Вопросы к экзамену 2022-2023 уч. г. для студентов II-го курса

спец. 1-40 01 01 «Программное обеспечение информационных технологий»

по дисциплине «Технологии разработки пользовательских интерфейсов»

1. Переменные. Имена переменных. Константы.
2. Типы данных. Оператор typeof.

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.

*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.
2. 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

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.' );

}

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.

**[&& (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.

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!' );

}

[**! (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

**?? (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';

...

}

1. Функциональные выражения (Function Expression) и функции-стрелки.
2. Числа. Способы записи числа. Системы счисления. Методы типа Number: преобразование к числу, округление, проверка специальных числовых значений.

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

[**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

[**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.

[**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€".

1. Строки. Методы типа String: изменение регистра, поиск подстроки. Сравнение строк.

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";

[**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

| **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}" );

[**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

}

[**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.

**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

[**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

[**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.

[**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".

1. Массивы.

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

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

Существует два варианта синтаксиса для создания пустого массива:

let arr = new Array();

let arr = [];

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

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

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

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

alert( fruits[0] ); // Яблоко

alert( fruits[1] ); // Апельсин

alert( fruits[2] ); // Слива

Можно заменить элемент:

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

Или добавить новый элемент к существующему массиву:

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

Общее число элементов массива содержится в его свойстве length:

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

alert( fruits.length ); // 3

Вывести массив целиком можно при помощи alert:

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

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

В массиве могут храниться элементы любого типа:

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

alert( arr[1].name ); // Джон

arr[3](); // привет

1. Методы массивов: добавление, удаление и замена элементов, объединение массивов, поиск в массиве.
2. 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

[**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" \l "kopirovanie-po-ssylke)**

Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке». Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению». Например:

let message = "Hello!";

let phrase = message;

В результате имеются две независимые переменные, каждая из которых хранит строку "Hello!". Объекты ведут себя иначе. Переменная хранит не сам объект, а его «адрес в памяти», другими словами «ссылку» на него. Например:

let user = {

name: "John"

};

Сам объект хранится где-то в памяти. А в переменной user лежит «ссылка» на эту область памяти. Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется:

let user = { name: "John" };

let admin = user; // копируется ссылка

В результате есть две переменные, каждая из которых содержит ссылку на один и тот же объект. Можно использовать любую из переменных для доступа к объекту и изменения его содержимого:

let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // изменено по ссылке из переменной "admin"

alert(user.name); // 'Pete', изменения видны по ссылке из переменной "user"

**[Сравнение объектов](https://learn.javascript.ru/object" \l "sravnenie-obektov)**

Операторы равенства == и строгого равенства === для объектов работают одинаково. Два объекта равны только в том случае, если это один и тот же объект. Например, две переменные ссылаются на один и тот же объект, они равны:

let a = {};

let b = a; // копирование по ссылке

alert( a == b ); // true

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

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

let a = {};

let b = {}; // два независимых объекта

alert( a == b ); // false

Для сравнений типа obj1 > obj2 или для сравнения с примитивом obj == 5 объекты преобразуются в примитивы. Такое сравнение используется очень редко и не рекомендуется.

**[Объекты-константы](https://learn.javascript.ru/object" \l "obekty-konstanty)**

Объект, объявленный через const, может быть изменен:

const user = {

name: "John"

};

user.age = 25; // (\*)

alert(user.age); // 25

Объявление const защищает от изменений только само значение user. В примере значение user – это ссылка на объект, и это значение не меняется. В строке (\*) вносятся изменения внутри объекта, а значение user не изменяется. Если же попытаться присвоить user другое значение, то const выдаст ошибку:

const user = {

name: "John"

};

// Ошибка (нельзя переопределять константу user)

user = {

name: "Pete"

};

Cделать константами свойства объекта тоже возможно с помощью флагов и дескрипторов свойств, кторые будут рассматриваться позже.

[**Клонирование и объединение объектов, Object.assign**](https://learn.javascript.ru/object#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.
2. 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

});

**WeakMap**

Как извстно, движок JavaScript хранит значения в памяти до тех пор, пока они достижимы (то есть, эти значения могут быть использованы). Например:

let john = { name: "John" };

// объект доступен, переменная john -- это ссылка на него

// перепишем ссылку

john = null;

// объект будет удалён из памяти

Обычно свойства объекта, элементы массива или другой структуры данных считаются достижимыми и сохраняются в памяти до тех пор, пока эта структура данных содержится в памяти. Например, если поместить объект в массив, то до тех пор, пока массив существует, объект также будет существовать в памяти, несмотря на то, что других ссылок на него нет. Например:

let john = { name: "John" };

let array = [ john ];

john = null; // перезаписываем ссылку на объект

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

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

let john = { name: "John" };

let map = new Map();

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

john = null; // перезаписываем ссылку на объект

// объект john сохранён внутри объекта `Map`

WeakMap – принципиально другая структура в этом аспекте. Она не предотвращает удаление объектов сборщиком мусора, когда эти объекты выступают в качестве ключей. Первое его отличие от Map в том, что ключи в WeakMap должны быть объектами, а не примитивными значениями:

let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok");

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

Теперь, если использовать объект в качестве ключа и если больше нет ссылок на этот объект, то он будет удалён из памяти (и из объекта WeakMap) автоматически.

let john = { name: "John" };

let weakMap = new WeakMap();

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

john = null;

// объект john удалён из памяти

Теперь john существует только как ключ в WeakMap и может быть удалён оттуда автоматически. WeakMap не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё. В WeakMap присутствуют только следующие методы:

* weakMap.get(key)
* weakMap.set(key, value)
* weakMap.delete(key)
* weakMap.has(key)

Такие ограничения связаны с особенностью технической реализации. Если объект станет недостижим (как объект john в примере выше), то он будет автоматически удалён сборщиком мусора. Но нет информации, в какой момент произойдет эта очистка. Решение о том, когда делать сборку мусора, принимает движок JavaScript. Он может посчитать необходимым как удалить объект прямо сейчас, так и отложить эту операцию, чтобы удалить большее количество объектов за раз позже. Так что технически количество элементов в коллекции WeakMap неизвестно. Движок может произвести очистку сразу или потом, или сделать это частично. По этой причине методы для доступа ко всем сразу ключам/значениям недоступны.

В основном, WeakMap используется в качестве дополнительного хранилища данных или кеширования, когда результат вызова функции должен где-то запоминаться («кешироваться») для того, чтобы дальнейшие её вызовы на том же объекте могли просто брать уже готовый результат, повторно используя его. Для хранения результатов можно использовать Map:

// cache.js

let cache = new Map();

// вычисляем и запоминаем результат

function process(obj) {

if (!cache.has(obj)) {

let result = /\* какие-то вычисления \*/ obj;

cache.set(obj, result);

}

return cache.get(obj);

}

// main.js

let obj = {/\* какой-то объект \*/};

let result1 = process(obj);

let result2 = process(obj); // ранее вычисленный результат взят из кеша

obj = null;

alert(cache.size); // 1, объект всё ещё в кеше и занимает память

Многократные вызовы process(obj) с тем же самым объектом в качестве аргумента ведут к тому, что результат вычисляется только в первый раз, а затем последующие вызовы берут его из кеша. Недостатком является то, что необходимо вручную очищать cache от ставших ненужными объектов. Но если использовать WeakMap вместо Map, то эта проблема исчезнет: закешированные результаты будут автоматически удалены из памяти сборщиком мусора.

// cache.js

let cache = new WeakMap();

function process(obj) {

if (!cache.has(obj)) {

let result = /\* вычисляем результат для объекта \*/ obj;

cache.set(obj, result);

}

return cache.get(obj);

}

// main.js

let obj = {/\* какой-то объект \*/};

let result1 = process(obj);

let result2 = process(obj);

obj = null;

// Нет возможности получить cache.size, так как это WeakMap,

// но он равен 0 или скоро будет равен 0

// Когда сборщик мусора удаляет obj, связанные с ним данные из кеша тоже удаляются

1. Деструктурирующее присваивание. Вложенная деструктуризация.

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.

[**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.

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

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения. В браузере он называется window, в Node.js – global, в другой среде исполнения может называться иначе. Недавно globalThis был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении. В некоторых браузерах (например, Edge) globalThis ещё не поддерживается, но легко реализуется с помощью полифила.

Прежде, решение было таким:

const getGlobal = function () {

if (typeof self !== undefined) {

return self;

}

if (typeof window !== undefined) {

return window;

}

if (typeof global !== undefined) {

return global;

}

throw new Error("unable to locate global object");

};

const globals = getGlobal();

// Сейчас есть `globalThis`

globalThis === window; // true

Рассмотрим подробнее глобальный объект window, так как наша среда – браузер. Ко всем свойствам глобального объекта можно обращаться напрямую:

alert("Привет");

// это то же самое, что и

window.alert("Привет");

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const), становятся свойствами глобального объекта:

var gVar = 5;

alert(window.gVar); // 5 (становится свойством глобального объекта)

Такое поведение поддерживается для совместимости. В современных проектах, использующих [JavaScript-модули](https://learn.javascript.ru/modules), такого не происходит.

Если объявить переменную при помощи let, то такого не произойдет:

let gLet = 5;

alert(window.gLet); // undefined (не становится свойством глобального объекта)

Если свойство настолько важное, что надо сделать его доступным для всей программы, то запишите его в глобальный объект напрямую:

window.currentUser = {

name: "John"

};

alert(currentUser.name); // John

alert(window.currentUser.name); // John

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

Глобальный объект можно использовать, чтобы проверить поддержку современных возможностей языка. Например, проверить наличие встроенного объекта Promise (такая поддержка отсутствует в очень старых браузерах):

if (!window.Promise) {

alert("Ваш браузер очень старый!");

}

Если такой объект не поддерживается, то можно создать полифил: добавить функции, которые не поддерживаются окружением, но существуют в современном стандарте.

if (!window.Promise) {

window.Promise = ... // реализация современной возможности языка

}

1. **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.
2. Объект функции. Именованное функциональное выражение (Named Function Expression).

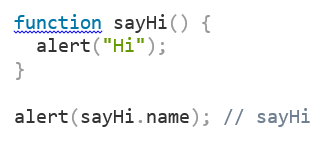
**1.Объект функции**

Как известно, в JavaScript функция – это значение. Каждое значение в JavaScript имеет свой тип. В JavaScript, функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

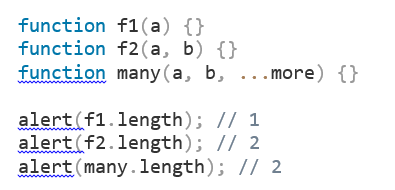
Объект функции содержит несколько полезных свойств:

Например, имя функции доступно как **свойство name**:



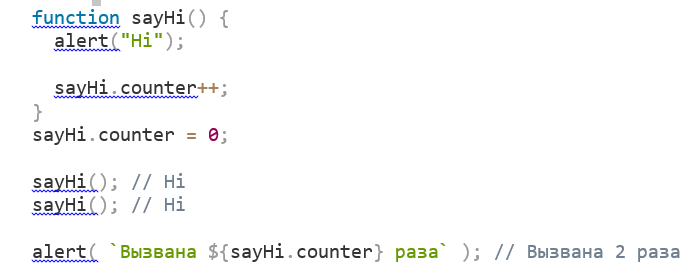
[**Свойство length**](https://learn.javascript.ru/function-object#svoystvo-length)

Ещё одно встроенное свойство length содержит количество параметров функции в её объявлении.



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.

Также можно добавить **свои собственные свойства**. Например, свойство counter для отслеживания общего количества вызовов:



**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.

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

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

let sayHi = function func(who) {

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

};

Заметьте, что функция всё ещё задана как Function Expression. Добавление "func" после function не превращает объявление в Function Declaration, потому что оно все еще является частью выражения присваивания. Добавление такого имени ничего не ломает. Функция все еще доступна как sayHi():

let sayHi = function func(who) {

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

};

sayHi("John"); // Hello, John

Есть две важные особенности имени func, ради которого оно даётся:

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

Например, ниже функция sayHi вызывает себя с "Guest", если не передан параметр who:

let sayHi = function func(who) {

if (who) {

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

} else {

func("Guest");

}

};

sayHi(); // Hello, Guest

func(); // Ошибка

Не следует использовать имя sayHi для вложенного вызова, так как значение sayHi может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнет выдавать ошибки:

let sayHi = function(who) {

if (who) {

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

} else {

sayHi("Guest"); // Ошибка

}

};

let welcome = sayHi;

sayHi = null;

welcome(); // Ошибка

Так происходит, потому что функция берет sayHi из внешнего лексического окружения. Так как локальная переменная sayHi отсутствует, используется внешняя. И на момент вызова эта внешняя sayHi равна null. Необязательное имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы. Все работает, потому что имя "func" локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию. Внешний код все еще содержит переменные sayHi и welcome, но теперь func – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Задать «внутреннее» имя можно только для Function Expression, и не нельзя для Function Declaration. Если нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

1. Остаточные параметры и оператор расширения.

Многие встроенные функции JavaScript поддерживают произвольное количество аргументов. Например: Math.max(arg1, arg2, ..., argN) – вычисляет максимальное число из переданных; Object.assign(dest, src1, ..., srcN) – копирует свойства из исходных объектов src1..N в целевой объект dest и др

1.[**Остаточные параметры (...)**](https://learn.javascript.ru/rest-parameters-spread-operator#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.

Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами.

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

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

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

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

Если вызвать эту функцию для массива чисел [3, 5, 1], то она его не обработает, так как ожидает список параметров:

let arr = [3, 5, 1];

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

Чтобы преобразовать массив в список необходимо использовать оператор расширения. Он похож на остаточные параметры – тоже использует ..., но делает совершенно противоположное. Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов. Для Math.max:

let arr = [3, 5, 1];

alert( Math.max(...arr) ); // 5

Этим же способом можно передать несколько итерируемых объектов и комбинировать оператор расширения с обычными значениями:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

Оператор расширения можно использовать и для слияния массивов:

let arr = [3, 5, 1];

let arr2 = [8, 9, 15];

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

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

Оператора расширения работает с любым перебираемым объектом. Например, оператор расширения подойдёт для того, чтобы превратить строку в массив символов:

let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т

Оператор расширения использует итераторы, чтобы собирать элементы. Так же, как это делает for..of. Цикл for..of перебирает строку как последовательность символов, поэтому из ...str получается "П", "р", "и", "в", "е", "т". Получившиеся символы собираются в массив при помощи стандартного объявления массива: [...str].

Для этой задачи можно использовать и Array.from. Он тоже преобразует перебираемый объект (такой как строка) в массив:

let str = "Привет";

alert( Array.from(str) ); // П,р,и,в,е,т

Результат аналогичен [...str]. Но между Array.from(obj) и [...obj] есть разница: Array.from работает как с псевдомассивами, так и с итерируемыми объектами; оператор расширения работает только с итерируемыми объектами. Таким образом, если нужно сделать из чего угодно массив, Array.from – более универсальный метод.

1. Каррирование 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.

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.
2. Методы объектов, this. Оператор опциональной последовательности.
3. **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;

**2.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)

Существуют три варианта преобразований:

* "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.

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.

**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

1. Создание объектов через "new".

создать множество однотипных объектов, таких как пользователи, элементы меню 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.
2. 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.

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 user.fullName  не вызывается  как функция, а читается как обычное свойство: геттер сам вернет нужное значение.

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.

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:

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 );

1. Декораторы. Методы call(), apply(), bind().

Bind()

привязке [контекста с помощью bind](https://learn.javascript.ru/bind#reshenie-2-privyazat-kontekst-s-pomoschyu-bind). В современном JavaScript у функций есть встроенный метод [bind](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/bind), который позволяет зафиксировать this. Базовый синтаксис bind:

let boundFunc = func.bind(context);

Результатом вызова func.bind(context) является особый, который вызывается как функция и прозрачно передает вызов в func, при этом устанавливая this=context. Другими словами, вызов boundFunc подобен вызову func с фиксированным this. Например, в коде ниже funcUser передает вызов в func, фиксируя this=user:

let user = {

firstName: "Вася"

};

function func() {

alert(this.firstName);

}

let funcUser = func.bind(user);

funcUser(); // Вася

Здесь func.bind(user) – это «связанный вариант» func, с фиксированным this=user. Все аргументы передаются исходному методу func «как есть», например:

let user = {

firstName: "Вася"

};

function func(phrase) {

alert(phrase + ', ' + this.firstName);

}

// привязка this к user

let funcUser = func.bind(user);

funcUser("Привет"); // Привет, Вася

Теперь давайте попробуем с методом объекта:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

let sayHi = user.sayHi.bind(user); // (\*)

sayHi(); // Привет, Вася!

setTimeout(sayHi, 1000); // Привет, Вася!

В строке (\*) метод user.sayHi привязываем к user. Теперь SayHi – это «связанная» функция, которая может быть вызвана отдельно или передана в setTimeout (контекст всегда будет правильным). Здесь можно видеть, что bind исправляет только this, а аргументы передаются «как есть»:

let user = {

firstName: "Вася",

say(phrase) {

alert(`${phrase}, ${this.firstName}!`);

}

};

let say = user.say.bind(user);

say("Привет"); // Привет, Вася

say("Пока"); // Пока, Вася

До сих пор речь шла только о привязывании this. Можно привязать не только this, но и аргументы. Это делается редко, но иногда может быть полезно.

Полный синтаксис bind:

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

Это позволяет привязать контекст this и начальные аргументы функции. Например, у нас есть функция умножения mul(a, b):

function mul(a, b) {

return a \* b;

}

Воспользуемся bind, чтобы создать функцию double на её основе:

function mul(a, b) {

return a \* b;

}

let double = mul.bind(null, 2);

alert( double(3) ); // = mul(2, 3) = 6

alert( double(4) ); // = mul(2, 4) = 8

alert( double(5) ); // = mul(2, 5) = 10

Вызов mul.bind(null, 2) создаёт новую функцию double, которая передаёт вызов mul, фиксируя null как контекст и 2 – как первый аргумент. Следующие аргументы передаются «как есть». Это называется [частичное применение](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5) – создаётся новая функция, которая фиксирует некоторые из существующих параметров.

Обратите внимание, что в данном случае не используется this. Но для bind это обязательный параметр, так что надо передать туда что-нибудь вроде null. В следующем коде функция triple умножает значение на три:

function mul(a, b) {

return a \* b;

}

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9

alert( triple(4) ); // = mul(3, 4) = 12

alert( triple(5) ); // = mul(3, 5) = 15

Польза от частично применённой функции в том, что можно создать независимую функцию с понятным названием (double, triple). Можно использовать её и не передавать каждый раз первый аргумент, т.к. он зафиксирован с помощью bind.

В других случаях частичное применение полезно, когда есть очень общая функция и для удобства надо создать её частный вариант. Например, есть функция send(from, to, text) и позже может возникнуть необходимость внутри объекта user использовать её частный вариант: sendTo(to, text), который отправляет текст от имени текущего пользователя.

Возможна ситуация, когда надо зафиксировать некоторые аргументы, но не контекст this. Например, для метода объекта. Встроенный bind не позволяет этого. Нельзя просто опустить контекст и перейти к аргументам. Но можно создать вспомогательную функцию partial, которая привязывает только аргументы. Вот так:

function partial(func, ...argsBound) {

return function(...args) { // (\*)

return func.call(this, ...argsBound, ...args);

}

}

let user = {

firstName: "John",

say(time, phrase) {

alert(`[${time}] ${this.firstName}: ${phrase}!`);

}

};

user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello"); // [10:00] John: Hello!

Результатом вызова partial(func[, arg1, arg2...]) будет обёртка (\*), которая вызывает func с:

* Тем же this, который она получает (для вызова user.sayNow – это будет user)
* Затем передаёт ей ...argsBound – аргументы из вызова partial ("10:00")
* Затем передаёт ей ...args – аргументы, полученные обёрткой ("Hello")

Благодаря оператору расширения ... это реализовать легко.

[**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.

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);

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.

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];

}

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

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

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

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

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:

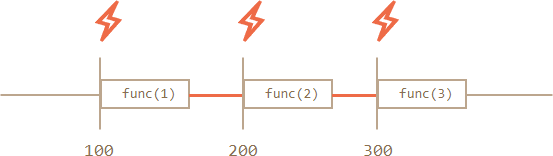
let i = 1; setTimeout(function run() {

func(i); setTimeout(run, 100);

}, 100);

Для setInterval внутренний планировщик выполнит func(i) каждые 100

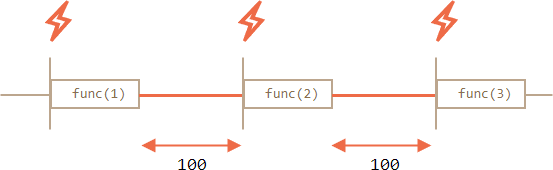
мс:



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 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 для Node.js. Так что это ограничение относится только к браузерам.

1. Прототипное наследование. Собственные и унаследованные свойства. Свойство F.prototype.
2. 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»

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); }

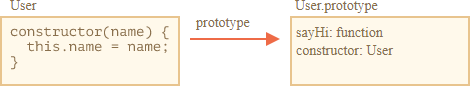
}

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

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

Свойства классов добавлены в язык недавно. Старым браузерам может понадобиться полифил. В приведённом выше примере у класса User были только методы. Добавим свойство:

class User {

name = "Аноним";

sayHi() {

alert(`Привет, ${this.name}!`);

}

}

new User().sayHi();

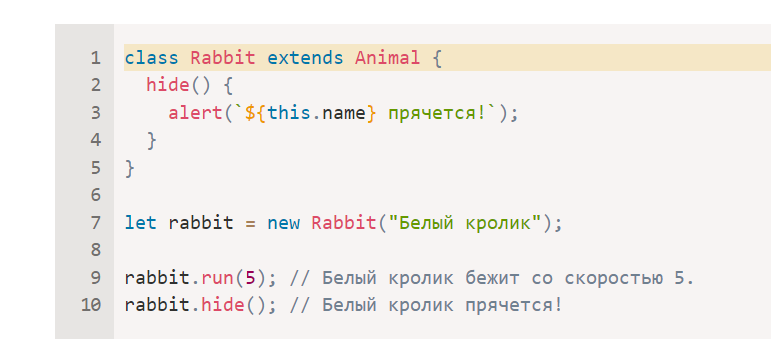
Свойство name не устанавливается в User.prototype. Вместо этого оно создаётся оператором new перед запуском конструктора, это именно свойство объекта

1. Наследование классов. Переопределение методов. Статические свойства и методы. Оператор instanceof

В языке JavaScript есть поддержка наследования классов, переопределения методов, а также использования статических свойств и методов. Оператор instanceof также доступен для проверки типа объекта.

Наследование классов:

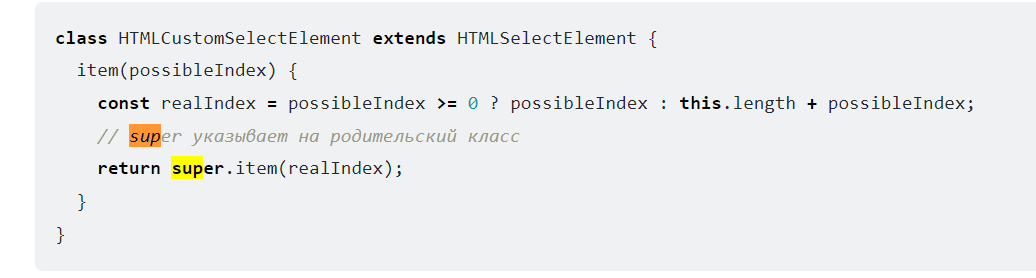
В JavaScript классы могут наследовать друг от друга с помощью ключевого слова extends.



Переопределение методов:

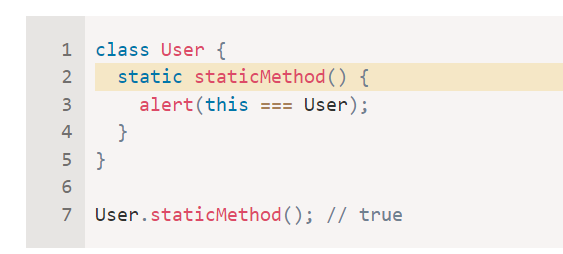
Классы могут переопределять методы родительского класса. Для этого достаточно определить метод с таким же именем в дочернем классе. Методы родительского класса могут быть вызваны из дочернего класса с помощью ключевого слова super.

Выше создан подкласс *HTMLCustomSelectElement*, который **переопределяет** метод item(index). Переопределение означает, что в подклассе создается метод с тем же именем, что и в родительском классе. Наш новый метод выполняет дополнительную работу по вычислению индекса, но ему все еще нужен исходный метод item(index), для выборки нужного элемента. Для этого применяется специальный синтаксис, который указывает явно что нужно взять метод из родительского класса: super.item(realIndex).



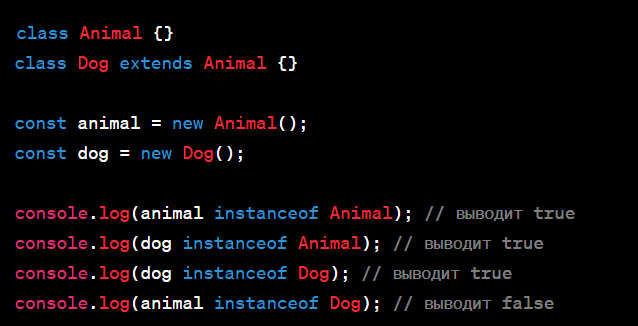
Статические свойства и методы:

В JavaScript классы могут иметь статические свойства и методы, которые принадлежат самому классу, а не его экземплярам. Доступ к статическим свойствам и методам осуществляется через имя класса.



Оператор instanceof:

Оператор instanceof используется для проверки, является ли объект экземпляром определенного класса или его потомком. Он возвращает true, если объект является экземпляром указанного класса, и false в противном случае. Пример:



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 – одна из самых старых модульных систем, изначально реализована библиотекой require.js.
* CommonJS – модульная система, созданная для сервера Node.js.
* 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>

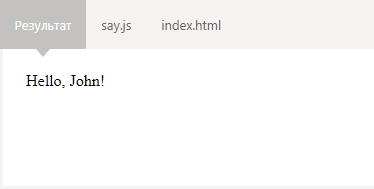
<script type="module">

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

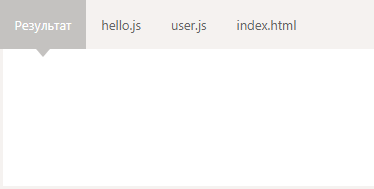
Есть основные возможности и особенности, работающие как в браузере, так и в серверном JavaScript.

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";

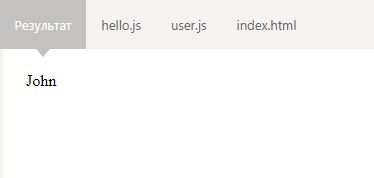
// index.html

<!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. 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

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. 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

В реальной жизни модули в браузерах редко используются как есть. Обычно, они объединяются вместе, специальными инструментами, например Webpack 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:

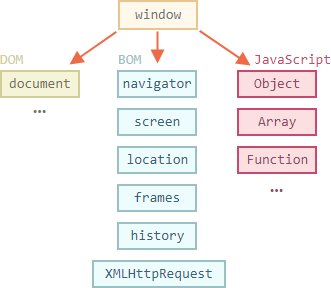
<script src="bundle.js"></script>

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

1. Модули: экспорт и импорт.
2. Окружение: DOM, BOM. Дерево DOM.

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)

*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.

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, 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)

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

* Объект 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 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. Спецификация HTML по адресу 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.

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

Разберемся со структурой DOM. Рассмотрим простой документ:

<!DOCTYPE HTML>

<html>

<head>

<title>О лосях</title>

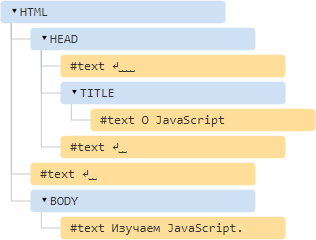
</head>

<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).

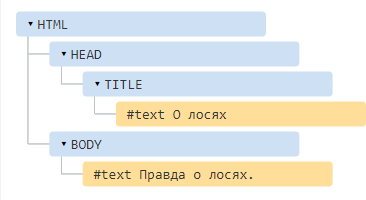
Существует всего два исключения из этого правила:

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>



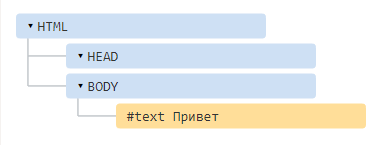
Пробелы по краям строк и пробельные текстовые узлы скрыты в инструментах разработки и обычно не отображаются. Таким образом инструменты разработки экономят место на экране.

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

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:

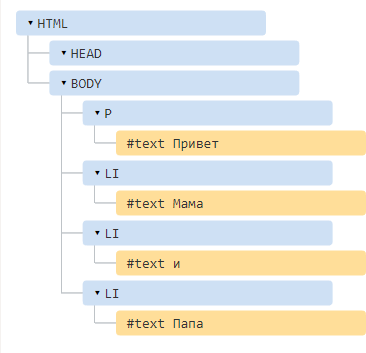
<p>Привет

<li>Мама

<li>и

<li>Папа

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

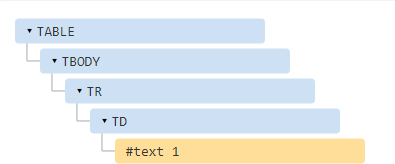


Важный «особый случай» – работа с таблицами. По стандарту DOM у них должен быть <tbody>, но в HTML их можно написать (официально) без

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

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>

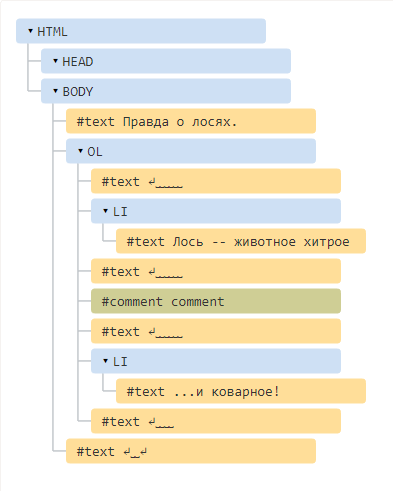
<!-- комментарий -->

<li>...и коварное!</li>

</ol>

</body>

</html>

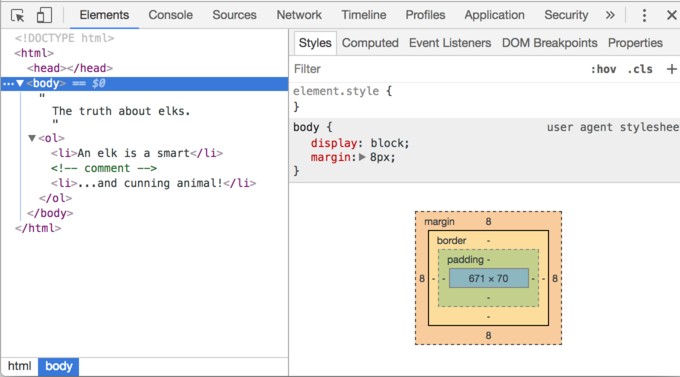


Здесь видно, что новый узел нового типа – комментарий, обозначенный как #comment, между двумя текстовыми узлами. Есть важное правило: если что-то есть в HTML, то оно должно быть в DOM-дереве. Поэтому все, что есть в HTML, даже комментарии, является частью DOM. Даже директива <!DOCTYPE...>, которая располагается 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. Но на практике в основном работают с четырьмя из них:

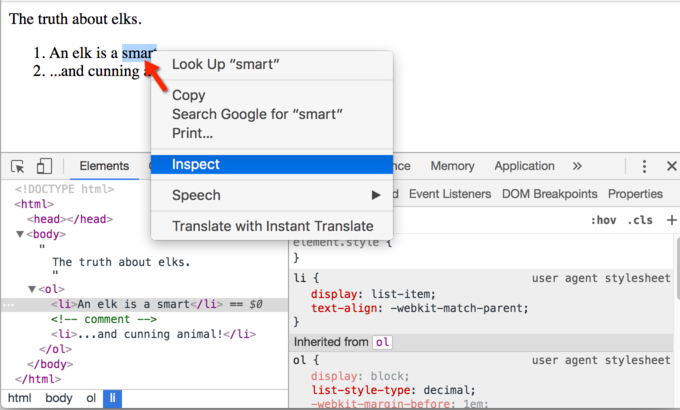
1. document – «входная точка» в DOM.
2. Узлы-элементы – HTML-теги, основные строительные блоки.
3. Текстовые узлы – содержат текст.
4. Комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

Чтобы посмотреть структуру DOM в реальном времени, можно использовать инструменты разработчика браузера. Для этого откройте страницу \*.html, 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

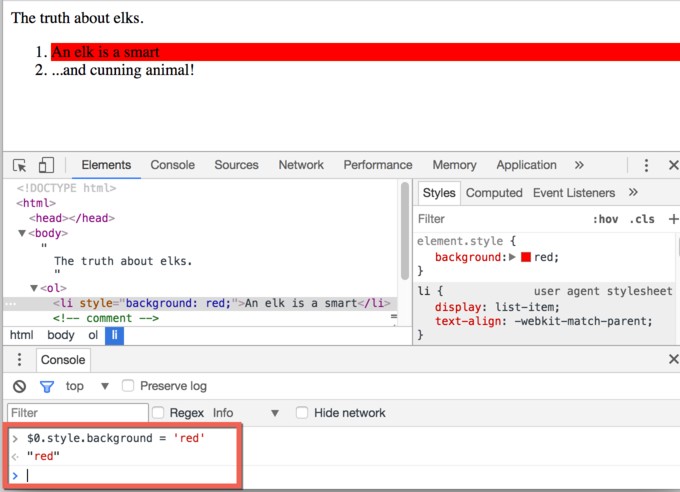
При работе с DOM, часто требуется применить к нему JavaScript. Например: получить узел и запустить какой-нибудь код для его изменения, чтобы посмотреть результат. Вот несколько подсказок по тому, как перемещаться между вкладками Elements и Console. Для начала:

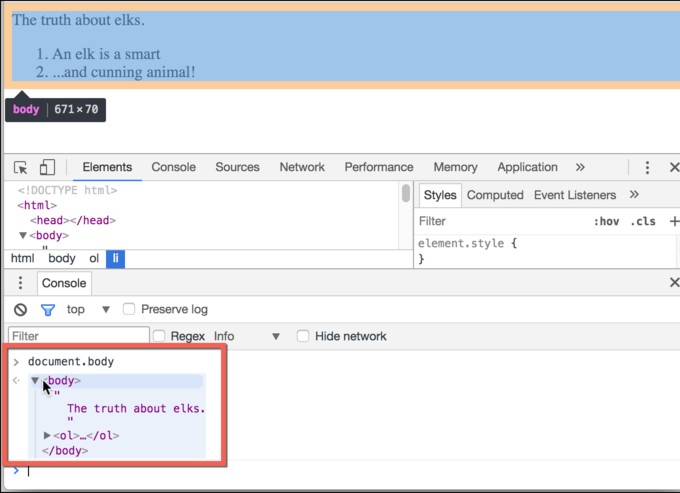
* На вкладке Elements выберите первый элемент <li>.
* Нажмите Esc – прямо под вкладкой Elements откроется Console.

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:



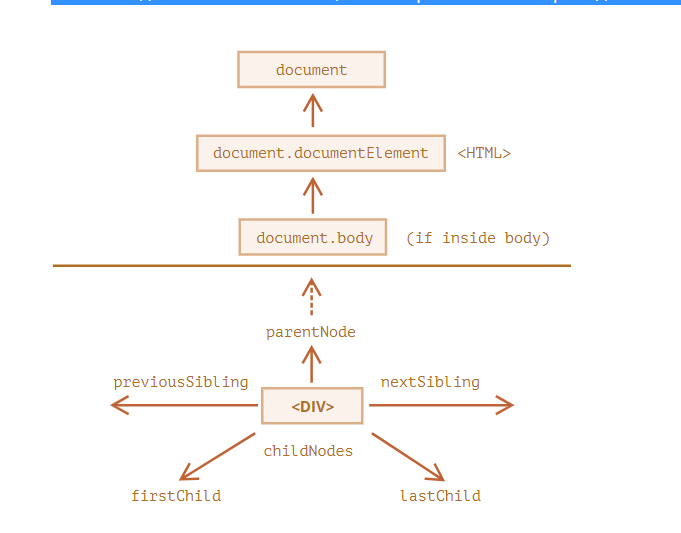
Это может быть полезно для отладки. Инструменты разработчика браузера отлично помогают в разработке: можно исследовать DOM, пробовать с ним что-то делать и смотреть, что идёт не так.

1. Навигация и методы поиска DOM-элементов.

DOM позволяет нам 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 мы можем получить доступ к любому узлу.

Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



**[Сверху: documentElement и body](https://learn.javascript.ru/dom-navigation" \l "sverhu-documentelement-i-body)**

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.

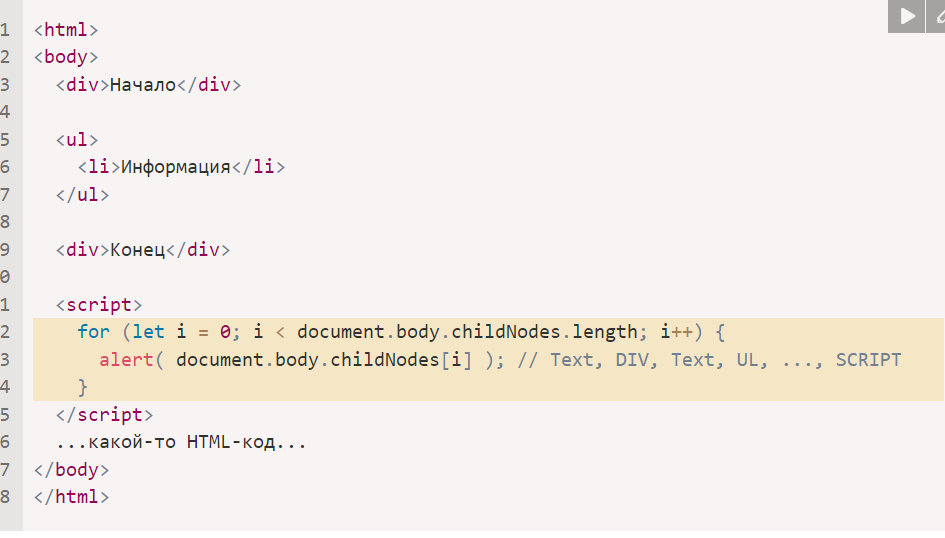
## [Дети: childNodes, firstChild, lastChild](https://learn.javascript.ru/dom-navigation" \l "deti-childnodes-firstchild-lastchild)

Здесь и далее мы будем использовать два принципиально разных термина:

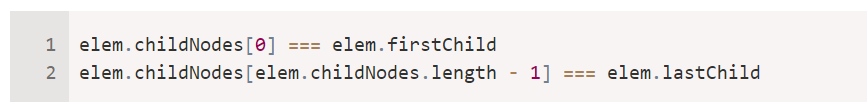
* **Дочерние узлы (или дети)** – элементы, которые являются непосредственными детьми узла. Другими словами, элементы, которые лежат непосредственно внутри данного. Например, <head> и <body> являются детьми элемента <html>.
* **Потомки** – все элементы, которые лежат внутри данного, включая детей, их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

В JavaScript существует несколько методов для навигации и поиска DOM-элементов. Эти методы позволяют получать доступ к элементам на веб-странице и взаимодействовать с ними.

**Коллекция childNodes содержит список всех детей, включая текстовые узлы.**



**Свойства firstChild и lastChild обеспечивают быстрый доступ к первому и последнему дочернему элементу.**



Навигация по DOM-дереву:

parentNode: Возвращает родительский элемент указанного элемента.

childNodes: Возвращает коллекцию дочерних элементов указанного элемента.

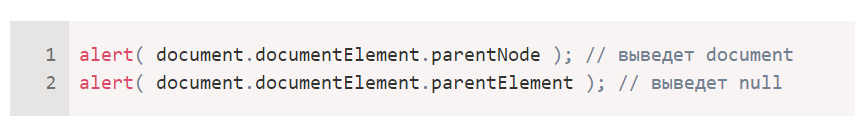
querySelector: Возвращает первый элемент, соответствующий указанному селектору CSS.

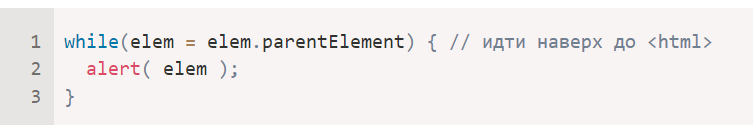
querySelectorAll: Возвращает все элементы, соответствующие указанному селектору CSS.

nextSibling: Возвращает следующий элемент на том же уровне вложенности.

previousSibling: Возвращает предыдущий элемент на том же уровне вложенности.

children: Возвращает коллекцию дочерних элементов-элементов, пропуская текстовые узлы и комментарии.





Поиск DOM-элементов:

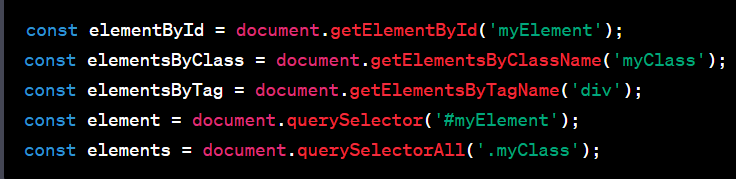
getElementById: Возвращает элемент с указанным идентификатором.

getElementsByClassName: Возвращает коллекцию элементов с указанным классом.

getElementsByTagName: Возвращает коллекцию элементов с указанным тегом.

querySelector: Возвращает первый элемент, соответствующий указанному селектору CSS.

querySelectorAll: Возвращает все элементы, соответствующие указанному селектору CSS.



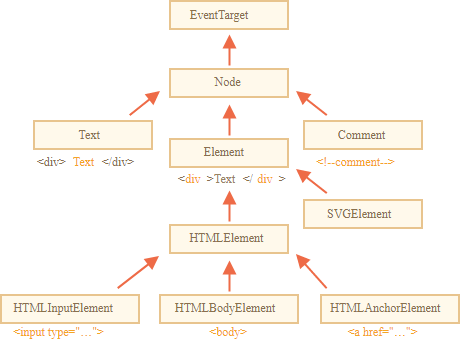
1. Свойства узлов: тип, тег и содержимое.

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

# Классы DOM-узлов

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, от него наследует Node и остальные DOM-узлы. На рисунке ниже изображены основные классы:



Существуют следующие классы:

* 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 – 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 – 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 – является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
  + HTMLInputElement – класс для тега <input>,
  + HTMLBodyElement – класс для тега <body>,
  + 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. 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

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

Свойство 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»

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

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.
2. 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

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

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';

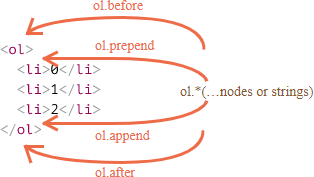
ol.prepend(liFirst); // вставить liFirst в начало <ol>

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>

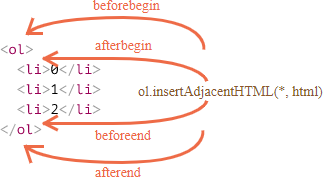
Приведёт к:

<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

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

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

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());

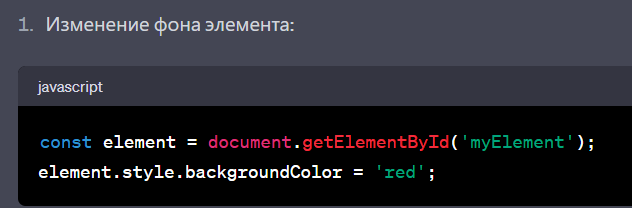
</script>

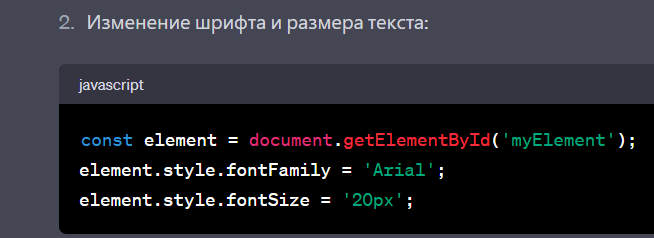
DocumentFragment используется в некоторых других областях, например, для элемента template.

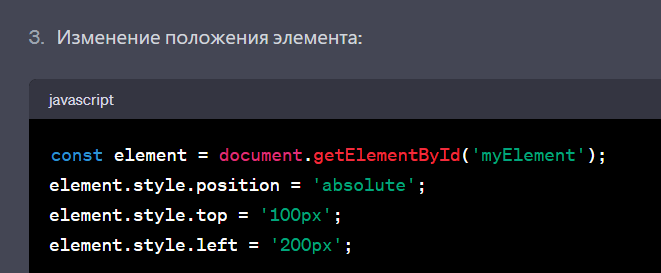
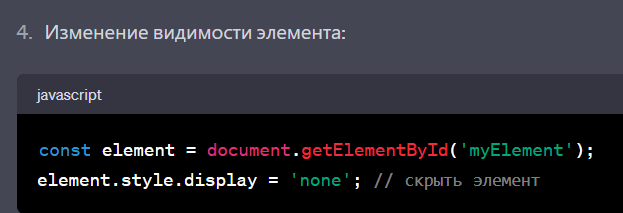
1. Стили DOM-узлов.

Для изменения стилей DOM-узлов в JavaScript можно использовать свойство style. Свойство style предоставляет доступ к инлайновым стилям элемента. Чтобы изменить стили элемента, вы можете установить значения свойств объекта style в соответствии с нужными вам стилями.

Вот несколько примеров, демонстрирующих, как изменять стили DOM-узлов:





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;

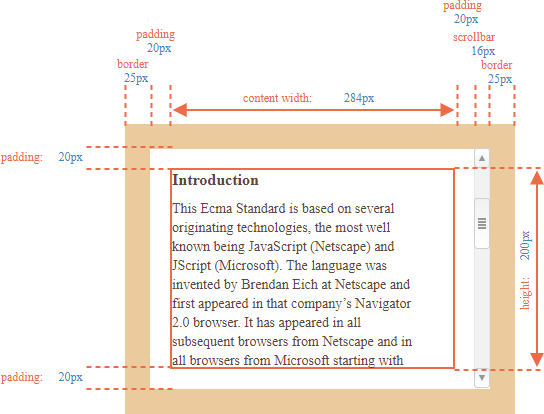
border: 25px solid #E8C48F; padding: 20px;

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:

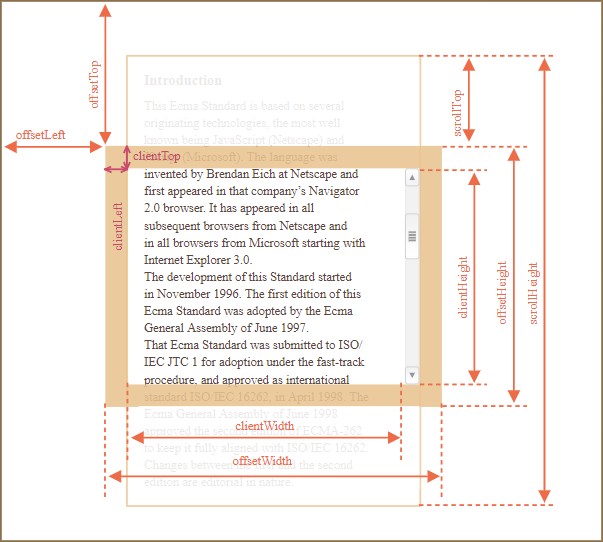


В иллюстрации выше намеренно продемонстрирован самый сложный и полный случай, когда у элемента есть ещё и полоса прокрутки. Некоторые браузеры (не все) отбирают место для неё, забирая его у области, отведённой для содержимого (помечена как «content width» выше).

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 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>

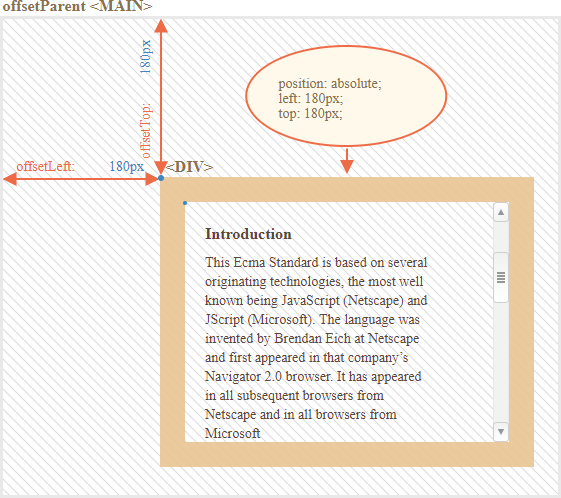
</article>

</main>

<script>

alert(example.offsetParent.id); // main alert(example.offsetLeft); // 180 (число, а не строка "180px") alert(example.offsetTop); // 180

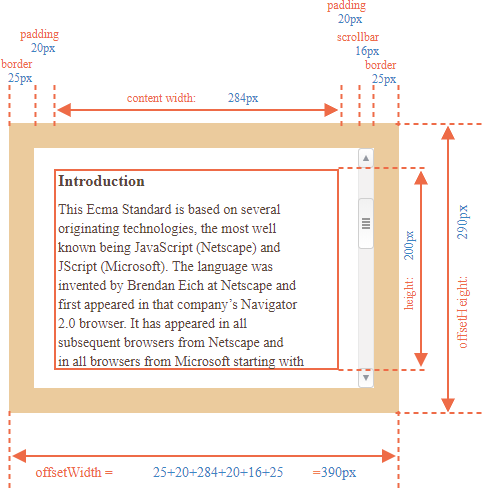
</script>



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 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) {

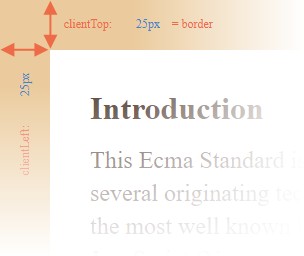
return !elem.offsetWidth && !elem.offsetHeight;

}

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:

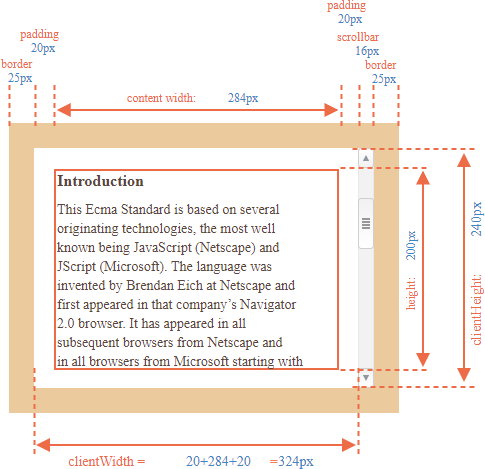
* clientLeft = 25 – ширина левой рамки,
* clientTop = 25 – ширина верхней рамки.

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.

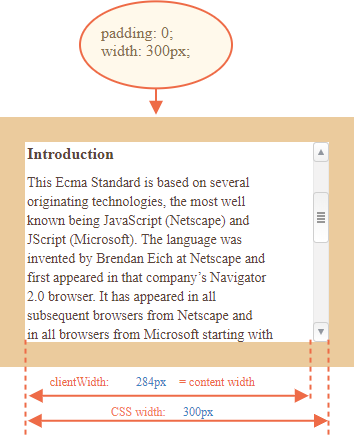
Вот соответствующий пример на иврите:

Свойства clientWidth/Height – 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.

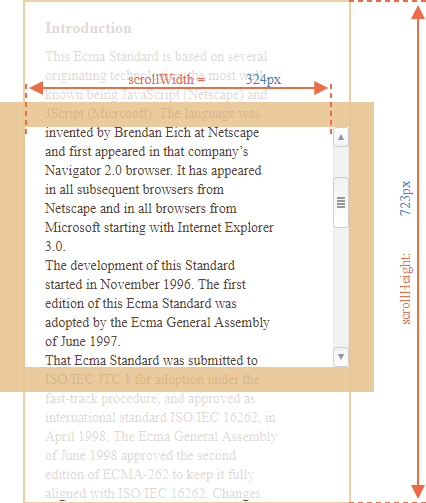
Если нет внутренних отступов padding, то clientWidth/Height в точности равны размеру области содержимого внутри рамок и полосы прокрутки (если она есть).

Поэтому в тех случаях, когда точно известно, что отступов нет, можно использовать clientWidth/clientHeight для получения размеров

внутренней области содержимого.

Свойства

scrollWidth/Height – как clientWidth/clientHeight, но также включают в себя прокрученную (которую не видно) часть элемента.

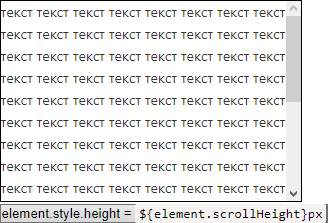


На рисунке выше:

* scrollHeight = 723 – полная внутренняя высота, включая прокрученную область;
* scrollWidth = 324 – полная внутренняя ширина, в данном случае прокрутки нет, поэтому она равна clientWidth.

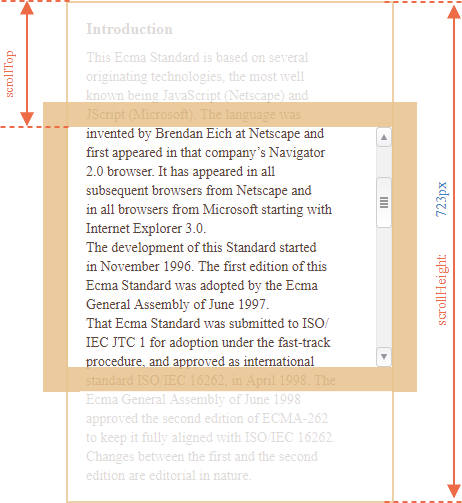
Эти свойства можно использовать, чтобы «распахнуть» элемент на всю ширину/высоту, например при нажатии на кнопку:

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.

1. 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.
2. 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

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 погружении 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).

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

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

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;

}

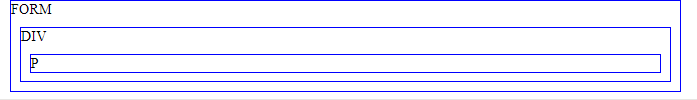
</style>

<form onclick="alert('form')">FORM

<div onclick="alert('div')">DIV

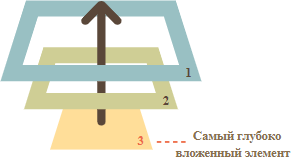
<p onclick="alert('p')">P</p>

</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

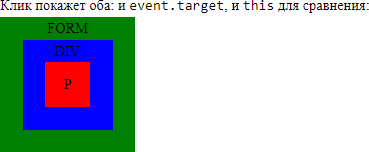
Всегда можно узнать, на каком конкретно элементе произошло событие. Самый глубокий элемент, который вызывает событие, называется целевым элементом, и он доступен через event.target. Отличия от this (=event.currentTarget):

* 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.



// 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

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.

1. Аналитика не будет работать над областью, где клики прекращаются stopPropagation, получилась «мёртвая зона».

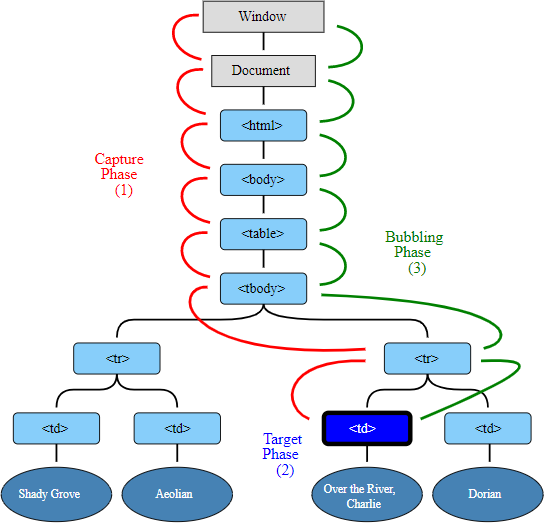
# Погружение

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 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));

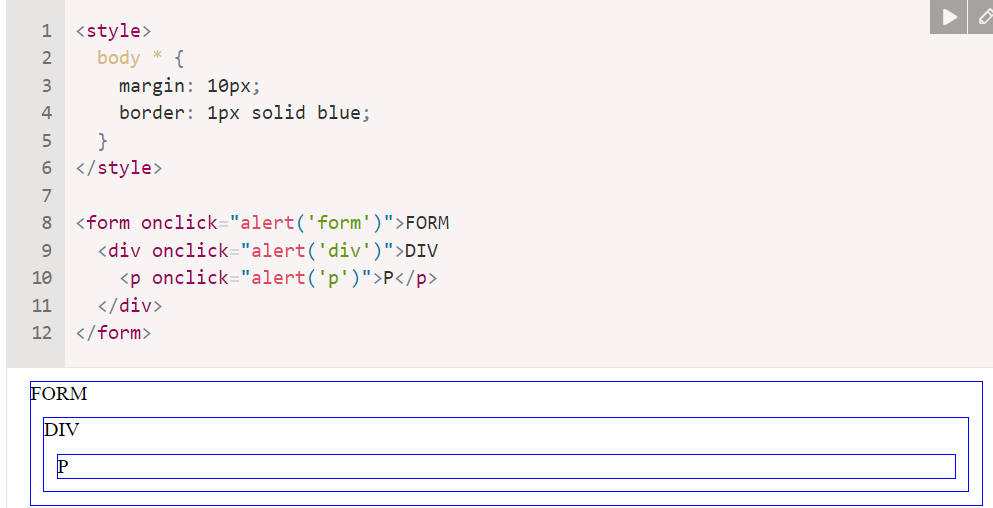
1. Всплытие и погружение событий.

**[Всплытие](https://learn.javascript.ru/bubbling-and-capturing" \l "vsplytie)**

Принцип всплытия очень простой.

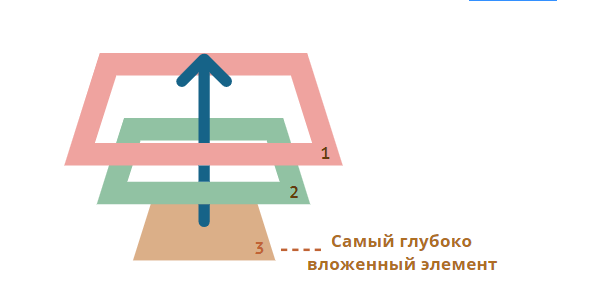
**Когда на элементе происходит событие, обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.**

Например, есть 3 вложенных элемента FORM > DIV > P с обработчиком на каждом:



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 мы увидим 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.

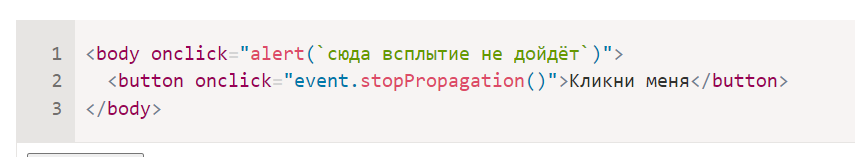
**[event.target](https://learn.javascript.ru/bubbling-and-capturing" \l "event-target)**

Всегда можно узнать, на каком конкретно элементе произошло событие.

**Самый глубокий элемент, который вызывает событие, называется *целевым* элементом, и он доступен через event.target.**

* event.target будет содержать ссылку на конкретный элемент внутри формы, на котором произошёл клик.

* **[Прекращение всплытия](https://learn.javascript.ru/bubbling-and-capturing" \l "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:



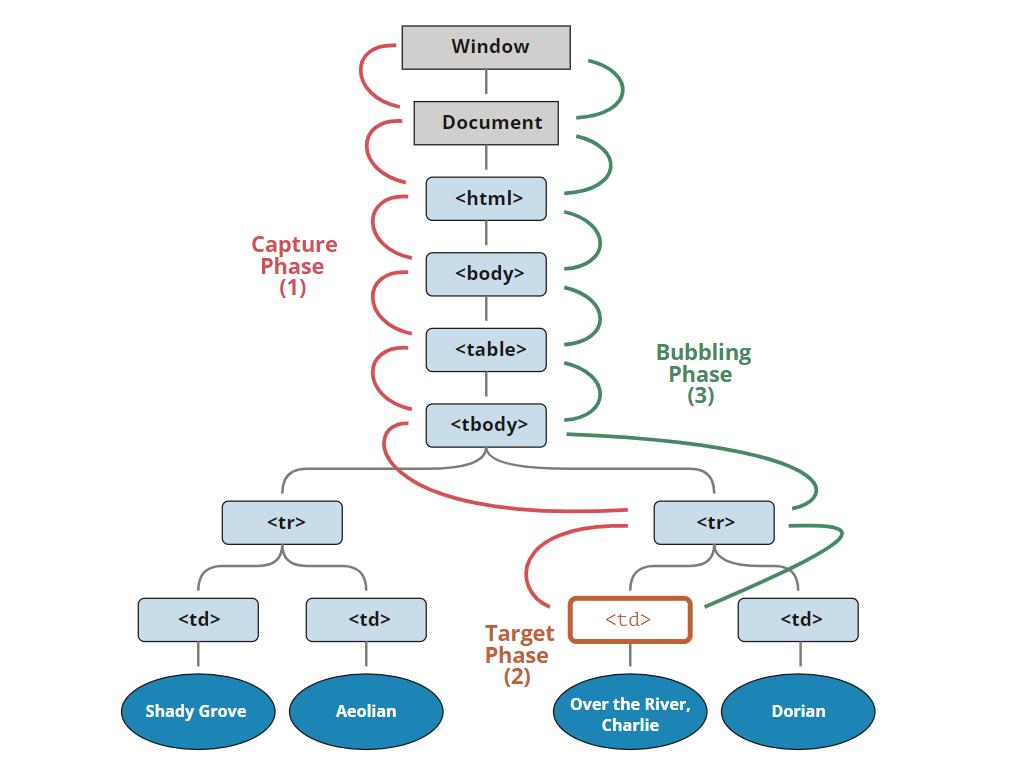
## [Погружение](https://learn.javascript.ru/bubbling-and-capturing" \l "pogruzhenie)

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](https://www.w3.org/TR/DOM-Level-3-Events/) описывает 3 фазы прохода события:

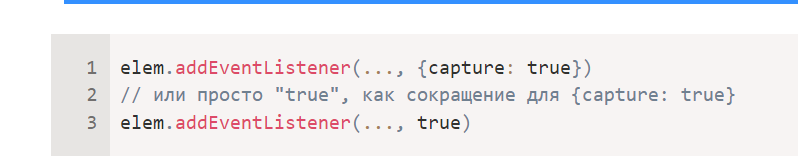
1. Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.
2. Фаза цели (target phase) – событие достигло целевого(исходного) элемента.
3. Фаза всплытия (bubbling stage) – событие начинает всплывать.

Картинка из спецификации демонстрирует, как это работает при клике по ячейке <td>, расположенной внутри таблицы:

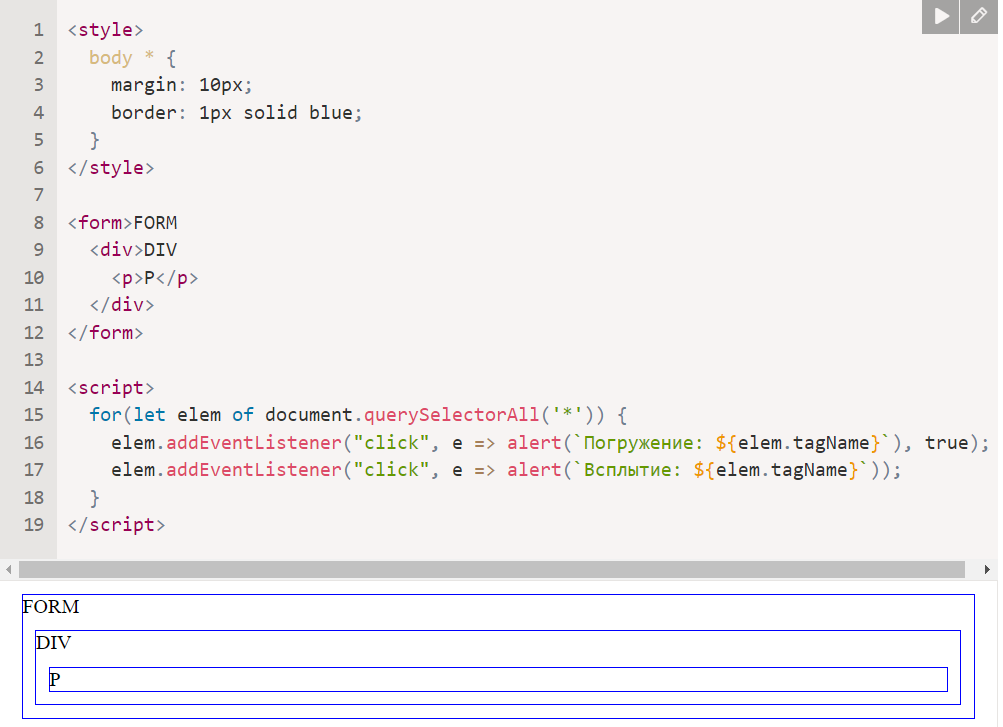


То есть при клике на <td> событие путешествует по цепочке родителей сначала вниз к элементу (погружается), затем оно достигает целевой элемент (фаза цели), а потом идёт наверх (всплытие), вызывая по пути обработчики.

тобы поймать событие на стадии погружения, нужно использовать третий аргумент capture вот так:



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
* 

Здесь обработчики навешиваются на *каждый* элемент в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события.

Если вы кликните 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, мы обычно и так знаем об этом в обработчике.

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>:

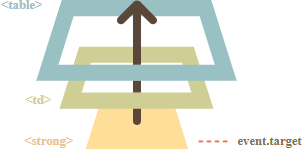
<td>

<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»

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.

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

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>

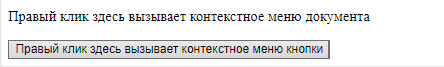
elem.oncontextmenu = function(event) { event.preventDefault(); alert("Контекстное меню кнопки");

};

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>

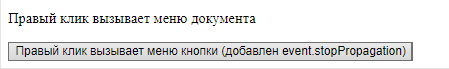
elem.oncontextmenu = function(event) { event.preventDefault(); event.stopPropagation(); alert("Контекстное меню кнопки");

};

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.
2. 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

Можно разделить события мыши на две категории: простые и комплексные.

Простые события (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:

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

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

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.

Рассмотрим подробнее события, возникающие при движении указателя (курсора) мыши над элементами страницы.

# 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.

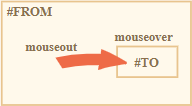
Свойство relatedTarget может быть null, это означает, что указатель мыши перешёл не с другого элемента, а из-за пределов окна браузера. Или, наоборот, ушёл за пределы окна.

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

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.

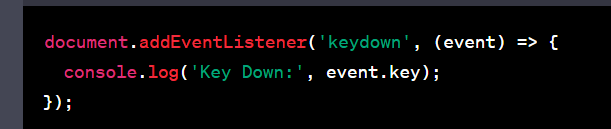
1. События клавиатуры: keyup, keydown. Прокрутка: событие scroll.

События клавиатуры и событие прокрутки являются важными взаимодействиями веб-страницы. В JavaScript доступны события keyup, keydown для клавиатуры и событие scroll для прокрутки.

События клавиатуры:

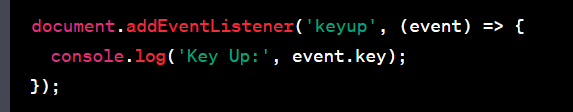
keydown: Событие keydown возникает, когда клавиша на клавиатуре нажата. Это событие происходит единожды, когда клавиша нажата вниз.

Пример использования:



keyup: Событие keyup возникает, когда клавиша на клавиатуре отпущена после нажатия. Это событие происходит единожды, когда клавиша отпущена.

Пример использования:

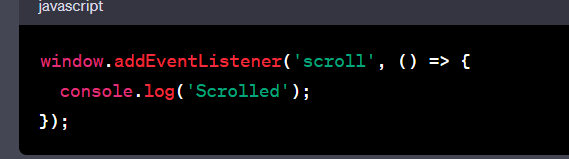


При обработке событий клавиатуры вы можете использовать свойство event.key, чтобы получить информацию о нажатой клавише.

Событие прокрутки:

scroll: Событие scroll возникает, когда пользователь прокручивает содержимое элемента, например, при прокрутке страницы. Вы можете использовать это событие для выполнения определенных действий, когда происходит прокрутка.

Пример использования:



Обратите внимание, что событие scroll обычно назначается на объект window для отслеживания прокрутки всей страницы. Однако, вы также можете назначить событие scroll на конкретные элементы, чтобы отслеживать прокрутку только внутри этих элементов.

При обработке событий прокрутки вы можете использовать свойства, такие как window.scrollX, window.scrollY, чтобы получить текущие значения горизонтальной и вертикальной прокрутки соответственно.

Используя эти события, вы можете реагировать на действия пользователя, связанные с клавиатурой и прокруткой, и выполнять определенные действия или обновлять интерфейс в соответствии с этими действиями.

1. События жизненного цикла HTML-страницы.

Жизненный цикл HTML-страницы состоит из нескольких событий, которые происходят при ее загрузке, изменении и выгрузке. Вот основные события жизненного цикла HTML-страницы:

**DOMContentLoaded:**

Событие DOMContentLoaded происходит, когда браузер полностью загрузил HTML-структуру документа (DOM), и все его внешние ресурсы (такие как стили CSS, изображения и скрипты) еще могут быть в процессе загрузки.

Это событие позволяет выполнять JavaScript-код, который зависит от полной доступности DOM-дерева.

**load:**

Событие load срабатывает, когда весь контент страницы, включая внешние ресурсы (стили, изображения, скрипты), был полностью загружен.

Это событие можно использовать для выполнения определенных задач, которые требуют полной загрузки контента страницы.

**beforeunload:**

Событие beforeunload возникает перед тем, как страница будет выгружена или закрыта.

Это событие дает возможность выполнить дополнительные действия перед тем, как пользователь покинет страницу, например, показать предупреждающее сообщение или запросить подтверждение выхода.

**unload:**

Событие unload возникает в момент выгрузки страницы, когда пользователь покидает страницу или переходит на другой сайт.

Это событие может использоваться для выполнения дополнительных операций перед закрытием страницы, таких как очистка ресурсов или отправка данных на сервер.

Эти события позволяют веб-разработчикам реагировать на различные этапы жизненного цикла HTML-страницы и выполнять соответствующие действия в зависимости от текущего состояния страницы.

**resize:**

Событие resize возникает, когда пользователь изменяет размер окна браузера.

Это событие позволяет реагировать на изменение размера окна и выполнить соответствующие действия, например, перераспределить элементы на странице для лучшей адаптивности.

**scroll:**

Событие scroll возникает, когда пользователь прокручивает содержимое страницы.

Это событие позволяет отслеживать прокрутку страницы и реагировать на нее, например, для создания эффектов параллакса или подгрузки дополнительного контента при достижении конца страницы.

**focus и blur:**

Событие focus происходит, когда элемент получает фокус ввода (например, при клике на текстовое поле).

Событие blur возникает, когда элемент теряет фокус ввода (например, при переключении на другой элемент).

Эти события позволяют реагировать на изменение фокуса и выполнять дополнительные действия, например, валидацию данных или отображение подсказок.

**submit:**

Событие submit возникает, когда пользователь отправляет форму на странице.

Это событие позволяет перехватить отправку формы, выполнить дополнительные действия, например, валидацию данных, и предотвратить ее отправку, если необходимо.

Это лишь некоторые из ключевых событий, которые могут возникать в жизненном цикле HTML-страницы. Каждое событие предоставляет возможность реагировать на определенное действие или состояние страницы, и разработчики могут использовать их для создания интерактивных и отзывчивых веб-приложений.

1. Загрузка скриптов, ресурсов.
2. 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

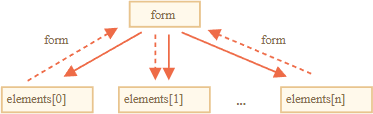
// form.elements обновили свои имена: alert(form.elements.login); // undefined alert(form.elements.username); // input

// 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

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

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 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.

окусировка элементов формы в JavaScript позволяет управлять фокусом ввода на различных элементах формы, таких как текстовые поля, кнопки, чекбоксы и другие. Когда элемент фокусируется, пользователь может сразу начать вводить данные в него без необходимости щелкать мышью.

focus(): Метод focus() используется для установки фокуса на элементе формы. Вызов метода focus() на элементе делает его активным для ввода.

blur(): Метод blur() используется для снятия фокуса с элемента формы. Вызов метода blur() на элементе делает его неактивным для ввода.

Пример использования:



1. Изменение значений элемента формы. Формы: отправка, событие и метод submit.

Рассмотрим различные события, сопутствующие обновлению данных.

# Событие: 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>

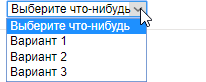
<option value="1">Вариант 1</option>

<option value="2">Вариант 2</option>

<option value="3">Вариант 3</option>

</select>





# Событие: input

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>

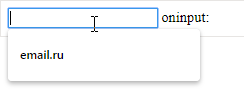
<script>

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

Эти события происходят при вырезании/копировании/вставке данных. Они относятся к классу ClipboardEvent 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">

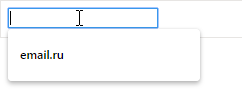
<script>

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 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.

1. JS-библиотека React. Понятие иммутабельности и согласования.
2. Расширение языка JavaScript – JSX. Рендеринг элементов.

Расширение JSX в языке JavaScript позволяет объединять код JavaScript с разметкой, представленной в виде XML-подобного синтаксиса. JSX обычно используется вместе с библиотекой React для создания пользовательских интерфейсов.

JSX позволяет разработчикам описывать структуру пользовательского интерфейса, создавая элементы, которые представляют собой комбинацию HTML-подобных тегов и JavaScript-кода. Элементы могут содержать атрибуты и дочерние элементы.

Пример JSX-элемента, представляющего кнопку, может выглядеть следующим образом:

const button = <button className="btn" onClick={handleClick}>Click me!</button>;

В приведенном примере <button> - это JSX-тег, className - это атрибут, который устанавливает CSS-класс, а onClick - это атрибут, определяющий обработчик события нажатия кнопки.

Для рендеринга элементов JSX на странице обычно используется метод ReactDOM.render(), который принимает JSX-элемент и контейнер, в котором следует отобразить этот элемент.

Пример использования ReactDOM.render():

|  |
| --- |
| import React from 'react';  import ReactDOM from 'react-dom';  const element = <h1>Hello, World!</h1>;  ReactDOM.render(element, document.getElementById('root')); |

В приведенном примере <h1> - это JSX-тег, который содержит текст "Hello, World!". Метод ReactDOM.render() используется для отображения этого элемента в корневом элементе с идентификатором 'root'.

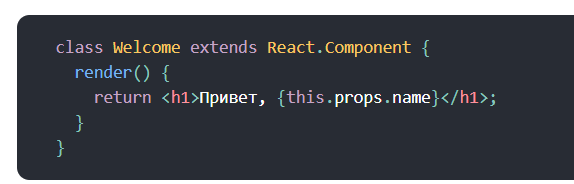
Таким образом, расширение JSX в языке JavaScript позволяет более удобным способом описывать пользовательский интерфейс и рендерить его на веб-странице с помощью метода ReactDOM.render().

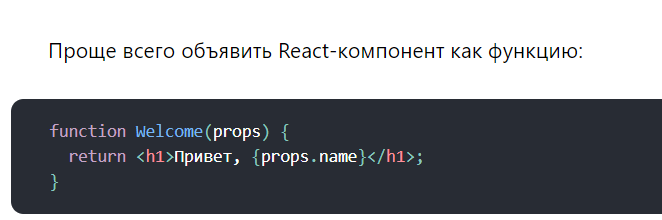
1. React: компоненты и пропсы.

В React компоненты являются основным строительным блоком при разработке приложений. Компоненты позволяют создавать переиспользуемые и модульные элементы интерфейса, которые могут содержать логику и данные.

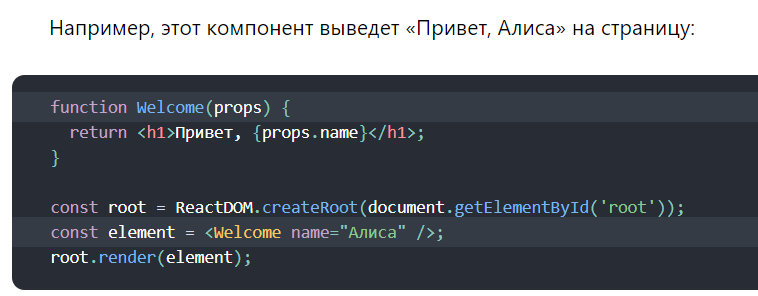
Компоненты в React могут быть классовыми или функциональными. Вот примеры обоих типов компонентов:

Классовый компонент:





Эта функция — компонент, потому что она получает данные в одном объекте («пропсы») в качестве параметра и возвращает React-элемент. Мы будем называть такие компоненты «функциональными», так как они буквально являются функциями.



Давайте разберём, что именно здесь происходит:

1. Мы вызываем root.render() c React-элементом <Welcome name="Алиса" />.
2. React вызывает наш компонент Welcome с пропсами {name: 'Алиса'}.
3. Наш компонент Welcome возвращает элемент <h1>Привет, Алиса</h1> в качестве результата.
4. React DOM делает минимальные изменения в DOM, чтобы получилось <h1>Привет, Алиса</h1>.

Компоненты в React могут принимать входные параметры, которые называются "пропсы" (props). Пропсы представляют собой объект, содержащий данные, передаваемые в компонент извне. Пропсы могут быть использованы внутри компонента для динамического отображения данных или настройки его поведения.

Пропсы являются неизменяемыми и только для чтения внутри компонента. Они могут быть использованы для передачи данных, функций обратного вызова или других компонентов.

Компоненты и пропсы в React позволяют создавать модульные и гибкие приложения, которые могут легко масштабироваться и переиспользоваться.

1. React: состояние и жизненный цикл: монтирование, обновление, размонтирование.

В React состояние (state) и жизненный цикл компонента взаимосвязаны. Состояние представляет данные, которые используются компонентом, а жизненный цикл определяет различные этапы, через которые проходит компонент при его создании, обновлении и удалении. Вот общий обзор жизненного цикла компонента React:

**Монтирование (Mounting):**

constructor: Это первый метод, вызываемый при создании компонента. Здесь инициализируются состояние и связанные данные.

static getDerivedStateFromProps: Этот метод вызывается перед render и используется для обновления состояния компонента на основе новых свойств (props).

render: Метод, возвращающий виртуальное представление компонента в виде React-элементов или компонентов.

componentDidMount: Этот метод вызывается сразу после того, как компонент был добавлен в DOM. Здесь можно выполнять операции, требующие доступа к DOM или инициализации внешних библиотек.

**Обновление (Updating):**

static getDerivedStateFromProps: При обновлении свойств (props) компонента этот метод может быть вызван для обновления состояния на основе новых свойств.

shouldComponentUpdate: Этот метод позволяет определить, должен ли компонент обновиться и перерисоваться на основе новых свойств или состояния. Возвращение false предотвратит обновление компонента.

render: Обновленное виртуальное представление компонента будет возвращено методом render.

componentDidUpdate: Вызывается после обновления компонента и обновления DOM. Здесь можно выполнять операции, которые требуют доступа к обновленным свойствам или DOM.

**Размонтирование (Unmounting):**

componentWillUnmount: Этот метод вызывается перед удалением компонента из DOM. Здесь можно освободить ресурсы, отменить подписки или очистить таймеры.

Кроме перечисленных методов, в React 16.3+ появились новые методы жизненного цикла, такие как getSnapshotBeforeUpdate, componentDidCatch и другие, которые предоставляют более гранулярный контроль над различными этапами жизненного цикла компонента.

Жизненный цикл компонента React позволяет разработчикам управлять состоянием и поведением компонента на различных этапах его жизни, что делает его мощным инструментом для создания интерактивных пользовательских интерфейсов.

1. React: обработка событий.
2. React: условный рендеринг, рендеринг списка, ключи.

В React существует несколько понятий, связанных с рендерингом, которые помогают создавать динамические интерфейсы: условный рендеринг, рендеринг списка и использование ключей.

Условный рендеринг:

Условный рендеринг в React позволяет отображать или скрывать компоненты в зависимости от определенных условий. Для этого обычно используются конструкции JavaScript, такие как условные операторы (if, else) или тернарный оператор (condition ? true : false).

Пример условного рендеринга с использованием тернарного оператора:

|  |
| --- |
| function Greeting({ isLoggedIn }) {  return (  <div>  {isLoggedIn ? <UserGreeting /> : <GuestGreeting />}  </div>  );  } |

В приведенном примере компонент Greeting рендерит компонент UserGreeting, если isLoggedIn равно true, или компонент GuestGreeting, если isLoggedIn равно false.

Рендеринг списка:

Рендеринг списка в React позволяет создавать элементы списка на основе массива данных. Для этого обычно используется метод map(), который преобразует каждый элемент массива в JSX-элемент.

Пример рендеринга списка:

|  |
| --- |
| function TodoList({ todos }) {  return (  <ul>  {todos.map(todo => (  <li key={todo.id}>{todo.text}</li>  ))}  </ul>  );  } |

В приведенном примере компонент TodoList принимает массив todos и создает список <ul> с элементами <li>. Каждый элемент списка получается из массива todos с помощью метода map(). Важно указать уникальный ключ (key) для каждого элемента списка, чтобы React мог эффективно обновлять только измененные элементы.

Ключи (keys):

Ключи (keys) являются особым атрибутом, который следует добавлять к элементам списка при их рендеринге. Ключи помогают React идентифицировать каждый элемент списка и эффективно обновлять только измененные элементы при изменении данных.

Ключи должны быть уникальными в пределах списка и обычно используются идентификаторы элементов из данных. Ключи не передаются в компоненты как пропсы (props), они используются React для оптимизации процесса обновления.

В приведенном ранее примере рендеринга списка элементам <li> добавлен атрибут key={todo.id}, где todo.id - это уникальный идентификатор каждого элемента списка.

Использование ключей позволяет React эффективно обновлять только измененные элементы списка, а не перерисовывать весь список при каждом изменении данных. Это особенно важно при работе с большими списками или при изменении порядка элементов в списке.

Важно отметить, что ключи должны быть уникальными только в пределах одного списка, но они не требуют глобальной уникальности в приложении.

Вот основные понятия, связанные с рендерингом в React: условный рендеринг, рендеринг списка и использование ключей. Эти концепции помогают создавать динамические и эффективные интерфейсы в React-приложениях.

1. React: формы.

В HTML элементы формы, такие как <input>, <textarea> и <select>, обычно сами управляют своим состоянием и обновляют его когда пользователь вводит данные. В React мутабельное состояние обычно содержится в свойстве компонентов state и обновляется только через вызов [setState()](https://ru.legacy.reactjs.org/docs/react-component.html" \l "setstate)

Управляемые компоненты:

В React мы строим формы с помощью управляемых компонентов. Управляемый компонент связывает состояние (state) с элементами формы и обновляет состояние при изменении пользовательского ввода.

Создание состояния формы:

Мы используем хук useState или классовые компоненты с состоянием для создания состояния формы. Состояние содержит значения полей формы и может быть обновлено при вводе данных пользователем.

Обработчики событий:

Мы создаем обработчики событий, которые реагируют на изменения состояния формы при вводе данных пользователем. Обработчики событий обновляют состояние формы с новыми значениями.

Обработка отправки формы:

При отправке формы мы предотвращаем стандартное поведение (перезагрузку страницы) с помощью метода event.preventDefault(). Затем мы можем выполнить дополнительную логику обработки данных формы, например, отправить их на сервер или выполнить действия на основе введенных данных.

В React <textarea> использует атрибут value. Таким образом, форму с <textarea> можно написать почти тем же способом, что и форму с однострочным <input>:

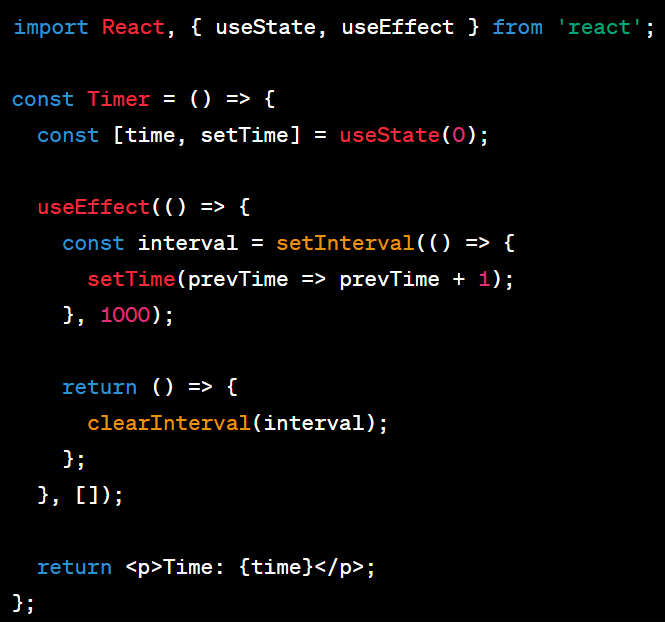


1. React: хуки, правила хуков.

Хук (hook) в React - это функция, которая позволяет использовать состояние (state) и другие возможности React в функциональных компонентах. Он предоставляет способ добавления состояния и побочных эффектов в функциональные компоненты без необходимости создания классовых компонентов. Примеры хуков в React:



В этом примере хук useState используется для добавления состояния (count) в функциональный компонент Counter. Мы можем обновлять состояние, вызывая функцию setCount.



Хук useEffect используется для добавления побочных эффектов в функциональный компонент Timer. В этом примере мы запускаем таймер, который увеличивает состояние time каждую секунду. Также мы используем функцию clearInterval для очистки интервала после размонтирования компонента.

В React хуки (hooks) представляют собой функции, которые позволяют вам использовать состояние и другие функциональные возможности React в функциональных компонентах без необходимости создания классовых компонентов. Хуки были представлены в React 16.8 и стали популярным способом работы с состоянием и эффектами в функциональных компонентах. Вот некоторые правила использования хуков в React:

Использование хуков только внутри функциональных компонентов:

Хуки могут использоваться только внутри тела функционального компонента или внутри другого хука.

Нельзя использовать хуки в классовых компонентах или обычных JavaScript-функциях.

Хуки следует вызывать только на верхнем уровне функционального компонента:

Хуки должны быть вызваны на верхнем уровне функционального компонента, не внутри циклов, условий или вложенных функций.

Это гарантирует, что хуки всегда вызываются в одном и том же порядке и сохраняют состояние между перерисовками компонента.

Названия хуков должны быть вызваны согласно правилам:

Названия хуков должны начинаться с префикса "use", например, useState, useEffect и т.д.

Это позволяет React определить, что функция является хуком и обеспечить правильное поведение.

Хуки должны вызываться в одном порядке при каждом рендеринге:

Порядок вызова хуков должен быть одинаковым при каждом рендеринге компонента.

Это обеспечивает правильное соответствие между вызовами хуков и их внутренним состоянием.

Хуки могут вызываться условно, но не внутри циклов и вложенных функций:

Вызов хуков внутри условных выражений является допустимым, но они должны быть вызваны на верхнем уровне функционального компонента, чтобы сохранить правильный порядок вызовов хуков.

Создание собственных хуков:

Вы также можете создавать собственные хуки, которые объединяют логику состояния и эффектов, чтобы их можно было повторно использовать в разных компонентах.

Следуя этим правилам, вы сможете эффективно использовать хуки в функциональных компонентах React и получить преимущества их простоты и удобства.

1. React: использование хука состояния.
2. React: использование хука эффекта.

Хук эффекта (useEffect) в React позволяет выполнять побочные эффекты в функциональных компонентах. Это может включать в себя взаимодействие с внешними источниками данных, подписку на события, выполнение асинхронных операций и другие сценарии, связанные с изменением состояния компонента.

Хук useEffect запускает функцию-эффект после каждого рендеринга компонента, и может быть использован для выполнения задач в момент монтирования, обновления или размонтирования компонента.

Вам скорее всего доводилось ранее запрашивать данные, делать подписки или вручную менять DOM из React-компонента. Мы расцениваем эти операции как «побочные эффекты» (или сокращённо «эффекты»), так как они могут влиять на работу других компонентов и их нельзя выполнить во время рендера.

С помощью хука эффекта useEffect вы можете выполнять побочные эффекты из функционального компонента. Он выполняет ту же роль, что и componentDidMount, componentDidUpdate и componentWillUnmount в React-классах, объединив их в единый API. Вы можете найти сравнение useEffect и этих методов на странице [использование хука эффекта](https://ru.legacy.reactjs.org/docs/hooks-effect.html).

К примеру, этот компонент устанавливает заголовок документа после того, как React обновляет DOM:



Когда вы вызываете useEffect, React получает указание запустить вашу функцию с «эффектом» после того, как он отправил изменения в DOM. Поскольку эффекты объявляются внутри компонента, у них есть доступ к его пропсам и состоянию. По умолчанию, React запускает эффекты после каждого рендера, включая первый рендер. Мы рассмотрим более подробно, как это отличается от классовых методов жизненного цикла на странице [использование хука эффекта](https://ru.legacy.reactjs.org/docs/hooks-effect.html).

При необходимости вы можете вернуть из эффекта функцию, которая указывает эффекту, как выполнить за собой «сброс». Например, этот компонент использует эффект, чтобы подписаться на статус друга в сети, и выполняет сброс, отписываясь от него.



В этом примере, React будет отписываться от нашего ChatAPI перед тем, как компонент размонтируется и перед тем, как перезапустить эффект при повторном рендере. Вы можете сделать так, чтобы [React пропускал повторные подписки](https://ru.legacy.reactjs.org/docs/hooks-effect.html" \l "tip-optimizing-performance-by-skipping-effects) если props.friend.id, который мы передали в ChatAPI, остался без изменений.

1. Redux: назначение, основные концепции.

Redux - это популярная библиотека управления состоянием для JavaScript приложений, особенно для React. Ее основная цель - упростить управление и обновление состояния приложения, делая его предсказуемым и легко тестируемым. Вот основные концепции, связанные с Redux:

Однонаправленный поток данных: Redux использует однонаправленный поток данных, где данные в приложении движутся только в одном направлении. Действия (actions) инициируются компонентами, передаются в Redux, где они вызывают изменения в хранилище состояния. Затем обновленное состояние передается обратно компонентам для отображения.

Хранилище состояния (Store): Центральная концепция Redux - это хранилище состояния, которое содержит все данные приложения. Хранилище является неизменяемым и может быть изменено только с помощью действий. Вся логика обновления состояния находится в редьюсерах.

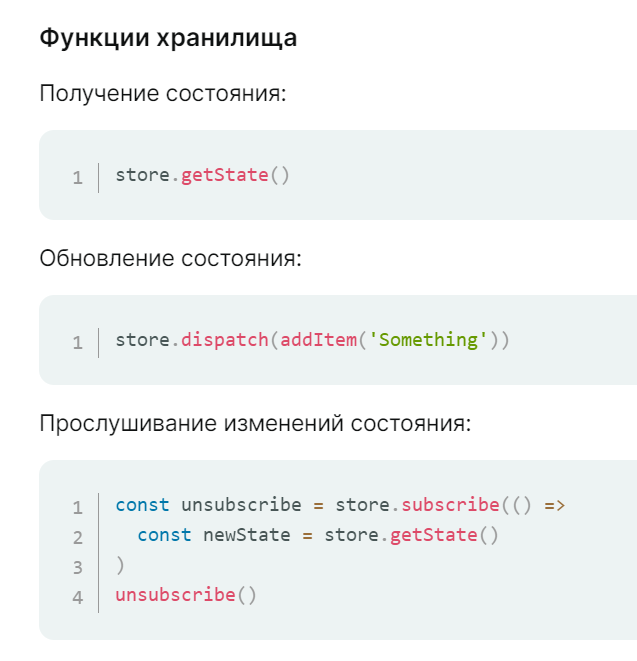
• содержит состояние приложения;

• отображает состояние через getState();

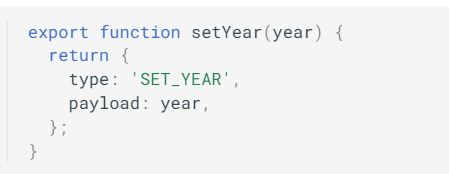
• может обновлять состояние через dispatch();

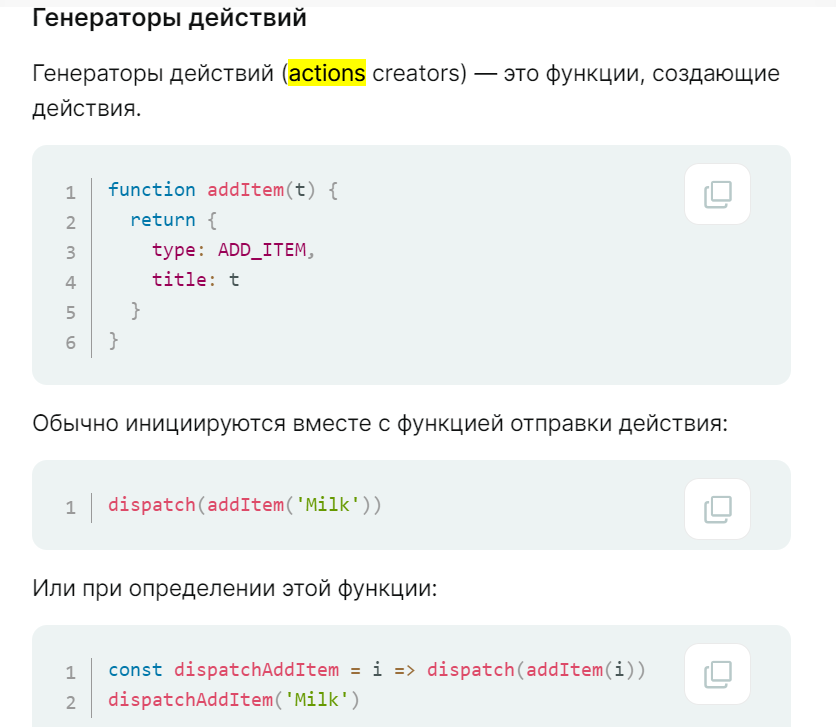
• позволяет регистрироваться (или удаляться) в качестве слушателя изменения состояния через subscribe().





Действия (Actions): Действия представляют собой объекты, которые описывают что-то произошло в приложении. Они содержат тип действия (action type) и опциональные данные, необходимые для обновления состояния. Действия создаются и инициируются компонентами, а затем передаются в Redux для обработки.





Редьюсеры (Reducers): Редьюсеры определяют, как состояние приложения изменяется в ответ на действия. Они являются чистыми функциями, которые принимают текущее состояние и действие, и возвращают новое состояние. Комбинация редьюсеров определяет форму и структуру хранилища.

Чего не должен делать редуктор

Редуктор — это всегда чистая функция, поэтому он не должен:

• мутировать аргументы;

• мутировать состояние. Вместо этого создаётся новое состояние с помощью Object.assign({}, ...);

• иметь побочные эффекты (никаких API-вызовов с какими-либо изменениями);

• вызывать нечистые функции. Это функции, результат которых зависит от чего-то кроме их аргументов (например, Date.now() или Math.random()).



Подписка на изменения состояния: Компоненты могут подписываться на изменения состояния из хранилища с помощью функции connect (в случае использования React и React Redux) или с помощью хука useSelector (с использованием React Hooks). Это позволяет компонентам быть в курсе изменений состояния и обновлять свое отображение при необходимости.

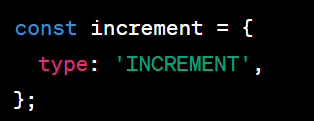
Диспетчеры (Dispatchers): Диспетчеры являются функциями, которые отправляют действия в хранилище. Компоненты вызывают диспетчеры для инициирования изменений состояния.

Миддлвары (Middlewares): Мид

1. Redux: экшены.

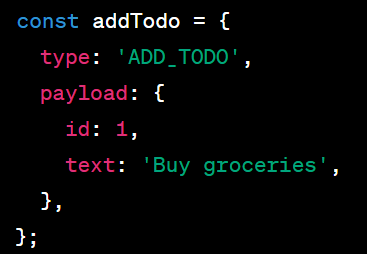
В Redux, экшены (actions) представляют собой объекты, которые описывают события или действия, происходящие в приложении. Они являются единственным источником информации для изменения состояния (state) в хранилище (store). Экшены обычно имеют тип (type) и дополнительные данные, необходимые для выполнения действия. Вот примеры экшенов в Redux:

Простой экшен без данных:



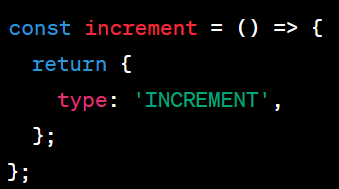
В этом примере экшен с типом "INCREMENT" описывает действие увеличения.

Экшен с данными:



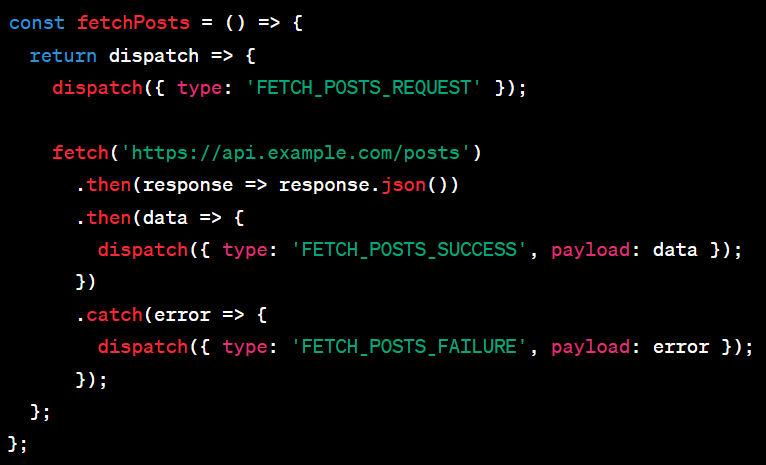
В этом примере экшен с типом "ADD\_TODO" содержит данные о задаче, которую нужно добавить.

Создание экшена с помощью функции (action creator):



Здесь экшен создается с помощью функции, называемой "action creator". Функция возвращает объект экшена.

Асинхронный экшен с использованием middleware (например, redux-thunk или redux-saga):



В этом примере экшен "fetchPosts" выполняет асинхронный запрос к API и диспатчит различные экшены в зависимости от результата запроса.

Экшены в Redux служат для уведомления хранилища о необходимости изменить состояние. Они передаются в функцию "dispatch" и обрабатываются редьюсерами (reducers), которые определяют, как состояние должно измениться в ответ на каждый экшен.

1. Redux: редьюсеры.
2. Redux: стор.

В Redux, стор (store) представляет собой центральное хранилище данных для приложения. Он содержит состояние приложения и предоставляет методы для изменения этого состояния. Redux основан на концепции однонаправленного потока данных, где все изменения состояния происходят через действия (actions) и обрабатываются с помощью редьюсеров (reducers).

Структура и функциональность стора в Redux включает в себя следующие основные элементы:

Состояние (State):

Состояние представляет собой иммутабельный объект, который содержит данные вашего приложения. Оно может быть любого вида, от примитивных типов данных до сложных объектов и массивов. Состояние является единственным источником истины в Redux.

Действия (Actions):

Действия представляют собой объекты, которые описывают события или команды, которые должны быть выполнены в приложении. Они передают информацию о том, что произошло в приложении и какие изменения необходимо внести в состояние. Действия обычно имеют тип (type) и дополнительные данные (payload).

Редьюсеры (Reducers):

Редьюсеры являются чистыми функциями, которые принимают предыдущее состояние и действие, и возвращают новое состояние. Они определяют, какие изменения должны быть выполнены в состоянии на основе полученных действий. Каждый редьюсер обрабатывает определенный срез состояния и может быть объединен с помощью функции combineReducers для создания корневого редьюсера.

Хранилище (Store):

Хранилище (store) представляет собой объект, который объединяет состояние, действия и редьюсеры. Оно предоставляет методы для доступа к состоянию, отправки действий и регистрации слушателей. Хранилище является единственным в приложении и позволяет обновлять состояние только через действия.

Пример создания стора в Redux:

|  |
| --- |
| import { createStore } from 'redux';  import rootReducer from './reducers';  const store = createStore(rootReducer); |

В приведенном примере createStore - это функция из Redux, которая создает новый стор на основе корневого редьюсера (rootReducer), объединяющего все редьюсеры в приложении.

После создания стора, вы можете получить доступ к состоянию с помощью метода getState(), отправлять действия с помощью метода dispatch(action) и подписываться на изменения состояния с помощью метода subscribe(listener).

Стор является центральным элементом в Redux, который помогает управлять и обновлять состояние приложения.

1. Redux: поток данных.

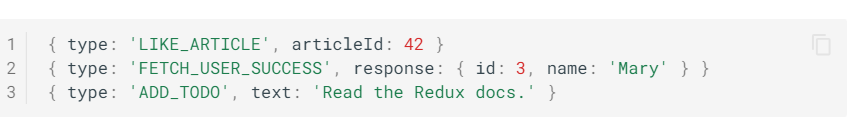
Архитектура Redux вращается вокруг **строго однонаправленного потока данных** (Data Flow).

Это значит, что все данные в приложении следуют одному паттерну жизненного цикла, делая логику вашего приложения более предсказуемой и легкой для понимания. Также это способствует большей упорядоченности данных (data normalization), так что в конечном итоге у вас не будет нескольких изолированных копий одних и тех же данных, которые ничего не знают друг о друге.

Жизненный цикл данных в любом Redux-приложении включает в себя 4 шага:

## 1. Вы вызываете [store.dispatch(action)](https://reactdev.ru/libs/redux/api/Store/" \l "dispatch)[¶](https://reactdev.ru/libs/redux/basics/DataFlow/#1-storedispatchaction)

Экшен — это простой javascript-объект, который описывает что случилось. Например:



Вы можете вызвать [store.dispatch(action)](https://reactdev.ru/libs/redux/api/Store/" \l "dispatch) из любого места Вашего приложения, включая компоненты и XHR-колбеки или даже с запланированными интервалами.

## 2. Redux-стор вызывает функцию-редьюсер, который вы ему передали[¶](https://reactdev.ru/libs/redux/basics/DataFlow/#2-redux-)

Стор передаст два аргумента при вызове редьюсера: текущее дерево состояния (current state tree) и экшен (action). Например, в todo-приложении главный редьюсер может принимать что-то такое:

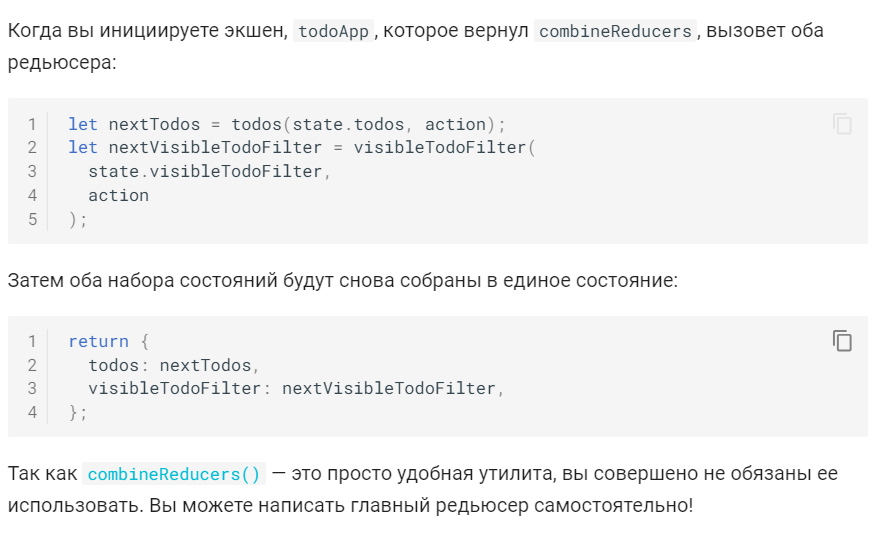


## 3. Главный редьюсер может комбинировать результат работы нескольких редьюсеров в единственное дерево состояния приложения[¶](https://reactdev.ru/libs/redux/basics/DataFlow/#3)

Каким образом вы будете структурировать главный редьюсер, зависит только от Вас. Redux поставляется с хелпером [combineReducers()](https://reactdev.ru/libs/redux/api/combineReducers/), полезным для "разделения" главного редьюсера на отдельные функции, которые управляют отдельными ветвями дерева состояния.

[combineReducers()](https://reactdev.ru/libs/redux/api/combineReducers/) работает следующим образом. Допустим, у вас есть два редьюсера: один для списка todo-дел, второй — для выбранного сейчас режима отображения этого списка:





## 4. Redux-стор сохраняет полное дерево состояния, которое возвращает главный редьюсер[¶](https://reactdev.ru/libs/redux/basics/DataFlow/" \l "4-redux-" \o "Permanent link)

Это новое дерево является следующим состоянием Вашего приложения! Каждый слушатель, зарегистрированный с помощью [store.subscribe(listener)](https://reactdev.ru/libs/redux/api/Store/" \l "subscribe), будет вызван. Слушатели могут вызывать [store.getState()](https://reactdev.ru/libs/redux/api/Store/" \l "getState) для получения текущего состояния приложения.

Теперь UI может быть обновлен для отражения нового состояния приложения. Если вы используете такие биндинги (bindings), как [React Redux](https://github.com/gaearon/react-redux), то это та точка, в которой стоит вызвать component.setState(newState)

Этот однонаправленный поток данных позволяет создавать предсказуемые и легко тестируемые приложения, управление состоянием которых становится более простым и понятным.

1. Redux: использование с React (react-redux).

Для использования Redux с React рекомендуется использовать библиотеку react-redux, которая облегчает интеграцию Redux в приложения React. Она предоставляет удобные инструменты для связывания состояния из хранилища Redux с компонентами React и управления экшенами. Вот примеры использования react-redux с React и Redux:

Установка зависимостей:

Установите react-redux, если его еще нет, вместе с redux:

shell

Copy code

npm install react-redux redux

Создание хранилища Redux:

Создайте файл store.js, где будет создано хранилище Redux с помощью функции createStore из библиотеки Redux. В хранилище должны быть указаны редьюсеры (reducers), а также возможно использование middleware и дополнительных настроек.

javascript

Copy code

import { createStore } from 'redux';

import rootReducer from './reducers';

const store = createStore(rootReducer);

export default store;

Создание редьюсеров:

Создайте файлы с редьюсерами, которые будут определять, как изменяется состояние приложения в ответ на экшены. Обычно редьюсеры объединяются в один корневой редьюсер с помощью combineReducers из библиотеки Redux.

javascript

Copy code

// exampleReducer.js

const initialState = {

count: 0,

};

const exampleReducer = (state = initialState, action) => {

switch (action.type) {

case 'INCREMENT':

return { ...state, count: state.count + 1 };

case 'DECREMENT':

return { ...state, count: state.count - 1 };

default:

return state;

}

};

export default exampleReducer;

// rootReducer.js

import { combineReducers } from 'redux';

import exampleReducer from './exampleReducer';

const rootReducer = combineReducers({

example: exampleReducer,

// другие редьюсеры...

});

export default rootReducer;

Подключение хранилища Redux к приложению:

В обертке вашего приложения, обычно в корневом файле (например, index.js), используйте компонент Provider из react-redux, чтобы обернуть ваше приложение и передать ему хранилище Redux.

javascript

Copy code

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

import store from './store';

import App from './App';

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

Использование состояния и действий в компонентах:

В компонентах React вы можете использовать хук useSelector из react-redux для получения состояния из хранилища и хук useDispatch для отправки экшенов.

javascript

Copy code

import React from 'react';

import { useSelector, useDispatch } from 'react-redux';

const Counter = () => {

const count = useSelector(state => state.example.count);

const dispatch = useDispatch();

const increment = () => {

dispatch({ type: 'INCREMENT' });

};

const decrement = () => {

dispatch({ type: 'DECREMENT' });

};

return (

<div>

<p>Count: {count}</p>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

};

export default Counter;

В этом примере компонент Counter использует хук useSelector для получения значения count из состояния хранилища, а также хук useDispatch для отправки экшенов с помощью функции dispatch.

Таким образом, с помощью react-redux вы можете связать состояние и действия в Redux с компонентами React и легко управлять ими.

Заведующий кафедрой Преподаватель

Дата утверждения 17 апреля 2023 протокол № 10