. Если у тега есть дочерние узлы, условие ниже всегда верно:

elem.childNodes[0] === elem.firstChild

elem.childNodes[elem.childNodes.length - 1] === elem.lastChild

Для проверки наличия дочерних узлов существует также специальная функция elem.hasChildNodes().

[**DOM-коллекции**](https://learn.javascript.ru/dom-navigation#dom-kollektsii)

childNodes это не массив, а коллекция – особый перебираемый объект-псевдомассив. И есть два важных следствия из этого:

1. Для перебора коллекции можно использовать for..of:

for (let node of document.body.childNodes) {

alert(node); // покажет все узлы из коллекции

}

Это работает, потому что коллекция является перебираемым объектом (есть требуемый для этого метод Symbol.iterator).

1. Методы массивов не будут работать, потому что коллекция – это не массив:

alert(document.body.childNodes.filter); // undefined (у коллекции нет метода filter)

Первый пункт – это хорошо для нас. Второй – бывает неудобен, но можно пережить. Если хочется использовать именно методы массива, то можно создать настоящий массив из коллекции, используя Array.from:

alert( Array.from(document.body.childNodes).filter ); // сделали массив

DOM-коллекции и все навигационные свойства, рассматриваемые в этой теме, доступны только для чтения. Нельзя заменить один дочерний узел на другой, просто написав childNodes[i] = .... Для изменения DOM требуются другие методы. Изучим их позже.

Почти все DOM-коллекции, за небольшим исключением, отражают текущее состояние DOM. Если сохранить ссылку на elem.childNodes и добавить/удалить узлы в DOM, то они появятся в сохраненной коллекции автоматически.

Коллекции перебираются циклом for..of. Не стоит использовать для этого цикл for..in. Цикл for..in перебирает все перечисляемые свойства. А у коллекций есть некоторые «лишние», редко используемые свойства, которые обычно не нужны:

<body>

<script>

// выводит 0, 1, length, item, values и другие свойства.

for (let prop in document.body.childNodes) alert(prop);

</script>

</body>

[**Соседи и родитель**](https://learn.javascript.ru/dom-navigation#sosedi-i-roditel)

Соседи – это узлы, у которых один и тот же родитель. Например, здесь <head> и <body> соседи:

<html>

<head>...</head><body>...</body>

</html>

Говорят, что <body> – «следующий» или «правый» сосед <head>. Также можно сказать, что <head> «предыдущий» или «левый» сосед <body>. Следующий узел того же родителя (следующий сосед) – в свойстве nextSibling, а предыдущий – в previousSibling. Родитель доступен через parentNode. Например:

// родителем <body> является <html>

alert( document.body.parentNode === document.documentElement ); // выведет true

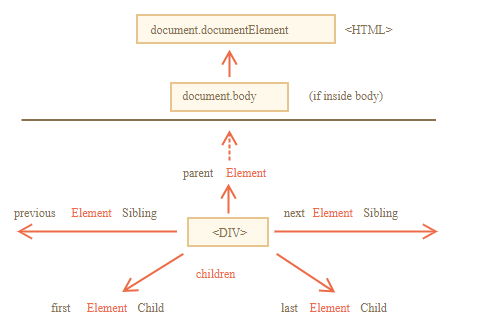
// после <head> идет <body>

alert( document.head.nextSibling ); // HTMLBodyElement

// перед <body> находится <head>

alert( document.body.previousSibling ); // HTMLHeadElement

Навигационные свойства, описанные выше, относятся ко всем узлам в документе. В частности, в childNodes находятся и текстовые узлы и узлы-элементы и узлы-комментарии, если они есть. Но для большинства задач текстовые узлы и узлы-комментарии не нужны. Как правило, надо манипулировать узлами-элементами, которые представляют собой теги и формируют структуру страницы. Поэтому рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы:



Эти ссылки похожи на те, что рассматривались раньше, только в ряде мест стоит слово Element:

* children – коллекция детей, которые являются элементами.
* firstElementChild, lastElementChild – первый и последний дочерний элемент.
* previousElementSibling, nextElementSibling – соседи-элементы.
* parentElement – родитель-элемент.

Свойство parentElement возвращает родитель-элемент, а parentNode возвращает «любого родителя». Обычно эти свойства одинаковы: они оба получают родителя. За исключением document.documentElement:

alert( document.documentElement.parentNode ); // выведет document

alert( document.documentElement.parentElement ); // выведет null

Причина в том, что родителем корневого узла document.documentElement (<html>) является document. Но document – это не узел-элемент, так что parentNode вернет его, а parentElement нет. Эта деталь может быть полезна, если надо пройти вверх по цепочке родителей от произвольного элемента elem к <html>, но не до document:

while(elem = elem.parentElement) { // идти наверх до <html>

alert( elem );

}

Изменим один из примеров выше: заменим childNodes на children. Теперь цикл выводит только элементы:

<html>

<body>

<div>Начало</div>

<ul>

<li>Информация</li>

</ul>

<div>Конец</div>

<script>

for (let elem of document.body.children) {

alert(elem); // DIV, UL, DIV, SCRIPT

}

</script>

...

</body>

</html>

До сих пор рассматривались основные навигационные ссылки. Некоторые типы DOM-элементов предоставляют для удобства дополнительные свойства, специфичные для их типа. Таблицы – отличный пример таких элементов. Элемент <table>, в дополнение к свойствам, о которых речь шла выше, поддерживает следующие:

* table.rows – коллекция строк <tr> таблицы;
* table.caption/tHead/tFoot – ссылки на элементы таблицы <caption>, <thead>, <tfoot>;
* table.tBodies – коллекция элементов таблицы <tbody> (по спецификации их может быть больше одного);
* <thead>, <tfoot>, <tbody> предоставляют свойство rows: tbody.rows – коллекция строк <tr> секции;
* <tr>:
* tr.cells – коллекция <td> и <th> ячеек, находящихся внутри строки <tr>;
* tr.sectionRowIndex – номер строки <tr> в текущей секции <thead>/<tbody>/<tfoot>;
* tr.rowIndex – номер строки <tr> в таблице (включая все строки таблицы);
* <td> and <th>: td.cellIndex – номер ячейки в строке <tr>.

Пример использования:

<table id="table">

<tr>

<td>один</td><td>два</td>

</tr>

<tr>

<td>три</td><td>четыре</td>

</tr>

</table>

<script>

// выводит содержимое первой строки, второй ячейки

alert( table.rows[0].cells[1].innerHTML ) // "два"

</script>

Существуют также дополнительные навигационные ссылки для HTML-форм. Они будут рассматриваться позже, при работе с формами.

**Методы поиска DOM-элементов.**

Свойства навигации по DOM удобны, когда элементы расположены рядом. Чтобы получить произвольный элемент страницы, можно использовать дополнительные методы поиска.

[**document.getElementById или просто id**](https://learn.javascript.ru/searching-elements-dom#document-getelementbyid-ili-prosto-id)

Когда у элемента есть атрибут id, значение атрибута используется в качестве имени глобальной переменной. С её помощью можно обратиться к элементу напрямую (если в имени переменной используется дефис `-`, то она доступна через квадратные скобки [...]):

<div id="elem">

<div id="elem-content">Элемент</div>

</div>

<script>

alert(elem); // DOM-элемент с id="elem"

alert(window.elem); // доступ к глобальной переменной тоже работает

alert(window['elem-content']);

</script>

Это поведение соответствует [стандарту](http://www.whatwg.org/specs/web-apps/current-work/#dom-window-nameditem), но поддерживается в основном для совместимости. Браузер пытается помочь, смешивая пространства имён JS и DOM. Это подходит для простого кода, но возможны конфликты. Если осуществляется работа с JS-кодом, не видя HTML, не очевидно, откуда возьмётся переменная. Если объявить переменную с тем же именем, она будет иметь приоритет:

<div id="elem"></div>

<script>

let elem = 5;

alert(elem); // 5

</script>

Лучшая альтернатива – использовать специальный метод document.getElementById(id). Например:

<div id="elem">

<div id="elem-content">Элемент</div>

</div>

<script>

let elem = document.getElementById('elem');

elem.style.background = 'red';

</script>

Далее в примерах часто будет использоваться прямое обращение через id, но это только для краткости. В реальных проектах предпочтителен метод document.getElementById.

Значение id должно быть уникальным. В документе может быть только один элемент с данным id. Если в документе есть несколько элементов с одинаковым значением id, то поведение методов поиска непредсказуемо. Браузер может вернуть любой из них случайным образом. Поэтому придерживайтесь правила сохранения уникальности id.

Метод getElementById можно вызвать только для объекта document. Он осуществляет поиск по id по всему документу.

**Методы** [**querySelectorAll**](https://learn.javascript.ru/searching-elements-dom#querySelectorAll) **и** [**querySelector**](https://learn.javascript.ru/searching-elements-dom#querySelector)

Самый универсальный метод поиска – это elem.querySelectorAll(css), он возвращает все элементы внутри elem, удовлетворяющие данному CSS-селектору. Следующий запрос получает все элементы <li>, которые являются последними потомками в <ul>:

<ul>

<li>Этот</li>

<li>тест</li>

</ul>

<ul>

<li>полностью</li>

<li>пройден</li>

</ul>

<script>

let elements = document.querySelectorAll('ul > li:last-child');

for (let elem of elements) {

alert(elem.innerHTML); // "тест", "пройден"

}

</script>

С этим методом можно использовать любой CSS-селектор. Псевдоклассы в CSS-селекторе, в частности :hover и :active, также поддерживаются. Например, document.querySelectorAll(':hover') вернёт коллекцию (в порядке вложенности: от внешнего к внутреннему) из текущих элементов под курсором мыши.

Метод elem.querySelector(css) возвращает первый элемент, соответствующий данному CSS-селектору. Иначе говоря, результат такой же, как при вызове elem.querySelectorAll(css)[0], но он сначала найдёт все элементы, а потом возьмёт первый, в то время как elem.querySelector найдёт только первый и остановится. Это быстрее, кроме того, его короче писать.

**Метод** [**matches**](https://learn.javascript.ru/searching-elements-dom#matches)

Предыдущие методы искали по DOM. Метод [elem.matches(css)](http://dom.spec.whatwg.org/" \l "dom-element-matches) ничего не ищет, а проверяет, удовлетворяет ли elem CSS-селектору, и возвращает true или false. Этот метод удобен, когда надо перебрать элементы (например, в массиве или в чём-то подобном) и выбрать те из них, которые интересуют. Например:

<a href="http://example.com/file.zip">...</a>

<a href="http://ya.ru">...</a>

<script>

// может быть любая коллекция вместо document.body.children

for (let elem of document.body.children) {

if (elem.matches('a[href$="zip"]')) {

alert("Ссылка на архив: " + elem.href );

}

}

</script>

**Метод** [**closest**](https://learn.javascript.ru/searching-elements-dom#closest)

Предки элемента – родитель, родитель родителя, его родитель и так далее. Вместе они образуют цепочку иерархии от элемента до вершины. Метод elem.closest(css) ищет ближайшего предка, который соответствует CSS-селектору. Сам элемент также включается в поиск. Другими словами, метод closest поднимается вверх от элемента и проверяет каждого из родителей. Если он соответствует селектору, поиск прекращается. Метод возвращает либо предка, либо null, если такой элемент не найден. Например:

<h1>Содержание</h1>

<div class="contents">

<ul class="book">

<li class="chapter">Глава 1</li>

<li class="chapter">Глава 2</li>

</ul>

</div>

<script>

let chapter = document.querySelector('.chapter'); // LI

alert(chapter.closest('.book')); // UL

alert(chapter.closest('.contents')); // DIV

alert(chapter.closest('h1')); // null (потому что h1 - не предок)

</script>

**Методы** [**getElementsBy\***](https://learn.javascript.ru/searching-elements-dom#getelementsby)

Существуют также другие методы поиска элементов по тегу, классу и так далее. На данный момент, они скорее исторические, так как querySelector более чем эффективен. Здесь рассмотрим их для полноты картины, также можно встретить их в старом коде.

* elem.getElementsByTagName(tag) ищет элементы с данным тегом и возвращает их коллекцию. Передав "\*" вместо тега, можно получить всех потомков.
* elem.getElementsByClassName(className) возвращает элементы, которые имеют данный CSS-класс.
* document.getElementsByName(name) возвращает элементы с заданным атрибутом name. Очень редко используется.

Например, получить все элементы div в документе:

let divs = document.getElementsByTagName('div');

Найдём все input в таблице:

<table id="table">

<tr>

<td>Ваш возраст:</td>

<td>

<label>

<input type="radio" name="age" value="young" checked> младше 18

</label>

<label>

<input type="radio" name="age" value="mature"> от 18 до 50

</label>

<label>

<input type="radio" name="age" value="senior"> старше 60

</label>

</td>

</tr>

</table>

<script>

let inputs = table.getElementsByTagName('input');

for (let input of inputs) {

alert( input.value + ': ' + input.checked );

}

</script>

Одна из самых частых ошибок – это забыть про букву "s". То есть пробовать вызывать метод getElementByTagName вместо getElement**s**ByTagName. Буква "s" отсутствует в названии метода getElementById, так как в данном случае возвращает один элемент. Но getElementsByTagName вернёт список элементов, поэтому "s" обязательна.

Метод возвращает коллекцию, а не элемент. Поэтому, другая распространённая ошибка – написать:

// не работает

document.getElementsByTagName('input').value = 5;

Попытка присвоить значение коллекции, а не элементам внутри неё, не сработает. Нужно перебрать коллекцию в цикле или получить элемент по номеру и уже ему присваивать значение, например, так:

// работает (если есть input)

document.getElementsByTagName('input')[0].value = 5;

Ищем элементы с классом .article:

<form name="my-form">

<div class="article">Article</div>

<div class="long article">Long article</div>

</form>

<script>

// поиск по имени атрибута

let form = document.getElementsByName('my-form')[0];

// поиск по классу внутри form

let articles = form.getElementsByClassName('article');

alert(articles.length); // 2, найдены два элемента с классом article

</script>

Все методы "getElementsBy\*" возвращают живую коллекцию. Такие коллекции всегда отражают текущее состояние документа и автоматически обновляются при его изменении. В приведённом ниже примере есть два скрипта.

1. Первый создаёт ссылку на коллекцию <div>. На этот момент её длина равна 1.
2. Второй скрипт запускается после того, как браузер встречает ещё один <div>, теперь её длина – 2.

<div>First div</div>

<script>

let divs = document.getElementsByTagName('div');

alert(divs.length); // 1

</script>

<div>Second div</div>

<script>

alert(divs.length); // 2

</script>

Напротив, querySelectorAll возвращает статическую коллекцию. Это похоже на фиксированный массив элементов. Если использовать его в примере выше, то оба скрипта вернут длину коллекции, равную 1:

<div>First div</div>

<script>

let divs = document.querySelectorAll('div');

alert(divs.length); // 1

</script>

<div>Second div</div>

<script>

alert(divs.length); // 1

</script>

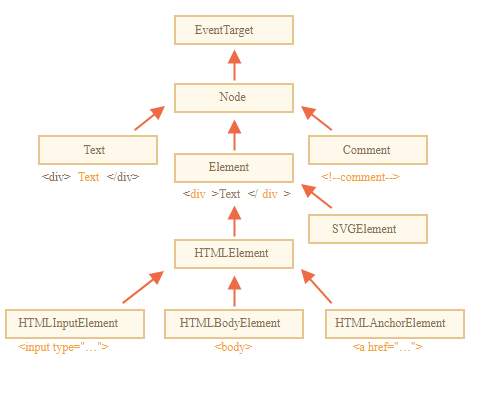
Теперь видна разница. Длина статической коллекции не изменилась после появления нового div в документе.

1. **Свойства узлов: тип, тег и содержимое.**

Разберём подробнее, что собой представляют DOM-узлы и изучим их основные свойства.

[**Классы DOM-узлов**](https://learn.javascript.ru/basic-dom-node-properties#klassy-dom-uzlov)

У разных DOM-узлов могут быть разные свойства. Например, у узла, соответствующего тегу <a>, есть свойства, связанные со ссылками, а у соответствующего тегу <input> – свойства, связанные с полем ввода и т.д. Текстовые узлы отличаются от узлов-элементов. Но у них есть общие свойства и методы, потому что все классы DOM-узлов образуют единую иерархию. Каждый DOM-узел принадлежит соответствующему встроенному классу. Корнем иерархии является [EventTarget](https://dom.spec.whatwg.org/" \l "eventtarget), от него наследует [Node](http://dom.spec.whatwg.org/" \l "interface-node) и остальные DOM-узлы. На рисунке ниже изображены основные классы:



Существуют следующие классы:

* [EventTarget](https://dom.spec.whatwg.org/#eventtarget) – это корневой «абстрактный» класс. Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события», о которых рассмотрим позже.
* [Node](http://dom.spec.whatwg.org/#interface-node) – также является «абстрактным» классом, и служит основой для DOM-узлов. Он обеспечивает базовую функциональность: parentNode, nextSibling, childNodes и т.д. (это геттеры). Объекты класса Node никогда не создаются. Но есть определенные классы узлов, которые наследуют от него: Text – для текстовых узлов, Element – для узлов-элементов и более экзотический Comment – для узлов-комментариев.
* [Element](http://dom.spec.whatwg.org/#interface-element) – это базовый класс для DOM-элементов. Он обеспечивает навигацию на уровне элементов: nextElementSibling, children и методы поиска: getElementsByTagName, querySelector. Браузер поддерживает не только HTML, но также XML и SVG. Класс Element служит базой для следующих классов: SVGElement, XMLElement и HTMLElement.
* [HTMLElement](https://html.spec.whatwg.org/multipage/dom.html#htmlelement) – является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
* [HTMLInputElement](https://html.spec.whatwg.org/multipage/forms.html#htmlinputelement) – класс для тега <input>,
* [HTMLBodyElement](https://html.spec.whatwg.org/multipage/semantics.html#htmlbodyelement) – класс для тега <body>,
* [HTMLAnchorElement](https://html.spec.whatwg.org/multipage/semantics.html#htmlanchorelement) – класс для тега <a>,
* …и т.д, каждому тегу соответствует свой класс, который предоставляет определенные свойства и методы.

Таким образом, полный набор свойств и методов данного узла собирается в результате наследования. Рассмотрим DOM-объект для тега <input>. Он принадлежит классу [HTMLInputElement](https://html.spec.whatwg.org/multipage/forms.html" \l "htmlinputelement). Он получает свойства и методы из (в порядке наследования):

* HTMLInputElement – этот класс предоставляет специфичные для элементов формы свойства;
* HTMLElement – предоставляет общие для HTML-элементов методы (и геттеры/сеттеры);
* Element – предоставляет типовые методы элемента;
* Node – предоставляет общие свойства DOM-узлов;
* EventTarget – обеспечивает поддержку событий (поговорим о них дальше);
* Object – доступны методы «обычного объекта», такие как hasOwnProperty.

Для того, чтобы узнать имя класса DOM-узла, вспомним, что обычно у объекта есть свойство constructor. Оно ссылается на конструктор класса, и в свойстве constructor.name содержится его имя:

alert( document.body.constructor.name ); // HTMLBodyElement

Также можно просто привести его к строке:

alert( document.body ); // [object HTMLBodyElement]

Проверить наследование можно также при помощи instanceof:

alert( document.body instanceof HTMLBodyElement ); // true

alert( document.body instanceof HTMLElement ); // true

alert( document.body instanceof Element ); // true

alert( document.body instanceof Node ); // true

alert( document.body instanceof EventTarget ); // true

Как видно, DOM-узлы – это обычные JavaScript объекты. Для наследования они используют классы, основанные на прототипах. В этом легко убедиться, если вывести в консоли браузера любой элемент через console.dir(elem). Или даже напрямую обратиться к методам, которые хранятся в HTMLElement.prototype, Element.prototype и т.д.

**console.dir(elem) и console.log(elem)**

Большинство браузеров поддерживают в инструментах разработчика две команды: console.log и console.dir. Они выводят свои аргументы в консоль. Для JavaScript-объектов эти команды обычно выводят одно и то же. Но для DOM-элементов они работают по-разному: console.log(elem) выводит элемент в виде DOM-дерева; console.dir(elem) выводит элемент в виде DOM-объекта, что удобно для анализа его свойств.

Чтобы узнать тип узла, можно использовать метод instanceof и другие способы проверить класс, но иногда nodeType проще использовать. Нельзя изменить значение nodeType, только прочитать его.

[**Свойства nodeName и tagName**](https://learn.javascript.ru/basic-dom-node-properties#teg-nodename-i-tagname)

Получив DOM-узел, можно узнать имя его тега из свойств nodeName и tagName. Например:

alert( document.body.nodeName ); // BODY

alert( document.body.tagName ); // BODY

Разница между tagName и nodeName заключается в следующем:

* свойство tagName есть только у элементов Element;
* свойство nodeName определено для любых узлов Node:
* для элементов оно равно tagName;
* для остальных типов узлов (текст, комментарий и т.д.) оно содержит строку с типом узла.

Другими словами, свойство tagName есть только у узлов-элементов (поскольку они происходят от класса Element), а nodeName может что-то сказать о других типах узлов. Например, сравним tagName и nodeName на примере объекта document и узла-комментария:

<body><!-- комментарий -->

<script>

// для комментария

alert( document.body.firstChild.tagName ); // undefined (не элемент)

alert( document.body.firstChild.nodeName ); // #comment

// for document

alert( document.tagName ); // undefined (не элемент)

alert( document.nodeName ); // #document

</script>

</body>

Если нужны только элементами, то можно использовать tagName или nodeName, нет разницы.

В браузере существуют два режима обработки документа: HTML и XML. HTML-режим обычно используется для веб-страниц. XML-режим включается, если браузер получает XML-документ с заголовком: Content-Type: application/xml+xhtml.

В HTML-режиме значения tagName/nodeName всегда записаны в верхнем регистре. Будет выведено BODY вне зависимости от того, как записан тег в HTML <body> или <BoDy>.

В XML-режиме регистр сохраняется «как есть». В настоящее время XML-режим применяется редко.

**Свойство [innerHTML](https://w3c.github.io/DOM-Parsing/" \l "widl-Element-innerHTML)**

Свойство [innerHTML](https://w3c.github.io/DOM-Parsing/" \l "widl-Element-innerHTML) позволяет получить HTML-содержимое элемента в виде строки. Также можно изменять его. Это один из самых мощных способов менять содержимое на странице. Пример ниже показывает содержимое document.body, а затем полностью заменяет его:

<body>

<p>Параграф</p>

<div>DIV</div>

<script>

alert( document.body.innerHTML ); // читаем текущее содержимое

document.body.innerHTML = 'Новый BODY!'; // заменяем содержимое

</script>

</body>

Можно вставить некорректный HTML, браузер исправит ошибки:

<body>

<script>

document.body.innerHTML = '<b>тест'; // не закрыт тег

alert( document.body.innerHTML ); // <b>тест</b> (исправлено)

</script>

</body>

Если innerHTML вставляет в документ тег <script> – он становится частью HTML, но не запускается.

Можно добавить HTML к элементу, используя elem.innerHTML+="ещё html":

chatDiv.innerHTML += "<div>Привет<img src='smile.gif'/> !</div>";

chatDiv.innerHTML += "Как дела?";

На практике этим следует пользоваться с большой осторожностью, так как фактически происходит не добавление, а перезапись. Технически эти две строки делают одно и то же:

elem.innerHTML += "...";

// это более короткая запись для:

elem.innerHTML = elem.innerHTML + "..."

Другими словами, innerHTML+= делает следующее:

1. Старое содержимое удаляется.
2. На его место становится новое значение innerHTML (с добавленной строкой).

Так как содержимое «обнуляется» и переписывается заново, все изображения и другие ресурсы будут перезагружены. В примере chatDiv выше строка chatDiv.innerHTML+="Как дела?" заново создаёт содержимое HTML и перезагружает smile.gif. Если в chatDiv много текста и изображений, то эта перезагрузка будет очень заметна.

Есть и другие побочные эффекты. Например, если существующий текст выделен мышкой, то при переписывании innerHTML большинство браузеров снимут выделение. А если это поле ввода <input> с текстом, введенным пользователем, то текст будет удалён.

**Свойство outerHTML**

Свойство outerHTML содержит HTML элемента целиком. Это как innerHTML плюс сам элемент. Рассмотрим пример:

<div id="elem">Привет <b>Мир</b></div>

<script>

alert(elem.outerHTML); // <div id="elem">Привет <b>Мир</b></div>

</script>

Будьте осторожны: в отличие от innerHTML, запись в outerHTML не изменяет элемент. Вместо этого элемент заменяется целиком во внешнем контексте. Рассмотрим пример:

<div>Привет, мир!</div>

<script>

let div = document.querySelector('div');

div.outerHTML = '<p>Новый элемент</p>'; // (\*)

alert(div.outerHTML); // <div>Привет, мир!</div> (\*\*)

</script>

В строке (\*) div заменен  на <p>Новый элемент</p>. Во внешнем документе располагается новое содержимое вместо <div>. Но, как видно в строке (\*\*), старая переменная div осталась прежней. Это потому, что использование outerHTML не изменяет DOM-элемент, а удаляет его из внешнего контекста и вставляет вместо него новый HTML-код. То есть, при div.outerHTML=... произошло следующее:

* div был удалён из документа;
* вместо него был вставлен другой HTML <p>A new element</p>;
* в div осталось старое значение. Новый HTML не схранён ни в какой переменной.

Здесь легко сделать ошибку: заменить div.outerHTML, а потом продолжить работать с div, как будто там новое содержимое. Но это не так. Подобное верно для innerHTML, но не для outerHTML.

Можно писать в elem.outerHTML, но это не меняет элемент, в который пишем. Вместо этого создается новый HTML на его месте. Можно получить ссылки на новые элементы, обратившись к DOM.

**Свойства nodeValue и data**

Свойство innerHTML есть только у узлов-элементов. У других типов узлов, в частности, у текстовых, есть свои аналоги: свойства nodeValue и data. Эти свойства очень похожи при использовании, есть лишь небольшие различия в спецификации. Будем использовать data, потому что оно короче. Прочитаем содержимое текстового узла и комментария:

<body>

Привет

<!-- Комментарий -->

<script>

let text = document.body.firstChild;

alert(text.data); // Привет

let comment = text.nextSibling;

alert(comment.data); // Комментарий

</script>

</body>

Иногда комментарии используют для вставки информации и инструкций шаблонизатора в HTML, как в примере ниже:

<!-- if isAdmin -->

<div>Добро пожаловать, Admin!</div>

<!-- /if -->

Затем JavaScript может прочитать это из свойства data и обработать инструкции.

**Свойство textContent**

Свойство textContent предоставляет доступ к тексту внутри элемента за вычетом всех <тегов>. Например:

<div id="news">

<h1>Срочно в номер!</h1>

<p>Марсиане атаковали человечество!</p>

</div>

<script>

// Срочно в номер! Марсиане атаковали человечество!

alert(news.textContent);

</script>

Возвращается только текст, как если бы все <теги> были вырезаны, но текст в них остался. На практике редко появляется необходимость читать текст таким образом. Намного полезнее возможность записывать текст в свойство textContent, т.к. оно позволяет писать текст «безопасным способом».

Допустим есть произвольная строка, введённая пользователем, и надо показать её. С innerHTML вставка происходит «как HTML», со всеми HTML-тегами. С textContent вставка получается «как текст», все символы трактуются буквально. Сравним два тега div:

<div id="elem1"></div>

<div id="elem2"></div>

<script>

let name = prompt("Введите ваше имя?", "<b>Винни-пух!</b>");

elem1.innerHTML = name;

elem2.textContent = name;

</script>

1. В первый <div> имя приходит «как HTML»: все теги стали именно тегами, поэтому отображается имя, выделенное жирным шрифтом.
2. Во второй <div> имя приходит «как текст», поэтому отображается <b>Винни-пух!</b>.

В большинстве случаев надо получить от пользователя текст и чтобы он интерпретировался как текст. Не надо, чтобы на сайте появлялся произвольный HTML-код. Присваивание через textContent – один из способов от этого защититься.

[**Свойство «hidden»**](https://learn.javascript.ru/basic-dom-node-properties#svoystvo-hidden)

Атрибут и DOM-свойство «hidden» указывает на то, виден ли элемент или нет. Можно использовать его в HTML или назначать при помощи JavaScript, как в примере ниже:

<div>Оба тега DIV внизу невидимы</div>

<div hidden>С атрибутом "hidden"</div>

<div id="elem">С назначенным JavaScript свойством "hidden"</div>

<script>

elem.hidden = true;

</script>

Технически, hidden работает так же, как style="display:none". Но его применение проще. Мигающий элемент:

<div id="elem">Мигающий элемент</div>

<script>

setInterval(() => elem.hidden = !elem.hidden, 1000);

</script>

[**Другие свойства**](https://learn.javascript.ru/basic-dom-node-properties#drugie-svoystva)

У DOM-элементов есть дополнительные свойства, в частности, зависящие от класса:

* value – значение для <input>, <select> и <textarea> (HTMLInputElement, HTMLSelectElement и др.);
* href – адрес ссылки «href» для <a href="..."> (HTMLAnchorElement);
* id – значение атрибута «id» для всех элементов (HTMLElement) и многие другие.

Например:

<input type="text" id="elem" value="значение">

<script>

alert(elem.type); // "text"

alert(elem.id); // "elem"

alert(elem.value); // значение

</script>

Большинство стандартных HTML-атрибутов имеют соответствующее DOM-свойство, и можно получить к нему доступ. Если надо узнать полный список поддерживаемых свойств для данного класса, можно найти их в спецификации. Например, класс HTMLInputElement описывается здесь: <https://html.spec.whatwg.org/#htmlinputelement>. Также можно вывести элемент в консоль, используя console.dir(elem), и прочитать все свойства. Или исследовать «свойства DOM» во вкладке Elements браузерных инструментов разработчика.

1. **Атрибуты и DOM-свойства.**

Когда браузер загружает страницу, он «читает» («парсит») HTML и генерирует из него DOM-объекты. Для узлов-элементов большинство стандартных HTML-атрибутов автоматически становятся свойствами DOM-объектов. Например, для такого тега <body id="page"> у DOM-объекта будет такое свойство body.id="page".

Но преобразование атрибута в свойство происходит не один-в-один.

[**DOM-свойства**](https://learn.javascript.ru/dom-attributes-and-properties#dom-svoystva)

Ранее рассматривались встроенные DOM-свойства. Но можно добавить своё собственное свойство. DOM-узлы – это обычные объекты JavaScript. Можно их изменять. Например, создадим новое свойство для document.body:

document.body.myData = {

name: 'Caesar',

title: 'Imperator'

};

alert(document.body.myData.title); // Imperator

Можно добавить и метод:

document.body.sayTagName = function() {

alert(this.tagName);

};

document.body.sayTagName(); // BODY (this = document.body)

Также можно изменять встроенные прототипы, такие как Element.prototype и добавлять новые методы ко всем элементам:

Element.prototype.sayHi = function() {

alert(`Hello, I'm ${this.tagName}`);

};

document.documentElement.sayHi(); // Hello, I'm HTML

document.body.sayHi(); // Hello, I'm BODY

Итак, DOM-свойства и методы ведут себя так же, как и обычные объекты JavaScript: им можно присвоить любое значение; они регистрозависимы (нужно писать elem.nodeType, не elem.NoDeTyPe).

[**HTML-атрибуты**](https://learn.javascript.ru/dom-attributes-and-properties#html-atributy)

В HTML у тегов могут быть атрибуты. Когда браузер парсит HTML, чтобы создать DOM-объекты для тегов, он распознаёт стандартные атрибуты и создаёт DOM-свойства для них. Таким образом, когда у элемента есть id или другой стандартный атрибут, создаётся соответствующее свойство. Но этого не происходит, если атрибут нестандартный. Например:

<body id="test" something="non-standard">

<script>

alert(document.body.id); // test

alert(document.body.something); // undefined

</script>

</body>

Пожалуйста, учтите, что стандартный атрибут для одного тега может быть нестандартным для другого. Например, атрибут "type" является стандартным для элемента <input> ([HTMLInputElement](https://html.spec.whatwg.org/" \l "htmlinputelement)), но не является стандартным для <body>([HTMLBodyElement](https://html.spec.whatwg.org/" \l "htmlbodyelement)). Стандартные атрибуты описаны в спецификации для соответствующего класса элемента. Можно увидеть это на примере ниже:

<body id="body" type="...">

<input id="input" type="text">

<script>

alert(input.type); // text

alert(body.type); // undefined

</script>

</body>

Таким образом, для нестандартных атрибутов не будет соответствующих DOM-свойств. Все атрибуты доступны с помощью следующих методов:

* elem.hasAttribute(name) – проверяет наличие атрибута;
* elem.getAttribute(name) – получает значение атрибута;
* elem.setAttribute(name, value) – устанавливает значение атрибута;
* elem.removeAttribute(name) – удаляет атрибут.

Этим методы работают именно с тем, что написано в HTML.

Кроме этого, получить все атрибуты элемента можно с помощью свойства elem.attributes: коллекция объектов, которая принадлежит ко встроенному классу [Attr](https://dom.spec.whatwg.org/" \l "attr) со свойствами name и value. Вот демонстрация чтения нестандартного свойства:

<body something="non-standard">

<script>

alert(document.body.getAttribute('something')); // non-standard

</script>

</body>

У HTML-атрибутов есть следующие особенности: их имена регистронезависимы (id то же самое, что и ID); их значения всегда являются строками. Расширенная демонстрация работы с атрибутами:

<body>

<div id="elem" about="Elephant"></div>

<script>

alert( elem.getAttribute('About') ); // (1) 'Elephant', чтение

elem.setAttribute('Test', 123); // (2), запись

alert( elem.outerHTML ); // (3), посмотрим, есть ли атрибут в HTML (да)

for (let attr of elem.attributes) { // (4) весь список

alert( `${attr.name} = ${attr.value}` );

}

</script>

</body>

Обратите внимание:

1. getAttribute('About') – здесь первая буква заглавная, а в HTML – строчная. Но это не важно: имена атрибутов регистронезависимы.
2. Можно присвоить что угодно атрибуту, но это станет строкой. Поэтому в этой строчке получаем значение "123".
3. Все атрибуты, в том числе те, которые были установлены, видны в outerHTML.
4. Коллекция attributes является перебираемой. В ней есть все атрибуты элемента (стандартные и нестандартные) в виде объектов со свойствами name и value.

[**Синхронизация между атрибутами и свойствами**](https://learn.javascript.ru/dom-attributes-and-properties#sinhronizatsiya-mezhdu-atributami-i-svoystvami)

Когда стандартный атрибут изменяется, соответствующее свойство автоматически обновляется. Это работает и в обратную сторону (за некоторыми исключениями). В примере ниже id модифицируется как атрибут, и можно увидеть, что свойство также изменено. То же самое работает и в обратную сторону:

<input>

<script>

let input = document.querySelector('input');

// атрибут => свойство

input.setAttribute('id', 'id');

alert(input.id); // id (обновлено)

// свойство => атрибут

input.id = 'newId';

alert(input.getAttribute('id')); // newId (обновлено)

</script>

Но есть и исключения, например, input.value синхронизируется только в одну сторону – атрибут → значение, но не в обратную:

<input>

<script>

let input = document.querySelector('input');

// атрибут => значение

input.setAttribute('value', 'text');

alert(input.value); // text

// свойство => атрибут

input.value = 'newValue';

alert(input.getAttribute('value')); // text

</script>

В примере выше изменение атрибута value обновило свойство, но изменение свойства не повлияло на атрибут. Иногда эта «особенность» может пригодиться, потому что действия пользователя могут приводить к изменениям value, и если после этого надо восстановить «оригинальное» значение из HTML, оно будет в атрибуте.

DOM-свойства не всегда являются строками. Например, свойство input.checked (для чекбоксов) имеет логический тип:

<input id="input" type="checkbox" checked> checkbox

<script>

alert(input.getAttribute('checked')); // значение атрибута: пустая строка

alert(input.checked); // значение свойства: true

</script>

Есть и другие примеры. Атрибут style – строка, но свойство style является объектом:

<div id="div" style="color:red;font-size:120%">Hello</div>

<script>

// строка

alert(div.getAttribute('style')); // color:red;font-size:120%

// объект

alert(div.style); // [object CSSStyleDeclaration]

alert(div.style.color); // red

</script>

Большинство свойств, всё же, строки, но могут отличаться от атрибутов. Например, DOM-свойство href всегда содержит полный URL, даже если атрибут содержит относительный URL или просто #hash. Ниже пример:

<a id="a" href="#hello">link</a>

<script>

// атрибут

alert(a.getAttribute('href')); // #hello

// свойство

alert(a.href ); // полный URL в виде http://site.com/page#hello

</script>

Если же нужно значение href или любого другого атрибута в точности, как оно записано в HTML, можно воспользоваться getAttribute.

[**Нестандартные атрибуты, dataset**](https://learn.javascript.ru/dom-attributes-and-properties#nestandartnye-atributy-dataset)

Нестандартные атрибуты используются для передачи пользовательских данных из HTML в JavaScript, или чтобы «помечать» HTML-элементы для JavaScript. Пример:

<!-- пометить div, чтобы показать здесь поле "name" -->

<div show-info="name"></div>

<!-- а здесь возраст "age" -->

<div show-info="age"></div>

<script>

// код находит элемент с пометкой и показывает запрошенную информацию

let user = {

name: "Pete",

age: 25

};

for(let div of document.querySelectorAll('[show-info]')) {

// вставить соответствующую информацию в поле

let field = div.getAttribute('show-info');

div.innerHTML = user[field]; // сначала Pete в name, потом 25 в age

}

</script>

Также они могут быть использованы, чтобы стилизовать элементы. Например, здесь для состояния заказа используется атрибут order-state:

<style>

/\* стили зависят от пользовательского атрибута "order-state" \*/

.order[order-state="new"] {

color: green;

}

.order[order-state="pending"] {

color: blue;

}

.order[order-state="canceled"] {

color: red;

}

</style>

<div class="order" order-state="new">

A new order.

</div>

<div class="order" order-state="pending">

A pending order.

</div>

<div class="order" order-state="canceled">

A canceled order.

</div>

Атрибут могут быть предпочтительнее таких классов, как .order-state-new, .order-state-pending, order-state-canceled, потому, что атрибутом удобнее управлять. Состояние может быть изменено достаточно просто (не надо удалять старый/добавлять новый класса):

div.setAttribute('order-state', 'canceled');

Но с пользовательскими атрибутами могут возникнуть проблемы, например, если был использован нестандартный атрибут, а позже он появится в стандарте и будет выполнять какую-то функцию.

Чтобы избежать конфликтов, существуют атрибуты вида [data-\*](https://html.spec.whatwg.org/" \l "embedding-custom-non-visible-data-with-the-data-*-attributes). Все атрибуты, начинающиеся с префикса «data-», зарезервированы для использования программистами. Они доступны в свойстве dataset. Например, если у elem есть атрибут "data-about", то обратиться к нему можно как elem.dataset.about. Например:

<body data-about="Elephants">

<script>

alert(document.body.dataset.about); // Elephants

</script>

Атрибуты, состоящие из нескольких слов, к примеру data-order-state, становятся свойствами, записанными с помощью верблюжьей нотации: dataset.orderState. Вот переписанный пример «состояния заказа»:

<style>

.order[data-order-state="new"] {

color: green;

}

.order[data-order-state="pending"] {

color: blue;

}

.order[data-order-state="canceled"] {

color: red;

}

</style>

<div id="order" class="order" data-order-state="new">

A new order.

</div>

<script>

// чтение

alert(order.dataset.orderState); // new

// изменение

order.dataset.orderState = "pending"; // (\*)

</script>

Использование data-\* атрибутов – валидный, безопасный способ передачи пользовательских данных. Можно не только читать, но и изменять data-атрибуты. Тогда CSS обновит представление соответствующим образом: в примере выше последняя строка (\*) меняет цвет на синий.

1. **Добавление и удаление DOM-узлов.**

Рассмотрим пример, добавим на страницу сообщение:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<div class="alert">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>

Это был пример HTML. Теперь создадим такой же div, используя JavaScript (предполагаем, что стили в HTML или во внешнем CSS-файле).

[**Создание элемента**](https://learn.javascript.ru/modifying-document#sozdanie-elementa)

DOM-узел можно создать двумя методами:

* document.createElement(tag) – создаёт новый элемент с заданным тегом:

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

* document.createTextNode(text) – создаёт новый текстовый узел с заданным текстом:

let textNode = document.createTextNode('А вот и я');

В рассматриваемом примере нужно создать сообщение – это div с классом alert и HTML в нём:

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

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

Элемент создан, но пока он только в переменной. Нельзя пока увидеть его на странице, поскольку он не является частью документа.

[**Методы вставки**](https://learn.javascript.ru/modifying-document#metody-vstavki)

Чтобы div появился на странице, нужно вставить его где-нибудь в document. Например, в document.body. Для этого есть метод append, в нашем случае: document.body.append(div). Вот полный пример:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

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

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

document.body.append(div);

</script>

Метод для различных вариантов вставки:

* node.append(...узлы или строки) – добавляет узлы или строки в конец node;
* node.prepend(...nodes or strings) – вставляет узлы или строки в начало node;
* node.before(...nodes or strings) –- вставляет узлы или строки до node;
* node.after(...nodes or strings) –- вставляет узлы или строки после node;
* node.replaceWith(...nodes or strings) –- заменяет node заданными узлами или строками.

Вот пример использования этих методов, чтобы добавить новые элементы в список и текст до/после него:

<ol id="ol">

<li>0</li>

<li>1</li>

<li>2</li>

</ol>

<script>

ol.before('before'); // вставить строку "before" перед <ol>

ol.after('after'); // вставить строку "after" после <ol>

let liFirst = document.createElement('li');

liFirst.innerHTML = 'prepend';

ol.prepend(liFirst); // вставить liFirst в начало <ol>

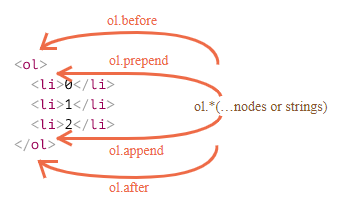
let liLast = document.createElement('li');

liLast.innerHTML = 'append';

ol.append(liLast); // вставить liLast в конец <ol>

</script>

Наглядная иллюстрация того, куда эти методы вставляют:



Итоговый список будет таким:

before

<ol id="ol">

<li>prepend</li>

<li>0</li>

<li>1</li>

<li>2</li>

<li>append</li>

</ol>

after

Эти методы могут вставлять несколько узлов и текстовых фрагментов за один вызов. Например, здесь вставляется строка и элемент:

<div id="div"></div>

<script>

div.before('<p>Привет</p>', document.createElement('hr'));

</script>

Весь текст вставляется как текст. Поэтому финальный HTML будет таким:

&lt;p&gt;Привет&lt;/p&gt;

<hr>

<div id="div"></div>

Другими словами, строки вставляются безопасным способом, как делает это elem.textContent. Поэтому эти методы могут использоваться только для вставки DOM-узлов или текстовых фрагментов.

Если надо вставить HTML именно «как html», со всеми тегами и прочим, как делает это elem.innerHTML, тогда надо использовать другой универсальный метод: elem.insertAdjacentHTML(where, html).

Первый параметр – это специальное слово, указывающее, куда по отношению к elem производить вставку. Значение должно быть одним из следующих:

* "beforebegin" – вставить html непосредственно перед elem,
* "afterbegin" – вставить html в начало elem,
* "beforeend" – вставить html в конец elem,
* "afterend" – вставить html непосредственно после elem.

Второй параметр – это HTML-строка, которая будет вставлена именно «как HTML». Например:

<div id="div"></div>

<script>

div.insertAdjacentHTML('beforebegin', '<p>Привет</p>');

div.insertAdjacentHTML('afterend', '<p>Пока</p>');

</script>

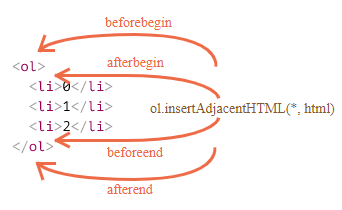
Приведёт к:

<p>Привет</p>

<div id="div"></div>

<p>Пока</p>

Так можно добавлять произвольный HTML на страницу. Варианты вставки:



Есть еще два схожих метода:

* elem.insertAdjacentText(where, text) – такой же синтаксис, но строка text вставляется «как текст», вместо HTML,
* elem.insertAdjacentElement(where, elem) – такой же синтаксис, но вставляет элемент elem.

На практике часто используется только insertAdjacentHTML. Потому что для элементов и текста есть методы append/prepend/before/after – их быстрее написать, и они могут вставлять как узлы, так и текст. Альтернативный вариант показа сообщения:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

document.body.insertAdjacentHTML("afterbegin", `<div class="alert">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>`);

</script>

[**Удаление узлов**](https://learn.javascript.ru/modifying-document#udalenie-uzlov)

Для удаления узла есть методы node.remove(). Например, сделаем так, чтобы сообщение удалялось через секунду:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<script>

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

div.className = "alert";

div.innerHTML = "<strong>Всем привет!</strong> Вы прочитали важное сообщение.";

document.body.append(div);

setTimeout(() => div.remove(), 1000);

</script>

Если нужно переместить элемент в другое место – нет необходимости удалять его со старого. Все методы вставки автоматически удаляют узлы со старых мест. Например, поменяем местами элементы:

<div id="first">Первый</div>

<div id="second">Второй</div>

<script>

second.after(first); // вставляет #first после #second

</script>

[**Клонирование узлов: cloneNode**](https://learn.javascript.ru/modifying-document#klonirovanie-uzlov-clonenode)

Вызов elem.cloneNode(true) создаёт клон элемента со всеми атрибутами и дочерними элементами. Если вызвать elem.cloneNode(false), тогда клон будет без дочерних элементов. Пример копирования сообщения:

<style>

.alert {

padding: 15px;

border: 1px solid #d6e9c6;

border-radius: 4px;

color: #3c763d;

background-color: #dff0d8;

}

</style>

<div class="alert" id="div">

<strong>Всем привет!</strong> Вы прочитали важное сообщение.

</div>

<script>

let div2 = div.cloneNode(true); // клонировать сообщение

div2.querySelector('strong').innerHTML = 'Всем пока!'; // изменить клонированный элемент

div.after(div2); // показать клонированный элемент после существующего div

</script>

**Узел** [**DocumentFragment**](https://learn.javascript.ru/modifying-document#document-fragment)

DocumentFragment является специальным DOM-узлом, который служит обёрткой для передачи списков узлов. Можно добавить к нему другие узлы, но при вставке он «исчезает», вместо него вставляется его содержимое. Например, getListContent ниже генерирует фрагмент с элементами <li>, которые позже вставляются в <ul>:

<ul id="ul"></ul>

<script>

function getListContent() {

let fragment = new DocumentFragment();

for(let i=1; i<=3; i++) {

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

li.append(i);

fragment.append(li);

}

return fragment;

}

ul.append(getListContent()); // (\*)

</script>

Обратите внимание, что на последней строке с (\*) добавляется DocumentFragment, но он «исчезает», поэтому структура будет:

<ul>

<li>1</li>

<li>2</li>

<li>3</li>

</ul>

DocumentFragment редко используется. Не смысла добавлять элементы в специальный вид узла, если вместо этого можно вернуть массив узлов. Переписанный пример:

<ul id="ul"></ul>

<script>

function getListContent() {

let result = [];

for(let i=1; i<=3; i++) {

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

li.append(i);

result.push(li);

}

return result;

}

ul.append(...getListContent());

</script>

DocumentFragment  используется в некоторых других областях, например, для элемента [template](https://learn.javascript.ru/template-element).

1. **Стили DOM-узлов.**

Как правило, существует два способа задания стилей для элемента:

1. Создать класс в CSS и использовать его: <div class="...">.
2. Писать стили непосредственно в атрибуте style: <div style="...">.

JavaScript может менять и классы и свойство style. Классы – всегда предпочтительный вариант по сравнению со style. Свойством style стоит манипулировать только в том случае, если классы «не могут справиться». Например, использование style является приемлемым, если вычисляются координаты элемента динамически и хотим установить их из JavaScript:

let top = /\* расчёты \*/;

let left = /\* расчёты \*/;

elem.style.left = left;

elem.style.top = top;

В других случаях, например, чтобы сделать текст красным, добавить значок фона – надо описать это в CSS и добавить класс (JavaScript может это сделать). Это более гибкое и легкое в поддержке решение.

[**className и classList**](https://learn.javascript.ru/styles-and-classes#classname-i-classlist)

Изменение класса является одним из наиболее часто используемых действий в скриптах. Свойство "className": elem.className соответствует атрибуту "class". Например:

<body class="main page">

<script>

alert(document.body.className); // main page

</script>

</body>

Если присвоить что-то elem.className, то это заменяет всю строку с классами. Иногда это то, что нужно, но часто надо добавить/удалить один класс. Для этого есть другое свойство: elem.classList.

elem.classList – это специальный объект с методами для добавления/удаления одного класса. Например:

<body class="main page">

<script>

document.body.classList.add('article');

alert(document.body.className); // main page article

</script>

</body>

Так что можно работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList.

Методы classList:

* elem.classList.add/remove("class") – добавить/удалить класс.
* elem.classList.toggle("class") – добавить класс, если его нет, иначе удалить.
* elem.classList.contains("class") – проверка наличия класса, возвращает true/false.

Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of:

<body class="main page">

<script>

for (let name of document.body.classList) {

alert(name); // main, затем page

}

</script>

</body>

**Свойство** [**Element style**](https://learn.javascript.ru/styles-and-classes#element-style)

Свойство elem.style – это объект, который соответствует тому, что написано в атрибуте "style". Установка стиля elem.style.width="100px" работает так же, как наличие в атрибуте style строки width:100px. Для свойства из нескольких слов используется camelCase:

background-color => elem.style.backgroundColor

z-index => elem.style.zIndex

border-left-width => elem.style.borderLeftWidth

Например:

document.body.style.backgroundColor = prompt('background color?', 'green');

Стили с браузерным префиксом, например, -moz-border-radius, -webkit-border-radius преобразуются по тому же принципу: дефис означает прописную букву. Например:

button.style.MozBorderRadius = '5px';

button.style.WebkitBorderRadius = '5px';

Иногда нужно добавить свойство стиля, а потом, позже, убрать его. Например, чтобы скрыть элемент, можно задать elem.style.display = "none". Затем можно удалить свойство style.display, чтобы вернуться к первоначальному состоянию. Вместо delete elem.style.display  надо присвоить ему пустую строку: elem.style.display = "".

// <body> "мигнёт"

document.body.style.display = "none"; // скрыть

setTimeout(() => document.body.style.display = "", 1000); // возврат к нормальному состоянию

Если установить в style.display пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства style.display вообще не было.

Обычно используется style.\* для присвоения индивидуальных свойств стиля. Нельзя установить список стилей как, например, div.style="color: red; width: 100px", потому что div.style – это объект, и он доступен только для чтения. Для задания нескольких стилей в одной строке используется специальное свойство style.cssText:

<div id="div">Button</div>

<script>

div.style.cssText=`color: red !important;

background-color: yellow;

width: 100px;

text-align: center;

`;

alert(div.style.cssText);

</script>

Это свойство редко используется, потому что такое присваивание удаляет все существующие стили: оно не добавляет, а заменяет их. Можно случайно удалить что-то нужное. Но его можно использовать, к примеру, для новых элементов, когда точно известно, что не удалится существующий стиль. То же самое можно сделать установкой атрибута: div.setAttribute('style', 'color: red...').

Не забывайте добавлять к значениям единицы измерения. Например, надо устанавливать 10px, а не просто 10 в свойство elem.style.top, иначе это не сработает:

<body>

<script>

// не работает

document.body.style.margin = 20;

alert(document.body.style.margin); // ''

document.body.style.margin = '20px';

alert(document.body.style.margin); // 20px

alert(document.body.style.marginTop); // 20px

alert(document.body.style.marginLeft); // 20px

</script>

</body>

Обратите внимание, браузер «распаковывает» свойство style.margin в последних строках и выводит style.marginLeft и style.marginTop из него.

[**Вычисленные стили: getComputedStyle**](https://learn.javascript.ru/styles-and-classes#vychislennye-stili-getcomputedstyle)

Метод getComputedStyle позволяет получить текущее значение свойств элемента. Синтаксис:

getComputedStyle(element, [pseudo])

element – элемент, значения для которого нужно получить,

pseudo – указывается, если нужен стиль псевдоэлемента, например ::before. Пустая строка или отсутствие аргумента означают сам элемент.

Результат вызова – объект со стилями, похожий на elem.style, но с учётом всех CSS-классов. Например:

<head>

<style> body { color: red; margin: 5px } </style>

</head>

<body>

<script>

let computedStyle = getComputedStyle(document.body);

alert( computedStyle.marginTop ); // 5px

alert( computedStyle.color ); // rgb(255, 0, 0)

</script>

</body>

Есть две концепции в [CSS](https://drafts.csswg.org/cssom/#resolved-values):

1. Вычисленное (computed) значение – это то, которое получено после применения всех CSS-правил и CSS-наследования. Например, height:1em или font-size:125%.
2. Окончательное ([resolved](https://drafts.csswg.org/cssom/" \l "resolved-values)) значение – непосредственно применяемое к элементу. Значения 1em или 125%являются относительными. Браузер берёт вычисленное значение и делает все единицы измерения фиксированными и абсолютными, например, height:20px or font-size:16px. Для геометрических свойств разрешенные значения могут иметь плавающую точку, например, width:50.5px.

Изначально getComputedStyle был создан для получения вычисленных значений, но окончательные значения гораздо удобнее, и стандарт изменился. Так что, в настоящее время getComputedStyle фактически возвращает окончательное значение свойства, для геометрии оно обычно в пискселях.

Для правильного получения значения нужно указать точное свойство. Например: paddingLeft, marginTop, borderTopWidth. При обращении к сокращенному: padding, margin, border – правильный результат не гарантируется.

Некоторые браузеры (Chrome) отображают 10px в документе ниже, а некоторые (Firefox) – нет:

<style>

body {

margin: 10px;

}

</style>

<script>

let style = getComputedStyle(document.body);

alert(style.margin); // пустая строка в Firefox

</script>

Посещенные ссылки могут быть окрашены с помощью псевдокласса :visited. Но getComputedStyle не дает доступ к этой информации, чтобы произвольная страница не могла определить, посещал ли пользователь ту или иную ссылку, проверив стили.

JavaScript не видит стили, применяемые с помощью :visited. Кроме того, в CSS есть ограничение, которое запрещает в целях безопасности применять к :visited CSS-стили, изменяющие геометрию элемента. Это гарантирует, что нет обходного пути для вредоносной страницы проверить, была ли ссылка посещена и, следовательно, нарушить конфиденциальность.

1. **Размеры и прокрутка элементов и страницы.**

Существует множество JavaScript-свойcтв, которые позволяют считывать информацию об элементе: ширину, высоту и другие геометрические характеристики. Будем называть их «метрики». Они часто требуются, когда нужно передвигать или позиционировать элементы с помощью JavaScript.

В качестве простого примера демонстрации свойств будем использовать следующий элемент:

<div id="example">

...Текст...

</div>

<style>

#example {

width: 300px;

height: 200px;

border: 25px solid #E8C48F;

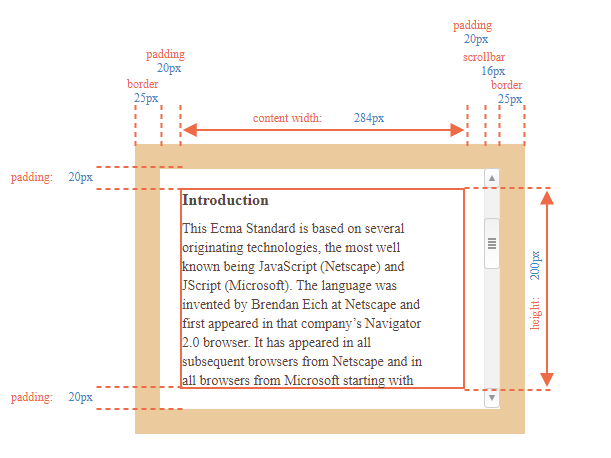
padding: 20px;

overflow: auto;

}

</style>

У элемента есть рамка (border), внутренний отступ (padding) и прокрутка. Полный набор характеристик. Обратите внимание, тут нет внешних отступов (margin), потому что они не являются частью элемента, для них нет особых JavaScript-свойств. Результат выглядит так:

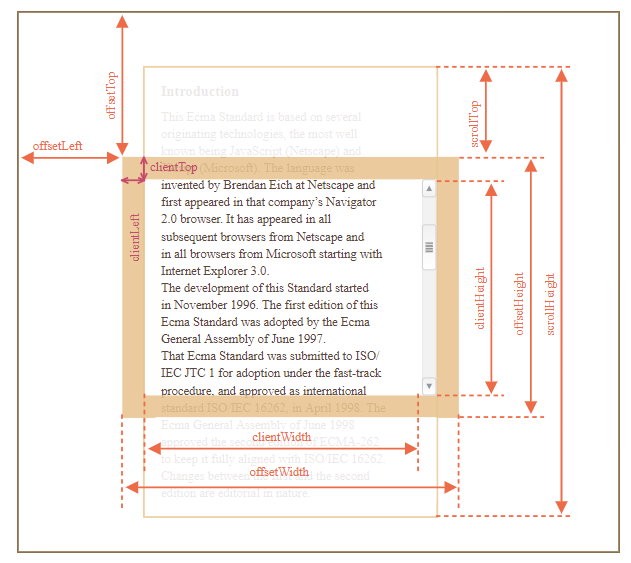


В иллюстрации выше намеренно продемонстрирован самый сложный и полный случай, когда у элемента есть ещё и полоса прокрутки. Некоторые браузеры (не все) отбирают место для неё, забирая его у области, отведённой для содержимого (помечена как «content width» выше).

Таким образом, без учёта полосы прокрутки ширина области содержимого (content width) будет 300px, но если предположить, что ширина полосы прокрутки равна 16px (её точное значение зависит от устройства и браузера), тогда остаётся только 300 - 16 = 284px, и надо это учитывать.

Нижние внутренние отступы padding-bottom изображены на иллюстрациях пустыми, но если элемент содержит много текста, то он будет перекрывать padding-bottom, это нормально.

Вот общая картина с геометрическими свойствами:



Значениями свойств являются числа, подразумевается, что они в пикселях.

Свойства [offsetParent, offsetLeft/Top](https://learn.javascript.ru/size-and-scroll#offsetparent-offsetleft-top) редко используются, они являются «самыми внешними» метриками.

В свойстве offsetParent находится предок элемента, который используется внутри браузера для вычисления координат при рендеринге. То есть, ближайший предок, который удовлетворяет следующим условиям:

* является CSS-позиционированным (CSS-свойство position равно absolute, relative, fixed или sticky);
* или <td>, <th>, <table>;
* или <body>.

Свойства offsetLeft/offsetTop содержат координаты x/y относительно верхнего левого угла offsetParent. В примере ниже внутренний <div> имеет элемент <main> в качестве offsetParent, а свойства offsetLeft/offsetTop являются сдвигами относительно верхнего левого угла (180):

<main style="position: relative" id="main">

<article>

<div id="example" style="position: absolute; left: 180px; top: 180px">...</div>

</article>

</main>

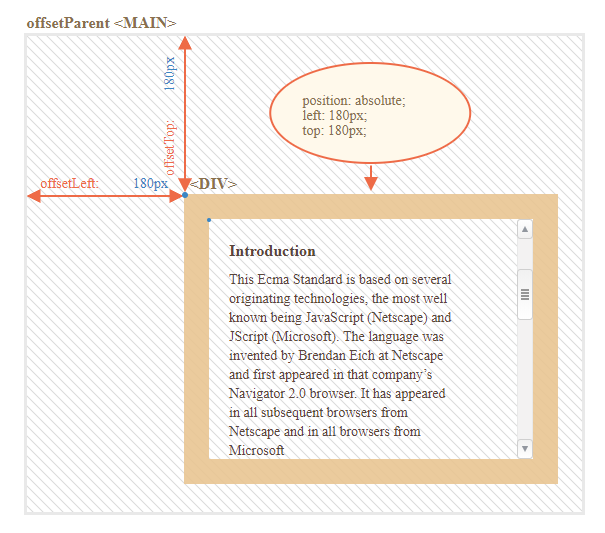
<script>

alert(example.offsetParent.id); // main

alert(example.offsetLeft); // 180 (число, а не строка "180px")

alert(example.offsetTop); // 180

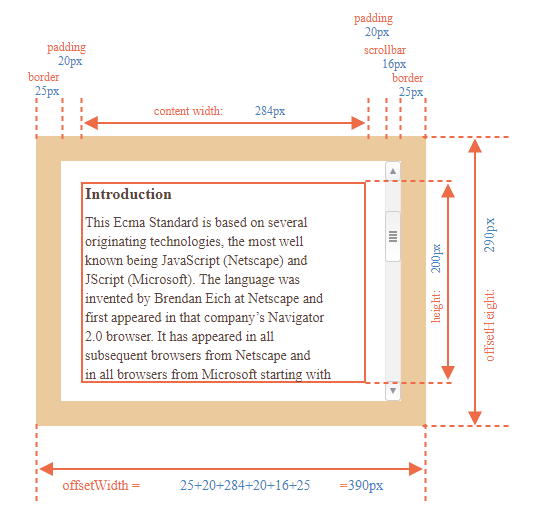
</script>



Существует несколько ситуаций, когда offsetParent равно null:

* 1. Для скрытых элементов (с CSS-свойством display:none или когда его нет в документе).
  2. Для элементов <body> и <html>.
  3. Для элементов с position:fixed.

Свойства [offsetWidth/Height](https://learn.javascript.ru/size-and-scroll#offsetwidth-height) самые простые. Они содержат «внешнюю» ширину/высоту элемента, то есть его полный размер, включая рамки.



Для рассматриваемого элемента:

* offsetWidth = 390 – внешняя ширина блока, её можно получить сложением CSS-ширины (300px), внутренних отступов (2 \* 20px) и рамок (2 \* 25px).
* offsetHeight = 290 – внешняя высота блока.

Метрики для не показываемых элементов равны нулю. Координаты и размеры в JavaScript устанавливаются только для видимых элементов. Если элемент (или любой его родитель) имеет display:none или отсутствует в документе, то все его метрики равны нулю (или null, если это offsetParent). Например, свойство offsetParent равно null, а offsetWidth и offsetHeight равны 0, когда элемент создан, но ещё не вставлен в документ, или если у элемента (или у его родителя) display:none. Это можно использовать, чтобы делать проверку на видимость:

function isHidden(elem) {

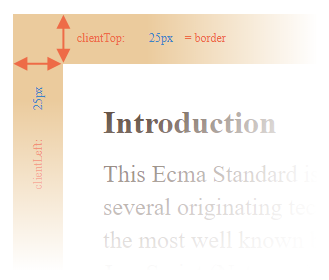
return !elem.offsetWidth && !elem.offsetHeight;

}

Функция isHidden также вернёт true для элементов, которые в принципе показываются, но их размеры равны нулю (например, пустые <div>).

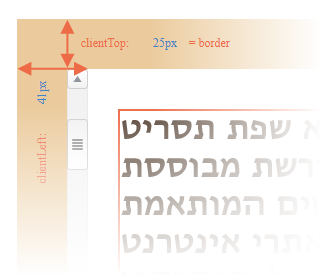
Внутри элемента есть рамки (border). Для них есть свойства-метрики clientTop и clientLeft. В примере:

* clientLeft = 25 – ширина левой рамки,
* clientTop = 25 – ширина верхней рамки.

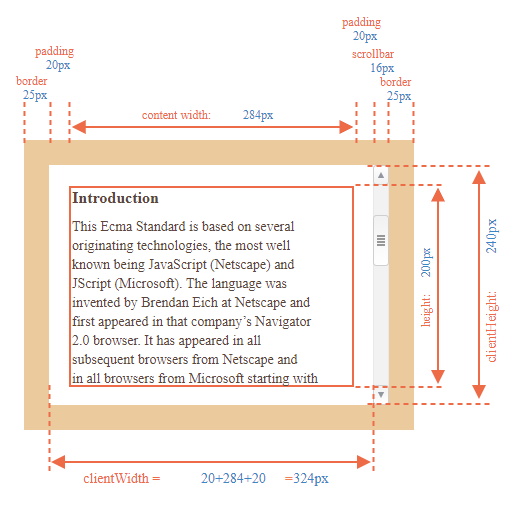


Но на самом деле эти свойства – вовсе не ширины рамок, а отступы внутренней части элемента от внешней. Разница в том, что когда документ располагается справа налево (операционная система на арабском языке или иврите). Полоса прокрутки в этом случае находится слева, и тогда свойство clientLeft включает в себя ещё и ширину полосы прокрутки. В этом случае clientLeft будет равно 25, но с прокруткой – 25 + 16 = 41.

Вот соответствующий пример на иврите:

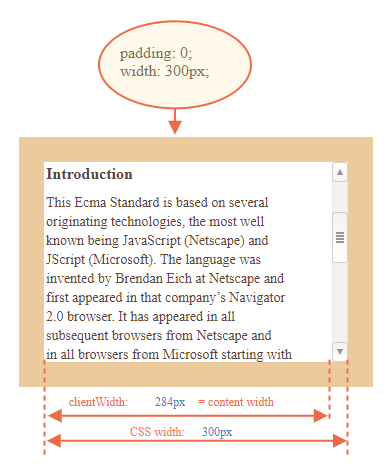


Свойства [clientWidth/Height](https://learn.javascript.ru/size-and-scroll#clientwidth-height) – это размер области внутри рамок элемента. Они включают в себя ширину области содержимого вместе с внутренними отступами padding, но без прокрутки:

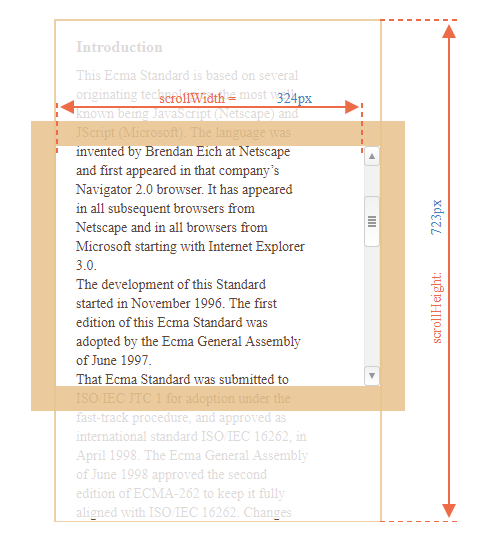


На рисунке выше горизонтальной прокрутки нет, так что высота clientHeight в точности то, что внутри рамок: CSS-высота 200px плюс верхние и нижние внутренние отступы (2 \* 20px), итого 240px. clientWidth – ширина содержимого здесь равна не 300px, а 284px, т.к. 16px отведено для полосы прокрутки. Таким образом: 284px плюс левый и правый отступы – всего 324px.

Если нет внутренних отступов padding, то clientWidth/Height в точности равны размеру области содержимого внутри рамок и полосы прокрутки (если она есть).

Поэтому в тех случаях, когда точно известно, что отступов нет, можно использовать clientWidth/clientHeight для получения размеров внутренней области содержимого.

Свойства [scrollWidth/Height](https://learn.javascript.ru/size-and-scroll#scrollwidth-height) – как clientWidth/clientHeight, но также включают в себя прокрученную (которую не видно) часть элемента.



На рисунке выше:

* scrollHeight = 723 – полная внутренняя высота, включая прокрученную область;
* scrollWidth = 324 – полная внутренняя ширина, в данном случае прокрутки нет, поэтому она равна clientWidth.

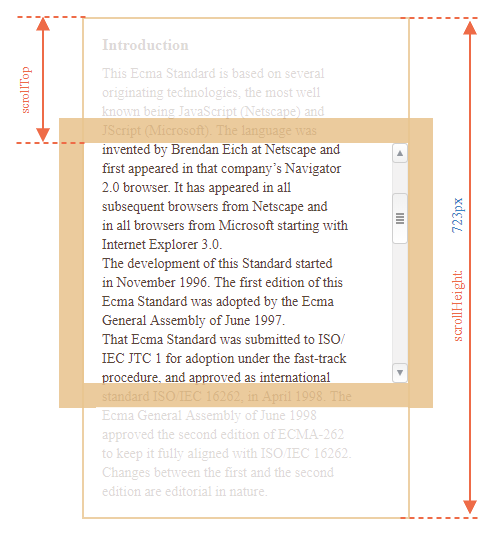
Эти свойства можно использовать, чтобы «распахнуть» элемент на всю ширину/высоту, например при нажатии на кнопку:

element.style.height = `${element.scrollHeight}px`;



Свойства scrollLeft/scrollTop – ширина/высота невидимой, прокрученной в данный момент, части элемента слева и сверху.

Следующая иллюстрация показывает значения scrollHeight и scrollTop для блока с вертикальной прокруткой.



Другими словами, свойство scrollTop – это «сколько уже прокручено вверх».

В отличие от большинства свойств, которые доступны только для чтения, значения scrollLeft/scrollTop можно изменять, и браузер выполнит прокрутку элемента. Установка значения scrollTop на 0 или Infinity прокрутит элемент в самый верх/низ соответственно.

Выше рассматривались метрики, которые есть у DOM-элементов, и которые можно использовать для получения различных высот, ширин и прочих расстояний. Но как известно, CSS-высоту и ширину можно извлечь, используя getComputedStyle:

let elem = document.body;

alert( getComputedStyle(elem).width );

Тем не менее стоит использовать свойства-метрики по следующим причинам:

1. Во-первых, CSS-свойства width/height зависят от другого свойства – box-sizing, которое определяет, «что такое», собственно, эти CSS-ширина и высота. Получается, что изменение box-sizing, к примеру, для более удобной вёрстки, сломает такой JavaScript.
2. Во-вторых, в CSS свойства width/height могут быть равны auto, например, для инлайнового элемента:

<span id="elem">Привет!</span>

<script>

alert( getComputedStyle(elem).width ); // auto

</script>

Конечно, с точки зрения CSS width:auto – совершенно нормально, но в JavaScript нужен конкретный размер в px, который можно использовать для вычислений. Получается, что в данном случае ширина из CSS вообще бесполезна.

Есть и ещё одна причина: полоса прокрутки. Бывает, без полосы прокрутки код работает прекрасно, но стоит ей появиться, как начинают проявляться баги. Так происходит потому, что полоса прокрутки «забирает» место от области внутреннего содержимого в некоторых браузерах. Таким образом, реальная ширина содержимого меньше CSS-ширины. Как раз это и учитывают свойства clientWidth/clientHeight.

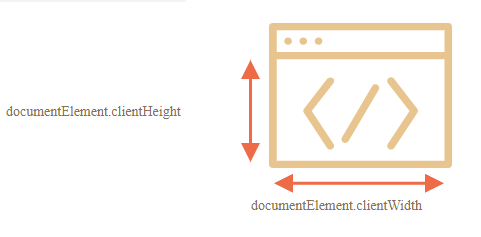
Но с getComputedStyle(elem).width ситуация иная. Некоторые браузеры (например, Chrome) возвращают реальную внутреннюю ширину с вычетом ширины полосы прокрутки, а некоторые (например, Firefox) – именно CSS-свойство (игнорируя полосу прокрутки). Эти кроссбраузерные отличия – ещё один повод не использовать getComputedStyle, а использовать свойства-метрики.

Описанные различия касаются только чтения свойства getComputedStyle(...).width из JavaScript, визуальное отображение корректно в обоих случаях.

1. **Размеры и прокрутка окна. Координаты.**

[**Ширина/высота окна**](https://learn.javascript.ru/size-and-scroll-window#shirina-vysota-okna)

Чтобы получить ширину/высоту окна, можно взять свойства clientWidth/clientHeight из document.documentElement:



Например, получить высоту окна можно так: document.documentElement.clientHeight.

Браузеры также поддерживают свойства window.innerWidth/innerHeight. Но их не стоит использовать их, так как если есть полоса прокрутки, и она занимает какое-то место, то свойства clientWidth/clientHeight указывают на ширину/высоту документа без неё (за её вычетом). Иными словами, они возвращают высоту/ширину видимой части документа, доступной для содержимого. A window.innerWidth/innerHeight включают в себя полосу прокрутки. Если полоса прокрутки занимает некоторое место, то эти две строки выведут разные значения:

alert( window.innerWidth ); // полная ширина окна

alert( document.documentElement.clientWidth ); // ширина окна за вычетом полосы прокрутки

В большинстве случаев нужна доступная ширина окна: для рисования или позиционирования. Полоса прокрутки «забирает» её часть. Поэтому следует использовать documentElement.clientHeight/Width.

Обратите внимание, что геометрические свойства верхнего уровня могут работать немного иначе, если в HTML нет <!DOCTYPE HTML>. Возможны странности. Поэтому в современном HTML всегда надо указывать DOCTYPE.

[**Ширина/высота документа**](https://learn.javascript.ru/size-and-scroll-window#shirina-vysota-dokumenta)

Теоретически, т.к. корневым элементом документа является documentElement.clientWidth/Height, и он включает в себя всё содержимое, то можно получить полный размер документа как documentElement.scrollWidth/scrollHeight.

Но именно на этом элементе, для страницы в целом, эти свойства работают не так, как предполагается. В Chrome/Safari/Opera, если нет прокрутки, то documentElement.scrollHeight может быть даже меньше, чем documentElement.clientHeight. С точки зрения элемента это невозможная ситуация. Чтобы надёжно получить полную высоту документа, следует взять максимальное из этих свойств:

let scrollHeight = Math.max(

document.body.scrollHeight, document.documentElement.scrollHeight,

document.body.offsetHeight, document.documentElement.offsetHeight,

document.body.clientHeight, document.documentElement.clientHeight

);

alert('Полная высота документа с прокручиваемой частью: ' + scrollHeight);

Обычные элементы хранят текущее состояние прокрутки в elem.scrollLeft/scrollTop. Чтобы получить текущее состояние прокрутки страницы в большинстве браузеров можно обратиться к documentElement.scrollLeft/Top, за исключением основанных на старом WebKit (Safari), где есть ошибка ([5991](https://bugs.webkit.org/show_bug.cgi?id=5991)), и там нужно использовать document.body вместо document.documentElement.

Текущую прокрутку можно прочитать из свойств window.pageXOffset/pageYOffset:

alert('Текущая прокрутка сверху: ' + window.pageYOffset);

alert('Текущая прокрутка слева: ' + window.pageXOffset);

Эти свойства доступны только для чтения.

Для прокрутки страницы из JavaScript её DOM должен быть полностью построен. Например, если попытаться прокрутить страницу из скрипта в <head>, это не сработает. Обычные элементы можно прокручивать, изменяя scrollTop/scrollLeft. Можно сделать то же самое для страницы в целом, используя document.documentElement.scrollTop/Left (кроме основанных на старом WebKit (Safari), где, как сказано выше, document.body.scrollTop/Left).

Есть и другие способы, в которых подобных несовместимостей нет: специальные методы window.scrollBy(x,y) и window.scrollTo(pageX,pageY).

Метод scrollBy(x,y) прокручивает страницу относительно её текущего положения. Например, scrollBy(0,10)прокручивает страницу на 10px вниз.

Метод scrollTo(pageX,pageY) прокручивает страницу на абсолютные координаты (pageX,pageY). То есть, чтобы левый-верхний угол видимой части страницы имел данные координаты относительно левого верхнего угла документа. Это всё равно, что поставить scrollLeft/scrollTop. Для прокрутки в самое начало можно использовать scrollTo(0,0).

Эти методы одинаково работают для всех браузеров.

Рассмотрим ещё один метод: [elem.scrollIntoView(top)](https://developer.mozilla.org/ru/docs/Web/API/Element/scrollIntoView). Вызов elem.scrollIntoView(top) прокручивает страницу, чтобы elem оказался вверху. У него есть один аргумент:

* если top=true (по умолчанию), то страница будет прокручена, чтобы elem появился в верхней части окна. Верхний край элемента совмещён с верхней частью окна.
* если top=false, то страница будет прокручена, чтобы elem появился внизу. Нижний край элемента будет совмещён с нижним краем окна.

Иногда нужно сделать документ непрокручиваемым. Например, при показе большого диалогового окна над документом – чтобы посетитель мог прокручивать это окно, но не документ. Чтобы запретить прокрутку страницы, достаточно установить document.body.style.overflow = "hidden":

document.body.style.overflow = 'hidden'

Возобновить прокрутку:

document.body.style.overflow = ''

Аналогичным образом можно запретить прокрутку для других элементов, а не только для document.body.

Недостатком этого способа является то, что сама полоса прокрутки исчезает. Если она занимала некоторую ширину, то теперь эта ширина освободится, и содержимое страницы расширится, текст «прыгнет», заняв освободившееся место.

Это выглядит немного странно, но это можно обойти, если сравнить clientWidth до и после остановки, и если clientWidth увеличится (значит полоса прокрутки исчезла), то добавить padding в document.body вместо полосы прокрутки, чтобы оставить ширину содержимого прежней.

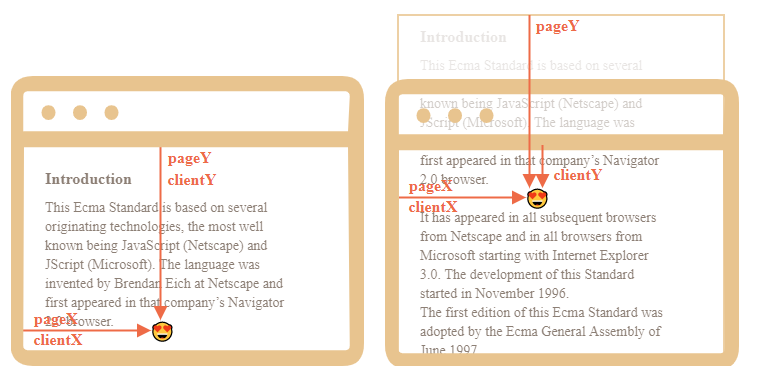
**Координаты**

Большинство соответствующих методов JavaScript работают в одной из двух указанных ниже систем координат:

1. Относительно окна браузера – как position:fixed, отсчёт идёт от верхнего левого угла окна. Далее будем обозначать эти координаты как clientX/clientY.
2. Относительно документа – как position:absolute на уровне документа, отсчёт идёт от верхнего левого угла документа. Далее будем обозначать эти координаты как pageX/pageY.

Когда страница полностью прокручена в самое начало, то верхний левый угол окна совпадает с левым верхним углом документа, при этом обе этих системы координат тоже совпадают. Но если происходит прокрутка, то координаты элементов в контексте окна меняются, так как они двигаются, но в то же время их координаты относительно документа остаются такими же.

На картинке ниже показаны координат точки до прокрутки (слева) и после (справа):



При прокрутке документа:

* pageY – координата точки относительно документа осталась без изменений, так как отсчёт по-прежнему ведётся от верхней границы документа (сейчас она прокручена наверх).
* clientY – координата точки относительно окна изменилась (стрелка на рисунке стала короче), так как точка стала ближе к верхней границе окна.

[**Координаты относительно окна: getBoundingClientRect**](https://learn.javascript.ru/coordinates#koordinaty-otnositelno-okna-getboundingclientrect)

Метод elem.getBoundingClientRect() возвращает координаты в контексте окна для минимального по размеру прямоугольника, который заключает в себе элемент elem, в виде объекта встроенного класса [DOMRect](https://www.w3.org/TR/geometry-1/" \l "domrect).

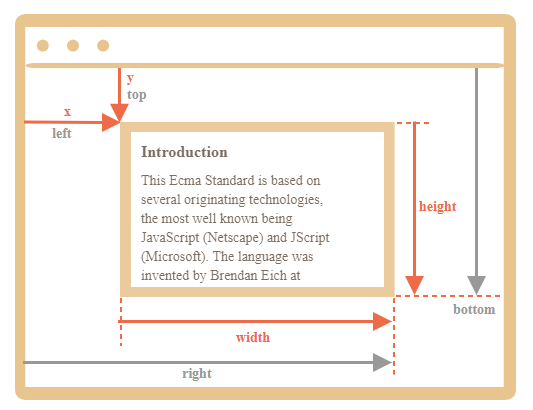
Основные свойства объекта типа DOMRect:

* x/y – X/Y-координаты начала прямоугольника относительно окна,
* width/height – ширина/высота прямоугольника (могут быть отрицательными).

Дополнительные, «зависимые», свойства:

* top/bottom – Y-координата верхней/нижней границы прямоугольника,
* left/right – X-координата левой/правой границы прямоугольника.

Картинка с результатами вызова elem.getBoundingClientRect():



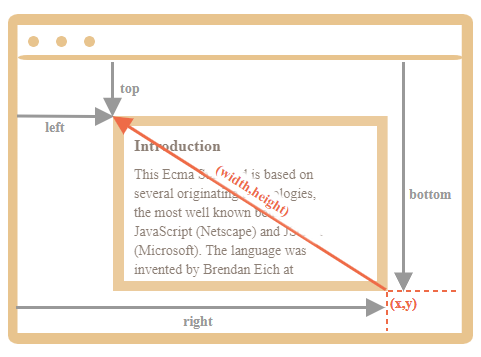
Как видно, x/y и width/height уже точно задают прямоугольник. Остальные свойства могут быть легко вычислены на их основе: left = x, top = y, right = x + width, bottom = y + height.

Заметим:

* Координаты могут считаться с десятичной частью, например, 10.5. Это нормально, ведь браузер использует дроби в своих внутренних вычислениях.
* Координаты могут быть отрицательными. Например, если страница прокручена так, что элемент elem ушёл вверх за пределы окна, то вызов elem.getBoundingClientRect().top вернёт отрицательное значение.

С математической точки зрения, прямоугольник однозначно задаётся начальной точкой (x,y) и вектором направления (width, height). Так что дополнительные зависимые свойства существуют лишь для удобства. Технически, значения width/height могут быть отрицательными, это позволяет задать «направленный» прямоугольник, например, для выделения мышью с отмеченным началом и концом.

Пример прямоугольника с отрицательными width и height (например, width=-200, height=-100):



Он начинается в правом-нижнем углу, и затем «растёт» влево-вверх, так как отрицательные width/height ведут его назад по координатам. Как видно, свойства left/top здесь не равны x/y. То есть они не дублируют друг друга. Формулы выше могут быть исправлены с учётом возможных отрицательных значений width/height. Это достаточно просто сделать, но редко требуется, так как результат вызова elem.getBoundingClientRect() всегда возвращает положительные значения для ширины/высоты.

Internet Explorer и Edge не поддерживают свойства x/y. Таким образом, можно либо сделать полифил (добавив соответствующие геттеры в DomRect.prototype), либо использовать top/left, так как это всегда одно и то же при положительных width/height, в частности – в результате вызова elem.getBoundingClientRect().

Есть очевидное сходство между координатами относительно окна и CSS position:fixed. Но в CSS свойство right означает расстояние от правого края, и свойство bottom означает расстояние от нижнего края окна браузера. Если взглянуть на картинку выше, то видно, что в JavaScript это не так. Все координаты в контексте окна считаются от верхнего левого угла, включая right/bottom.

Вызов document.elementFromPoint(x, y) возвращает самый глубоко вложенный элемент в окне, находящийся по координатам (x, y). Синтаксис:

let elem = document.elementFromPoint(x, y);

Например, код ниже выделяет с помощью стилей и выводит имя тега элемента, который сейчас в центре окна браузера:

let centerX = document.documentElement.clientWidth / 2;

let centerY = document.documentElement.clientHeight / 2;

let elem = document.elementFromPoint(centerX, centerY);

elem.style.background = "red";

alert(elem.tagName);

Поскольку используются координаты в контексте окна, то элемент может быть разным, в зависимости от того, какая сейчас прокрутка.

Метод document.elementFromPoint(x,y) работает, только если координаты (x,y) относятся к видимой части содержимого окна. Если любая из координат представляет собой отрицательное число или превышает размеры окна, то возвращается null. Вот типичная ошибка, которая может произойти, если в коде нет соответствующей проверки:

let elem = document.elementFromPoint(x, y);

elem.style.background = ''; // ошибка

Чаще всего нужны координаты для позиционирования чего-либо. Чтобы показать что-то около нужного элемента, можно вызвать getBoundingClientRect, чтобы получить его координаты элемента, а затем использовать CSS-свойство position вместе с left/top (или right/bottom). Например, функция createMessageUnder(elem, html) ниже показывает сообщение под элементом elem:

let elem = document.getElementById("coords-show-mark");

function createMessageUnder(elem, html) {

let message = document.createElement('div');

message.style.cssText = "position:fixed; color: red";

let coords = elem.getBoundingClientRect();

message.style.left = coords.left + "px";

message.style.top = coords.bottom + "px";

message.innerHTML = html;

return message;

}

let message = createMessageUnder(elem, 'Hello, world!');

document.body.append(message);

setTimeout(() => message.remove(), 5000);

При прокрутке страницы сообщение будет уплывать от кнопки. Причина в том, что сообщение позиционируется с помощью position:fixed, поэтому оно остаётся всегда на том же самом месте в окне при прокрутке страницы. Чтобы изменить это, нужно использовать другую систему координат, где сообщение позиционировалось бы относительно документа, и свойство position:absolute.

[**Координаты относительно документа**](https://learn.javascript.ru/coordinates#getCoords)

В такой системе координат отсчёт ведётся от левого верхнего угла документа, не окна. В CSS координаты относительно окна браузера соответствуют свойству position:fixed, а координаты относительно документа – свойству position:absolute на самом верхнем уровне вложенности.

Можно воспользоваться свойствами position:absolute и top/left, чтобы привязать что-нибудь к конкретному месту в документе. При этом прокрутка страницы не имеет значения. Но сначала нужно получить верные координаты.

Не существует стандартного метода, который возвращал бы координаты элемента относительно документа, но можно написать его сами.

Две системы координат связаны следующими формулами:

* pageY = clientY + высота вертикально прокрученной части документа,
* pageX = clientX + ширина горизонтально прокрученной части документа.

Функция getCoords(elem) берёт координаты в контексте окна с помощью elem.getBoundingClientRect() и добавляет к ним значение соответствующей прокрутки:

function getCoords(elem) {

let box = elem.getBoundingClientRect();

return {

top: box.top + pageYOffset,

left: box.left + pageXOffset

};

}

Если бы в примере выше функция использовалась вместе с position:absolute, то при прокрутке сообщение оставалось бы рядом с элементом. Модифицированная функция createMessageUnder:

function createMessageUnder(elem, html) {

let message = document.createElement('div');

message.style.cssText = "position:absolute; color: red";

let coords = getCoords(elem);

message.style.left = coords.left + "px";

message.style.top = coords.bottom + "px";

message.innerHTML = html;

return message;

}

1. **Браузерные события.**

*Событие* – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы (хотя события бывают и не только в DOM). Ниже перечисляются наиболее часто используемые DOM-событий.

События мыши:

* click – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании);
* contextmenu – происходит, когда кликнули на элемент правой кнопкой мыши;
* mouseover / mouseout – когда мышь наводится на / покидает элемент;
* mousedown / mouseup – когда нажали / отжали кнопку мыши на элементе;
* mousemove – при движении мыши.

События на элементах управления:

* submit – пользователь отправил форму <form>;
* focus – пользователь фокусируется на элементе, например, нажимает на <input>.

Клавиатурные события:

* keydown и keyup – когда пользователь нажимает / отпускает клавишу.

События документа:

* DOMContentLoaded – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

* transitionend – когда CSS-анимация завершена.

Существует множество других событий.

[**Обработчики событий**](https://learn.javascript.ru/introduction-browser-events#obrabotchiki-sobytiy)

Событию можно назначить обработчик, то есть функцию, которая сработает, как только событие произошло. Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя. Есть несколько способов назначить событию обработчик.

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>. Например, чтобы назначить обработчик события click на элементе input, можно использовать атрибут onclick, вот так:

<input value="Нажми меня" onclick="alert('Клик!')" type="button">

При клике мышкой на кнопке выполнится код, указанный в атрибуте onclick. Обратите внимание, для содержимого атрибута onclick используются одинарные кавычки, так как сам атрибут находится в двойных. Если поставить двойные кавычки внутри атрибута, вот так: onclick="alert("Click!")", код не будет работать.

Атрибут HTML-тега – не самое удобное место для написания большого количества кода, поэтому лучше создать отдельную JavaScript-функцию и вызвать её там. Следующий пример по клику запускает функцию countRabbits():

<script>

function countRabbits() {

for(let i=1; i<=3; i++) {

alert("Кролик номер " + i);

}

}

</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">

Как известно, атрибут HTML-тега не чувствителен к регистру, поэтому ONCLICK будет работать так же, как onClick и onCLICK. Но, как правило, атрибуты пишут в нижнем регистре: onclick.

Можно назначать обработчик, используя свойство DOM-элемента on<событие>. К примеру, elem.onclick:

<input id="elem" type="button" value="Нажми меня!">

<script>

elem.onclick = function() {

alert('Спасибо');

};

</script>

Если обработчик задан через атрибут, то браузер читает HTML-разметку, создаёт новую функцию из содержимого атрибута и записывает в свойство. Этот способ, по сути, аналогичен предыдущему. Обработчик всегда хранится в свойстве DOM-объекта, а атрибут – лишь один из способов его инициализации.

Эти два примера кода работают одинаково:

1. Только HTML:

<input type="button" onclick="alert('Клик!')" value="Кнопка">

1. HTML + JS:

<input type="button" id="button" value="Кнопка">

<script>

button.onclick = function() {

alert('Клик!');

};

</script>

Так как у элемента DOM может быть только одно свойство с именем onclick, то назначить более одного обработчика так нельзя. В примере ниже назначение через JavaScript перезапишет обработчик из атрибута:

<input type="button" id="elem" onclick="alert('Было')" value="Нажми меня">

<script>

elem.onclick = function() {

alert('Станет');

};

</script>

Кстати, обработчиком можно назначить и уже существующую функцию:

function sayThanks() {

alert('Спасибо!');

}

elem.onclick = sayThanks;

Убрать обработчик можно назначением elem.onclick = null.

Внутри обработчика события this ссылается на текущий элемент, то есть на тот, на котором, как говорят, «висит» (т.е. назначен) обработчик. В коде ниже button выводит своё содержимое, используя this.innerHTML:

<button onclick="alert(this.innerHTML)">Нажми меня</button>

Обратите внимание, что функция должна быть присвоена как sayThanks, а не sayThanks().

// правильно

button.onclick = sayThanks;

// неправильно

button.onclick = sayThanks();

Если добавить скобки, то sayThanks() – это уже вызов функции, результат которого (равный undefined, так как функция ничего не возвращает) будет присвоен onclick. Так что это не будет работать. А вот в разметке, в отличие от свойства, скобки нужны:

<input type="button" id="button" onclick="sayThanks()">

Это различие просто объяснить. При создании обработчика браузером из атрибута, он автоматически создаёт функцию с телом из значения атрибута: sayThanks(). Так что разметка генерирует такое свойство:

button.onclick = function() {

sayThanks();

};

Назначение обработчика строкой elem.onclick = "alert(1)" также сработает. Это сделано из соображений совместимости, но делать так не рекомендуется.

Не используйте setAttribute для обработчиков. Такой вызов работать не будет, так как атрибуты всегда строки, и функция станет строкой:

document.body.setAttribute('onclick', function() { alert(1) });

Используйте elem.onclick, а не elem.ONCLICK, потому что DOM-свойства чувствительны к регистру.

Фундаментальный недостаток описанных выше способов назначения обработчика – невозможность повесить несколько обработчиков на одно событие. Например, одна часть кода хочет при клике на кнопку делать её подсвеченной, а другая – выдавать сообщение, следовательно, надо назначить два обработчика для этого. Но новое DOM-свойство перезапишет предыдущее:

input.onclick = function() { alert(1); }

// ...

input.onclick = function() { alert(2); }

Решить эту проблему позволяют методы addEventListener и removeEventListener.

Синтаксис добавления обработчика:

element.addEventListener(event, handler[, options]);

* event – имя события, например "click";
* handler – ссылка на функцию-обработчик;
* options – дополнительный объект со свойствами:
* once: если true, тогда обработчик будет автоматически удалён после выполнения;
* capture: фаза, на которой должен сработать обработчик (подробнее об этом будет рассказано в вопросе о [всплытии и погружени](https://learn.javascript.ru/bubbling-and-capturing)и событий). options может быть false/true, это тоже самое, что {capture: false/true}.
* passive: если true, то указывает, что обработчик никогда не вызовет preventDefault() (подробнее это будет рассматриваться в вопросе о [действии браузера по умолчанию](https://learn.javascript.ru/default-browser-action)).

Для удаления обработчика следует использовать removeEventListener:

element.removeEventListener(event, handler[, options]);

Для удаления обработчика нужно передать именно ту функцию-обработчик которая была назначена. Вот так не сработает:

elem.addEventListener( "click" , () => alert('Спасибо!'));

elem.removeEventListener( "click", () => alert('Спасибо!'));

Обработчик не будет удалён, т.к. в removeEventListener передана не та же функция, а другая, с одинаковым кодом. Вот так правильно:

function handler() {

alert( 'Спасибо!' );

}

input.addEventListener("click", handler);

input.removeEventListener("click", handler);

Обратим внимание – если функцию обработчик не сохранить где-либо, нельзя будет её удалить. Нет метода, который позволяет получить из элемента обработчики событий, назначенные через addEventListener. Метод addEventListener позволяет добавлять несколько обработчиков на одно событие одного элемента, например:

<input id="elem" type="button" value="Нажми меня"/>

<script>

function handler1() {

alert('Спасибо!');

};

function handler2() {

alert('Спасибо ещё раз!');

}

elem.onclick = () => alert("Привет");

elem.addEventListener("click", handler1); // Спасибо!

elem.addEventListener("click", handler2); // Спасибо ещё раз!

</script>

Как видно из примера выше, можно одновременно назначать обработчики и через DOM-свойство и через addEventListener. Однако, во избежание путаницы, рекомендуется выбрать один способ.

Существуют события, которые нельзя назначить через DOM-свойство, но можно через addEventListener. Например, таково событие transitionend, то есть окончание CSS-анимации. Код ниже демонстрирует это. В большинстве браузеров, сработает лишь второй обработчик, но не первый:

<style>

input {

transition: width 1s;

width: 100px;

}

.wide {

width: 300px;

}

</style>

<input type="button" id="elem" onclick="this.classList.toggle('wide')" value="Нажми меня">

<script>

elem.ontransitionend = function() {

alert("DOM property"); // не сработает

};

elem.addEventListener("transitionend", function() {

alert("addEventListener"); // сработает по окончании анимации

});

</script>

[**Объект события**](https://learn.javascript.ru/introduction-browser-events#obekt-sobytiya)

Чтобы хорошо обработать событие, могут понадобиться детали того, что произошло. Не просто «клик» или «нажатие клавиши», а также – какие координаты указателя мыши, какая клавиша нажата и так далее. Когда происходит событие, браузер создаёт объект события, записывает в него детали и передаёт его в качестве аргумента функции-обработчику. Пример ниже демонстрирует получение координат мыши из объекта события:

<input type="button" value="Нажми меня" id="elem">

<script>

elem.onclick = function(event) {

// вывести тип события, элемент и координаты клика

alert(event.type + " на " + event.currentTarget);

alert("Координаты: " + event.clientX + ":" + event.clientY);

};

</script>

Некоторые свойства объекта event:

* event.type – тип события, в данном случае "click";
* event.currentTarget – элемент, на котором сработал обработчик. Значение – обычно такое же, как и у this, но если обработчик является функцией-стрелкой или при помощи bind привязан другой объект в качестве this, то можно получить элемент из event.currentTarget;
* event.clientX / event.clientY – координаты курсора в момент клика относительно окна, для событий мыши.

Есть также и ряд других свойств, в зависимости от типа событий, которые будут рассмотрены далее.

При назначении обработчика в HTML, тоже можно использовать объект event, вот так:

<input type="button" onclick="alert(event.type)" value="Тип события">

Это возможно потому, что когда браузер из атрибута создаёт функцию-обработчик, то она выглядит так: function(event) { alert(event.type) }. То есть, её первый аргумент называется "event", а тело взято из атрибута.

Можно назначить обработчиком не только функцию, но и объект при помощи addEventListener. В этом случае, когда происходит событие, вызывается метод объекта handleEvent. К примеру:

<button id="elem">Нажми меня</button>

<script>

elem.addEventListener('click', {

handleEvent(event) {

alert(event.type + " на " + event.currentTarget);

}

});

</script>

Как видим, если addEventListener получает объект в качестве обработчика, он вызывает object.handleEvent(event), когда происходит событие. Также можно использовать класс для этого:

<button id="elem">Нажми меня</button>

<script>

class Menu {

handleEvent(event) {

switch(event.type) {

case 'mousedown':

elem.innerHTML = "Нажата кнопка мыши";

break;

case 'mouseup':

elem.innerHTML += "...и отжата.";

break;

}

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

Здесь один и тот же объект обрабатывает оба события. Обратите внимание, нужно явно назначить оба обработчика через addEventListener. Тогда объект menu будет получать события mousedown и mouseup, но не другие (не назначенные) типы событий.

Метод handleEvent не обязательно должен выполнять всю работу сам. Он может вызывать другие методы, которые заточены под обработку конкретных типов событий, вот так:

<button id="elem">Нажми меня</button>

<script>

class Menu {

handleEvent(event) {

// mousedown -> onMousedown

let method = 'on' + event.type[0].toUpperCase() + event.type.slice(1);

this[method](event);

}

onMousedown() {

elem.innerHTML = "Кнопка мыши нажата";

}

onMouseup() {

elem.innerHTML += "...и отжата.";

}

}

let menu = new Menu();

elem.addEventListener('mousedown', menu);

elem.addEventListener('mouseup', menu);

</script>

Теперь обработка событий разделена по методам, что упрощает поддержку кода.

1. **Всплытие и погружение событий.**

Рассмотрим пример. Этот обработчик для <div> сработает, если вы кликните по любому из вложенных тегов, будь то <em> или <code>:

<div onclick="alert('Обработчик!')">

<em>Если вы кликните на <code>EM</code>, сработает обработчик на <code>DIV</code></em>

</div>

**[Всплытие](https://learn.javascript.ru/bubbling-and-capturing" \l "vsplytie)**

Принцип всплытия очень простой. Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков. Например, есть 3 вложенных элемента FORM > DIV > P с обработчиком на каждом:

<style>

body \* {

margin: 10px;

border: 1px solid blue;

}

</style>

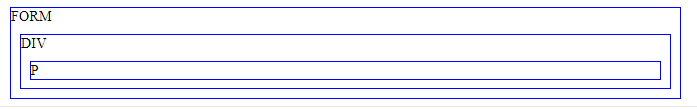
<form onclick="alert('form')">FORM

<div onclick="alert('div')">DIV

<p onclick="alert('p')">P</p>

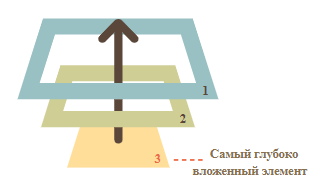
</div>

</form>



Клик по внутреннему <p> вызовет обработчик onclick:

1. Сначала на самом <p>.
2. Потом на внешнем <div>.
3. Затем на внешнем <form>.
4. И так далее вверх по цепочке до самого document.



Поэтому если кликнуть на <p>, то появятся три оповещения: p → div → form. Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей подобно тому, как всплывает пузырёк воздуха в воде. Не все события всплывают. Например, событие focus не всплывает. Однако, это исключение, всё-таки большинство событий всплывают.

[**event.target**](https://learn.javascript.ru/bubbling-and-capturing#event-target)

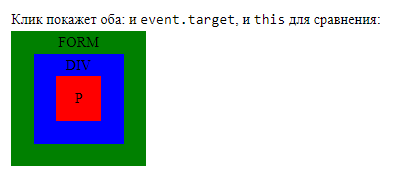
Всегда можно узнать, на каком конкретно элементе произошло событие. Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через event.target. Отличия от this (=event.currentTarget):

* event.target – это «целевой» элемент, на котором произошло событие, в процессе всплытия он неизменен.
* this – это «текущий» элемент, до которого дошло всплытие, на нём сейчас выполняется обработчик.

Например, если стоит только один обработчик form.onclick, то он «поймает» все клики внутри формы. Где бы ни был клик внутри – он всплывёт до элемента <form>, на котором сработает обработчик. При этом внутри обработчикаform.onclick:

* this (=event.currentTarget) всегда будет элемент <form>, так как обработчик сработал на ней.
* event.target будет содержать ссылку на конкретный элемент внутри формы, на котором произошёл клик.

Результат



// script.js

form.onclick = function(event) {

event.target.style.backgroundColor = 'yellow';

// браузеру нужно некоторое время, чтобы зарисовать всё жёлтым

setTimeout(() => {

alert("target = " + event.target.tagName + ", this=" + this.tagName);

event.target.style.backgroundColor = ''

}, 0);

};

// example.css

form {

background-color: green;

position: relative;

width: 150px;

height: 150px;

text-align: center;

cursor: pointer;

}

div {

background-color: blue;

position: absolute;

top: 25px;

left: 25px;

width: 100px;

height: 100px;

}

p {

background-color: red;

position: absolute;

top: 25px;

left: 25px;

width: 50px;

height: 50px;

line-height: 50px;

margin: 0;

}

body {

line-height: 25px;

font-size: 16px;

}

// index.html

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<link rel="stylesheet" href="example.css">

</head>

<body>

Клик покажет оба: и <code>event.target</code>, и <code>this</code> для сравнения:

<form id="form">FORM

<div>DIV

<p>P</p>

</div>

</form>

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

</body>

</html>

Возможна и ситуация, когда event.target и this – один и тот же элемент, например, если клик был непосредственно на самом элементе <form>, не на его подэлементе.

[**Прекращение всплытия**](https://learn.javascript.ru/bubbling-and-capturing#prekraschenie-vsplytiya)

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет всплывать до элемента <html>, а затем до объекта document, а иногда даже до window, вызывая все обработчики на своём пути. Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие. Для этого нужно вызвать метод event.stopPropagation(). Например, здесь при клике на кнопку <button> обработчик body.onclick не сработает:

<body onclick="alert(`сюда всплытие не дойдёт`)">

<button onclick="event.stopPropagation()">Кликни меня</button>

</body>

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены. То есть, event.stopPropagation() препятствует продвижению события дальше, но на текущем элементе все обработчики будут вызваны.

Для того, чтобы полностью остановить обработку, существует метод event.stopImmediatePropagation(). Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

Не прекращайте всплытие без необходимости. Зачастую прекращение всплытия через event.stopPropagation() имеет свои подводные камни, которые со временем могут стать проблемами. Например:

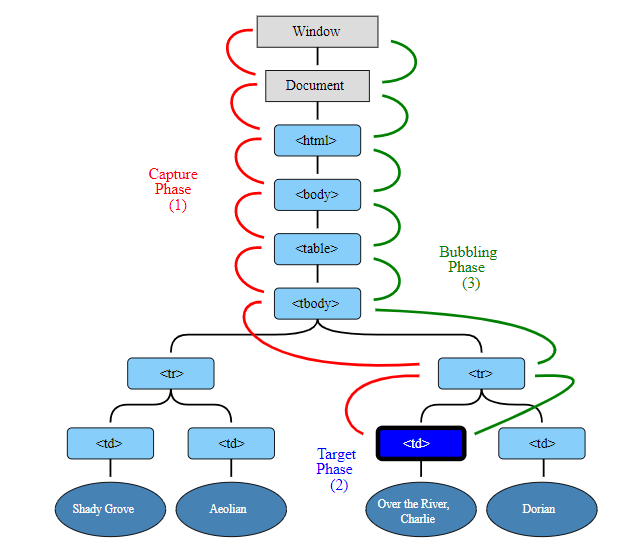
1. При создании вложенного меню, каждое подменю обрабатывает клики на своих элементах и делает для них stopPropagation, чтобы не срабатывало внешнее меню.
2. Позже было решено отслеживать все клики в окне для какой-то своей функциональности, к примеру, для статистики – где вообще кликают пользователи. Обычно используют document.addEventListener('click'…), чтобы отлавливать все клики.
3. Аналитика не будет работать над областью, где клики прекращаются stopPropagation, получилась «мёртвая зона».

[**Погружение**](https://learn.javascript.ru/bubbling-and-capturing#pogruzhenie)

Существует ещё одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной. Стандарт [DOM Events](http://www.w3.org/TR/DOM-Level-3-Events/) описывает 3 фазы прохода события:

1. Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.
2. Фаза цели (target phase) – событие достигло целевого(исходного) элемента.
3. Фаза всплытия (bubbling stage) – событие начинает всплывать.

Картинка из спецификации демонстрирует, как это работает при клике по ячейке <td>, расположенной внутри таблицы:



То есть при клике на <td> событие путешествует по цепочке родителей сначала вниз к элементу (погружается), затем оно достигает целевой элемент (фаза цели), а потом идёт наверх (всплытие), вызывая по пути обработчики.

Обработчики, добавленные через on<event>-свойство или через HTML-атрибуты, или через addEventListener(event, handler) с двумя аргументами, ничего не знают о фазе погружения, а работают только на 2-ой и 3-ей фазах. Чтобы поймать событие на стадии погружения, нужно использовать третий аргумент capture вот так:

elem.addEventListener(..., {capture: true})

// или просто "true", как сокращение для {capture: true}

elem.addEventListener(..., true)

Существуют два варианта значений опции capture:

* Если аргумент false (по умолчанию), то событие будет поймано при всплытии.
* Если аргумент true, то событие будет перехвачено при погружении.

Обратите внимание, что хоть и формально существует 3 фазы, 2-ую фазу («фазу цели»: событие достигло элемента) нельзя обработать отдельно, при её достижении вызываются все обработчики: и на всплытие, и на погружение. Посмотрим и всплытие и погружение в действии:

<style>

body \* {

margin: 10px;

border: 1px solid blue;

}

</style>

<form>FORM

<div>DIV

<p>P</p>

</div>

</form>

<script>

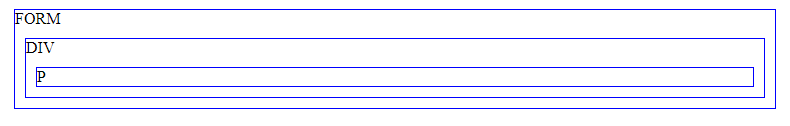
for(let elem of document.querySelectorAll('\*')) {

elem.addEventListener("click", e => alert(`Погружение: ${elem.tagName}`), true);

elem.addEventListener("click", e => alert(`Всплытие: ${elem.tagName}`));

}

</script>



Здесь обработчики назначаются каждому элементу в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события. Если кликнуть по <p>, то последовательность следующая:

1. HTML → BODY → FORM → DIV (фаза погружения, первый обработчик).
2. P (фаза цели, срабатывают обработчики, установленные и на погружение, и на всплытие, так что выведется два раза).
3. DIV → FORM → BODY → HTML (фаза всплытия, второй обработчик).

Существует свойство event.eventPhase, содержащее номер фазы, на которой событие было поймано. Но оно используется редко, обычно это и так известно в обработчике.

Если обработчик добавлен с помощью addEventListener(..., true), то надо передать то же значение аргумента capture в removeEventListener(..., true), когда снимаем обработчик.

Если есть несколько обработчиков одного события, назначенных addEventListener на один элемент, в рамках одной фазы, то их порядок срабатывания – тот же, в котором они установлены:

elem.addEventListener("click", e => alert(1)); // всегда сработает перед следующим

elem.addEventListener("click", e => alert(2));

1. **Делегирование событий. Действия браузера по умолчанию.**

Всплытие и перехват событий позволяет реализовать один из самых важных приёмов разработки – *делегирование*. Идея в том, что если есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, назначается один обработчик на их общего предка. Из него можно получить целевой элемент event.target, понять на каком именно потомке произошло событие и обработать его.

Рассмотрим пример –  таблица. Её HTML (схематично):

<table>

<tr>

<th colspan="3">Квадрат <em>Bagua</em>: Направление, Элемент, Цвет, Значение</th>

</tr>

<tr>

<td>...<strong>Северо-Запад</strong>...</td>

<td>...</td>

<td>...</td>

</tr>

<tr>...ещё 2 строки такого же вида...</tr>

<tr>...ещё 2 строки такого же вида...</tr>

</table>

В этой таблице всего 9 ячеек, но могло бы быть и 99, и даже 9999. Задача – реализовать подсветку ячейки <td> при клике. Вместо того, чтобы назначать обработчик onclick для каждой ячейки <td> (их может быть очень много) – назначим «единый» обработчик на элемент <table>. Он будет использовать event.target, чтобы получить элемент, на котором произошло событие, и подсветить его. Код будет таким:

let selectedTd;

table.onclick = function(event) {

let target = event.target;

if (target.tagName != 'TD') return;

highlight(target);

};

function highlight(td) {

if (selectedTd) { // убрать существующую подсветку, если есть

selectedTd.classList.remove('highlight');

}

selectedTd = td;

selectedTd.classList.add('highlight'); // подсветить новый td

}

Такому коду нет разницы, сколько ячеек в таблице. Можно добавлять, удалять <td> из таблицы динамически в любое время, и подсветка будет стабильно работать. Однако, у текущей версии кода есть недостаток. Клик может быть не на теге <td>, а внутри него. В рассматриваемом примере, если взглянуть на HTML-код таблицы внимательно, видно, что ячейка <td> содержит вложенные теги, например, <strong>:

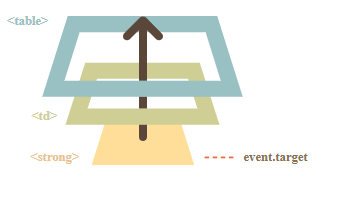
<td>

<strong>Северо-Запад</strong>

...

</td>

Естественно, если клик произойдёт на элементе <strong>, то он станет значением event.target.



Внутри обработчика table.onclick нужно с помощью event.target определить, был клик внутри <td>или нет. Вот улучшенный код:

table.onclick = function(event) {

let td = event.target.closest('td'); // (1)

if (!td) return; // (2)

if (!table.contains(td)) return; // (3)

highlight(td); // (4)

};

Рассмотрим пример:

1. Метод elem.closest(selector) возвращает ближайшего предка, соответствующего селектору. В данном случае нужен <td>, находящийся выше по дереву от исходного элемента.
2. Если event.target не содержится внутри элемента <td>, то вызов вернёт null, и ничего не произойдёт.
3. Если таблицы вложенные, event.target может содержать элемент <td>, находящийся вне текущей таблицы. В таких случаях надо проверить действительно ли это <td> рассматриваемой таблицы.
4. И если это так, то подсветить его.

В итоге получится короткий код подсветки, быстрый и эффективный, не зависящий от того, сколько всего в таблице <td>.

Есть и другие применения делегирования. Например, нужно сделать меню с разными кнопками: «Сохранить (save)», «Загрузить (load)», «Поиск (search)» и т.д. И есть объект с соответствующими методами save, load, search. Надо добавить один обработчик для всего меню и атрибуты data-action для каждой кнопки в соответствии с методами, которые они вызывают:

<button data-action="save">Нажмите, чтобы Сохранить</button>

Обработчик считывает содержимое атрибута и выполняет метод. Рабочий пример:

<div id="menu">

<button data-action="save">Сохранить</button>

<button data-action="load">Загрузить</button>

<button data-action="search">Поиск</button>

</div>

<script>

class Menu {

constructor(elem) {

this.\_elem = elem;

elem.onclick = this.onClick.bind(this); // (\*)

}

save() {

alert('сохраняю');

}

load() {

alert('загружаю');

}

search() {

alert('ищу');

}

onClick(event) {

let action = event.target.dataset.action;

if (action) {

this[action]();

}

};

}

new Menu(menu);

</script>



Обратите внимание, что метод this.onClick в строке, отмеченной звёздочкой (\*), привязывается к контексту текущего объекта this. Это важно, т.к. иначе this внутри него будет ссылаться на DOM-элемент (elem), а не на объект Menu, и this[action] будет не тем, что нужно.

Здесь преимущество делегирования заключается в следующем:

* Не нужно писать код, чтобы присвоить обработчик каждой кнопке. Достаточно просто создать один метод и поместить его в разметку.
* Структура HTML становится по-настоящему гибкой. Можно добавлять/удалять кнопки в любое время.

Также можно использовать классы .action-save, .action-load, но подход с использованием атрибутов data-action является более семантичным. Их можно использовать и для стилизации в правилах CSS.

[**Приём проектирования «поведение»**](https://learn.javascript.ru/event-delegation#priyom-proektirovaniya-povedenie)

Делегирование событий можно использовать для добавления элементам «поведения» (behavior), декларативно задавая обработчики установкой специальных HTML-атрибутов и классов. Приём проектирования «поведение» состоит из двух частей:

1. Элементу устанавливается пользовательский атрибут, описывающий его поведение.
2. При помощи делегирования назначается обработчик на документ, который ловит все клики (или другие события) и, если элемент имеет нужный атрибут – производит соответствующее действие.

Например, здесь HTML-атрибут data-counter добавляет кнопкам поведение: «увеличить значение при клике»:

Счётчик: <input type="button" value="1" data-counter>

Ещё счётчик: <input type="button" value="2" data-counter>

<script>

document.addEventListener('click', function(event) {

if (event.target.dataset.counter != undefined) { // если есть атрибут

event.target.value++;

}

});

</script>





Если нажать на кнопку – значение увеличится.

Элементов с атрибутом data-counter может быть сколько угодно. Новые могут добавляться в HTML-код в любой момент. При помощи делегирования, фактически, добавляется новый «псевдостандартный» атрибут в HTML, который добавляет элементу новую возможность («поведение»).

Когда устанавливается обработчик событий на объект document, всегда надо использовать метод addEventListener, а не document.on<событие>, т.к. в случае последнего могут возникать конфликты: новые обработчики будут перезаписывать уже существующие. Для реального проекта совершенно нормально иметь много обработчиков на элементе document, установленных из разных частей кода.

Ещё один пример поведения: при клике на элемент с атрибутом data-toggle-id скрывается/показывается элемент с заданным id:

<button data-toggle-id="subscribe-mail">

Показать форму подписки

</button>

<form id="subscribe-mail" hidden>

Ваша почта: <input type="email">

</form>

<script>

document.addEventListener('click', function(event) {

let id = event.target.dataset.toggleId;

if (!id) return;

let elem = document.getElementById(id);

elem.hidden = !elem.hidden;

});

</script>





Теперь для того, чтобы добавить скрытие-раскрытие любому элементу, даже не надо знать JavaScript, можно просто написать атрибут data-toggle-id. Это бывает очень удобно – не нужно писать JavaScript-код для каждого элемента, который должен так себя вести. Просто используем поведение. Обработчики на уровне документа сделают это возможным для элемента в любом месте страницы.

Можно комбинировать несколько вариантов поведения на одном элементе. Шаблон «поведение» может служить альтернативой для фрагментов JS-кода в вёрстке.

**Действия браузера по умолчанию.**

Многие события автоматически влекут за собой действие браузера.

Например:

* клик по ссылке инициирует переход на новый URL;
* нажатие на кнопку «отправить» в форме – отсылку её на сервер;
* зажатие кнопки мыши над текстом и её движение в таком состоянии – инициирует его выделение.

Если событие обрабатывается в JavaScript, то зачастую такое действие браузера не нужно. Его можно отменить.

[**Отмена действия браузера**](https://learn.javascript.ru/default-browser-action#otmena-deystviya-brauzera)

Есть два способа отменить действие браузера:

* Основной способ – это воспользоваться объектом event. Для отмены действия браузера существует стандартный метод event.preventDefault().
* Если же обработчик назначен через on<событие> (не через addEventListener), то также можно вернуть false из обработчика.

В следующем примере при клике по ссылке переход не произойдет:

<a href="/" onclick="return false">Нажми здесь</a>

или

<a href="/" onclick="event.preventDefault()">здесь</a>

Обычно значение, которое возвращает обработчик события, игнорируется. Единственное исключение – это return false из обработчика, назначенного через on<событие>. В других случаях return не нужен, он никак не обрабатывается.

Рассмотрим пример меню для сайта:

<ul id="menu" class="menu">

<li><a href="/html">HTML</a></li>

<li><a href="/javascript">JavaScript</a></li>

<li><a href="/css">CSS</a></li>

</ul>

Данный пример при помощи CSS может выглядеть так:



В HTML-разметке все элементы меню являются не кнопками, а ссылками, то есть тегами <a>. В этом подходе есть некоторые преимущества, например:

* некоторые посетители очень любят сочетание «правый клик – открыть в новом окне». Если использовать <button> или <span>, то данное сочетание работать не будет;
* поисковые движки переходят по ссылкам <a href="..."> при индексации.

Поэтому в разметке используется <a>. Но в примере требуется обрабатывать клики в JavaScript, а стандартное действие браузера (переход по ссылке) – отменить. Например, вот так:

menu.onclick = function(event) {

if (event.target.nodeName != 'A') return;

let href = event.target.getAttribute('href');

alert( href );

return false; // отменить действие браузера (переход по ссылке)

};

Если убрать return false, то после выполнения обработчика события браузер выполнит «действие по умолчанию» – переход по адресу из href. А это здесь не нужно, так как клик обрабатывается.

Использование здесь делегирования событий делает меню очень гибким. Можно добавить вложенные списки и стилизовать их с помощью CSS – обработчик не потребует изменений.

Некоторые события естественным образом вытекают друг из друга. Если отменить первое событие, то последующие не возникнут. Например, событие mousedown для поля <input> приводит к фокусировке на нём и запускает событие focus. Если отменить событие mousedown, то фокусирования не произойдёт.

В следующем примере если нажать на первом <input> – произойдет событие focus. Но если нажать по второму элементу, то события focus не будет.

<input value="Фокус работает" onfocus="this.value=''">

<input onmousedown="return false" onfocus="this.value=''" value="Кликни меня">



Это потому, что отменено стандартное действие mousedown. Впрочем, фокусировка на элементе всё ещё возможна, если использовать другой способ. Например, нажатием клавиши Tab можно перейти от первого поля ввода ко второму. Но только не через клик мышью на элемент, это больше не работает.

Необязательная опция passive: true для addEventListener сигнализирует браузеру, что обработчик не собирается выполнять preventDefault(). Это может быть полезно, так как есть некоторые события, как touchmove на мобильных устройствах (когда пользователь перемещает палец по экрану), которое по умолчанию начинает прокрутку, но можно отменить это действие, используя preventDefault() в обработчике. Поэтому, когда браузер обнаружит такое событие, он должен для начала запустить все обработчики и после, если preventDefault не вызывается нигде, он может начать прокрутку. Это может вызвать ненужные задержки в пользовательском интерфейсе.

Опция passive: true сообщает браузеру, что обработчик не собирается отменять прокрутку. Тогда браузер начинает её немедленно, обеспечивая максимально плавный интерфейс, параллельно обрабатывая событие. Для некоторых браузеров (Firefox, Chrome) опция passive по умолчанию включена в true для таких событий, как touchstart и touchmove.

Свойство event.defaultPrevented установлено в true, если действие по умолчанию было предотвращено, и false, если нет. Рассмотрим практическое применение этого свойства для улучшения архитектуры. event.defaultPrevented можно использовать вместо  остановки всплытия с помощью event.stopPropagation(), чтобы просигналить другим обработчикам, что событие обработано.

Рассмотрим практический пример. По умолчанию браузер при событии contextmenu (клик правой кнопкой мыши) показывает контекстное меню со стандартными опциями. Можно отменить событие по умолчанию и показать своё меню, как здесь:

<button>Правый клик вызывает контекстное меню браузера</button>

<button oncontextmenu="alert('Рисуем наше меню'); return false">

Правый клик вызывает наше контекстное меню

</button>



Теперь в дополнение к этому контекстному меню реализуем контекстное меню для всего документа. При правом клике должно показываться ближайшее контекстное меню.

<p>Правый клик здесь вызывает контекстное меню документа</p>

<button id="elem">Правый клик здесь вызывает контекстное меню кнопки</button>

<script>

elem.oncontextmenu = function(event) {

event.preventDefault();

alert("Контекстное меню кнопки");

};

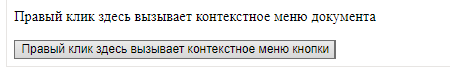
document.oncontextmenu = function(event) {

event.preventDefault();

alert("Контекстное меню документа");

};

</script>



Проблема заключается в том, что когда происходит клик по элементу elem, то получаем два меню: контекстное меню для кнопки и (событие всплывает вверх) контекстное меню для документа. Чтобы это исправить, надо остановить всплытие когда обрабатывается правый клик в обработчике на кнопке, и вызвать event.stopPropagation():

<p>Правый клик вызывает меню документа</p>

<button id="elem">Правый клик вызывает меню кнопки (добавлен event.stopPropagation)</button>

<script>

elem.oncontextmenu = function(event) {

event.preventDefault();

event.stopPropagation();

alert("Контекстное меню кнопки");

};

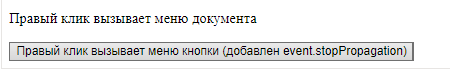
document.oncontextmenu = function(event) {

event.preventDefault();

alert("Контекстное меню документа");

};

</script>



Теперь контекстное меню для кнопки работает как задумано. Но навсегда запрещен доступ к информации о правых кликах для любого внешнего кода, включая счётчики, которые могли бы собирать статистику, и т.п. Это неудобно.

Альтернативным решением было бы проверить в обработчике document, было ли отменено действие по умолчанию? Если да, тогда событие было обработано, и не нужно на него реагировать.

<p>Правый клик вызывает меню документа (добавлена проверка event.defaultPrevented)</p>

<button id="elem">Правый клик вызывает меню кнопки</button>

<script>

elem.oncontextmenu = function(event) {

event.preventDefault();

alert("Контекстное меню кнопки");

};

document.oncontextmenu = function(event) {

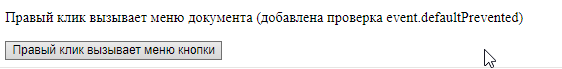
if (event.defaultPrevented) return;

event.preventDefault();

alert("Контекстное меню документа");

};

</script>



Сейчас всё работает правильно. Если есть вложенные элементы и каждый из них имеет контекстное меню, то код также будет работать. Просто надо убедиться, что осуществляется проверка event.defaultPrevented в каждом обработчике contextmenu.

event.stopPropagation() и event.preventDefault() (также известный как return false) – это две разные функции. Они никак не связаны друг с другом.

1. **Генерация событий.**

Можно не только назначать обработчики, но и генерировать события из JavaScript-кода. Пользовательские события могут быть использованы при создании графических компонентов. Например, корневой элемент меню может генерировать события, относящиеся к этому меню: open (меню раскрыто), select (выбран пункт меню) и т.п. Также можно генерировать встроенные события, такие как click, mousedown и другие, что бывает полезно для автоматического тестирования.

[**Конструктор Event**](https://learn.javascript.ru/dispatch-events#konstruktor-event)

Встроенные классы для событий формируют иерархию, аналогично классам для DOM-элементов. Её корнем является встроенный класс [Event](http://www.w3.org/TR/dom/#event). Событие встроенного класса Event можно создать так:

let event = new Event(type[, options]);

Где:

* type – тип события, строка, например, "click" или другая;
* options – объект с двумя необязательными свойствами:
* bubbles: true/false – если true, тогда событие всплывает;
* cancelable: true/false – если true, тогда можно отменить действие по умолчанию.

По умолчанию оба свойства установлены в false: {bubbles: false, cancelable: false}.

После того, как объект события создан, надо запустить событие на элементе, вызвав метод elem.dispatchEvent(event). Затем обработчики отреагируют на него, как будто это обычное встроенное событие. Если при создании указан флаг bubbles, то оно будет всплывать. В примере ниже событие click инициируется JavaScript-кодом так же, как если бы кликнули по кнопке:

<button id="elem" onclick="alert('Клик!');">Автоклик</button>

<script>

let event = new Event("click");

elem.dispatchEvent(event);

</script>

В то же время можно легко отличить «настоящее» событие от сгенерированного кодом. Свойство event.isTrusted принимает значение true для событий, порождаемых реальными действиями пользователя, и false для генерируемых кодом.

Можно создать всплывающее событие с именем "hello" и поймать его на document. Всё, что нужно сделать – это установить флаг bubbles в true:

<h1 id="elem">Привет из кода!</h1>

<script>

document.addEventListener("hello", function(event) { // (1)

alert("Привет от " + event.target.tagName); // Привет от H1

});

// запуск события на элементе

let event = new Event("hello", {bubbles: true}); // (2)

elem.dispatchEvent(event);

</script>

Надо использовать addEventListener для собственных событий, т.к. on<event>-свойства существуют только для встроенных событий, то есть document.onhello не сработает. Надо передавать флаг bubbles:true, иначе событие не будет всплывать.

Механизм всплытия идентичен как для встроенного события (click), так и для пользовательского события (hello). Также одинакова работа фаз всплытия и погружения.

**Конструкторы** [**MouseEvent, KeyboardEvent и другие**](https://learn.javascript.ru/dispatch-events#mouseevent-keyboardevent-i-drugie)

Для некоторых конкретных типов событий есть свои специфические конструкторы. Вот небольшой список конструкторов для различных событий пользовательского интерфейса, которые можно найти в спецификации [UI Event](https://www.w3.org/TR/uievents): UIEvent, FocusEvent, MouseEvent, WheelEvent, KeyboardEvent.

Стоит использовать их вместо new Event, если надо создавать такие события. К примеру, new MouseEvent("click"). Специфический конструктор позволяет указать стандартные свойства для данного типа события. Например, clientX/clientY для события мыши:

let event = new MouseEvent("click", {

bubbles: true,

cancelable: true,

clientX: 100,

clientY: 100

});

alert(event.clientX); // 100

Обратите внимание: это нельзя было бы сделать с обычным конструктором Event. Проверим:

let event = new Event("click", {

bubbles: true, // только свойства bubbles и cancelable

cancelable: true, // работают в конструкторе Event

clientX: 100,

clientY: 100

});

alert(event.clientX); // undefined, неизвестное свойство проигнорировано

Впрочем, использование конкретного конструктора не является обязательным, можно обойтись Event, а свойства записать в объект отдельно, после создания, вот так: event.clientX=100. Здесь это скорее вопрос удобства и желания следовать правилам. События, которые генерирует браузер, всегда имеют правильный тип. Полный список свойств по типам событий можно найти в спецификации, например для [MouseEvent](https://www.w3.org/TR/uievents/" \l "mouseevent).

[**Пользовательские события**](https://learn.javascript.ru/dispatch-events#polzovatelskie-sobytiya)

Для генерации пользовательских событий, таких как "hello", следует использовать конструктор new CustomEvent. Технически [CustomEvent](https://dom.spec.whatwg.org/" \l "customevent) абсолютно идентичен Event за исключением одной небольшой детали. У второго аргумента-объекта есть дополнительное свойство detail, в котором можно указывать информацию для передачи в событие. Например:

<h1 id="elem">Привет для Васи!</h1>

<script>

elem.addEventListener("hello", function(event) {

alert(event.detail.name);

});

elem.dispatchEvent(new CustomEvent("hello", {

detail: { name: "Вася" }

}));

</script>

Свойство detail может содержать любые данные. Надо сказать, что никто не мешает и в обычное new Event записать любые свойства. Но CustomEvent предоставляет специальное поле detail во избежание конфликтов с другими свойствами события. Класс события содержит информацию о том, что это за событие, и если оно не браузерное, а пользовательское, то стоит всё-таки использовать CustomEvent.

На сгенерированном событии обработчик может вызвать метод event.preventDefault(), если задан флаг cancelable:true. Для пользовательских событий, названия которых браузеру неизвестны, соответственно, нет никаких действий браузера по умолчанию. Но код, который генерирует событие, может предусматривать какие-то ещё действия после dispatchEvent.

Вызов event.preventDefault() является возможностью для обработчика события сообщить в сгенерировавший событие код, что эти действия надо отменить. Тогда вызов elem.dispatchEvent(event) возвратит false. И код, сгенерировавший событие, приостановит выполнение.

В примере ниже есть функция hide(), которая при вызове генерирует событие "hide" на элементе #rabbit, уведомляя всех интересующихся, что кролик собирается спрятаться. Любой обработчик может узнать об этом, подписавшись на событие hide через rabbit.addEventListener('hide',...) и, при желании, отменить действие по умолчанию через event.preventDefault(). Тогда кролик не исчезнет:

<pre id="rabbit">

|\ /|

\|\_|/

/. .\

=\\_Y\_/=

{>o<}

</pre>

<script>

function hide() {

let event = new CustomEvent("hide", {

cancelable: true

});

if (!rabbit.dispatchEvent(event)) {

alert('действие отменено обработчиком');

} else {

rabbit.hidden = true;

}

}

rabbit.addEventListener('hide', function(event) {

if (confirm("Вызвать preventDefault?")) {

event.preventDefault();

}

});

setTimeout(hide, 2000);

</script>

Обычно события обрабатываются асинхронно. То есть, если браузер обрабатывает onclick и в процессе этого произойдёт новое событие, то оно ждёт, пока закончится обработка onclick. Исключением является ситуация, когда событие инициировано из обработчика другого события. Тогда управление сначала переходит в обработчик вложенного события и уже после этого возвращается назад. В примере ниже событие menu-open обрабатывается синхронно во время обработки onclick:

<button id="menu">Меню (нажми меня)</button>

<script>

menu.onclick = function() {

alert(1);

menu.dispatchEvent(new CustomEvent("menu-open", {

bubbles: true

}));

alert(2);

};

document.addEventListener('menu-open', () => alert('вложенное событие'))

</script>

Обратите внимание, что вложенное событие menu-open всплывает и обрабатывается на document. Обработка вложенного события полностью завершается до того, как управление возвращается во внешний код (onclick). Это справедливо не только для dispatchEvent, но и для других случаев. JavaScript в обработчике события может вызвать другие методы, которые приведут к другим событиям – они тоже обрабатываются синхронно.

Также можно либо поместить dispatchEvent (или любой другой код инициирующий события) в конец обработчика onclick, либо обернуть такой код в setTimeout(..., 0):

<button id="menu">Меню (нажми меня)</button>

<script>

menu.onclick = function() {

alert(1);

setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {

bubbles: true

})));

alert(2);

};

document.addEventListener('menu-open', () => alert('вложенное событие'))

</script>

Теперь dispatchEvent запускается асинхронно после исполнения текущего кода, включая mouse.onclick, поэтому обработчики полностью независимы.

1. **События мыши. События mouseover/out, mouseenter/leave.**

События мыши происходят не только от манипуляций мышью, но и эмулируются на сенсорных устройствах, чтобы сделать их совместимыми.

[**Типы событий мыши**](https://learn.javascript.ru/mouse-events-basics#tipy-sobytiy-myshi)

Можно разделить события мыши на две категории: простые и комплексные.

[Простые события](https://learn.javascript.ru/mouse-events-basics#prostye-sobytiya) (наиболее часто используемые):

* mousedown/mouseup – кнопка мыши нажата/отпущена над элементом;
* mouseover/mouseout – курсор мыши появляется над элементом и уходит с него;
* mousemove – каждое движение мыши над элементом генерирует это событие.

[Комплексные события](https://learn.javascript.ru/mouse-events-basics#kompleksnye-sobytiya):

* click – вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши;
* contextmenu – вызывается при mousedown правой кнопкой мыши;
* dblclick – вызывается двойным кликом на элементе.

Комплексные события состоят из простых. С ними удобнее работать.

[**Порядок событий**](https://learn.javascript.ru/mouse-events-basics#poryadok-sobytiy)

Одно действие может вызвать несколько событий. Например, клик мышью вначале вызывает mousedown, когда кнопка нажата, затем mouseup и click, когда она отпущена. В случае, когда одно действие инициирует несколько событий, порядок их выполнения фиксирован. То есть обработчики событий вызываются в следующем порядке: mousedown → mouseup → click. События обрабатываются в той же последовательности: onmouseup завершается до того, как запускается onclick.

События, связанные с кликом, всегда имеют свойство which, которое позволяет определить нажатую кнопку мыши. Это свойство не используется для событий click и contextmenu, поскольку первое происходит только при нажатии левой кнопкой мыши, а второе – правой. События mousedown и mouseup срабатывают на любой кнопке и свойство which позволяет различать между собой «нажатие правой кнопки» и «нажатие левой кнопки» мыши.

Есть три возможных значения:

* event.which == 1 – левая кнопка;
* event.which == 2 – средняя кнопка;
* event.which == 3 – правая кнопка.

[**Модификаторы: shift, alt, ctrl и meta**](https://learn.javascript.ru/mouse-events-basics#modifikatory-shift-alt-ctrl-i-meta)

Все события мыши включают в себя информацию о нажатых клавишах-модификаторах. Их свойства: shiftKey, altKey, ctrlKey, metaKey (Cmd для Mac). Например, кнопка внизу работает только при комбинации Alt+Shift+клик:

<button id="button">Нажми Alt+Shift+Click на мне!</button>

<script>

button.onclick = function(event) {

if (event.altKey && event.shiftKey) {

alert('Ура!');

}

};

</script>



В Windows и Linux клавишами-модификаторами являются Alt, Shift и Ctrl. На Mac есть ещё одна: Cmd, она соответствует свойству metaKey. В большинстве случаев, когда в Windows/Linux используется Ctrl, на Mac пользователи используют Cmd. Поэтому, когда пользователь Windows нажимает Ctrl+Enter и Ctrl+A, пользователь Mac нажимает Cmd+Enter или Cmd+A, и так далее, большинство приложений используют Cmd вместо Ctrl. Поэтому, если надо поддерживать такие комбинации, как Ctrl+клик, то для Mac имеет смысл использовать Cmd+клик. Это удобней для пользователей Mac.

Левый клик в сочетании с Ctrl интерпретируется как правый клик на Mac и генерирует событие contextmenu, а не click как на Windows/Linux. Поэтому, если надо, чтобы пользователям всех операционных систем было удобно, то вместе с ctrlKey нужно использовать metaKey. Для JS-кода это означает, что надо проверить if (event.ctrlKey || event.metaKey).

Все события мыши имеют координаты двух видов: относительно окна (event.clientX и event.clientY) и относительно документа (event.pageX и event.pageY).

Клики мышью имеют побочный эффект, который может быть неудобен в некоторых интерфейсах: двойной клик мышью выделяет текст. Это действие браузера по умолчанию при наступлении события mousedown. Поэтому альтернативным решением проблемы будет обработать событие mousedown и предотвратить его:

<b ondblclick="alert('Клик!')" onmousedown="return false">

Сделайте двойной клик на мне

</b>

Теперь выделенный жирным элемент не выделяется при двойном клике. Текст внутри него по-прежнему можно выделить. Однако, выделение должно начаться не на самом тексте, а до него или после.

Вместо предотвращения выделения, можно отменить его «постфактум» в обработчике событий. Например, так:

<b ondblclick="getSelection().removeAllRanges()">

Сделайте двойной клик на мне

</b>

При двойном клике на элементе, выделенном жирным шрифтом, выделение появится и тут же будет немедленно снято. Выглядит это не очень красиво.

Если надо отключить выделение для защиты контента от копирования, то можно использовать другое событие: oncopy.

<div oncopy="alert('Копирование запрещено!');return false">

Уважаемый пользователь,

Копирование информации запрещено.

</div>

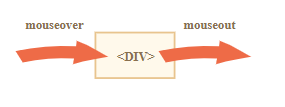
Скопировать текст в <div> не получится, потому что срабатывание события oncopy по умолчанию запрещено.

**Мышь: mouseover/out, mouseenter/leave**

Рассмотрим подробнее события, возникающие при движении указателя (курсора) мыши над элементами страницы.

[**Mouseover/mouseout, relatedTarget**](https://learn.javascript.ru/mousemove-mouseover-mouseout-mouseenter-mouseleave#mouseover-mouseout-relatedtarget)

Событие mouseover происходит в момент, когда курсор оказывается над элементом, а событие mouseout – в момент, когда курсор уходит с элемента.



Эти события являются особенными, потому что у них имеется свойство relatedTarget. Оно дополняет target. Когда мышь переходит с одного элемента на другой, то один из них будет храниться в target, а другой в relatedTarget.

Для события mouseover:

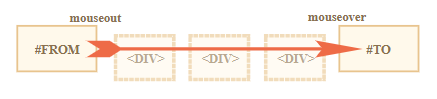
* event.target – это элемент, на который курсор перешёл;
* event.relatedTarget – это элемент, с которого курсор ушёл.

Для события mouseout наоборот:

* event.target – это элемент, с которого курсор ушёл;
* event.relatedTarget – это элемент, на который курсор перешёл.

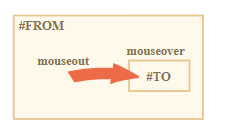
Свойство relatedTarget может быть null, это означает, что указатель мыши перешёл не с другого элемента, а из-за пределов окна браузера. Или, наоборот, ушёл за пределы окна.

Событие mousemove происходит при движении мыши. Однако, это не означает, что указанное событие генерируется при прохождении каждого пикселя. Браузер периодически проверяет позицию курсора и, заметив изменения, генерирует события mousemove. Это означает, что если пользователь двигает мышкой очень быстро, то некоторые DOM-элементы могут быть пропущены:



Если курсор мыши двигается очень быстро с #FROM на #TO элемент, как это показано выше, то лежащие между ними элементы <div> (или некоторые из них) могут быть пропущены. Событие mouseout может запуститься на элементе #FROM и затем сразу же сгенерируется mouseover на элементе #TO.

Представьте ситуацию – курсор мыши перешёл на элемент. Сгенерировано событие mouseover. Затем курсор перешёл на дочерний элемент, и сгенерировано mouseout. То есть курсор всё ещё на элементе, но событие mouseout.



По логике браузера, курсор мыши может быть только над одним элементом в любой момент времени – над самым глубоко вложенным (и верхним по z-index). Таким образом, если курсор переходит на другой элемент (пусть даже дочерний), то он покидает предыдущий.

[**События mouseenter и mouseleave**](https://learn.javascript.ru/mousemove-mouseover-mouseout-mouseenter-mouseleave#sobytiya-mouseenter-i-mouseleave)

События mouseenter/mouseleave похожи на mouseover/mouseout. Они тоже генерируются, когда курсор мыши переходит на элемент или покидает его. Но есть и пара важных отличий:

1. Переходы внутри элемента по дочерним элементам не считаются.
2. События mouseenter/mouseleave не всплывают.

Когда курсор становится над элементом – генерируется mouseenter, и не имеет значения, где именно находится курсор внутри элемента. Событие mouseleave происходит, когда курсор покидает элемент.

События mouseenter/leave простые и легкие в использовании, но они не всплывают, а это значит что их нельзя делегировать.

1. **События клавиатуры: keyup, keydown. Прокрутка: событие scroll.**

**Прокрутка: событие scroll.**

События прокрутки позволяют реагировать на прокрутку страницы или элемента. Есть много хороших вещей, которые можно сделать при этом.

Например:

* Показать/скрыть дополнительные элементы управления или информацию, основываясь на том, в какой части документа находится пользователь.
* Загрузить больше данных, когда пользователь прокручивает страницу вниз до конца.

Вот небольшая функция для отображения текущей прокрутки:

window.addEventListener('scroll', function() {

document.getElementById('showScroll').innerHTML = pageYOffset + 'px';

});

В действии:

Текущая прокрутка = 563.3333740234375px

Событие scroll работает как на window, так и на прокручиваемых элементах.

[Предотвращение прокрутки](https://learn.javascript.ru/onscroll#predotvraschenie-prokrutki)

Как можно сделать что-то непрокручиваемым? Нельзя предотвратить прокрутку, используя event.preventDefault() в обработчике onscroll, потому что он срабатывает после того, как прокрутка уже произошла.

Но можно предотвратить прокрутку, используя event.preventDefault() на событии, которое вызывает прокрутку.

Например:

* На событии wheel – прокрутка колеса мыши («прокручивание» тачпада также его генерирует).
* На событии keydown для клавиш pageUp и pageDown.

Если поставить на них обработчики, в которых вызвать event.preventDefault(), то прокрутка не начнётся.

Иногда это может помочь, но более надёжный способ – использовать CSS, чтобы сделать что-то непрокручиваемым, например, свойство overflow.

Вот несколько задач, которые вы можете решить или просмотреть, чтобы увидеть применение onscroll.

**Клавиатура: keyup, keydown.**

События клавиатуры должны использоваться, если необходимо обрабатывать взаимодействие пользователя с клавиатурой (в том числе виртуальной). К примеру, если нужно реагировать на стрелочные клавиши Up и Down или горячие клавиши (включая комбинации клавиш).

Событие keydown происходит при нажатии клавиши, а keyup – при отпускании.

У объекта события есть свойство key, чьё значение – символ, и свойство code – «физический код клавиши». Например, клавишу Z можно нажать с клавишей Shift и без неё. В результате получится два разных символа: z в нижнем регистре и Z в верхнем регистре.

Свойство event.key – это непосредственно символ, и он может различаться. Но event.code всегда будет тот же:

| **Клавиша** | **event.key** | **event.code** |
| --- | --- | --- |
| Z | z (нижний регистр) | KeyZ |
| Shift+Z | Z (Верхний регистр) | KeyZ |

Если пользователь работает с разными языками, то при переключении на другой язык символ изменится с "Z" на совершенно другой. Получившееся станет новым значением event.key, тогда как event.code останется тем же: "KeyZ".

У каждой клавиши есть код, который зависит от её расположения на клавиатуре. Подробно о клавишных кодах можно прочитать в [спецификации о кодах событий UI](https://www.w3.org/TR/uievents-code/). Например:

* буквенные клавиши имеют коды по типу "Key<буква>": "KeyA", "KeyB" и т.д.;
* коды числовых клавиш строятся по принципу: "Digit<число>": "Digit0", "Digit1" и т.д.;
* код специальных клавиш – это их имя: "Enter", "Backspace", "Tab" и т.д.

Существует несколько широко распространённых раскладок клавиатуры, и в спецификации приведены клавишные коды к каждой из них.

Регистр важен: правильно KeyZ, а не keyZ. Условие event.code=="keyZ" работать не будет: первая буква в слове "Key" должна быть заглавная.

Если клавиша не буквенно-цифровая, например, Shift или F1, или какая-либо другая специальная клавиша, то значение свойства event.key примерно тоже, что и у event.code:

| **Клавиша** | **event.key** | **event.code** |
| --- | --- | --- |
| F1 | F1 | F1 |
| Backspace | Backspace | Backspace |
| Shift | Shift | ShiftRight или ShiftLeft |

Большинство клавиатур имеют по две клавиши Shift: слева и справа. event.code уточняет, какая именно из них была нажата, в то время как event.key сообщает о том, что вообще было нажато (Shift).

Допустим, надо обработать горячую клавишу Ctrl+Z (или Cmd+Z для Mac). Можно поставить обработчик событий на keydown и проверять, какая клавиша была нажата. Возникает вопрос: надо проверять значение event.key или event.code?

С одной стороны, значение event.key изменяется в зависимости от языка, и, если у пользователя установлено в ОС несколько языков, и он переключается между ними, нажатие на одну и ту же клавишу будет давать разные символы. Так что имеет смысл проверять event.code, ведь его значение всегда одно и тоже. Вот пример кода:

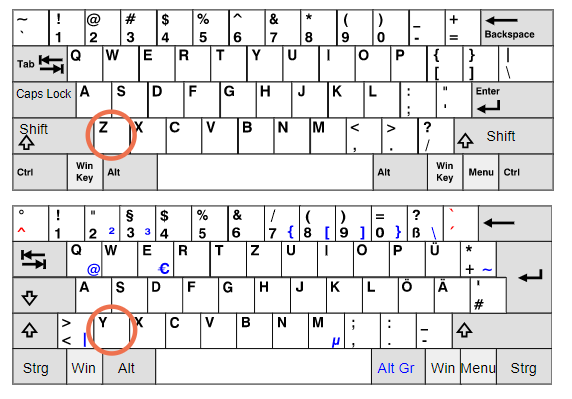
document.addEventListener('keydown', function(event) {

if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {

alert('Отменить!')

}

});



С другой стороны, с event.code тоже есть проблемы. На разных раскладках к одной и той же клавише могут быть привязаны разные символы (буквы). Например, вот схема стандартной (US) раскладки («QWERTY») и под ней немецкой («QWERTZ») раскладки (предоставлены Википедией):

Для одной и той же клавиши в американской раскладке значение event.code равно «Z», в то время как в немецкой «Y». Таким образом, для пользователей с немецкой раскладкой event.code при нажатии на «Y» будет равен KeyZ.

В [спецификации](https://www.w3.org/TR/uievents-code/#table-key-code-alphanumeric-writing-system) напрямую упоминается такое поведение.

* Преимущество event.code заключается в том, что его значение всегда остается неизменным, будучи привязанным к физическому местоположению клавиши, даже если пользователь меняет язык. Так что горячие клавиши, использующие это свойство, будут работать даже в случае переключения языка.
* event.code может содержать неправильный символ на нестандартной раскладке. Одни и те же буквы на разных раскладках могут сопоставляться с разными физическими клавишами, что приводит к разным кодам. Это происходит не со всеми кодами, а с несколькими, например, keyA, keyQ, keyZ, и не происходит со специальными клавишами, такими как Shift. Полный список проблемных кодов можно найти в [спецификации](https://www.w3.org/TR/uievents-code/#table-key-code-alphanumeric-writing-system).

Таким образом, чтобы отслеживать символы, зависящие от раскладки, event.key надёжнее.

При долгом нажатии клавиши возникает автоповтор: keydown срабатывает снова и снова, и когда клавишу отпускают, то отрабатывает keyup. Так что ситуация, когда много keydown и один keyup, абсолютно нормальна. Для событий, вызванных автоповтором, у объекта события свойство event.repeat равно true.

Действия по умолчанию весьма разнообразны, много чего можно инициировать нажатием на клавиатуре: появление символа, удаление символа, прокрутка страницы, открытие диалогового окна браузера «Сохранить» и так далее. Предотвращение стандартного действия с помощью event.preventDefault() работает практически во всех сценариях, кроме тех, которые происходят на уровне операционной системы. Например, комбинация Alt+F4 инициирует закрытие браузера в Windows, как бы эта комбинация не обрабатывалась в JavaScript. Для примера, <input> ниже ожидает телефонный номер, так что ничего кроме чисел, +, () или - принято не будет:

<script>

function checkPhoneKey(key) {

return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key == '-';

}

</script>

<input onkeydown="return checkPhoneKey(event.key)" placeholder="Введите номер телефона" type="tel">





Заметьте, что специальные клавиши по типу Backspace, Left, Right, Ctrl+V не работают как должны в поле для ввода. Это побочный эффект жесткого фильтра checkPhoneKey. Доработаем код:

<script>

function checkPhoneKey(key) {

return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')' || key == '-' ||

key == 'ArrowLeft' || key == 'ArrowRight' || key == 'Delete' || key == 'Backspace';

}

</script>

<input onkeydown="return checkPhoneKey(event.key)" placeholder="Введите номер телефона" type="tel">





1. **События жизненного цикла HTML-страницы.**

У жизненного цикла HTML-страницы есть три важных события:

* DOMContentLoaded – браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки <img> и стили, могут быть ещё не загружены;
* load – браузер загрузил HTML и внешние ресурсы (картинки, стили и т.д.);
* beforeunload/unload – пользователь покидает страницу.

Каждое из этих событий может быть полезно:

* событие DOMContentLoaded – DOM готов, так что обработчик может искать DOM-узлы и инициализировать интерфейс;
* событие load – внешние ресурсы были загружены, стили применены, размеры картинок известны и т.д. Событие beforeunload – пользователь покидает страницу. Можно проверить, сохранил ли он изменения и спросить, на самом ли деле он хочет уйти;
* unload – пользователь почти ушёл, но всё ещё можно запустить некоторые операции, например, отправить статистику.

Рассмотрим эти события подробнее.

[**DOMContentLoaded**](https://learn.javascript.ru/onload-ondomcontentloaded#domcontentloaded)

Событие DOMContentLoaded срабатывает на объекте document. Необходимо использовать addEventListener, чтобы поймать его:

document.addEventListener("DOMContentLoaded", ready);

// не "document.onDOMContentLoaded = ..."

Например:

<script>

function ready() {

alert('DOM готов');

// изображение ещё не загружено (если не было закешировано), поэтому размер будет 0x0

alert(`Размер изображения: ${img.offsetWidth}x${img.offsetHeight}`);

}

document.addEventListener("DOMContentLoaded", ready);

</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">

В этом примере обработчик DOMContentLoaded запустится, когда документ загрузится, так что он увидит все элементы, включая расположенный ниже <img>. Но он не дожидается, пока загрузится изображение. Поэтому alert покажет нулевой размер. На первый взгляд событие DOMContentLoaded очень простое. DOM-дерево готово – получаем событие. Хотя тут есть несколько особенностей.

Когда браузер обрабатывает HTML-документ и встречает тег <script>, он должен выполнить его перед тем, как продолжить строить DOM. Это делается на случай, если скрипт захочет изменить DOM или даже дописать в него (document.write), так что DOMContentLoaded должен подождать. Поэтому DOMContentLoaded определённо случится после таких скриптов:

<script>

document.addEventListener("DOMContentLoaded", () => {

alert("DOM готов!");

});

</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<script>

alert("Библиотека загружена, встроенный скрипт выполнен");

</script>

В примере выше сначала будет выведено «Библиотека загружена…», а затем «DOM готов!» (все скрипты выполнены).

Есть два исключения из этого правила:

1. Скрипты с атрибутом async (будет рассмотрен [немного позже](https://learn.javascript.ru/script-async-defer)) не блокируют DOMContentLoaded.
2. Скрипты, сгенерированные динамически при помощи document.createElement('script') и затем добавленные на страницу, также не блокируют это событие.

Внешние таблицы стилей не затрагивают DOM, поэтому DOMContentLoaded их не ждёт. Но если после стилей есть скрипт, то этот скрипт должен дождаться, пока загрузятся стили:

<link type="text/css" rel="stylesheet" href="style.css">

<script>

// скрипт не выполняется, пока не загрузятся стили

alert(getComputedStyle(document.body).marginTop);

</script>

Причина в том, что скрипту может понадобиться получить координаты или другие свойства элементов, зависящих от стилей, как в примере выше. Естественно, он должен дождаться, пока стили загрузятся. Так как DOMContentLoaded дожидается скриптов, то теперь он так же дожидается и стилей перед ними.

Firefox, Chrome и Opera автоматически заполняют поля при наступлении DOMContentLoaded. Например, если на странице есть форма логина и пароля и браузер запомнил значения, то при наступлении DOMContentLoaded он попытается заполнить их (если получил разрешение от пользователя). Так что, если DOMContentLoaded откладывается из-за долгой загрузки скриптов, в свою очередь – откладывается автозаполнение. Вы наверняка замечали, что на некоторых сайтах (если вы используете автозаполнение в браузере) поля логина и пароля не заполняются мгновенно, есть некоторая задержка до полной загрузки страницы. Это и есть ожидание события DOMContentLoaded.

[**window.onload**](https://learn.javascript.ru/onload-ondomcontentloaded#window-onload)

Событие load на объекте window наступает, когда загрузилась вся страница, включая стили, картинки и другие ресурсы. В примере ниже правильно показаны размеры картинки, потому что window.onload дожидается всех изображений:

<script>

window.onload = function() {

alert('Страница загружена');

// к этому моменту страница загружена

alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);

};

</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">

[**window.onunload**](https://learn.javascript.ru/onload-ondomcontentloaded#window-onunload)

Когда посетитель покидает страницу, на объекте window генерируется событие unload. В этот момент стоит совершать простые действия, не требующие много времени, вроде закрытия связанных всплывающих окон.

Обычно здесь отсылают статистику. Предположим, собраны данные о том, как используется страница: клики, прокрутка, просмотры областей страницы и так далее. Естественно, событие unload – это тот момент, когда пользователь покидает страницу и надо сохранить эти данные. Для этого существует специальный метод navigator.sendBeacon(url, data), описанный в спецификации <https://w3c.github.io/beacon/>. Он посылает данные в фоне. Переход к другой странице не задерживается: браузер покидает страницу, но всё равно выполняет sendBeacon. Его можно использовать вот так:

let analyticsData = { /\* объект с собранными данными \*/ };

window.addEventListener("unload", function() {

navigator.sendBeacon("/analytics", JSON.stringify(analyticsData));

};

* Отсылается POST-запрос.
* Можно послать не только строку, но так же формы и другие форматы, но обычно это строковый объект.
* Размер данных ограничен 64 Кб.

К тому моменту, как sendBeacon завершится, браузер наверняка уже покинет страницу, так что возможности обработать ответ сервера не будет (для статистики он обычно пустой). Для таких запросов с закрывающейся страницей есть специальный флаг keepalive.  Если надо отменить переход на другую страницу, то здесь этого сделать нельзя. Но сможем в другом месте – в событии onbeforeunload.

[**window.onbeforeunload**](https://learn.javascript.ru/onload-ondomcontentloaded#window.onbeforeunload)

Если посетитель собирается уйти со страницы или закрыть окно, обработчик beforeunload попросит дополнительное подтверждение. Если отменить это событие, то браузер спросит посетителя, уверен ли он. Запустите следующий код и затем перезагрузите страницу:

window.onbeforeunload = function() {

return false;

};

**[readyState](https://learn.javascript.ru/onload-ondomcontentloaded" \l "readystate)**

Если установить обработчик DOMContentLoaded после того, как документ загрузился, то он никогда не запустится.

Бывают ситуации, когда неизвестно, готов документ или нет. Но надо, чтобы функция исполнилась, когда DOM загрузился, неважно сейчас или позже. Свойство document.readyState показывает текущее состояние загрузки. Есть три возможных значения:

* "loading" – документ загружается;
* "interactive" – документ был полностью прочитан;
* "complete" – документ был полностью прочитан и все ресурсы (такие как изображения) были тоже загружены.

Так что можно проверить document.readyState и, либо установить обработчик, либо, если документ готов, выполнить код сразу же. Например, вот так:

function work() { /\*...\*/ }

if (document.readyState == 'loading') {

// ещё загружается, ждём события

document.addEventListener('DOMContentLoaded', work);

} else {

// DOM готов

work();

}

Также есть событие readystatechange, которое генерируется при изменении состояния, так что можно вывести все эти состояния таким образом:

// текущее состояние

console.log(document.readyState);

// вывести изменения состояния

document.addEventListener('readystatechange', () => console.log(document.readyState));

Событие readystatechange – альтернативный вариант отслеживания состояния загрузки документа, который появился очень давно. На сегодняшний день он используется редко.

Рассмотрим весь поток событий. Здесь документ с <iframe>, <img> и обработчиками, которые логируют события:

<script>

log('начальный readyState:' + document.readyState);

document.addEventListener('readystatechange', () => log('readyState:' + document.readyState));

document.addEventListener('DOMContentLoaded', () => log('DOMContentLoaded'));

window.onload = () => log('window onload');

</script>

<iframe src="iframe.html" onload="log('iframe onload')"></iframe>

<img src="http://en.js.cx/clipart/train.gif" id="img">

<script>

img.onload = () => log('img onload');

</script>

Типичный вывод:

1. [1] начальный readyState:loading
2. [2] readyState:interactive
3. [2] DOMContentLoaded
4. [3] iframe onload
5. [4] img onload
6. [4] readyState:complete
7. [4] window onload

Цифры в квадратных скобках обозначают примерное время события. События, отмеченные одинаковой цифрой, произойдут примерно в одно и то же время (± несколько миллисекунд).

* document.readyState станет interactive прямо перед DOMContentLoaded. Эти две вещи, на самом деле, обозначают одно и то же.
* document.readyState станет complete, когда все ресурсы (iframe и img) загрузятся. Это произойдёт примерно в одно время с img.onload (img последний ресурс) и window.onload. Переключение на состояние complete означает то же самое, что и window.onload. Разница заключается в том, что window.onload всегда срабатывает после всех load других обработчиков.

1. **Загрузка скриптов, ресурсов.**

**Загрузка скриптов**

В современных сайтах скрипты обычно «тяжелее», чем HTML: они весят больше, дольше обрабатываются. Когда браузер загружает HTML и доходит до тега <script>...</script>, он не может продолжать строить DOM. Он должен сначала выполнить скрипт. То же самое происходит и с внешними скриптами <script src="..."></script>: браузер должен подождать, пока загрузится скрипт, выполнить его, и только затем обработать остальную страницу. Это ведёт к двум важным проблемам:

1. Скрипты не видят DOM-элементы ниже себя, поэтому к ним нельзя добавить обработчики и т.д.
2. Если вверху страницы объёмный скрипт, он «блокирует» страницу. Пользователи не видят содержимое страницы, пока он не загрузится и не запустится:

<p>...содержимое перед скриптом...</p>

<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<!-- Это не отобразится, пока скрипт не загрузится -->

<p>...содержимое после скрипта...</p>

Конечно, есть пути, как это обойти. Например, можно поместить скрипт внизу страницы. Тогда он сможет видеть элементы над ним и не будет препятствовать отображению содержимого страницы:

<body>

...всё содержимое над скриптом...

<script src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

</body>

Но у этого решения есть недостатки. Например, браузер замечает скрипт (и может начать загружать его) только после того, как он полностью загрузил HTML-документ. В случае с длинными HTML-страницами это может создать заметную задержку. Эту проблему можно решить с помощью атрибутов тега <script>: defer и async.

[**defer**](https://learn.javascript.ru/script-async-defer#defer)

Атрибут defer сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме, а затем запустить этот скрипт, когда он загрузится. Вот тот же пример, что и выше, но с defer:

<p>...содержимое перед скриптом...</p>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<!-- отображается сразу же -->

<p>...содержимое после скрипта...</p>

* Скрипты с defer никогда не блокируют страницу.
* Скрипты с defer всегда выполняются, когда дерево DOM готово, но до события DOMContentLoaded.

Следующий пример это показывает:

<p>...содержимое до скрипта...</p>

<script>

document.addEventListener('DOMContentLoaded', () => alert("Дерево DOM готово после скрипта с 'defer'!")); // (2)

</script>

<script defer src="https://javascript.info/article/script-async-defer/long.js?speed=1"></script>

<p>...содержимое после скрипта...</p>

Содержимое страницы отобразится мгновенно. Событие DOMContentLoaded подождёт отложенный скрипт. Оно будет сгенерировано, только когда скрипт (2) будет загружен и выполнен.

Отложенные с помощью defer скрипты сохраняют порядок относительно друг друга, как и обычные скрипты. Поэтому, если сначала загружается большой скрипт, а затем меньшего размера, то последний будет ждать.

<script defer src="https://javascript.info/article/script-async-defer/long.js"></script>

<script defer src="https://javascript.info/article/script-async-defer/small.js"></script>

Браузеры сканируют страницу на предмет скриптов и загружают их параллельно в целях увеличения производительности. Поэтому и в примере выше оба скрипта скачиваются параллельно. small.js загрузится первым. Но спецификация требует последовательного выполнения скриптов согласно порядку в документе, поэтому он подождёт выполнения long.js.

Атрибут defer предназначен только для внешних скриптов. Поэтому он будет проигнорирован, если в теге <script> нет src.

[**async**](https://learn.javascript.ru/script-async-defer#async)

Атрибут async означает, что скрипт абсолютно независим:

* Страница не ждёт асинхронных скриптов, содержимое обрабатывается и отображается.
* Событие DOMContentLoaded и асинхронные скрипты не ждут друг друга:

DOMContentLoaded может произойти как до асинхронного скрипта (если асинхронный скрипт завершит загрузку после того, как страница будет готова), так и после асинхронного скрипта (если он короткий или уже содержится в HTTP-кеше)

* Остальные скрипты не ждут async, и скрипты casync не ждут другие скрипты.

Так что, если есть несколько скриптов с async, они могут выполняться в любом порядке. То, что первое загрузится – запустится в первую очередь:

<p>...содержимое перед скриптами...</p>

<script>

document.addEventListener('DOMContentLoaded', () => alert("DOM готов!"));

</script>

<script async src="https://javascript.info/article/script-async-defer/long.js"></script>

<script async src="https://javascript.info/article/script-async-defer/small.js"></script>

<p>...содержимое после скриптов...</p>

1. Содержимое страницы отображается сразу же: async его не блокирует.
2. DOMContentLoaded может произойти как до, так и после async, никаких гарантий нет.
3. Асинхронные скрипты не ждут друг друга. Меньший скрипт small.js идёт вторым, но скорее всего загрузится раньше long.js, поэтому и запустится первым. То есть, скрипты выполняются в порядке загрузки.

Можно также добавить скрипт и динамически, с помощью JavaScript:

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

script.src = "/article/script-async-defer/long.js";

document.body.append(script); // (\*)

Скрипт начнёт загружаться, как только он будет добавлен в документ (\*).

Динамически загружаемые скрипты по умолчанию ведут себя как «async». То есть:

* Они никого не ждут, и их никто не ждёт.
* Скрипт, который загружается первым – запускается первым (в порядке загрузки).

Можно изменить относительный порядок скриптов с «первый загрузился – первый выполнился» на порядок, в котором они идут в документе (как в обычных скриптах) с помощью явной установки свойства async в false:

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

script.src = "/article/script-async-defer/long.js";

script.async = false;

document.body.append(script);

Например, в примере ниже добавляются два скрипта. Без script.async=false они запускались бы в порядке загрузки (small.js скорее всего запустился бы раньше). Но с этим флагом порядок будет как в документе:

function loadScript(src) {

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

script.src = src;

script.async = false;

document.body.append(script);

}

loadScript("/article/script-async-defer/long.js");

loadScript("/article/script-async-defer/small.js");

**Загрузка ресурсов.**

Браузер позволяет отслеживать загрузку сторонних ресурсов: скриптов, ифреймов, изображений и др. Для этого существуют два события:

* load – успешная загрузка,
* error – во время загрузки произошла ошибка.

[**Загрузка скриптов**](https://learn.javascript.ru/onload-onerror#zagruzka-skriptov)

Допустим, нужно загрузить сторонний скрипт и вызвать функцию, которая объявлена в этом скрипте. Можно загрузить этот скрипт динамически:

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

script.src = "my.js";

document.head.append(script);

Но чтобы вызвать функцию, которая объявлена внутри того скрипта, нужно подождать, пока скрипт загрузится, и только потом можно её вызвать.

Для наших собственных скриптов можно использовать [JavaScript-модули](https://learn.javascript.ru/modules), но они не слишком широко распространены в сторонних библиотеках.

[**script.onload**](https://learn.javascript.ru/onload-onerror#script-onload)

Событие load срабатывает после того, как скрипт был загружен и выполнен. Например:

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

// можно загрузить любой скрипт с любого домена

script.src = "https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"

document.head.append(script);

script.onload = function() {

// в скрипте создается вспомогательная функция с именем "\_"

alert(\_); // функция доступна

};

Таким образом, в обработчике onload можно использовать переменные, вызывать функции и т.д., которые предоставляет сторонний скрипт.

[**script.onerror**](https://learn.javascript.ru/onload-onerror#script-onerror)

Ошибки, которые возникают во время загрузки скрипта, могут быть отслежены с помощью события error. Например, запросим скрипт, которого не существует:

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

script.src = "https://example.com/404.js"; // такого файла не существует

document.head.append(script);

script.onerror = function() {

alert("Error loading " + this.src); // Ошибка загрузки https://example.com/404.js

};

Обратите внимание, что нельзя получить описание HTTP-ошибки. Неизвестно, была ли это ошибка 404 или 500, или какая-то другая. Известно только, что во время загрузки произошла ошибка.

Обработчики onload/onerror отслеживают только сам процесс загрузки. Ошибки обработки и выполнения загруженного скрипта ими не отслеживаются. Чтобы «поймать» ошибки в скрипте, нужно воспользоваться глобальным обработчиком window.onerror.

События load и error также срабатывают и для других ресурсов, у которых есть внешний src. Например:

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

img.src = "https://js.cx/clipart/train.gif"; // (\*)

img.onload = function() {

alert(`Изображение загружено, размеры ${img.width}x${img.height}`);

};

img.onerror = function() {

alert("Ошибка во время загрузки изображения");

};

Однако есть некоторые особенности:

* Большинство ресурсов начинают загружаться после их добавления в документ. За исключением тега <img>. Изображения начинают загружаться, когда получают src (\*).
* Для <iframe> событие load срабатывает по окончании загрузки как в случае успеха, так и в случае ошибки.

Есть правило: скрипты с одного сайта не могут получить доступ к содержимому другого сайта. Например, скрипт с https://facebook.com не может прочитать почту пользователя на <https://gmail.com>. Или, если быть более точным, один источник (домен/порт/протокол) не может получить доступ к содержимому c другого источника. Даже поддомен или просто другой порт будут считаться разными источниками, не имеющими доступа друг к другу. Это правило также касается ресурсов с других доменов. Если используется скрипт с другого домена, и в нем имеется ошибка, то нельзя узнать детали этой ошибки.

Для примера рассматрим мини-скрипт error.js, который состоит из одного-единственного вызова функции, которой не существует:

// error.js

noSuchFunction();

Теперь загрузим этот скрипт с того же сайта, на котором он располагается:

<script>

window.onerror = function(message, url, line, col, errorObj) {

alert(`${message}\n${url}, ${line}:${col}`);

};

</script>

<script src="/article/onload-onerror/crossorigin/error.js"></script>

Видим отчёт об ошибке:

Uncaught ReferenceError: noSuchFunction is not defined

https://javascript.info/article/onload-onerror/crossorigin/error.js, 1:1

Теперь загрузим этот же скрипт с другого домена:

<script>

window.onerror = function(message, url, line, col, errorObj) {

alert(`${message}\n${url}, ${line}:${col}`);

};

</script>

<script src="https://cors.javascript.info/article/onload-onerror/crossorigin/error.js"></script>

Отчёт отличается:

Script error.

, 0:0

Детали отчёта могут варьироваться в зависимости от браузера, но основная идея остаётся неизменной: любая информация о внутреннем устройстве скрипта, включая стек ошибки, спрятана. Именно потому, что скрипт загружен с другого домена.

Детали ошибки могут быть полезны. Существует много сервисов, которые обрабатывают глобальные ошибки при помощи window.onerror, сохраняют отчёт о них и предоставляют доступ к этому отчёту для анализа. Это здорово, потому что можно увидеть реальные ошибки, которые случились у наших пользователей. Но если скрипт – с другого домена, то информации об ошибках в нём почти нет.

Чтобы разрешить кросс-доменный доступ, нужно установить тегу <script> атрибут crossorigin, и, кроме того, удалённый сервер должен указать специальные заголовки.

Существует три уровня кросс-доменного доступа:

1. Атрибут crossorigin отсутствует – доступ запрещён.
2. crossorigin="anonymous" – доступ разрешён, если сервер отвечает с заголовком Access-Control-Allow-Origin со значениями \* или наш домен. Браузер не отправляет авторизационную информацию и куки на удалённый сервер.
3. crossorigin="use-credentials" – доступ разрешён, если сервер отвечает с заголовками Access-Control-Allow-Origin со значением наш домен и Access-Control-Allow-Credentials: true. Браузер отправляет авторизационную информацию и куки на удалённый сервер.

В рассматриваемом примере атрибут crossorigin отсутствовал. Поэтому кросс-доменный доступ был запрещён. Добавим его. Можно выбрать "anonymous" (куки не отправляются, требуется один серверный заголовок) или "use-credentials" (куки отправляются, требуются два серверных заголовка) в качестве значения атрибута.

<script>

window.onerror = function(message, url, line, col, errorObj) {

alert(`${message}\n${url}, ${line}:${col}`);

};

</script>

<script crossorigin="anonymous" src="https://cors.javascript.info/article/onload-onerror/crossorigin/error.js"></script>

Теперь при условии, что сервер предоставил заголовок Access-Control-Allow-Origin, получен полный отчёт по ошибкам.

1. **Свойства и методы формы.**

Формы и элементы управления, такие как <input>, имеют множество специальных свойств и событий.

Формы в документе входят в специальную коллекцию document.forms. Это – так называемая «именованная» коллекция: можно использовать для получения формы как её имя, так и порядковый номер в документе.

document.forms.my - форма с именем "my" (name="my")

document.forms[0] - первая форма в документе

Когда форма уже получена, любой элемент доступен в именованной коллекции form.elements. Например:

<form name="my">

<input name="one" value="1">

<input name="two" value="2">

</form>

<script>

// получаем форму

let form = document.forms.my; // <form name="my"> element

// получаем элемент

let elem = form.elements.one; // <input name="one"> element

alert(elem.value); // 1

</script>

Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями radio. В этом случае form.elements[name] является коллекцией, например:

<form>

<input type="radio" name="age" value="10">

<input type="radio" name="age" value="20">

</form>

<script>

let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0].value); // 10, the first input value

</script>

Эти навигационные свойства не зависят от структуры тегов внутри формы. Все элементы, как бы глубоко они ни находились в форме, доступны в коллекции form.elements.

Форма может содержать один или несколько элементов <fieldset> внутри себя. Они также поддерживают свойство elements. Например:

<body>

<form id="form">

<fieldset name="userFields">

<legend>info</legend>

<input name="login" type="text">

</fieldset>

</form>

<script>

alert(form.elements.login); // <input name="login">

let fieldset = form.elements.userFields;

alert(fieldset); // HTMLFieldSetElement

// можно получить информацию как из формы, так и из fieldset

alert(fieldset.elements.login == form.elements.login); // true

</script>

</body>

Есть более короткая запись: можно получить доступ к элементу через form[index/name]. Вместо form.elements.login можно написать form.login. Это также работает, но есть небольшая проблема: если надо получить элемент, а затем менять его свойство name, то он всё ещё будет доступен под старым именем (также, как и под новым). В этом легче разобраться на примере:

<form id="form">

<input name="login">

</form>

<script>

alert(form.elements.login == form.login); // true, ведь это одинаковые <input>

form.login.name = "username"; // изменяем свойство name у элемента input

// form.elements обновили свои имена:

alert(form.elements.login); // undefined

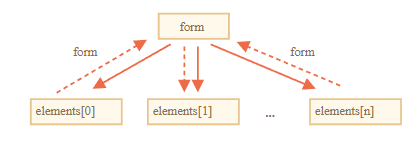
alert(form.elements.username); // input

// теперь для прямого доступа можно использовать оба имени: новое и старое

alert(form.username == form.login); // true

</script>

Для любого элемента форма доступна через element.form. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму. Вот иллюстрация:



Пример:

<form id="form">

<input type="text" name="login">

</form>

<script>

// form -> element

let login = form.login;

// element -> form

alert(login.form); // HTMLFormElement

</script>

Рассмотрим элементы управления, используемые в формах, обращая внимание на их особенности.

[**input и textarea**](https://learn.javascript.ru/form-elements#input-i-textarea)

К их значению можно получить доступ через свойство input.value (строка) или input.checked (булево значение) для чекбоксов. Вот так:

input.value = "Новое значение";

textarea.value = "Новый текст";

input.checked = true; // для чекбоксов и переключателей

Обратите внимание: хоть <textarea>...</textarea> и хранит значение как вложенный HTML, не следует использовать textarea.innerHTML. Там хранится только тот HTML, который был изначально на странице, а не текущее значение.

[**select и option**](https://learn.javascript.ru/form-elements#select-i-option)

Элемент <select> имеет 3 важных свойства:

1. select.options – коллекция из элементов <option>,
2. select.value – значение выбранного в данный момент <option>,
3. select.selectedIndex – номер выбранного <option>.

Имеется три способа задать значение для <select>:

1. Найти необходимый <option> и установить в option.selected значение true.
2. Установить в select.value значение нужного нам <option>.
3. Установить в select.selectedIndex номер <option>.

Первый способ наиболее понятный, но (2) и (3) являются более удобными при работе. Вот эти способы на примере:

<select id="select">

<option value="apple">Яблоко</option>

<option value="pear">Груша</option>

<option value="banana">Банан</option>

</select>

<script>

// все три строки делают одно и то же

select.options[2].selected = true;

select.selectedIndex = 2;

select.value = 'banana';

</script>

В отличие от большинства других элементов управления, <select multiple> позволяет выбрать несколько вариантов. В этом случае необходимо пройтись по select.options, чтобы получить все выбранные значения. Например так:

<select id="select" multiple>

<option value="blues" selected>Блюз</option>

<option value="rock" selected>Рок</option>

<option value="classic">Классика</option>

</select>

<script>

// получаем все выбранные значения из списка множественного выбора

let selected = Array.from(select.options)

.filter(option => option.selected)

.map(option => option.value);

alert(selected); // Блюз,Рок

</script>

Полное описание элемента <select> доступно в спецификации <https://html.spec.whatwg.org/multipage/forms.html#the-select-element>.

Элемент <option> редко используется сам по себе. В описании [элемента option](https://html.spec.whatwg.org/multipage/forms.html#the-option-element) есть короткий синтаксис для создания элемента:

option = new Option(text, value, defaultSelected, selected);

Параметры:

* text – текст внутри,
* value – значение,
* defaultSelected – если true, то ставится HTML-атрибут selected,
* selected – если true, то элемент <option> будет выбранным.

Пример:

let option = new Option("Текст", "value");

// создаст <option value="value">Текст</option>

Тот же элемент, но выбранный:

let option = new Option("Текст", "value", true, true);

Элементы <option> имеют дополнительные свойства:

* selected – выбрана ли опция,
* index – номер опции среди других в списке <select>,
* text – содержимое опции (то, что видит посетитель).

1. **Фокусировка элементов формы.**

Элемент получает фокус, когда пользователь кликает по нему или использует клавишу Tab. Также существует HTML-атрибут autofocus, который устанавливает фокус на элемент, когда страница загружается. Есть и другие способы получения фокуса.

Фокусировка обычно означает: «приготовься к вводу данных на этом элементе», это хороший момент, чтобы инициализовать или загрузить что-нибудь. Момент потери фокуса («blur») может быть важнее. Это момент, когда пользователь кликает куда-то ещё или нажимает Tab, чтобы переключиться на следующее поле формы. Есть другие причины потери фокуса.

Потеря фокуса обычно означает «данные введены», и можно выполнить проверку введенных данных или даже отправить эти данные на сервер и так далее. В работе с событиями фокусировки есть важные особенности.

[**События focus/blur**](https://learn.javascript.ru/focus-blur#sobytiya-focus-blur)

Событие focus вызывается в момент фокусировки, а blur – когда элемент теряет фокус. Используем их для валидации (проверки) введённых данных. В примере ниже:

* Обработчик blur проверяет, введён ли email, и если нет – показывает ошибку.
* Обработчик focus скрывает это сообщение об ошибке (в момент потери фокуса проверка повторится):

<style>

.invalid { border-color: red; }

#error { color: red }

</style>

Ваш email: <input type="email" id="input">

<div id="error"></div>

<script>

input.onblur = function() {

if (!input.value.includes('@')) { // не email

input.classList.add('invalid');

error.innerHTML = 'Пожалуйста, введите правильный email.'

}

};

input.onfocus = function() {

if (this.classList.contains('invalid')) {

// удаляем индикатор ошибки, т.к. пользователь хочет ввести данные заново

this.classList.remove('invalid');

error.innerHTML = "";

}

};

</script>





Современный HTML позволяет делать валидацию с помощью атрибутов required, pattern и т.д. Иногда – этого достаточно. JavaScript можно использовать, когда требуется больше гибкости. А также можно отправлять изменённое значение на сервер, если оно правильное.

[**Методы focus/blur**](https://learn.javascript.ru/focus-blur#metody-focus-blur)

Методы elem.focus() и elem.blur() устанавливают/снимают фокус. Например, запретим посетителю переключаться с поля ввода, если введённое значение не прошло валидацию:

<style>

.error {

background: red;

}

</style>

Ваш email: <input type="email" id="input">

<input type="text" style="width:280px" placeholder="введите неверный email и кликните сюда">

<script>

input.onblur = function() {

if (!this.value.includes('@')) { // не email

// показать ошибку

this.classList.add("error");

// вернуть фокус обратно

input.focus();

} else {

this.classList.remove("error");

}

};

</script>





Если что-нибудь ввести и нажать Tab или кликнуть в другое место, тогда onblur вернёт фокус обратно.

Нельзя отменить потерю фокуса, вызвав event.preventDefault() в обработчике onblur потому, что onblur срабатывает после потери фокуса элементом.

Потеря фокуса может произойти по множеству причин. Одна из них – когда посетитель кликает куда-то ещё. Но и JavaScript может быть причиной, например:

* alert переводит фокус на себя – элемент теряет фокус (событие blur), а когда alert закрывается – элемент получает фокус обратно (событие focus);
* если элемент удалить из DOM, фокус также будет потерян. Если элемент добавить обратно, то фокус не вернётся.

Из-за этих особенностей обработчики focus/blur могут сработать тогда, когда это не требуется.

Многие элементы по умолчанию не поддерживают фокусировку. Какие именно – зависит от браузера, но одно всегда верно: поддержка focus/blur гарантирована для элементов, с которыми посетитель может взаимодействовать: <button>, <input>, <select>, <a> и т.д. С другой стороны, элементы форматирования <div>, <span>, <table> – по умолчанию не могут получить фокус. Метод elem.focus() не работает для них, и события focus/blur никогда не срабатывают.

Это можно изменить HTML-атрибутом tabindex. Цель этого атрибута – указать порядковый номер элемента, когда клавиша Tab используется для переключения между элементами. То есть: если имеется два элемента, первый имеет tabindex="1", а второй tabindex="2", то находясь в первом элементе и нажав Tab – фокус переместится на второй.

Есть два специальных значения:

* tabindex="0" делает элемент последним,
* tabindex="-1" значит, что Tab игнорирует этот элемент.

Любой элемент поддерживает фокусировку, если имеет tabindex. Например, список ниже. Кликните первый пункт в списке и нажмите Tab:

Кликните первый пункт в списке и нажмите Tab. Продолжайте следить за порядком. Обратите внимание, что много последовательных нажатий Tab могут вывести фокус из iframe с примером.

<ul>

<li tabindex="1">Один</li>

<li tabindex="0">Ноль</li>

<li tabindex="2">Два</li>

<li tabindex="-1">Минус один</li>

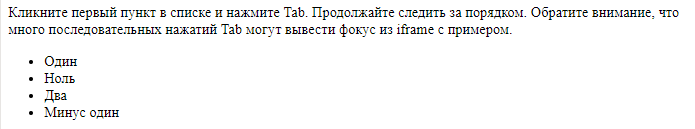
</ul>

<style>

li { cursor: pointer; }

:focus { outline: 1px dashed green; }

</style>



Порядок такой: 1 - 2 - 0 (ноль всегда последний). Обычно <li> не поддерживает фокусировку, но tabindex полностью включает её, а также события и стилизацию псевдоклассом :focus.

Можно добавить tabindex из JavaScript, используя свойство elem.tabIndex. Это даст тот же эффект.

[**События focusin/focusout**](https://learn.javascript.ru/focus-blur#sobytiya-focusin-focusout)

События focus и blur не всплывают. Например, нельзя использовать onfocus на <form>, чтобы подсветить её:

<!-- добавить класс при фокусировке на форме -->

<form onfocus="this.className='focused'">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

<style> .focused { outline: 1px solid red; } </style>





Пример выше не работает, потому что когда пользователь перемещает фокус на <input>, событие focus срабатывает только на этом элементе. Это событие не всплывает. Следовательно, form.onfocus никогда не срабатывает.

У этой проблемы два решения. Первое: забавная особенность – focus/blur не всплывают, но передаются вниз на фазе перехвата. Это сработает:

<form id="form">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

<style> .focused { outline: 1px solid red; } </style>

<script>

// установить обработчик на фазе перехвата (последний аргумент true)

form.addEventListener("focus", () => form.classList.add('focused'), true);

form.addEventListener("blur", () => form.classList.remove('focused'), true);

</script>





Второе решение: события focusin и focusout – такие же, как и focus/blur, но они всплывают. Заметьте, что эти события должны использоваться с elem.addEventListener, но не с on<event>. Второй рабочий вариант:

<form id="form">

<input type="text" name="name" value="Имя">

<input type="text" name="surname" value="Фамилия">

</form>

<style> .focused { outline: 1px solid red; } </style>

<script>

form.addEventListener("focusin", () => form.classList.add('focused'));

form.addEventListener("focusout", () => form.classList.remove('focused'));

</script>





1. **Изменение значений элемента формы. Формы: отправка, событие и метод submit.**

**Изменение значений элемента.**

Рассмотрим различные события, сопутствующие обновлению данных.

[**Событие: change**](https://learn.javascript.ru/events-change-input#sobytie-change)

Событие change срабатывает по окончании изменения элемента. Для текстовых <input> это означает, что событие происходит при потере фокуса. Пока пользователь печатает в текстовом поле в примере ниже, событие не происходит. Но когда перемещает фокус в другое место, например, нажимая на кнопку, то произойдёт событие change:

<input type="text" onchange="alert(this.value)">

<input type="button" value="Button">





Для других элементов: select, input type=checkbox/radio событие запускается сразу после изменения значения:

<select onchange="alert(this.value)">

<option value="">Выберите что-нибудь</option>

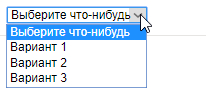
<option value="1">Вариант 1</option>

<option value="2">Вариант 2</option>

<option value="3">Вариант 3</option>

</select>





[**Событие: input**](https://learn.javascript.ru/events-change-input#sobytie-input)

Событие input срабатывает каждый раз при изменении значения. В отличие от событий клавиатуры, оно работает при любых изменениях значений, даже если они не связаны с клавиатурными действиями: вставка с помощью мыши или распознавание речи при диктовке текста. Например:

<input type="text" id="input"> oninput: <span id="result"></span>

<script>

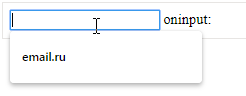
input.oninput = function() {

result.innerHTML = input.value;

};

</script>





Если надо обрабатывать каждое изменение в <input>, то это событие является лучшим выбором. С другой стороны, событие input не происходит при вводе с клавиатуры или иных действиях, если при этом не меняется значение в текстовом поле, т.е. нажатия клавиш ⇦, ⇨ и подобных при фокусе на текстовом поле не вызовут это событие.

Событие input происходит после изменения значения. Поэтому нельзя использовать event.preventDefault() там – будет уже слишком поздно, никакого эффекта не будет.

[**События: cut, copy, paste**](https://learn.javascript.ru/events-change-input#sobytiya-cut-copy-paste)

Эти события происходят при вырезании/копировании/вставке данных. Они относятся к классу [ClipboardEvent](https://www.w3.org/TR/clipboard-apis/" \l "clipboard-event-interfaces) и обеспечивают доступ к копируемым/вставляемым данным. Также можно использовать event.preventDefault() для предотвращения действия по умолчанию, и в итоге ничего не скопируется/не вставится. Например, код, приведённый ниже, предотвращает все подобные события и показывает, что надо вырезать/копировать/вставить:

<input type="text" id="input">

<script>

input.oncut = input.oncopy = input.onpaste = function(event) {

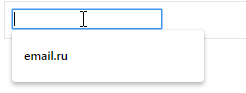
alert(event.type + ' - ' + event.clipboardData.getData('text/plain'));

return false;

};

</script>





Технически, можно скопировать/вставить всё. Например, можно скопировать файл из файловой системы и вставить его. Существует список методов [в спецификации](https://www.w3.org/TR/clipboard-apis/#dfn-datatransfer) для работы с различными типами данных, чтения/записи в буфер обмена. Но обратите внимание, что буфер обмена работает глобально, на уровне ОС. Большинство браузеров в целях безопасности разрешают доступ на чтение/запись в буфер обмена только в рамках определённых действий пользователя, к примеру, в обработчиках событий onclick.

Также запрещается генерировать «пользовательские» события буфера обмена при помощи dispatchEvent во всех браузерах, кроме Firefox.

**Формы: отправка, событие и метод submit.**

При отправке формы срабатывает событие submit, оно обычно используется для проверки (валидации) формы перед её отправкой на сервер или для предотвращения отправки и обработки её с помощью JavaScript.

Метод form.submit() позволяет инициировать отправку формы из JavaScript. Можно использовать его для динамического создания и отправки наших собственных форм на сервер. Рассмотрим их подробнее.

[**Событие: submit**](https://learn.javascript.ru/forms-submit#sobytie-submit)

Есть два основных способа отправить форму:

1. Первый – нажать кнопку <input type="submit"> или <input type="image">.
2. Второй – нажать Enter, находясь на каком-нибудь поле.

Оба действия сгенерируют событие submit на форме. Обработчик может проверить данные, и если есть ошибки, показать их и вызвать event.preventDefault(), тогда форма не будет отправлена на сервер.

В примере ниже оба действия показывают alert и форма не отправится благодаря return false:

<form onsubmit="alert('submit!');return false">

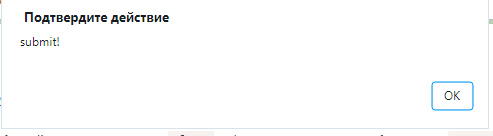
Первый пример: нажмите Enter: <input type="text" value="Текст"><br>

Второй пример: нажмите на кнопку "Отправить": <input type="submit" value="Отправить">

</form>



Первый и второй примеры, результат одинаковый:



При отправке формы по нажатию Enter в текстовом поле, генерируется событие click на кнопке <input type="submit">. При этом никакого клика на самом деле не было. Пример:

<form onsubmit="alert('submit!');return false">

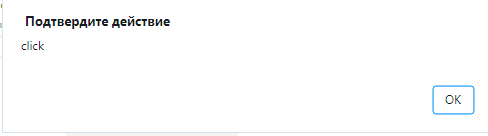
<input type="text" size="30" value="Установите фокус здесь и нажмите Enter">

<input type="submit" value="Отправить" onclick="alert('click')">

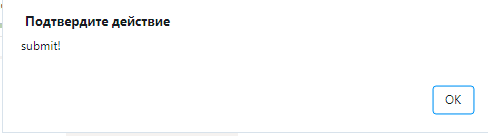
</form>



Появилось окно:



После клика на ок:



[**Метод: submit**](https://learn.javascript.ru/forms-submit#metod-submit)

Чтобы отправить форму на сервер вручную, можно вызвать метод form.submit(). При этом событие submit не генерируется. Предполагается, что если программист вызывает метод form.submit(), то он уже выполнил всю соответствующую обработку. Иногда это используют для генерации формы и отправки её вручную, например так:

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

form.action = 'https://google.com/search';

form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// перед отправкой формы, её нужно вставить в документ

document.body.append(form);

form.submit();

**57. JS-библиотека React. Понятие иммутабельности и согласования**

**React** — JavaScript-библиотека для создания пользовательских интерфейсов. Она строится вокруг компонентов и позволяет эффективно обновлять DOM с помощью виртуального DOM.

* **Иммутабельность (immutability)** — данные не изменяются напрямую, а создаются их новые копии при изменении. Это позволяет React эффективно определять, что именно изменилось, и минимизировать обновления DOM.
* **Согласование (reconciliation)** — процесс, в ходе которого React сравнивает старое дерево виртуального DOM с новым и находит минимальное число изменений, необходимых для обновления реального DOM.

**58. Расширение языка JavaScript – JSX. Рендеринг элементов**

* **JSX (JavaScript XML)** — синтаксическое расширение JavaScript, которое позволяет писать HTML-подобный код в JS-функциях. Пример:

jsx

КопироватьРедактировать

const element = <h1>Hello, world!</h1>;

* **Рендеринг элементов** — это процесс отображения JSX-элементов в DOM с помощью функций ReactDOM.render() (в старых версиях) или createRoot().render() (в новых).

jsx

КопироватьРедактировать

import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));

root.render(<App />);

**59. React: компоненты и пропсы**

* **Компоненты** — переиспользуемые части интерфейса. Бывают:
  + Функциональные:

jsx

КопироватьРедактировать

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

* + Классовые (устаревшие):

jsx

КопироватьРедактировать

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

* **Пропсы (props)** — свойства, передаваемые в компонент извне, только для чтения. Они позволяют параметризовать компоненты.

**60. React: состояние и жизненный цикл: монтирование, обновление, размонтирование**

* **Состояние (state)** — внутренние данные компонента, которые могут изменяться.
* **Жизненный цикл компонента**:
  + **Монтирование (mounting)**: вставка компонента в DOM (componentDidMount / useEffect(() => {}, [])).
  + **Обновление (updating)**: изменение props или state, вызывающее повторный рендер (componentDidUpdate).
  + **Размонтирование (unmounting)**: удаление компонента из DOM (componentWillUnmount / useEffect(() => { return () => {...} }, [])).

**61. React: обработка событий**

* React использует **camelCase** для названий событий и передаёт функции вместо строк:

jsx

КопироватьРедактировать

<button onClick={handleClick}>Click me</button>

* События синтетические (SyntheticEvent), кросс-браузерные и оборачивают нативные события.

**62. React: условный рендеринг, рендеринг списка, ключи**

* **Условный рендеринг**:

jsx

КопироватьРедактировать

{isLoggedIn ? <Logout /> : <Login />}

* **Рендеринг списков**:

jsx

КопироватьРедактировать

const items = list.map(item => <li key={item.id}>{item.name}</li>);

* **Ключи (keys)** — уникальные идентификаторы для отслеживания элементов списка при обновлении, особенно при удалении/добавлении.

**63. React: формы**

* **Управляемые компоненты** — значение элементов форм хранится в state:

jsx

КопироватьРедактировать

const [value, setValue] = useState('');

<input value={value} onChange={e => setValue(e.target.value)} />

* Обработка submit через onSubmit, предотвращая стандартное поведение (event.preventDefault()).

**64. React: хуки, правила хуков**

* **Хуки (hooks)** — функции, добавляющие функциональность классов в функциональные компоненты.
  + Примеры: useState, useEffect, useContext, useRef, useReducer.
* **Правила хуков**:
  + Вызывать только **внутри** функциональных компонентов или пользовательских хуков.
  + Вызывать **всегда в одном порядке**, не в условиях или циклах.

**65. React: использование хука состояния**

jsx

КопироватьРедактировать

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

return (

<button onClick={() => setCount(count + 1)}>

Count: {count}

</button>

);

}

* useState(0) возвращает текущее значение и функцию для его изменения.

**66. React: использование хука эффекта**

jsx

КопироватьРедактировать

import { useEffect } from 'react';

useEffect(() => {

console.log("Component mounted or updated");

return () => console.log("Component unmounted");

}, []);

* useEffect выполняется после рендера.
* Зависимости ([]) указывают, когда эффект будет вызываться (только при монтировании, обновлении и т.д.).

**67. Redux: назначение, основные концепции**

* **Redux** — библиотека управления состоянием, особенно полезна в больших приложениях.
* **Основные концепции**:
  + Единое глобальное хранилище (store)
  + Состояние только для чтения
  + Изменения происходят через **actions** и **reducers**

**68. Redux: экшены**

* **Экшены (actions)** — обычные объекты с полем type, описывающим, что должно произойти:

js

КопироватьРедактировать

{ type: 'INCREMENT' }

* Могут содержать данные (payload):

js

КопироватьРедактировать

{ type: 'ADD\_TODO', payload: { text: 'Learn Redux' } }

**69. Redux: редьюсеры**

* **Редьюсер (reducer)** — чистая функция, принимающая старое состояние и action, возвращающая новое состояние:

js

КопироватьРедактировать

function counter(state = 0, action) {

switch (action.type) {

case 'INCREMENT':

return state + 1;

case 'DECREMENT':

return state - 1;

default:

return state;

}

}

**70. Redux: стор**

* **Store** — объект, который:
  + Хранит состояние (state)
  + Предоставляет методы: dispatch(action), getState(), subscribe(listener)
  + Создаётся через createStore(reducer):

js

КопироватьРедактировать

import { createStore } from 'redux';

const store = createStore(reducer);

**71. Redux: поток данных**

* Однонаправленный поток данных:
  1. Компонент вызывает dispatch(action)
  2. Action передаётся в reducer
  3. Reducer возвращает новое состояние
  4. Store обновляет состояние и уведомляет подписчиков

**72. Redux: использование с React (react-redux)**

* Подключение через библиотеку react-redux.
* **Provider** — оборачивает всё приложение и предоставляет доступ к хранилищу:

jsx

КопироватьРедактировать

<Provider store={store}>

<App />

</Provider>

* **useSelector** — получение данных из store.
* **useDispatch** — отправка actions.

jsx

КопироватьРедактировать

const count = useSelector(state => state.counter);

const dispatch = useDispatch();

dispatch({ type: 'INCREMENT' });