Вопросы:

1. Переменные. Имена переменных. Константы. (Т1.4)

Для создания переменной в JavaScript, используйте ключевое слово **let**. В старых скриптах можно найти ключевое слово: var вместо let. Областью видимости переменных, объявленных ключевым словом let, является блок, в котором они объявлены, и все его подблоки. Область видимости переменных var ограничивается либо функцией (var становится локальной переменной в этой функции), либо, если переменная *глобальная*, то скриптом. Такие переменные доступны за пределами блока (например, блок if, for и др.). Используя var, можно переобъявлять переменную сколько угодно раз.

В JavaScript есть два ограничения, касающиеся имён переменных:

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

- Имя переменной должно содержать только буквы, цифры или символы $ и \_

- Первый символ не должен быть цифрой.

- Если имя содержит несколько слов, обычно используется верблюжья нотация, например, myVeryLongName.

- Регистр имеет значение. apple and AppLE – это две разные переменные.

- Нелатинские буквы разрешены, но не рекомендуются.

- Существует список зарезервированных слов: let, class, return и function и др.

**const - переменная константа**. Изменение приведет к ошибке. Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта.

2. Типы данных. Оператор typeof. (Т1.5)

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число. 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 **7** основных типов данных в JavaScript, **5** из них называются «примитивными», потому что их значениями могут быть только простые значения.

**Число**

Числовой тип данных (**number**) представляет, *как целочисленные значения, так и числа с плавающей точкой*. Помимо обычных чисел существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity и NaN.

***Infinity*** представляет собой математическую бесконечность ∞.

***NaN*** означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции. (alert( "не число" / 2 ); // NaN)

***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. (let newMax = 9007199254740992n)

Математические операции в JavaScript позволяют делать что угодно: делить на ноль т.д. Скрипт не остановится с фатальной ошибкой (будет NaN).

**Строка**

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

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. **Обратные кавычки** имеют расширенный функционал: они позволяют встраивать выражения в строку, заключая их в ${…}. Выражение внутри ${…} вычисляется, и его результат становится частью строки.

**Булевый (логический) тип**

Булевый тип (**boolean**) может принимать только два значения: true (истина) и false (ложь). Такой тип, как правило, используется для хранения значений да/нет.

**Null**

Специальное значение null не относится ни к одному из типов, описанных выше. Оно формирует отдельный тип, который содержит только значение null. В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

**undefined**

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.

**Объекты и символы**

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

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

**Оператор 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.

Результатом вызова typeof null в строке 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 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.

3. Преобразование типов. (Т1.6)

**Строковое**

Можно использовать функцию *String(value)* чтобы преобразовать значение к строке*.*

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)

**Численное**

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

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

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

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. |

**Логическое**

*Boolean(value).* Правила преобразования

Значения, которые интуитивно «**пустые**» (как 0, пустой строки, null, undefined, и NaN), **становятся false, остальное true**.

Строка с нулём "0" и пробелом " "  преобразуется в true.

4. Операторы. Бинарные и унарные операторы. Приоритет операторов. Совмещение операторов. (Т1.7)

*Операнд* – то, к чему применяется оператор.

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

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

**Сложение строк, бинарный +**

- '+' складывают числа.

- '+' объединяет строки.

- строка + число = строка всегда

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

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.

**Преобразование к числу, унарный +**

если операнд не число, унарный плюс преобразует его в число. Это то же самое, что и Number(...), только короче.

**Присваивание = Остаток от деления % Возведение в степень \*\***

* a % b возвращает остаток от деления a на b.
* a = b значение справа к левому
* a\*\*b левое в степень правого

**Инкремент/декремент и форма**

*Инкремент* ++ увеличивает на 1, *Декремент* -- уменьшает

*постфиксная форма*: counter++, возвращает старое значение(до+)

*префиксная форма:* ++counter., возвращает новое значение.

**Побитовые операторы**

Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), на уровне их внутреннего двоичного представления.

- 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) ( >>> )

**Оператор запятая**

Оператор запятая предоставляет возможность вычислять несколько выражений, разделяя их запятой. Каждое выражение выполняется, но возвращается результат только последнего.   
let a = (1 + 2, 3 + 4);  
alert( a ); // 7

**Совмещенные операторы**

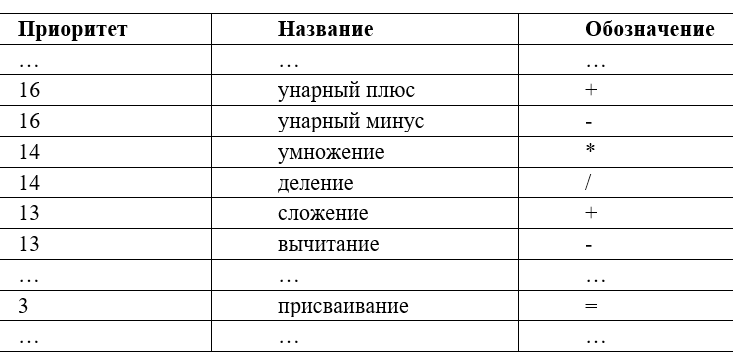
let n = 2;  
n += 5; // теперь n=7 (работает как n = n + 5)  
n \*= 2; // теперь n=14 (работает как n = n \* 2)

**Приоритет операторов**

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется приоритетом.

Умножение в выражении 2 \* 2 + 1 выполнится раньше сложения, 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. **У каждого оператора есть приоритет, чем выше, тем раньше выполнится, одинаковый, значит слева направо**

5. Операторы сравнения. Сравнение с null и undefined. (Т1.8)

Операторы сравнения известные из математики:

- Больше/меньше: a > b, a < b.

- Больше/меньше или равно: a >= b, a <= b.

- Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства =. Один знак равенства a = b означал бы присваивание.

- Не равно: a! = b.

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

- true – означает «да», «верно», «истина».

- false – означает «нет», «неверно», «ложь».

**Сравнение строк**

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

**Алгоритм сравнения двух строк довольно прост:**

1.  Сначала сравниваются первые символы строк.

2.  Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй. alert( 'Я' > 'А' ); // true

3.  Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.

4.  Сравнение продолжается, пока не закончится одна из строк.

5.  Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

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

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

При сравнении значений разных типов, **JS приводит каждое к числу**.

Логическое значение true становится 1, а false – 0.

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

let a = 0;  
alert( Boolean(a) ); // false  
let b = "0";  
alert( Boolean(b) ); // true  
alert(a == b); // true!

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

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

alert( 0 == false ); // true

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

всё потому, что операнды разных типов преобразуются оператором == к числу. В итоге, и пустая строка, и false становятся нулём.

**Оператор строгого равенства ===** проверяет равенство без приведения типов. если a и b имеют разные типы, то проверка a === b возвращает false   
alert( 0 === false ); // false, так как сравниваются разные типы

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

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

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

* При строгом равенстве === эти значения различны, так как различны их типы.    Alert (null === undefined); // false
* При нестрогом равенстве == эти значения равны друг другу и не равны другим значениям. Это специальное правило языка.

Alert (null == undefined); // true

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

alert( null > 0 );  // (1) false  
аlert( null == 0 ); // (2) false  
alert( null >= 0 ); // (3) true

Сравнения преобразуют null в число, рассматривая его как 0. Поэтому выражение (3) null >= 0 истинно, а null > 0 ложно.

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

**Значение undefined несравнимо с другими значениями**:

alert( undefined > 0 ); // false (1)  
alert( undefined < 0 ); // false (2)  
alert( undefined == 0 ); // false (3)

Сравнение undefined с нулём всегда ложно, по следующим причинам:

- Сравнения (1) и (2) возвращают false, потому что undefined преобразуется в NaN, а NaN – это специальное числовое значение, которое возвращает false при любых сравнениях.

- Нестрогое равенство (3) возвращает false, потому что undefined равно только null и ни чему больше.

6. Условные операторы: if, '?'. Конструкция switch. (Т1.10, Т1.13)

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

**Оператор if(...)** вычисляет условие в скобках и, если результат true, то выполняется блок кода. Оператор if может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно.

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

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок else if. Присутствие блока else не является обязательным.

**Условный оператор „?“**

Иногда нужно назначить переменную в зависимости от условия

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

**let result = условие ? значение1 : значение2;**

**let result = условие1 ? значение1 : условие2 ? значение1 : ...;**

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

(age < 18) ? 'Привет!' :

(age < 100) ? 'Здравствуйте!' :

'Какой необычный возраст!'; //если больше 100

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение 2. Во втором случае происходит последовательная проверка условий.

**switch** заменяет собой сразу несколько if. Конструкция switch имеет один или более блок case и необязательный блок default. Синтаксис:

switch(x) {

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

...

Break;

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

...

break;

default:

...

break;

}

Переменная x проверяется на строгое равенство первому значению value1, затем второму value2 и так далее. Если ни один case не совпал – выполняется (если есть) вариант default.

Чтобы завершить проход case нужен break. Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются.

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Маловато' );

case 4:

alert( 'В точку!' );

case 5:

alert( 'Перебор' );

default:

alert( "Нет таких значений" );

}

В примере выше последовательно выполнятся три alert:

alert( 'В точку!' );

alert( 'Перебор' );

alert( "Нет таких значений" );

7. Логические операторы. (Т1.11)

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)

result = a || b;

Существует всего четыре возможные логические комбинации:

alert( true || true );   // true  
alert( false || true );  // true  
alert( true || false );  // true  
**alert( false || false ); // FALSE**

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

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

-    Вычисляет операнды слева направо.

-    Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.

-    Если все операнды являются ложными (false), возвращает последний из них.

Значение возвращается в исходном виде, без преобразования.

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

Alert (1 || 0); // 1

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

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

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

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

Еще один пример использования ИЛИ для “сокращённого вычисления”. Операндами могут быть как отдельные значения, так и произвольные выражения. ИЛИ вычисляет их слева направо. Вычисление останавливается при достижении первого истинного значения.   
let x;

true || (x = 1);

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

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

let x;

false || (x = 1);

alert(x); // 1

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

[**&& (И)**](https://learn.javascript.ru/logical-operators#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

[И «&&» находит первое ложное значение](https://learn.javascript.ru/logical-operators#i-nahodit-pervoe-lozhnoeznachenie).

 При нескольких подряд операторах И **result = value1 && value2 && value3;** оператор && выполняет следующие действия:

-    Вычисляет операнды слева направо.

-    Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.

-    Если все операнды были истинными, возвращается последний.

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

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

alert( 1 && 0 ); // 0

alert( 1 && 5 ); // 5//

alert( null && 5 ); // null

alert( 0 && "no matter what" ); // 0  
alert( 1 && 2 && null && 3 ); // null   
alert( 1 && 2 && 3 ); // 3

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

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

let x = 1;  
(x > 0) && alert( 'Greater than zero!' );

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

let x = 1;  
if (x > 0)  alert( 'Greater than zero!' )

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

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

**result = !value;**

Оператор принимает один аргумент и выполняет следующие действия:

-    Сначала приводит аргумент к логическому типу true/false.

-    Затем возвращает противоположное значение.

alert( !true ); // false  
alert( !0 ); // true

**Двойное НЕ используют для преобразования значений к логическому типу:**

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

Первое НЕ преобразует значение в логический тип и возвращает обратное, а второе НЕ снова инвертирует его. В результате получится простое преобразование значения в логическое. С помощью встроенной функции Boolean можно сделать то же самое: alert (Boolean ("non-empty string")); // true

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

**?? (нуллевое слияние)**

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

В отличие от логического ИЛИ (||), левая часть оператора вычисляется и возвращается даже если его результат после приведения к логическому типу оказывается ложным, но не является null или undefined. Другими словами, если вы используете || чтобы установить значение по умолчанию, вы можете столкнуться с неожиданным поведением если считаете некоторые ложные значения пригодными для использования (например, "" или 0).   
const foo = null ?? 'default string';  
console.log(foo); // "default string"  
const baz = 0 ?? 42;  
console.log(baz); // 0

8. Циклы while, for. (Т1.12)

Для многократного повторения одного участка кода предусмотрены while do while for foreach.

[**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.

Одно выполнение тела цикла называется **итерация**.

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--;

}

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

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.

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

for (начало; условие; шаг) {

  // ... тело цикла ...

}

}

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 или его любую часть**, получив бесконечный цикл, но обязательны 2 ;; .   
for (;;) {

  // будет выполняться вечно

}

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

Обычно цикл завершается при вычислении условия в false. **Можно выйти из цикла в любой момент с помощью break**.

let sum = 0;

while (true) {

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

  if (!value) break; // (\*)

  sum += value;

}

alert( 'Сумма: ' + sum );

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

При выполнении директивы continue цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true). 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

9. Функции (Function Declaration). Параметры по умолчанию.  (Т1.14)

Чтобы не повторять один и тот же код во многих местах, придуманы функции. Функции являются основными «строительными блоками» программы.

**Function Declaration: функция объявляется отдельной конструкцией** «function…» в основном потоке кода

function showMessage() {

alert( 'Всем привет!' );

}

Важная особенность Function Declaration заключается в их блочной области видимости.

Внутренние(локальные) переменные видны только внутри этой функции. Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).  
У функции есть доступ к внешним переменным(также может менять 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   
Мы можем передать внутрь функции любую информацию, используя параметры (также называемые *аргументами функции*).

[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 мы хотим задать параметру text значение по умолчанию, мы должны указать его после =:  
function showMessage(from, text = "текст не добавлен") {}

* Передаваемые значения копируются в параметры функции и становятся локальными переменными.
* Функции имеют доступ к внешним переменным. Но это работает только изнутри наружу. Код вне функции не имеет доступа к её локальным переменным.
* Функция может возвращать значение(return может находиться в любом месте тела функции). Если этого не происходит, тогда результат равен undefined. Пустой return аналогичен return undefined

Именование функций:

* Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает, и что возвращает.
* Функция – это действие, поэтому её имя обычно является глаголом.
* Есть много общепринятых префиксов, таких как: create…, show…, get…, check… и т.д. Пользуйтесь ими как подсказками, поясняющими, что делает функция.

10. Функциональные выражения (Function Expression) и функции-стрелки. (Т1.15)

Синтаксис, который мы использовали до этого, называется *Function Declaration* (Объявление Функции):

function sayHi() {alert( "Привет" );}

***Function Expression* (Функциональное Выражение).Оно выглядит вот так:**

let sayHi = function() {

  alert( "Привет" );

};

**\*чудеса сценарные\***

function sayHi() {   // (1) создаём, объявление Function Declaration

  alert( "Привет" );

}

let func = sayHi;    // (2) копируем значение в func

func(); // Привет    // (3) вызываем копию (работает)!

sayHi(); // Привет   //     прежняя тоже работает (почему бы нет)

* **Функции – это значения**. Они могут быть присвоены, скопированы или объявлены в другом месте кода.
* Если функция **объявлена как отдельная инструкция** в основном потоке кода, то **это Function Declaration**.
* **Если функция была создана как часть выражения**, то считается, что эта функция объявлена при помощи **Function Expression**.
* В Function Expression ставится точка с запятой ; в конце, а в Function Declaration нет
* Важная особенность Function Declaration заключается в их блочной области видимости.
* Function Declaration обрабатываются перед выполнением блока кода. В результате функции, созданные как Function Declaration, могут быть вызваны раньше своих определений. Они видны внутри блока, в котором располагаются.
* Функции, объявленные при помощи Function Expression, создаются, только когда поток выполнения достигает их.

«**функции-стрелки**» или «стрелочные функции» (arrow functions), т.к. выглядит следующим образом:

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

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

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

…Такой код создаёт функцию func с аргументами arg1..argN и вычисляет expression с правой стороны с их использованием, возвращая результат.

Другими словами, это более короткий вариант такой записи:

let func = function(arg1, arg2, ...argN) { return expression;};

11. Числа. Способы записи числа. Системы счисления. Методы типа Number: преобразование к числу, округление, проверка специальных числовых значений. (Т2.2, кроме: неточные вычисления)

В современном JavaScript существует два типа чисел:

1. **Обычные** 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). Это числа, которые мы будем использовать чаще всего. Мы поговорим о них в этой главе.
2. **BigInt** числа дают возможность работать с целыми числами произвольной длины. Они нужны когда необходимо работать со значениями более чем 253 или менее чем -253. Так как BigInt числа нужны достаточно редко, мы рассмотрим их в отдельной главе [BigInt](https://learn.javascript.ru/bigint).

**Способы записи числа**

let billion = 1000000000;

В JavaScript можно использовать букву "e", чтобы укоротить запись числа. Она добавляется к числу и заменяет указанное количество нулей:

let billion = 1e9;  // 1 миллиард, буквально: 1 и 9 нулей

alert( 7.3e9 );  // 7.3 миллиардов (7,300,000,000)

яФlet ms = 0.000001;

Записать микросекунду в укороченном виде нам поможет "e".

let ms = 1e-6; // шесть нулей, слева от 1  
Если мы подсчитаем количество нулей 0.000001, их будет 6. Естественно, верная запись 1e-6.  
Другими словами, отрицательное число после "e" подразумевает деление на 1 с указанным количеством нулей:  
// 1 делится на 1 с 3 нулями  
1e-3 = 1 / 1000 (=0.001)  
// 1.23 делится на 1 с 6 нулями

1.23e-6 = 1.23 / 1000000 (=0.00000123)  
[**Шестнадцатеричные, двоичные и восьмеричные числа**](https://learn.javascript.ru/number#shestnadtsaterichnye-dvoichnye-i-vosmerichnye-chisla)[Шестнадцатеричные](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D1%81%D1%82%D0%BD%D0%B0%D0%B4%D1%86%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) числа широко используются в JavaScript для представления цветов, кодировки символов и многого другого. Естественно, есть короткий стиль записи: 0x, после которого указывается число.

Например:

alert( 0xff ); // 255

alert( 0xFF ); // 255 (то же самое, 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, с двух сторон число 255

[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.

**Функция parseInt возвращает целое число, а parseFloat возвращает число с плавающей точкой.** Если в процессе чтения возникает ошибка, они возвращают полученное до ошибки число:

alert( parseInt('100px') ); // 100  
alert( parseFloat('12.5em') ); // 12.5  
alert( parseInt('12.3') ); // 12, вернётся только целая часть  
alert( parseFloat('12.3.4') ); // 12.3, произойдёт остановка чтения на второй точке

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

[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.

**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.

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

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

12. Строки. Методы типа String: изменение регистра, поиск подстроки. Сравнение строк. (Т2.3, кроме: спецсимволы, правильное сравнение)

Внутренний формат для строк — всегда [UTF-16](https://ru.wikipedia.org/wiki/UTF-16), вне зависимости от кодировки страницы.

[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

[Поиск подстроки](https://learn.javascript.ru/string#poisk-podstroki)

Существует несколько способов поиска подстроки.

**str.indexOf**

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, потому что подстрока 'Widget' найдена в начале

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

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

**str.lastIndexOf(substr, position)**

Также есть похожий метод [str.lastIndexOf(substr, position)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf), который ищет с конца строки к её началу.

Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.

[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.

Это — правильный выбор, если нам необходимо проверить, есть ли совпадение, но позиция не нужна:

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

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

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

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

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

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

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

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

[Получение подстроки](https://learn.javascript.ru/string#poluchenie-podstroki)

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

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

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

let str = "stringify";

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

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

let str = "stringify";

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

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

**str.substring(start [, end])**

Возвращает часть строки *между* start и end.

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

Например:

let str = "stringify";

// для substring эти два примера — одинаковы

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

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

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

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

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

**str.substr(start [, length])** - Возвращает часть строки от start длины length.

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

let str = "stringify";

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

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

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

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

Строчные буквы больше заглавных(код больше):

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

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

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

Метод[**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

* Есть три типа кавычек. Строки, использующие обратные кавычки, могут занимать более одной строки в коде и включать выражения ${…}.
* Строки в JavaScript кодируются в UTF-16.
* Есть специальные символы, такие как \n, и можно добавить символ по его юникодному коду, используя \u….
* Для получения символа используйте [].

Строки также имеют ещё кое-какие полезные методы:

* str.trim() — убирает пробелы в начале и конце строки.
* str.repeat(n) — повторяет строку n раз.

Также есть методы для поиска и замены с использованием регулярных выражений.

13. Массивы. (Т2.4)

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

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:

alert( fruits[0] ); // Яблоко

Можно заменить элемент:

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

Или добавить новый элемент к существующему массиву:

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

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

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

**«Висячая» запятая**

Массив может заканчиваться запятой:  
let fruits = ["Яблоко", "Апельсин", "Слива"  ,    ];

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

14. Методы массивов: добавление, удаление и замена элементов, объединение массивов, поиск в массиве.  (Т2.5)

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

-    arr.push(...items) – добавляет элементы в конец,

-    arr.pop() – извлекает элемент из конца,

-    arr.shift() – извлекает элемент из начала,

-    arr.unshift(...items) – добавляет элементы в начало.

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

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

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

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

Рассмотрим пример удаления элементов:

let arr = ["Я", "изучаю", "JavaScript"];  
arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент  
alert( arr ); // осталось ["Я", "JavaScript"]

**Начиная с позиции 1, метод удалил 1 элемент.**

**Удалим 3 элемента и заменим их двумя другими:**

let arr = ["Я", "изучаю", "JavaScript", "прямо", "сейчас"];  
arr.splice(0, 3, "Давай", "танцевать");  
alert( arr ) // теперь ["Давай", "танцевать", "прямо", "сейчас"]

 Здесь видно, что splice возвращает массив из удалённых элементов:

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

 // удалить 2 первых элемента

let removed = arr.splice(0, 2);  
alert( removed ); // "Я", "изучаю" – массив из удалённых элементов

 Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0:  
let arr = ["Я", "изучаю", "JavaScript"];  
// с позиции 2 удалить 0 элементов и вставить "сложный", "язык"  
arr.splice(2, 0, "сложный", "язык");  
alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"

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

let arr = [1, 2, 5];  
// начиная с индекса -1 (перед последним элементом) удалить 0 элементов,  
// затем вставить числа 3 и 4  
arr.splice(-1, 0, 3, 4);  
alert( arr ); // 1,2,3,4,5

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

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

**arr.slice(start, end)**

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

let arr = ["t", "e", "s", "t"];  
alert( arr.slice(1, 3) ); // e,s  
alert( arr.slice(-2) ); // s,t

**Метод** [**concat**](https://learn.javascript.ru/array-methods#concat)

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

**arr.concat(arg1, arg2...)**

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

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

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

**Методы** [**поиска**](https://learn.javascript.ru/array-methods#poisk-v-massive) **в массиве**Методы [arr.indexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf), [arr.lastIndexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/lastIndexOf) и [arr.includes](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/includes) имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

-   arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.

-    arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.

-   arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

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

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

  // если true - возвращается текущий элемент и перебор прерывается

  // если все итерации оказались ложными возвращается undefined});

Функция вызывается по очереди для каждого элемента массива:

-    item – очередной элемент.

-    index – его индекс.

-    array – сам массив.

let users = [

{id: 1, name: "Вася"},

{id: 2, name: "Петя"},

{id: 3, name: "Маша"}

];

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

alert(user.name); // Вася

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

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

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

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

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

});

15. Методы перебора и преобразование массива. (Т2.5)

[**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});**

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

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

});

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

Одним из самых старых способов перебора элементов массива является **цикл for** по цифровым индексам:

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

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

// проходит по значениям  
for (let fruit of fruits) {alert( fruit );}

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

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

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

for (let key in arr) { alert( arr[key] ); // Яблоко, Апельсин, Груша}

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

16. Объекты. Литералы и свойства. Вычисляемые и короткие свойства. Проверка существования свойства. Перебор и упорядочение свойств объекта. (Т2.6)

**1.**  **Объекты.**

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"**(или литеральная нотация)

[**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. Если имя свойства состоит из нескольких слов, тогда оно должно быть заключено в кавычки.

Для обращения к свойствам используется запись «через точку»:

**Для удаления свойства** можно использовать оператор delete:

delete user.age;

[**Квадратные скобки**](https://learn.javascript.ru/object#kvadratnye-skobki)

Квадратные скобки также **позволяют обратиться к свойству, имя которого может быть результатом выражения**. Например, имя свойства может храниться в переменной.

let key = "likes birds";

user[key] = true;

[**Вычисляемые свойства**](https://learn.javascript.ru/object#vychislyaemye-svoystva)

Можно использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства.

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"

[**Свойство из переменной**](https://learn.javascript.ru/object#svoystvo-iz-peremennoy)

В реальном коде часто необходимо использовать существующие переменные как значения для свойств с тем же именем.

function makeUser(name, age) {

return {

name: name,

age: age

// ...другие свойства

};

}

let user = makeUser("John", 30);

alert(user.name); // John

В примере выше название свойств name и age совпадают с названиями переменных, которые указываются в качестве значений этих свойств. Такой подход настолько распространен, что существуют специальные **короткие свойства** для упрощения этой записи. Вместо name:name можно написать просто name. Можно использовать как обычные свойства, так и короткие в одном и том же объекте:

function makeUser(name, age) {

return {

name, //то же самое, что и name:name

age

height: 170// ...

};

}

[**Проверка существования свойства**](https://learn.javascript.ru/object#proverka-suschestvovaniya-svoystva)

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**

let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age существует

alert( "blabla" in user ); // false, user.blabla не существует

[**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:

for (**let key in user**) {

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

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

}

[**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/

Таким образом, чтобы телефонные коды выводились, в том порядке в котором 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

}

17. Копирование, клонирование, сравнение, объединение объектов. Объекты-константы. (Т2.6)

**Копирование по ссылке**

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

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"

**Клонирование**

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];

}

Можно использовать метод **Object.assign**.

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.

**Сравнение объектов**

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.

**Объединени**е

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 });

**Объекты-констант**ы

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"

};

18. Тип данных Symbol. (Т2.7)

По спецификации, в качестве ключей для свойств объекта могут использоваться только строки либо символы. Ни числа, ни логические значения не подходят, разрешены только эти два типа данных.

Символ представляет собой уникальный идентификатор. Создаются новые символы с помощью функции Symbol(). Создание нового символа – id:

let id = Symbol();

При создании символу можно дать описание (также называемое имя), в основном использующееся для отладки кода. Например, создадим символ id с описанием (именем) "id":

let id = Symbol("id");

Символы гарантированно уникальны. Если создать множество символов с одинаковым описанием, это всё равно будут разные символы. Описание – это просто метка, которая ни на что не влияет. Например, два символа с одинаковым описанием,  но они не равны:

let id1 = Symbol("id");

let id2 = Symbol("id");

alert(id1 == id2); // false

Большинство типов данных в JavaScript могут быть неявно преобразованы в строку. Например, функция alert принимает практически любое значение, автоматически преобразовывает его в строку, а затем выводит это значение, не сообщая об ошибке. Символы же особенные и не преобразуются автоматически. К примеру, alert ниже выдаст ошибку:

let id = Symbol("id");

alert(id); // TypeError: Cannot convert a Symbol value to a string

Если нужно вывести символ с помощью alert, то необходимо явно преобразовать его с помощью метода .toString():

let id = Symbol("id");

alert(id.toString()); // Symbol(id)

Также можно обратиться к свойству symbol.description, чтобы вывести только описание:

let id = Symbol("id");

alert(id.description); // id

Символы позволяют создавать «скрытые» свойства объектов, к которым нельзя нечаянно обратиться и перезаписать их из других частей программы. Например, выполняются действия с объектами user, которые принадлежат стороннему коду, и в которых нет поля "id". Надо добавить к ним идентификаторы. Можно использовать для этого символьный ключ:

let user = { name: "Егор" };

let id = Symbol("id");

user[id] = "ID Value";

alert( user[id] );

Если необходимо использовать символ при литеральном объявлении объекта {...}, его необходимо заключить в квадратные скобки:

let id = Symbol("id");

let user = {

name: "Егор",

**[id]:** 123

};

Это связано с необходимостью использовать значение переменной id в качестве ключа, а не строку «id».

Иногда необходимо, чтобы символы с одинаковыми именами были одной сущностью. Для этого существует глобальный реестр символов. Можно создавать в нём символы и обращаться к ним позже, и при каждом обращении гарантированно будет возвращаться один и тот же символ.

Для чтения (или, при отсутствии, создания) символа из реестра используется вызов Symbol.for(key). Он проверяет глобальный реестр, и если в нём есть символ с именем key, то возвращает его, иначе создает новый символ Symbol(key) и записывает его в реестр под ключом key. Например:

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

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

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

19. Коллекции Set, WeakSet. (Т2.8)

**Set**

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

- **new Set**(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.

- set.**add**(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.

- set.**delete**(value) – удаляет значение, возвращает true если value было в множестве на момент вызова, иначе false.

- set.**has**(value) – возвращает true, если значение присутствует в множестве, иначе false.

- set.**clear**() – удаляет все имеющиеся значения.

- set.**size** – возвращает количество элементов в множестве.

let set = new Set();

let john = { name: "John" };

let pete = { name: "Pete" };

let mary = { name: "Mary" };

// считаем гостей, некоторые приходят несколько раз

set.add(john);

set.add(pete);

set.add(mary);

set.add(john);

set.add(mary);

// set хранит только 3 уникальных значения

alert(set.size); // 3

for (let user of set) {

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

}

Суть в том, что при повторных вызовах set.add() с одним и тем же значением ничего не происходит, за счёт этого как раз и получается, что каждое значение появляется один раз.

Альтернативой множеству Set может выступать массив для хранения гостей и дополнительный код для проверки уже имеющегося элемента с помощью [arr.find](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/find). Но в этом случае будет хуже производительность, потому что arr.find проходит весь массив для проверки наличия элемента. Множество Set лучше оптимизировано для добавлений, оно автоматически проверяет на уникальность.

Можно перебрать содержимое объекта set как с помощью метода for..of, так и используя forEach

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

alert(value);

});

Функция в forEach у Set имеет 3 аргумента: значение value, потом снова то же самое значение valueAgain, и целевой объект. Значение появляется в списке аргументов дважды. Это сделано для совместимости с объектом Map, в котором колбэк forEach имеет 3 аргумента. Выглядит странно, но в некоторых случаях может помочь легко заменить Map на Set и наоборот. Set имеет те же встроенные методы, что и Map:

- set.**keys()** – возвращает перебираемый объект для значений,

- set.**values()** – то же самое, что и set.keys(), присутствует для обратной совместимости с Map,

- set.**entries()** – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

**WeekSet**

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

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

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

let visitedSet = new WeakSet();

let john = { name: "John" };

let pete = { name: "Pete" };

visitedSet.add(john); // John

visitedSet.add(pete); // Pete

visitedSet.add(john); // John

// visitedSet сейчас содержит двух пользователей

alert(visitedSet.has(john)); // true

alert(visitedSet.has(mary)); // false

john = null;

// структура данных visitedSet будет очищена автоматически

- Она аналогична Set, но можно добавлять в WeakSet только объекты (не примитивные значения).

- Объект присутствует в множестве только до тех пор, пока доступен где-то ещё.

- Как и Set, она поддерживает add, has и delete, но не size, keys() и не является перебираемой.

WeakSet тоже служит в качестве дополнительного хранилища. Но не для произвольных данных, а скорее для значений типа «да/нет». Присутствие во множестве WeakSet может что-то сказать об объекте.

Наиболее значительным ограничением WeakMap и WeakSet является то, что их нельзя перебрать или взять всё содержимое.

20. Коллекции Map, WeakMap. (Т2.8)

[Map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Map) – это коллекция ключ/значение, как и Object (ключ - строка). 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.

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

В отличие от объектов, ключи не были приведены к строкам. Можно использовать любые типы данных для ключей, даже объекты.

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.

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 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

});

Если есть обычный объект, и необходимо создать Map из него, то поможет встроенный метод [Object.entries(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/entries), который получает объект и возвращает массив пар ключ-значение для него, как раз в этом формате.

let obj = {

name: "John",

age: 30

};

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

Метод **Object.fromEntries** делает противоположное: получив массив пар вида [ключ, значение], он создаёт из них объект.

Можно использовать Object.fromEntries, чтобы получить обычный объект из Map. Например, есть данные в Map, но их нужно передать в сторонний код, который ожидает обычный объект.

Вызов map.entries() возвращает массив пар ключ/значение, как раз в нужном формате для Object.fromEntries

**WeekMap**

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

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

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)

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.

21. Деструктурирующее присваивание.  Вложенная деструктуризация. (Т2.9)

***Деструктурирующее присваивание*** – специальный синтаксис, который позволяет преобразовать массивы или объекты в кучу переменных, так как иногда они более удобны. Деструктуризация также прекрасно работает со сложными функциями, которые имеют много параметров, значений по умолчанию, и так далее.

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

let arr = ["Steve", "Jobs"]

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

let [firstName, surname] = arr;

alert(firstName); // Steve

alert(surname);  // Jobs

Теперь можно использовать переменные вместо элементов массива. Удобно использовать в сочетании со **split** или другими методами, возвращающими массив:

let [firstName, surname] = "Steve Jobs".split(' ');

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

Ненужные элементы массива также могут быть отброшены **через запятую**:

// второй элемент не нужен

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

alert( title ); // Consul

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

Можно использовать любой перебираемый объект, не только массивы:

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

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

Можно использовать что угодно с левой стороны. Например, можно присвоить *свойству объекта*:

let user = {};

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

alert(user.name); // Ilya

Ранее рассматривался метод [Object.entries(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/entries). Можем использовать его с деструктуризацией для цикличного перебора ключей и значений объекта:

let user = {

  name: "John",

  age: 30

};

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

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

}

То же самое для map:

let user = new Map();

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

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

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

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

}

Если надо не просто получить первые значения, но и собрать все остальные – мы можем добавить ещё один параметр, который получает остальные значения, **используя троеточие "...":**

let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];

alert(name1); // Julius

alert(name2); // Caesar

alert(rest[0]); // Consul

alert(rest[1]); // of the Roman Republic

alert(rest.length); // 2

Переменная rest является массивом из оставшихся элементов. Вместо rest можно использовать любое другое название переменной, и она *должна находится на последнем месте* в деструктурирующем присваивании.

Если в массиве меньше значений, чем в присваивании, то ошибки не будет. Отсутствующие значения считаются неопределёнными:

let [firstName, surname] = [];

alert(firstName); // undefined

alert(surname); // undefined

Если необходимо указать значения по умолчанию, то можно использовать = :

let [name = "Guest", surname = "Anonymous"] = ["Julius"];

alert(name); // Julius (из массива)

alert(surname); // Anonymous (значение по умолчанию)

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

Деструктурирующее присваивание также работает с объектами. **Имена переменных и ключи должны совпадать.** Синтаксис:

let {var1, var2} = {var1:…, var2:…}

let options = {

  title: "Menu",

  width: 100,

  height: 200

};

let {title, width, height} = options;

alert(title);  // Menu

alert(width);  // 100

alert(height); // 200

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

// изменён порядок в let {...}

let {height, width, title} = { title: "Menu", height: 200, width: 100 }

Если нужно присвоить свойство объекта переменной **с другим названием**, например, свойство options.width присвоить переменной *w*, то можно использовать двоеточие:

let options = {

title: "Menu",

width: 100,

height: 200

};

let {width: w, height: h, title} = options;

alert(title); // Menu

alert(w); // 100

alert(h); // 200

Двоеточие показывает *«что : куда идёт»*.

Для потенциально отсутствующих свойств можно установить **значения по умолчанию**, используя "=":

let options = {

  title: "Menu"

};

let {width = 100, height = 200, title} = options;

alert(title);  // Menu

alert(width);  // 100

alert(height); // 200

Как и в случае с массивами, значениями по умолчанию могут быть любые выражения или даже функции. Они выполнятся, если значения отсутствуют.

[**Остаток объекта «…»**](https://learn.javascript.ru/destructuring-assignment#ostatok-obekta)

Если в объекте больше свойств, чем переменных, то можно использовать троеточие, так же как для массивов. В некоторых старых браузерах (IE) это не поддерживается, необходимо использовать полифилы. Например:

let options = {

  title: "Menu",

  height: 200,

  width: 100

};

let {title, ...rest} = options;

// title="Menu", rest={height: 200, width: 100}

alert(rest.height);  // 200

alert(rest.width);   // 100

[**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.

22. Дата и Время. Объект Date. Получение и установка компонентов даты. Автоисправление даты. Разность дат. (Т2.10)

Встроенный объект [Date](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date) содержит дату и время, а также предоставляет методы управления ими. Например, его можно использовать для хранения времени создания/изменения, для измерения времени или просто для вывода текущей даты.

[**Создание**](https://learn.javascript.ru/date#sozdanie)

Для создания нового объекта Date нужно вызвать конструктор new Date() с одним из следующих аргументов:

- **new Date()** без аргументов – создать **объект Date с текущими датой** и временем:

let now = new Date();

alert( now ); // показывает текущие дату и время

- **new Date(milliseconds)** – создать объект Date с временем, равным **количеству миллисекунд** (тысячная доля секунды), прошедших **с 1 января 1970 года** UTC+0.

// 0 соответствует 01.01.1970 UTC+0

let Jan01\_1970 = new Date(0);

// добавим 24 часа и получим 02.01.1970 UTC+0

let Jan02\_1970 = new Date(24 \* 3600 \* 1000);

Целое число, представляющее собой **количество миллисекунд, прошедших с начала 1970 года**, называется ***таймстамп***(англ. timestamp). Это легковесное численное представление даты. Из таймстампа всегда можно получить дату с помощью new Date(timestamp) и преобразовать существующий объект Date в таймстамп, используя метод date.getTime() (см. ниже).

- **new Date(datestring)** – если аргумент всего один, и это строка, то из неё считывается дата. В примере ниже время не указано, поэтому оно ставится в полночь по Гринвичу и меняется в соответствии с временной зоной места выполнения кода:

let date = new Date("2017-01-26");

alert(date);

- **new Date(year, month, date, hours, minutes, seconds, ms)** – создать объект Date с заданными компонентами в местной временной зоне. Обязательны только первые два аргумента.

·       year должен состоять из четырёх цифр: значение 2013 корректно, 98 – нет.

·       month начинается с 0 (январь) по 11 (декабрь).

·       Параметр date здесь представляет собой день месяца. Если параметр не задан, то принимается значение 1.

·       Если параметры hours/minutes/seconds/ms отсутствуют, их значением становится 0.

new Date(2011, 0, 1, 0, 0, 0, 0); // // 1 Jan 2011, 00:00:00

new Date(2011, 0, 1); // то же самое, так как часы и проч. равны 0

[П**олучение компонентов даты**](https://learn.javascript.ru/date#poluchenie-komponentov-daty)

Существуют методы получения года, месяца и т.д. из объекта Date:

- [getFullYear()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getFullYear) – получить год (4 цифры)

- [getMonth()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMonth) – получить месяц, от 0 до 11.

- [getDate()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getDate) – получить день месяца, от 1 до 31, что несколько противоречит названию метода.

- [getHours()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getHours), [getMinutes()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMinutes), [getSeconds()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getSeconds), [getMilliseconds()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMilliseconds) – получить, **соответственно, часы, минуты, секунды или миллисекунды.**

- [getDay()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getDay) – вернуть **номер дня недели** от 0 (воскресенье) до 6 (суббота).

Все вышеперечисленные методы возвращают значения в соответствии с местной временной зоной. Однако существуют и их UTC-варианты, возвращающие день, месяц, год для временной зоны UTC+0: [getUTCFullYear()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCFullYear), [getUTCMonth()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCMonth), [getUTCDay()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCDay).

Помимо вышеприведённых методов, существуют два особых метода без UTC-варианта:

[**getTime()**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getTime)– для заданной даты возвращает таймстамп – количество миллисекунд, прошедших с 1 января 1970 года UTC+0.

[**getTimezoneOffset()**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getTimezoneOffset)– возвращает разницу в минутах между местной временной зоной и UTC:

[У**становка компонентов даты**](https://learn.javascript.ru/date#ustanovka-komponentov-daty)

Следующие методы позволяют установить компоненты даты и времени:

- [setFullYear(year, [month], [date])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setFullYear)

- [setMonth(month, [date])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMonth)

- [setDate(date)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setDate)

- [setHours(hour, [min], [sec], [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setHours)

- [setMinutes(min, [sec], [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMinutes)

- [setSeconds(sec, [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setSeconds)

- [setMilliseconds(ms)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMilliseconds)

- [setTime(milliseconds)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setTime) (устанавливает дату в виде целого количества миллисекунд, прошедших с 01.01.1970 UTC)

[**Автоисправление даты**](https://learn.javascript.ru/date#avtoispravlenie-daty)

**Автоисправление** – это очень полезная особенность объектов Date. Можно устанавливать компоненты даты вне обычного диапазона значений, а объект сам себя исправит. Пример:

let date = new Date(2013, 0, 32); // 32 Jan 2013

alert(date); // 1st Feb 2013

[**Преобразование к числу, разность дат**](https://learn.javascript.ru/date#preobrazovanie-k-chislu-raznost-dat)

` `Если объект Date преобразовать в число, то получим таймстамп, по аналогии с date.getTime():

let date = new Date();

alert(+date); // количество миллисекунд, то же самое, что date.getTime()

Важный побочный эффект: даты можно вычитать, в результате получим **разность в миллисекундах**. Этот приём можно использовать для измерения времени:

let start = new Date(); // начинаем отсчёт времени

for (let i = 0; i < 100000; i++) {

  let doSomething = i \* i \* i;

}

let end = new Date(); // заканчиваем отсчёт времени

alert( `Цикл отработал за ${end - start} миллисекунд` );

[**Date.now()**](https://learn.javascript.ru/date#date-now)

Если нужно **просто померить время**, для этого существует особый метод **Date.now(), возвращающий текущий таймстамп**. Семантически он эквивалентен new Date().getTime(), однако метод не создаёт промежуточный объект Date. Так что этот способ работает быстрее и не нагружает сборщик мусора. Поэтому, предыдущий пример лучше переписать так:

let start = Date.now(); // количество миллисекунд с 1 января 1970 года

for (let i = 0; i < 100000; i++) {

  let doSomething = i \* i \* i;

}

let end = Date.now(); // заканчиваем отсчёт времени

alert( `Цикл отработал за ${end - start} миллисекунд` ); // вычитаются числа, а не даты

**Метод** [**Date.parse(str)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/parse) **считывает дату из строки.** Формат строки должен быть следующим: YYYY-MM-DDTHH:mm:ss.sssZ, где:

- YYYY-MM-DD – это дата: год-месяц-день.

- Символ "T" используется в качестве разделителя.

- HH:mm:ss.sss – время: часы, минуты, секунды и миллисекунды.

- Необязательная часть 'Z' обозначает временную зону в формате +-hh:mm. Если указать просто букву Z, то получим UTC+0.

Возможны и более короткие варианты, например, YYYY-MM-DD или YYYY-MM или даже YYYY.

**Вызов Date.parse(str) обрабатывает строку в заданном формате и возвращает таймстамп** (количество миллисекунд с 1 января 1970 года UTC+0). Если формат неправильный, возвращается NaN. Например:

let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 1327611110417  (таймстамп)

23. Глобальный объект. Создание функции с помощью конструктора (new Function). (Т3.1, Т3.3)

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.

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)

Если объявить переменную при помощи let, то такого не произойдет

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

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

if (!window.Promise) {

  alert("Ваш браузер очень старый!");

}

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

if (!window.Promise) {

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

}

**Конструкция new Function**

Существует ещё один вариант объявлять функции. Он используется крайне редко, но иногда другого решения не найти.  Синтаксис для объявления функции:

**let func = new Function([arg1, arg2, ...argN], functionBody);**

Функция создается с заданными аргументами arg1...argN и телом functionBody.

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;

}

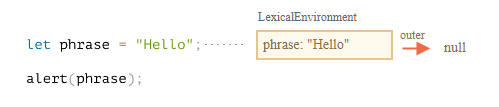
getFunc()(); // "test", из лексического окружения функции getFunc

24. [Лексическое](https://learn.javascript.ru/closure#leksicheskoe-okruzhenie) окружение (LexicalEnvironment). Замыкание. (Т3.2)

В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый *лексическим окружением LexicalEnvironment*. Объект лексического окружения состоит из двух частей:

* **Environment Record** – объект, в котором как свойства хранятся все локальные переменные (а также некоторая другая информация, такая как значение this).
* **Ссылка на внешнее лексическое окружение** – то есть то, которое соответствует коду снаружи (снаружи от текущих фигурных скобок).

Переменная – это просто свойство специального внутреннего объекта: Environment Record. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта».



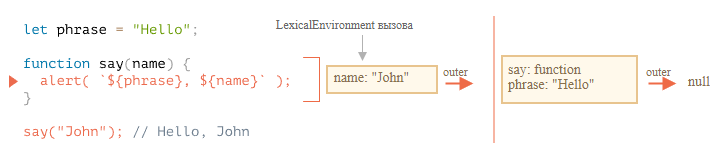
На картинке выше прямоугольник означает Environment Record (хранилище переменных), а стрелка означает ссылку на внешнее окружение. У глобального лексического окружения нет внешнего окружения, так что она указывает на null.

**Function Declaration**

Теперь рассмотрим Function Declaration. В отличие от переменных, объявленных с помощью let, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, *когда создаётся лексическое окружение*. Для верхнеуровневых функций это означает момент, когда скрипт начинает выполнение. Вот почему можно вызвать функцию, объявленную через Function Declaration, *до того, как она определена*.

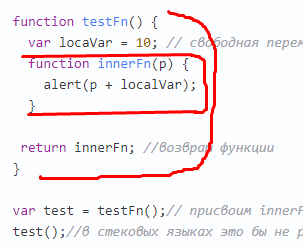
**[Внутреннее и внешнее лексическое окружение](https://learn.javascript.ru/closure" \l "vnutrennee-i-vneshnee-leksicheskoe-okruzhenie)**

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



Итак, в процессе вызова функции **есть два лексических окружения: внутреннее** (для вызываемой функции) и **внешнее (глобальное).** Внутреннее лексическое окружение соответствует текущему выполнению say. В нём находится одна переменная name, аргумент функции. При вызове say("John") значение переменной name равно "John". *Внешнее лексическое окружение – это глобальное лексическое окружение*. В нём находятся переменная phrase и сама функция. **У внутреннего лексического окружения есть ссылка outer на внешнее. Когда код хочет получить доступ к переменной – сначала происходит поиск во внутреннем лексическом окружении, затем во внешнем, затем в следующем и так далее, до глобального.** Если переменная не была найдена, это будет ошибкой в strict mode. Без strict mode, для обратной совместимости, присваивание несуществующей переменной создаёт новую глобальную переменную с таким именем.

Можно изобразить лексическое окружение на примере простого кода, где фрагменты кода, вложенные в друг в друга, будут являться уровнями.



При разрешении переменной localVar произойдёт сначала поиск подобной переменной в месте вызова (alert) , так как в этой области нет значения, поиск поднимется в область выше, где объявлена эта переменная со значением 10.

**Замыкание это функция, которая запоминает свои внешние переменные и может получить к ним доступ.** То есть, они автоматически запоминают, где были созданы, с помощью скрытого свойства [[Environment]] и все они могут получить доступ к внешним переменным. (все функции в js являются замыканиями)

25. Объект функции. Именованное функциональное выражение (Named Function Expression). (Т3.4)

В JS **функция это** объект. В случае с функцией это объект, который может делать действия.

В структуре функции есть и свойства:

* **name** это имя функции
* **length** содержит количество параметров функции в её объявлении
* **любые** пользовательские свойства. Например, свойство counter для отслеживания общего количества вызовов:

function sayHi() {

alert("Hi");

sayHi.counter++;

}

sayHi.counter = 0;

sayHi(); // Hi

sayHi(); // Hi

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

 При этом свойства функции и переменные-это два независимых разных понятия.

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

let sayHi = function func(who) {

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

};

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.

То есть если let saySmth= function func() {alert(‘lol’)}; вызвать func не получится извне.(функцию saySmth можно) А вот вызвать func внутри func можно.

26. Остаточные параметры и оператор расширения. (Т3.5)

**Остаточные параметры (…)** предназначены для вызова функций с любым количеством входных параметров. Остаточные параметры в функции содержатся в виде массива, обозначены через три точки «…».

function showName(firstName, lastName, **...titles**) {

alert( firstName + ' ' + lastName ); // Юлий Цезарь

// titles = ["Консул", "Император"]

alert( titles[0] ); // Консул

alert( titles[1] ); // Император

alert( titles.length ); // 2

}

showName("Юлий", "Цезарь", "Консул", "Император");

Остаточные параметры собирают все остальные аргументы, поэтому бессмысленно писать что-либо после них. Это вызовет ошибку

function f(arg1, ...rest, arg2) { // Ошибка

Все аргументы функции также можно получить при помощи **псевдомассива arguments** под своими порядковыми номерами

function showName() {

alert( arguments.length );

alert( arguments[0] );

alert( arguments[1] );

}

// Вывод: 2, Юлий, Цезарь

showName("Юлий", "Цезарь");

**Оператор расширения**

Необходим для преобразования одного типа в другой. Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов.

Так можно преобразовать:

* массив в список

let arr = [3, 5, 1];

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

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

* строку в массив символов

let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т

* можно соединять массивы

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

27. Каррирование и частичное применение функции. (Т3.6)

**Каррирование или карринг (currying)** – 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, можно сказать, что это процесс превращения функции с несколькими аргументами в функцию с меньшей арностью.

**Арность** – это количество аргументов функции.

Исходная функция:

function multiply(a, b, c) {

   return a \* b \* c;

}

multiply(1,2,3); *// 6*

Каррированный вариант этой функции

function multiply(a) {

   return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

}

log(multiply(1)(2)(3)) *// 6*

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.

Каррирование и **частичное применение функций** очень похожи друг на друга, но концепции это разные.

function volume(l) {

   return (w, h) => {

       return l \* w \* h

   }

}

Пользоватьтся можно так:

const hV = volume(70)*;*

hV(203,142)*;*

И так:

volume(70)(90,30)*;*

volume(70)(390,320)*;*

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).

**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).

28. Генераторы. Функции-генераторы. Перебор объектов-генераторов. (Т3.7, кроме: композиция генераторов, generator.throw)

Обычные функции возвращают только одно-единственное значение (или ничего). **Генераторы могут порождать (yield) множество значений одно за другим**, по мере необходимости. Генераторы отлично работают с перебираемыми объектами и позволяют легко создавать потоки данных.

[**Функция-генератор**](https://learn.javascript.ru/generators#funktsiya-generator)

Для объявления генератора используется специальная синтаксическая конструкция: **function\* f(…) (или function \*f(…))**, которая называется «функция-генератор». Выглядит она так:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

Функции-генераторы ведут себя не так, как обычные. Когда такая функция вызвана, **она не выполняет свой код, а возвращает специальный объект** «генератор» для управления её выполнением:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

alert(generator); // [object Generator]

Основным методом генератора является **next()**. При вызове он запускает выполнение **кода до ближайшей инструкции yield** <значение> (значение может отсутствовать, в этом случае оно полагается равным undefined). По достижении yield выполнение функции приостанавливается, а соответствующее значение – возвращается во внешний код:

Результатом метода next() всегда является объект с двумя свойствами:

-    **value**: значение из yield.

-     **done:** true, если выполнение функции завершено, иначе false.

Создадим генератор и получим первое из возвращаемых им значений:

function\* generateSequence() {

  yield 1;

  yield 2;

  return 3;

}

let generator = generateSequence();

let one = generator.next();

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

На данный момент получено только первое значение, выполнение функции остановлено на второй строке. **Повторный вызов** generator.next() возобновит выполнение кода и вернёт результат следующего yield. И, наконец, **последний вызов** завершит выполнение функции и вернёт результат return:

let three = generator.next();

alert(JSON.stringify(three)); // {value: 3, done: true}

Сейчас генератор полностью выполнен. Можно увидеть это по свойству done:true и обработать value:3 как окончательный результат. Новые вызовы generator.next() больше не имеют смысла. Впрочем, если они и будут, то не вызовут ошибки, но будут возвращать один и тот же объект: {done: true}.

[**Перебор генераторов**](https://learn.javascript.ru/generators#perebor-generatorov)

**Генераторы являются** [**перебираемыми**](https://learn.javascript.ru/iterable) **объектами**. Возвращаемые ими значения можно перебирать **через for..of:**

function\* generateSequence() {

  yield 1;

  yield 2;

  return 3;

}

let generator = generateSequence();

for(let value of generator) {

  alert(value); // 1, затем 2

}

Обратите внимание: пример выше выводит значение 1, затем 2. **Значение 3 выведено не будет**. Это потому что перебор через for..of игнорирует последнее значение, при котором done: true. Поэтому, если необходимо, чтобы были все значения при переборе через for..of, **то надо возвращать их через yield**:

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

Так как генераторы являются перебираемыми объектами, можно использовать всю связанную с ними функциональность, например, **оператор расширения ...:**

let sequence = [0, **...generateSequence**()];

alert(sequence); // 0, 1, 2, 3

В коде выше ...generateSequence() превращает перебираемый объект-генератор в массив элементов.

29. Методы объектов, this. Оператор опциональной последовательности. (Т4.1)

**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");

}

};

**Оператор *опциональной последовательности***

Оператор опциональной последовательности **?.** **позволяет получить значение свойства, находящегося на любом уровне вложенности в цепочке связанных между собой объектов**, без необходимости проверять каждое из промежуточных свойств в ней на существование.  ?. работает подобно оператору ., за исключением того, что не выбрасывает исключение, если объект, к свойству или методу которого идёт обращение, равен null или undefined. В этих случаях он возвращает undefined.

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** = {//this смотрит сюда

  name: "Джон",

  age: 30,

  sayHi() {

// this - это "текущий объект"

alert(**this.name**);

  }

};

user.sayHi(); // Джон

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. 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.

**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*.

30. Итерируемые объекты. Псевдомассивы. (Т4.2, кроме: Использование генераторов для перебираемых объектов )

***Итерируемые* объекты** – э**то концепция, которая позволяет использовать любой объект в цикле for..of.**

Легко понять принцип устройства перебираемых объектов, создав один из них. Например, есть объект range, который представляет собой диапазон чисел. Это не массив, но он выглядит подходящим для for..of:

let range = {  
from: 1,  
to: 5  
};

Чтобы сделать range **итерируемым** (и позволить с ним работать for..of), нужно добавить в объект **метод с именем Symbol.iterator** (специальный встроенный Symbol, созданный как раз для этого).

1.   Когда цикл for..of запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть **итератор** – объект с методом next.

2.   Дальше, for..of работает только с этим возвращённым объектом.

3.   Когда for..of хочет получить следующее значение, он вызывает **метод next()** этого объекта.

4.   Результат вызова next() должен иметь вид {done: Boolean, value: any}, где done=true означает, что итерация закончена, в противном случае value содержит очередное значение.

Вот полная реализация range с пояснениями

let range = {

from: 1,

to: 5

};

// 1. вызов for..of сначала вызывает эту функцию

range[Symbol.iterator] = function() {

// она возвращает объект итератора:

// 2. Далее, for..of работает только с этим итератором, запрашивая у него новые значения

return {

current: this.from,

last: this.to,

// 3. next() вызывается на каждой итерации цикла for..of

next() {

// 4. он должен вернуть значение в виде объекта {done:.., value :...}

if (this.current <= this.last) {

return { done: false, value: this.current++ };

} else {

return { done: true };

}

}

};

};

// теперь работает

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Обратите внимание на ключевую особенность итераторов: разделение ответственности.

- У самого range нет метода next().

- Вместо этого другой объект, так называемый «итератор», создаётся вызовом range[Symbol.iterator](), и именно его next() генерирует значения.

Таким образом, **итератор отделен от самого итерируемого объекта**. Технически, можно объединить их и использовать сам range как итератор, чтобы упростить код. Например, вот так:

let range = {

  from: 1,

  to: 5,

  [Symbol.iterator]() {

this.current = this.from;

return this;

  },

  next() {

if (this.current <= this.to) {

   return { done: false, value: this.current++ };

} else {

   return { done: true };

}

  }

};

for (let num of range) {

  alert(num); // 1, затем 2, 3, 4, 5

}

Теперь range[Symbol.iterator]() возвращает сам объект range: у него есть необходимый метод next() и он запоминает текущее состояние итерации в this.current.

Можно сделать бесконечный итератор. Например, range будет бесконечным при range.to = Infinity. Или можно создать итерируемый объект, который генерирует бесконечную последовательность псевдослучайных чисел. Это бывает полезно.

***Итерируемые объекты*  – это объекты, которые реализуют метод Symbol.iterator**, как было описано выше. ***Псевдомассивы* – это объекты, у которых есть индексы и свойство length**, то есть они выглядят как массивы.

А вот объект, который является псевдомассивом, но его нельзя итерировать:

let arrayLike = { // есть индексы и свойство length => псевдомассив

  0: "Hello",

  1: "World",

  length: 2

};

// Ошибка (отсутствует Symbol.iterator)

for (let item of arrayLike) {}

И итерируемые объекты, и псевдомассивы – это обычно не массивы, у них нет методов push, pop и т.д. Довольно неудобно, если есть такой объект и надо работать с ним как с массивом.

Есть универсальный метод [**Array.from**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/from), который принимает итерируемый объект или псевдомассив и делает из него «настоящий» Array. После этого уже можно использовать методы массивов. Например:

let arrayLike = {

  0: "Hello",

  1: "World",

  length: 2

};

let arr = Array.from(arrayLike); // (\*)

alert(arr.pop()); // World (метод работает)

Array.from в строке (\*) принимает объект, проверяет, является ли он итерируемым объектом или псевдомассивом, затем создаёт новый массив и копирует туда все элементы. То же самое происходит с итерируемым объектом:

let arr = Array.from(range);

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

31. Преобразование объектов. (Т4.3)

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*. Результатом obj1 + obj2 (или другой математической операции) *не может быть другой объект*!

**Правила преобразования объектов:**

1. Не существует преобразования к логическому значению.  Все объекты в логическом контексте **являются true**. Существуют лишь их **численные и строковые преобразования**.

2.  Численные преобразования происходят, когда осуществляется вычитание объектов или выполняются другие математические операции. Например, объекты Date могут вычитаться и результатом date1 - date2 будет временной отрезок между двумя датами.

3.   Что касается строковых преобразований – они обычно происходят, при выводе объекта по типу alert(obj) и в подобных случаях.

[**Преобразование к примитивам**](https://learn.javascript.ru/object-toprimitive#preobrazovanie-k-primitivam)

Можно настраивать строковые и численные преобразования, используя специальные методы объекта. Существуют три варианта преобразований («три хинта»), описанные в [спецификации](https://tc39.github.io/ecma262/#sec-toprimitive):

- **"string"** – для преобразования объекта к строке, когда операция ожидает получить строку, например, alert:

alert(obj);

// объект в качества имени свойства

anotherObj[obj] = 123;

- **"number"** – для преобразования объекта к числу, в случае математических операций:

// явное преобразование

let num = Number(obj);

// математическое (исключая бинарный "+")

let n = +obj; // унарный плюс

let delta = date1 - date2;

// больше/меньше сравнения

let greater = user1 > user2;

- **"default"** – происходит редко, когда оператор «не уверен», какой тип ожидать.

Например, **бинарный плюс +** может работать с обоими типами: строками (конкатенировать их) и числами (складывать). Таким образом, и те, и другие будут вычисляться. Или, когда происходит сравнение объектов с помощью **нестрогого равенства ==** со строкой, числом или символом, и неясно какое преобразование должно быть выполнено.

// бинарный плюс

let total = car1 + car2;

// obj == string/number/symbol

if (user == 1) { ... };

Оператор **больше/меньше <>** также может работать, как со строками, так и с числами. Однако, по историческим причинам он *использует хинт «number», а не «default».*

На практике **все встроенные объекты, исключая Date,** реализуют "default" преобразования тем же способом, что и "number".

Обратите внимание, что существуют лишь **три варианта хинтов**. Не существует хинта со значением «boolean» (все объекты являются true в логическом контексте) или каких-либо ещё.

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

1.   Вызывает **obj[Symbol.toPrimitive](hint)** – метод с символьным ключом Symbol.toPrimitive (системный символ), если такой метод существует, и передаёт ему хинт.

2.   Иначе, если хинт равен "string", пытается вызвать **obj.toString()**, а если его нет, то **obj.valueOf()**, если он существует.

3.   В случае, если хинт равен "number" или "default", пытается вызвать **obj.valueOf()**, а если его нет, то **obj.toString()**, если он существует.

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

Метод используется для всех преобразований:

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

[**Методы toString/valueOf**](https://learn.javascript.ru/object-toprimitive#metody-tostring-valueof)

Методы toString и valueOf существуют давно. Они не символы, а просто обычные методы объектов со строковыми именами. Они предоставляют «устаревший» способ реализации преобразований объектов.

Если нет метода Symbol.toPrimitive, движок JavaScript пытается найти эти методы и вызвать следующим образом:

- toString – valueOf для хинта со значением «string».

- valueOf – toString – в ином случае.

Для примера, используем их в реализации всё того же объекта user. Воспроизведём его поведение комбинацией методов toString и valueOf:

let user = {

  name: "John",

  money: 1000,

   // "string"

  toString() {

return `{name: "${this.name}"}`;

  },

   // "number" или "default"

  valueOf() {

return this.money;

  }};

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

alert(+user); // valueOf -> 1000

alert(user + 500); // valueOf -> 1500

Как видно, получилось то же поведение, что и у предыдущего примера с Symbol.toPrimitive.

Часто надо описать одно универсальное преобразование объекта к примитиву, для всех ситуаций. Для этого достаточно создать один toString:

let user = {

  name: "John",

  toString() {

return this.name;

  }};

alert(user); // toString -> John

alert(user + 500); // toString -> John500

В отсутствие Symbol.toPrimitive и valueOf, **toString обработает все случаи преобразований к примитивам**.

Важно понимать, что все описанные методы для преобразований объектов не обязаны возвращать именно «требуемый хинтом» тип примитива. Нет требований, чтобы toString() возвращал строго строку, или к тому, чтобы метод Symbol.toPrimitive возвращал число для хинта равного «number». Единственное обязательное требование**: методы должны возвращать примитив, а не объект.**

Если toString или valueOf вернёт объект, то ошибки не будет, но такое значение будет проигнорировано (как если бы метода вообще не существовало). Метод Symbol.toPrimitive, напротив, обязан возвращать примитив, иначе будет ошибка.

[**Последующие операции**](https://learn.javascript.ru/object-toprimitive#posleduyuschie-operatsii)

Операция, инициализировавшая преобразование, получает примитив, и затем продолжает работу с ним, производя дальнейшие преобразования, если это необходимо. Например:

- Математические операции, исключая бинарный плюс, преобразуют примитив к числу:

let obj = {

toString() {

return "2";

}

};

alert(obj \* 2); // 4

- Бинарный плюс + в аналогичном случае сложит строки:

let obj = {

toString() {

return "2";

}

};

alert(obj + 2); // 22

32. Создание объектов через "new". (Т4.4)

Иногда необходимо **создать множество однотипных объектов**. Это делается при помощи **функции-конструктора и оператора new**.

**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"*.

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

alert(user.isAdmin); // false

**Основная цель конструкторов – удобное повторное** создание однотипных объектов.

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 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() { ... }

}

\*/

33. Флаги и дескрипторы свойств. (Т4.5)

**Флаги свойств**

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

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

Эти атрибуты(флаги) обычно скрыты. Когда создается свойство «обычным способом», все эти атрибуты имеют значение true. Но можно изменить их в любое время.

**Метод Object.getOwnPropertyDescriptor** позволяет получить полную информацию о свойстве.

Чтобы изменить флаги, можно использовать **метод Object.defineProperty.** Его синтаксис:

**Object.defineProperty(obj, propertyName, descriptor)**

* *obj, propertyName* – объект и его свойство, для которого нужно применить дескриптор.
* *descriptor* – применяемый дескриптор.

Если свойство существует, defineProperty обновит его флаги. В противном случае метод создает новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение false. Например, здесь создаётся свойство name, все флаги которого имеют значение false:

let user = {};

Object.defineProperty(user, "name", {

  value: "John"

});

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

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

/\*

{

  "value": "John",

  "writable": false,

  "enumerable": false,

  "configurable": false

} \*/

**Только для чтения**

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

let user = {

  name: "John"

};

Object.defineProperty(user, "name", {

**writable: false**

});

user.name = "Pete"; // Ошибка

**Неперечисляемое свойств**о

Можно установить для свойства enumerable: false. Тогда оно перестанет появляться в цикле for..in.

let user = {

  name: "John",

  toString() {

return this.name;

  }

};

Object.defineProperty(user, "toString", {

  enumerable: false

});

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

**Неконфигурируемое свойств**о

Флаг неконфигурируемого свойства (configurable: false) иногда предустановлен для некоторых встроенных объектов и свойств. **Неконфигурируемое свойство не может быть удалено или изменено с помощью defineProperty.**

let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

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

/\*

{

  "value": 3.141592653589793,

  "writable": false,

  "enumerable": false,

  "configurable": false

}

\*/

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

Math.PI = 3; // Ошибка

Существует **метод Object.defineProperties(obj, descriptors)**, который позволяет определять множество свойств сразу. Его синтаксис:

Object.defineProperties(obj, {

  prop1: descriptor1,

  prop2: descriptor2

  // ...

});

Object.defineProperties(user, {

  name: { value: "John", writable: false },

  surname: { value: "Smith", writable: false },

  // ...

});

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться **методом Object.getOwnPropertyDescriptors(obj)**. Вместе с Object.defineProperties этот метод можно использовать для клонирования объекта вместе с его флагами:

let clone = Object.defineProperties({},Object.getOwnPropertyDescriptors(obj));

Обычно при клонировании объекта используется присваивание, чтобы скопировать его свойства:

for (let key in user) {

  clone[key] = user[key]

}

Но это не копирует флаги. Поэтому если нужен клон с флагами, предпочтительнее использовать Object.defineProperties. Другое отличие в том, что for..in игнорирует символьные свойства, а Object.getOwnPropertyDescriptors возвращает дескрипторы всех свойств, включая свойства-символы.

34. Геттеры и сеттеры. (Т4.6)

Есть два типа свойств объекта.

* **свойства-данные** (data properties). Все свойства, которые использовались до текущего момента были свойствами-данными.
* **свойства-аксессоры** (accessor properties). По своей сути *это функции, которые используются для присвоения и получения значения*, но во внешнем коде они выглядят как обычные свойства объекта.

**Геттеры и сеттеры**

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

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

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

Свойства-аксессоры не имеют value и writable(дескрипторы), 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.

Аксессоры позволяют в любой момент взять «обычное» свойство и изменить его поведение, поменяв на геттер и сеттер.

35. Декораторы. Методы call(), apply(). (Т4.7)

Есть функция slow(x), выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Вместо того, чтобы усложнять slow(x) дополнительной функциональностью заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование.

function slow(x) {

  // здесь могут быть CPU ресурсоёмкие вычисления

  alert(`Called with ${x}`);

  return x;

}

function cachingDecorator(func) {

  let cache = new Map();

  return function(x) {

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

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

}

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

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

return result;

  };

}

slow = cachingDecorator(slow);

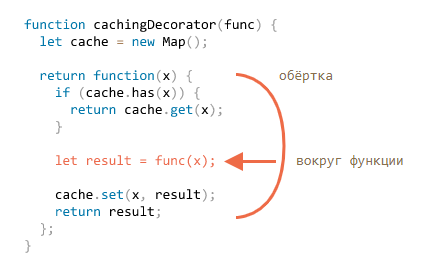
alert( slow(1) ); // slow(1) кешируем

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

alert( slow(2) ); // slow(2) кешируем

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

**cachingDecorator** – это **декоратор**, специальная функция, которая принимает другую функцию и изменяет её поведение. Идея состоит в том, что можно вызвать cachingDecorator с любой функцией, в результате чего **получим кеширующую обёртку(кеширует результаты).** Результат вызова cachingDecorator(func) является «обёрткой», т.е. function(x) «оборачивает» вызов func(x) в кеширующую логику:



В итоге

- Функцию cachingDecorator можно использовать повторно. Можно применить её к другой функции.

- Логика кеширования является отдельной, она не увеличивает сложность самой slow (если таковая была).

- При необходимости можно объединить несколько декораторов

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

Существует специальный встроенный метод функции func.*call*(context, arg), который позволяет вызывать функцию, явно устанавливая this(первый аргумент) и последующие аргументы (arg1, arg2 …)

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

Пример:

function sayHi() {

  alert(this.name);

}

let user = { name: "John" };

let admin = { name: "Admin" };

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

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

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

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. Фундаментальное различие между call и apply  заключается в том, что функция **call() принимает список аргументов, а apply() принимает псевдомассив.**

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

func.apply(context, args);

36. Привязка контекста. (Т4.8)

При передаче методов объекта в качестве колбэков, например, для setTimeout, возникает известная проблема – потеря this. Как только метод передается отдельно от объекта – this теряется.

let user = {

  firstName: "Вася",

  sayHi() {

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

  }

};

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

Так выглядит потеря контекста. Чтобы его не терять нужно добавить при вызове user.sayHi() скобки

Как можно не терять контекст ещё(вызывать метод в правильном контексте):

* сделать функцию-обёртку, обернуть вызов в анонимную функцию, создав замыкание  
  setTimeout(() => user.sayHi(), 1000); // Привет, Вася!
* использовать **bind** в привязке контекста**.** 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.

let boundFunc = func.bind(context);

Результатом вызова func.bind(context) является особый «экзотический объект», 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.

let user = {

firstName: "Вася"

};

function func() {

alert(this.firstName);

}

let funcUser = func.bind(user);

funcUser(); // Вася

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.

function mul(a, b) {

return a \* b;

}

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

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.

37. Функции setTimeout and setInterval. (Т4.9)

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

-     **setTimeou**t позволяет вызвать функцию один раз через определённый интервал времени.

-     **setInterval** позволяет вызывать функцию регулярно, повторяя вызов через определённый интервал времени.

**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-)

**Метод setInterval**

Метод setInterval имеет такой же синтаксис как setTimeout:

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

38. Прототипное наследование. Собственные и унаследованные свойства. (Т5.1)

**Свойство** [**[[Prototype]]**](https://learn.javascript.ru/prototype-inheritance#prototype)

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

Если **при чтении свойство из object отсутствует, JavaScript автоматически берет его из прототипа. В программировании такой механизм называется *прототипным наследованием***. Многие возможности языка и техники программирования основываются на нем.

Свойство [[Prototype]] является внутренним и скрытым, но есть много способов задать его. Одним из них является использование \_\_proto\_\_, например так:

let animal = {

  eats: true

};

let rabbit = {

  jumps: true

};

rabbit.\_\_proto\_\_ = animal;

Свойство \_\_proto\_\_ – не то же самое, что [[Prototype]]. Это геттер/сеттер для него. Он существует по историческим причинам, в современном языке его заменяют функции Object.getPrototypeOf/Object.setPrototypeOf, которые также получают/устанавливают прототип. По спецификации \_\_proto\_\_ должен поддерживаться только браузерами, но по факту все среды, включая серверную, поддерживают его.

В примере ниже осуществляется поиск свойства в rabbit, а оно отсутствует, и JavaScript автоматически берет его из animal

let animal = {

  eats: true

};

let rabbit = {

  jumps: true

};

rabbit.\_\_proto\_\_ = animal; // (\*)

alert( rabbit.eats ); // true (\*\*)

alert( rabbit.jumps ); // true

Есть только два ограничения:

1.  Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если попытаться назначить \_\_proto\_\_ по кругу.

2.  Значение \_\_proto\_\_ может быть объектом или null. Другие типы игнорируются.

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

**Прототип используется только для чтения свойств. Операции записи/удаления работают напрямую с объектом.**

39. Свойство F.prototype. (Т5.2)

**Чтобы новым объектам автоматически ставить прототип, конструктору ставится свойство prototype.  
При создании объекта через new, в его прототип \_\_proto\_\_ записывается ссылка из prototype функции-конструктора.**

Свойство с именем prototype можно указать на любом объекте, но особый смысл оно имеет, лишь если назначено функции-конструктору.

Само по себе, без вызова оператора new, оно вообще ничего не делает, его единственное назначение – указывать \_\_proto\_\_ для новых объектов.

**Значением prototype может быть только объект**

Технически, в это свойство можно записать что угодно.

Однако, при работе new, свойство prototype будет использовано лишь в том случае, если это объект. Примитивное значение, такое как число или строка, будет проигнорировано.

**У каждой функции по умолчанию уже есть свойство prototype.**

Оно содержит объект такого вида:

function Rabbit() {}

Rabbit.prototype = {  
constructor: Rabbit  
};

В коде выше я создал Rabbit.prototype вручную, но ровно такой же – генерируется автоматически.

**Для произвольной функции** – назовём её Person, верно следующее:

* Прототип \_\_proto\_\_ новых объектов, создаваемых через new Person, можно задавать при помощи свойства Person.prototype.
* Значением Person.prototype по умолчанию является объект с единственным свойством constructor, содержащим ссылку на Person. Его можно использовать, чтобы из самого объекта получить функцию, которая его создала. Однако, JavaScript никак не поддерживает корректность этого свойства, поэтому программист может его изменить или удалить.
* Современный метод Object.create(proto) можно эмулировать при помощи prototype, если хочется, чтобы он работал в IE8-.

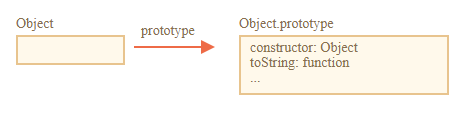
40. Встроенные прототипы. (Т5.3)

Выведем пустой объект:

let obj = {};

alert( obj ); // "[object Object]" ?

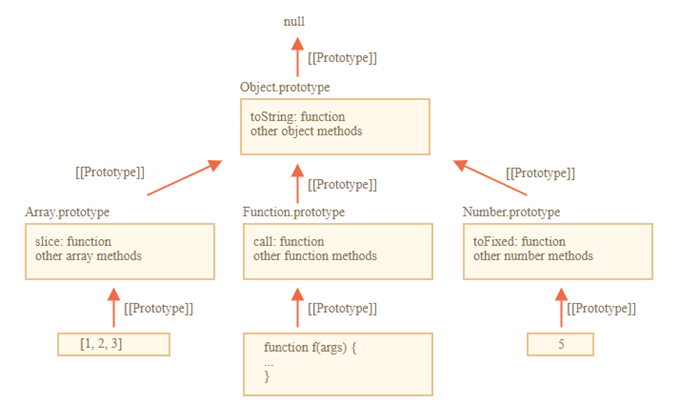
В рассматриваемом примере **нету кода, который генерирует строку "[object Object]"**. Понятно, что это встроенный метод toString, но явно не видно, где он объявлен,  ведь obj  пуст. Дело в том, что краткая нотация obj = {} это то же самое, что и obj = new Object(), где Object – встроенная функция-конструктор для объектов с собственным свойством prototype, который ссылается на огромный объект с методом toString и другими. Вот что происходит:



Таким образом, когда вызывается **obj.toString(), метод берётся из Object.prototype. То есть существую встроенные методы по умолчанию.**

**Другие встроенные объекты, такие как Array, Date, Function** и другие, также хранят свои методы в прототипах. Например, при создании массива [1, 2, 3] внутренне используется конструктор массива Array. Поэтому прототипом массива становится Array.prototype, предоставляя ему свои методы. Это позволяет эффективно использовать память.

Согласно спецификации, наверху иерархии встроенных прототипов находится Object.prototype. Поэтому иногда говорят, что «всё наследует от объектов».



Встроенные прототипы можно изменять. Например, если добавить метод к String.prototype, метод становится доступен для всех строк:

String.prototype.show = function() {  
alert(this);  
};"BOOM!".show(); // BOOM!

41. Методы прототипов. (Т5.4)

Свойство \_\_proto\_\_ считается устаревшим, и по стандарту должно поддерживаться только браузерами. **Современные методы это:**

-   [**Object.create(proto, [descriptors])**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/create) – создаёт пустой объект со свойством [[Prototype]], указанным как proto, и необязательными дескрипторами свойств descriptors.

-   [**Object.getPrototypeOf(obj)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf) – возвращает свойство [[Prototype]] объекта obj.

-   [**Object.setPrototypeOf(obj, proto)**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf) – устанавливает свойство [[Prototype]] объекта obj как proto.

let animal = {

  eats: true

};

let rabbit = Object.create(animal);

alert(rabbit.eats); // true

alert(Object.getPrototypeOf(rabbit) === animal);

Object.setPrototypeOf(rabbit, {});

У Object.create есть необязательный второй аргумент: дескрипторы свойств. Можно добавить дополнительное свойство новому объекту таким образом:

let animal = {

  eats: true

};

let rabbit = Object.create(animal, {

  jumps: {

value: true

  }

});

alert(rabbit.jumps); // true

Также можно использовать **Object.create для глубокого клонирования** объекта, более мощного, чем копирование свойств в цикле for..in:

**let clone = Object.create(Object.getPrototypeOf(obj),** Object.getOwnPropertyDescriptors(obj));

 Такой вызов создаёт точную копию объекта obj, включая все свойства: перечисляемые и не перечисляемые, свойства, геттеры/сеттеры для свойств – и все это с правильным свойством [[Prototype]].

Технически, можно установить/получить [[Prototype]] в любое время. Но обычно прототип устанавливается только раз во время создания объекта, а после не меняется: rabbit наследует от animal, и это не изменится.

42. Классы. Class Expression. Приватные и защищенные методы и свойства. (Т5.5, Т5.8)

**Класс** – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

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");

  }

};

**Аналогично Named Function Expression, Class Expression может иметь имя. Если у Class Expression есть имя, то оно видно только внутри класса.**

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

* **Внутренний** интерфейс – методы и свойства, доступные из других методов класса, но не снаружи класса.Внутренний интерфейс **используется для работы объекта**, его методы, свойства используют друг друга.
* **Внешний** интерфейс – методы и свойства, доступные снаружи класса.

 Всё, что **нужно для использования объекта**, это знать его внешний интерфейс. Совершенно не обязательно знать, его внутреннюю структуру и логику.

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

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

Во многих других языках также существуют «**защищённые**» поля, доступные только внутри класса или для дочерних классов (то есть, как приватные, но разрёшен доступ для наследующих классов) и также полезны для внутреннего интерфейса. В некотором смысле они более распространены, чем приватные, потому что обычно надо, чтобы наследующие классы получали доступ к внутренним полям.

Защищённые поля не реализованы в JavaScript на уровне языка, но на практике они очень удобны, поэтому их эмулируют.

Защищённые свойства обычно начинаются с префикса \_ ( это не синтаксис, а правило)

Приватные свойства и методы должны начинаться со знака #

43. Наследование классов. Переопределение методов. Статические свойства и методы. (Т5.6, Т5.7, кроме: [[HomeObject]], Методы не «свободны»)

Для того, **чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед** {..}. Ключевое слово extends работает, используя прототипы. Оно устанавливает Rabbit.prototype.[[Prototype]] в Animal.prototype. Так что если метод не найден в Rabbit.prototype, JavaScript берёт его из Animal.prototype.

Синтаксис создания класса допускает указывать после extends не только класс, но любое выражение. Например, можно написать так:

extends function(){this.id = 1;}

В таком случае класс наследует результат функции,в примере объект со свойством id = 1.

**Для переопределения метода** суперкласса в классе-потомке, необходимо создать метод с таким же именем, как в суперклассе, в классе-потомке.

Бывают случаи, когда нужны не изменить функционал метода полностью, а что-то добавить. В таком случае в переопределении метода используют такие методы, как:

* **super.method(...)** - вызывает родительский метод (method - название метода, который переопределяется)
* **super()** - вызывает родительский конструктор

Мы также можем присвоить метод самой функции-классу, а не её "prototype". Такие методы называются статическими.

В классе такие методы обозначаются ключевым словом static.

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

class User { }

User.staticMethod = function() {

  alert(this === User);

};

Значением this при вызове User.staticMethod() является сам конструктор класса User (правило «объект до точки»).

**Статические члены класса используются без создания объекта**. (Думаю помним из курса ООП, ок?) Статические члены классы наследуются классами-потомками.

44. Расширение встроенных классов. Оператор instanceof. (Т5.9, Т5.10, кроме Свойство Symbol.toStringTag)

От встроенных классов, таких как Array, Map и других, тоже можно наследовать. Например, в этом примере PowerArray наследуется от встроенного Array:

**class PowerArray extends Array** {

  isEmpty() {

    return this.length === 0;

  }

}

let arr = new PowerArray(1, 2, 5, 10, 50);

alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);

alert(filteredArr); // 10, 50

alert(filteredArr.isEmpty()); // false

Обратите внимание: встроенные методы, такие как filter, map и другие возвращают новые объекты унаследованного класса PowerArray. Их внутренняя реализация такова, что для этого они используют свойство объекта constructor.

В примере выше arr.constructor === PowerArray. Поэтому при вызове метода arr.filter() он внутри создаёт массив результатов, именно используя arr.constructor, а не обычный массив. Это удобно, поскольку можно продолжать использовать методы PowerArray далее на результатах.

Более того, можно настроить это поведение. При помощи специального статического геттера **Symbol.species можно вернуть конструктор**, который JavaScript будет использовать в filter, map и других методах для создания новых объектов. Если надо, чтобы методы map, filter и т. д. возвращали обычные массивы, то нужно вернуть Array в Symbol.species, вот так:

class PowerArray extends Array {

  isEmpty() {

    return this.length === 0;

  }

  // встроенные методы массива будут использовать этот метод как конструктор

  static get [Symbol.species]() {

    return Array;

  }

}

let arr = new PowerArray(1, 2, 5, 10, 50);

alert(arr.isEmpty()); // false

// filter создаст новый массив, используя arr.constructor[Symbol.species] как конструктор

let filteredArr = arr.filter(item => item >= 10);

// filteredArr не является PowerArray, это Array

alert(filteredArr.isEmpty()); // Error: filteredArr.isEmpty is not a function

Теперь .filter возвращает Array. Расширенная функциональность не будет передаваться далее.

Другие коллекции, такие как Map, Set, работают аналогично. Они также используют Symbol.species.

Оператор **instanceof** позволяет проверить, **к какому классу принадлежит объект**, с учётом наследования.

Такая проверка может потребоваться во многих случаях. Здесь мы используем её для создания полиморфной функции, которая интерпретирует аргументы по-разному в зависимости от их типа. Обычно оператор **instanceof просматривает для проверки *цепочку прототипов***. Но это поведение может быть изменено при помощи статического метода Symbol.hasInstance.

45. Модули. Основные возможности модулей. (Т5.14)

**Модуль обычно содержит класс или набор функций/методов**. Реализация с помощью:

* появились библиотеки для работы с модулями (CommonJS/Node, AMD, UMD)
* 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.

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”>.

Особенности:

* используется **use strict** режим в модулях
* каждый модуль имеет свою область видимости
* модули должны экспортировать функционал, который будет использоваться извне модуля (export)
* *В браузере также существует независимая область видимости для каждого скрипта <script type="module">*
* Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам

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.

46. Модули: экспорт и импорт. (Т5.15)

Директивы экспорт и импорт имеют несколько вариантов вызова.

**Экспорт до объявления**

export function foo(){ alert(“Hello!”);}

**Экспорт отдельно от объявления**

export {foo, others\_objects...}

**Импорт \***

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

// main.js

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

или

import \* as say from ‘.say.js’;

Во втором случае из модуля say.js импортируются все объекты, которые он экспортирует, в объект say.

**Импорт “как”**

import {sayHi as Hi, sayBye **as** Bye} from ‘./say.js’;

**Экспортировать «как»**

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

// say.js

export {sayHi **as** hi, sayBye as bye};

**Экспорт по умолчанию**

На практике модули встречаются в основном одного из двух типов:

1.Модуль, содержащий библиотеку или набор функций, как say.js выше.

2.Модуль, который объявляет что-то одно, например, модуль user.js экспортирует только class User.

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

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

// user.js

export default class User {

  constructor(name) {

    this.name = name;

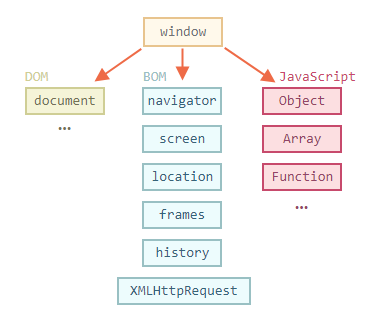
  }

}

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

47. Окружение: DOM, BOM. Дерево DOM. (Т6.1, Т6.2, кроме: Взаимодействие с консолью)

Браузеры, например, дают средства для управления веб-страницами.



window - глобальный объект и окно браузера.

**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);

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

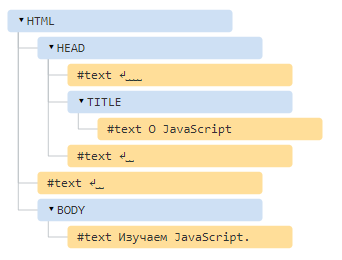
Объект navigator даёт информацию о самом браузере и операционной системе. Среди множества его свойств самыми известными являются: navigator.userAgent – информация о текущем браузере, и navigator.platform – информация о платформе (может помочь в понимании того, в какой ОС открыт браузер – Windows/Linux/Mac и так далее).

Объект location позволяет получить текущий URL и перенаправить браузер по новому адресу.

Функции alert/prompt/confirm также являются ф-ми BOM.

**Дерево ДОМ**

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



 Каждый узел этого дерева – это объект.

Существует 12 типов узлов. Но на практике в основном работают с четырьмя из них:

1.document – «входная точка» в DOM.

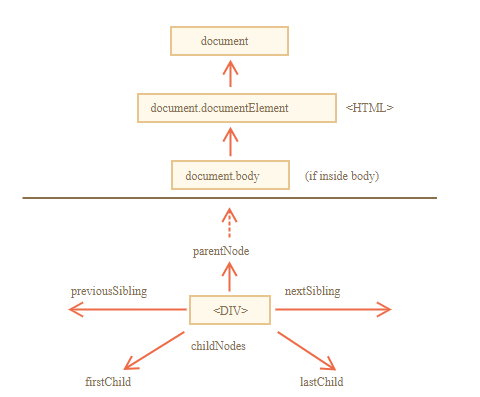
2.Узлы-элементы – HTML-теги, основные строительные блоки.

3.Текстовые узлы – содержат текст.

4.Комментарии – иногда в них можно включить информацию, которая не будет показана, но доступна в DOM для чтения JS.

48. Навигация и методы поиска DOM-элементов. (Т6.3, Т6.4)

Так выглядят основные ссылки, по которым можно переходить между DOM-элементами.



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

Дети: childNodes, firstChild, lastChild. Можно применять  к любому DOM-элементу. которые имеет потомки или детей (document.body.lastChild for example).

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

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

1. Можно перебирать в for..of
2. Нельзя использовать методы массивов, это ж не массив, а псевдомассив, но можно коллекцию преобразовать в массив при помощи Array.from(...)

DOM-коллекции и все навигационные свойства, приведенные выше, используются только для чтения.

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

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

// родителем <body> является <html>

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

// после <head> идет <body>

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

// перед <body> находится <head>

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

Элемент <table>, в дополнение к свойствам, о которых речь шла выше, поддерживает следующие:

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

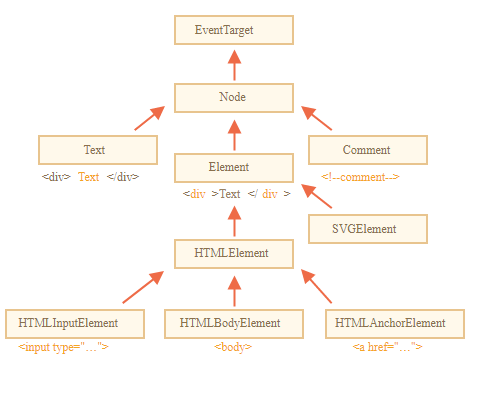
**Методы поиска (просто перечислю, их все знают):**

* document.getElementById
* document.getElementsByClass
* document.getElementsByTagName
* document.querySelector
* document.querySelectorAll

49. Свойства узлов: тип, тег и содержимое. (Т6.5)

**Классы DOM-узлов**

У разных DOM-узлов могут быть разные свойства. Текстовые узлы отличаются от узлов-элементов. Но у них есть общие свойства и методы, потому что все чклассы 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 – корневой «абстрактный» класс**. Его объекты никогда не создаются. Он служит основой для поддерживания DOM-узлами «событий».

**Node – «абстрактный» класс, основа DOM**-узлов. Обеспечивает базовую функциональность: **parentNode, nextSibling, childNodes** и т.д. (это геттеры). Его объекты никогда не создаются. Есть определенные классы узлов, которые наследуют от него: Text, Element, Comment.

**Element** –базовый класс DOM-элементов. Обеспечивает навигацию на уровне элементов: **nextElementSibling, children и методы поиска: getElementsByTagName, querySelector**. Браузер поддерживает HTML, XML и SVG. Класс Element – база для следующих классов: SVGElement, XMLElement и HTMLElement.

**HTMLElement** –базовый 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, также можно просто привести его к строке, проверить наследование можно также при помощи instanceof.

**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.

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.

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.

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

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

На практике этим следует пользоваться с большой осторожностью, так как фактически происходит не добавление, а перезапись.

Другими словами, innerHTML+= делает следующее:

Старое содержимое удаляется.

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.

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

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.

Иногда комментарии используют для вставки информации и инструкций шаблонизатора в HTML.

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>.

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.

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.

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

**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.

50. Атрибуты и DOM-свойства. (Т6.6)

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

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

**DOM-свойства**

Можно добавить своё собственное свойство. DOM-узлы – это обычные объекты JavaScript. Можно их изменять. Можно добавить и метод. Можно изменять встроенные прототипы, такие как Element.prototype и добавлять новые методы ко всем элементам.

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

**HTML-атрибуты**

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

Пожалуйста, учтите, что стандартный атрибут для одного тега может быть нестандартным для другого. Например, атрибут "type" является стандартным для элемента <input> (HTMLInputElement), но не является стандартным для <body>(HTMLBodyElement). Стандартные атрибуты описаны в спецификации для соответствующего класса элемента.

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

**elem.hasAttribute(name)** – проверяет наличие атрибута;

**elem.getAttribute(name)** – получает значение атрибута;

**elem.setAttribute(name, value)** – устанавливает значение атрибута;

**elem.removeAttribute(name)** – удаляет атрибут.

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

Кроме этого, получить все атрибуты элемента можно с помощью свойства **elem.attributes**: коллекция объектов, которая принадлежит ко встроенному классу Attr со свойствами name и value.

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

Обратите внимание:

**Имена атрибутов регистроНЕзависимы.**

Можно присвоить что угодно атрибуту, но это станет строкой.

Все атрибуты, в том числе те, которые были установлены, видны в outerHTML.

**Коллекция attributes является перебираемой**. В ней есть все атрибуты элемента (стандартные и нестандартные) в виде объектов со свойствами name и value.

**Синхронизация между атрибутами и свойствами**

Когда стандартный атрибут изменяется, соответствующее свойство автоматически обновляется. Это работает и в обратную сторону (за некоторыми исключениями).

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

Иногда эта «особенность» может пригодиться, потому что действия пользователя могут приводить к изменениям value, и если после этого надо восстановить «оригинальное» значение из HTML, оно будет в атрибуте.

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

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

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

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

**Нестандартные атрибуты, dataset**

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

Также они могут быть использованы, чтобы стилизовать элементы.

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

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

Чтобы избежать конфликтов, существуют **атрибуты вида data-\*. Все атрибуты, начинающиеся с префикса «data-», зарезервированы для использования программистами. Они доступны в свойстве dataset.** Например, если у elem есть атрибут "data-about", то обратиться к нему можно как elem.dataset.about.

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

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

51. Добавление и удаление DOM-узлов. (Т6.7)

**Создание элемента**

DOM-узел можно создать двумя методами:

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

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

**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).

**Метод для различных вариантов вставки:**

node.**append**(...узлы или строки) – добавляет узлы или строки в конец node;

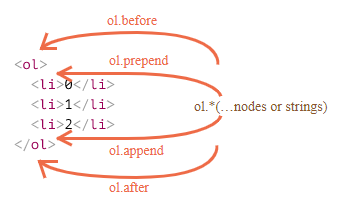
node.**prepend**(...nodes or strings) – вставляет узлы или строки в начало node;

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.

Наглядная иллюстрация того, куда эти методы вставляют:



Эти методы могут вставлять несколько узлов и текстовых фрагментов за один вызов.

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,

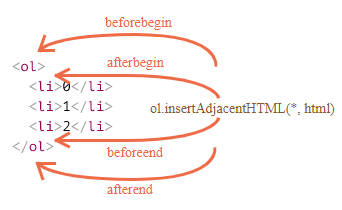
"afterbegin" – вставить html в начало elem,

"beforeend" – вставить html в конец elem,

"afterend" – вставить html непосредственно после elem.

Второй параметр – это HTML-строка, которая будет вставлена именно «как HTML».

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.

**Удаление узлов**

Для удаления узла есть методы **node.remove().**

Если нужно переместить элемент в другое место – нет необходимости удалять его со старого. Все методы вставки автоматически удаляют узлы со старых мест.

**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.

**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.

DocumentFragment редко используется. Нет смысла добавлять элементы в специальный вид узла, если вместо этого можно вернуть массив узлов.

52. Стили DOM-узлов. (Т6.8)

Как правило, существует два способа задания стилей для элемента:

1. Создать класс в CSS и использовать его: <div class="...">.

2. Писать стили непосредственно в атрибуте style: <div style="...">.

JavaScript может менять и классы и свойство style. Классы – всегда предпочтительный вариант по сравнению со style и хотим установить их из JavaScript.

В других случаях, например. Свойством style стоит  манипулировать только в том случае, если классы «не могут справиться». Например, использование style является приемлемым, если вычисляются координаты элемента динамически, чтобы сделать текст красным, добавить значок фона – надо описать это в CSS и добавить класс (JavaScript может это сделать). Это более гибкое и легкое в поддержке решение.

**className и classList**

Изменение класса является одним из наиболее часто используемых действий в скриптах. Свойство "className": elem.className соответствует атрибуту "class".

Если присвоить что-то elem.className, то это заменяет всю строку с классами. Иногда это то, что нужно, но часто надо добавить/удалить один класс. Для этого есть другое свойство: elem.classList.

**elem.classList** – это специальный объект с методами для добавления/удаления одного класса.

Так что можно работать как со строкой полного класса, используя className, так и с отдельными классами, используя classList.

Методы classList:

-      elem.classList.**add/remove**("class") – добавить/удалить класс.

-      elem.classList.**toggle**("class") – добавить класс, если его нет, иначе удалить.

-      elem.classList.**contains**("class") – проверка наличия класса, возвращает true/false.

Кроме того, classList является перебираемым, поэтому можно перечислить все классы при помощи for..of.

**Свойство  Element style**

Свойство **elem.style – это объект, который соответствует тому, что написано в атрибуте "style"**. Установка стиля elem.style.width="100px" работает так же, как наличие в атрибуте style строки width:100px. Для свойства из нескольких слов используется **camelCase**:

background-color  => elem.style.backgroundColor

z-index       => elem.style.zIndex

border-left-width => elem.style.borderLeftWidth

 Стили с браузерным префиксом, например, -moz-border-radius, -webkit-border-radius преобразуются по тому же принципу: **дефис означает прописную букву.**

Иногда нужно добавить свойство стиля, а потом, позже, убрать его. Например, чтобы скрыть элемент, можно задать elem.style.display = "none". Затем можно удалить свойство style.display, чтобы вернуться к первоначальному состоянию. Вместо delete elem.style.display  надо присвоить ему пустую строку: elem.style.display = "".

Если установить в style.display пустую строку, то браузер применит CSS-классы и встроенные стили, как если бы такого свойства style.display вообще не было.

Обычно используется style.\* для присвоения индивидуальных свойств стиля. Нельзя установить список стилей как, например, div.style="color: red; width: 100px", потому что div.style – это объект, и он доступен только для чтения. Для задания нескольких стилей в одной строке используется специальное свойство style.cssText.

Это свойство редко используется, потому что такое присваивание удаляет все существующие стили: оно не добавляет, а заменяет их. Можно случайно удалить что-то нужное. Но его можно использовать, к примеру, для новых элементов, когда точно известно, что не удалится существующий стиль. То же самое можно сделать установкой атрибута: div.setAttribute('style', 'color: red...').

Не забывайте добавлять к значениям единицы измерения. Например, надо устанавливать 10px, а не просто 10 в свойство elem.style.top, иначе это не сработает.

Обратите внимание, браузер «распаковывает» свойство style.margin и выводит style.marginLeft и style.marginTop из него.

**Вычисленные стили: getComputedStyle**

Метод getComputedStyle позволяет получить текущее значение свойств элемента.Синтаксис:

getComputedStyle(element, [pseudo])

**element** –  элемент, значения для которого нужно получить,

**pseudo** – указывается, если нужен стиль псевдоэлемента, например ::before. Пустая строка или отсутствие аргумента означают сам элемент.

Результат вызова – объект со стилями, похожий на elem.style, но с учётом всех CSS-классов.

Есть две концепции в CSS:

1.   **Вычисленное (computed) значение** – это то, которое получено после применения всех CSS-правил и CSS-наследования. Например, height:1em или font-size:125%.

2.   **Окончательное (resolved) значение** – непосредственно применяемое к элементу. Значения 1em или 125%являются относительными. Браузер берёт вычисленное значение и делает все единицы измерения фиксированными и абсолютными, например, height:20px or font-size:16px. Для геометрических свойств разрешенные значения могут иметь плавающую точку, например, width:50.5px.

Изначально getComputedStyle был создан для получения вычисленных значений, но окончательные значения гораздо удобнее, и стандарт изменился. Так что, в настоящее время getComputedStyle фактически возвращает окончательное значение свойства, для геометрии оно обычно в пискселях.

Для правильного получения значения нужно указать точное свойство. Например: paddingLeft, marginTop, borderTopWidth. При обращении к сокращенному: padding, margin, border – правильный результат не гарантируется.

Некоторые браузеры (Chrome) отображают 10px в документе ниже, а некоторые (Firefox) – нет.

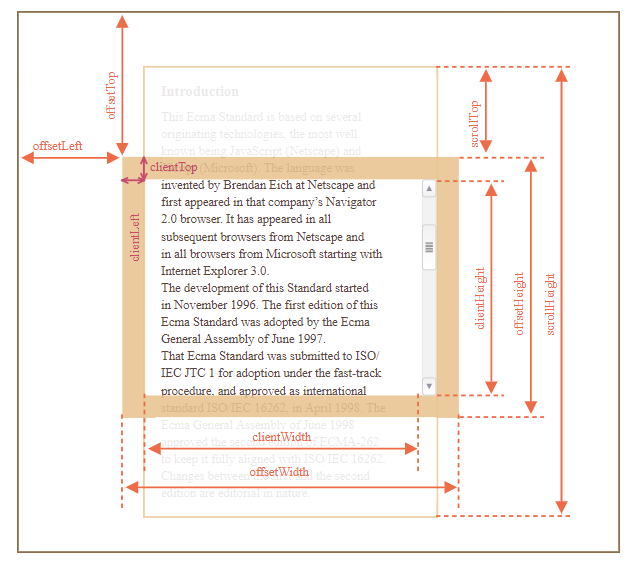
Посещенные ссылки могут быть окрашены с помощью псевдокласса :visited. Но getComputedStyle не дает доступ к этой информации, чтобы произвольная страница не могла определить, посещал ли пользователь ту или иную ссылку, проверив стили.

JavaScript не видит стили, применяемые с помощью :visited. Кроме того, в CSS есть ограничение, которое запрещает в целях безопасности применять к :visited CSS-стили, изменяющие геометрию элемента. Это гарантирует, что нет обходного пути для вредоносной страницы проверить, была ли ссылка посещена и, следовательно, нарушить конфиденциальность.

53. Размеры и прокрутка элементов и страницы. (Т6.9)

Резюмируя все, что будет ниже. JS cвойства элементов:

* **offsetParent** (ближайший предок эл-та);
* **offsetLeft**, **offsetTop** (x/y относительно верхнего левого угла offsetParent);
* **offsetWidth**, **offsetHeight** (полный размер эл-та, включая рамки);
* **clientLeft**, **clientTop**  (ширина левой/верхней рамок, но иногда+полоса прокрутки (см. пример с ивритом));
* **clientWidth**, **clientHeight** (ширина/высота области внутри рамок, включая padding, без полосы прокрутки);
* **scrollWidth**, **scrollHeight**  (полная ширина/высота контента, в т.ч. выходящая за рамки);
* **scrollLeft**, **scrollTop**  (ширина/высота прокрученной в данный момент части эл-та).



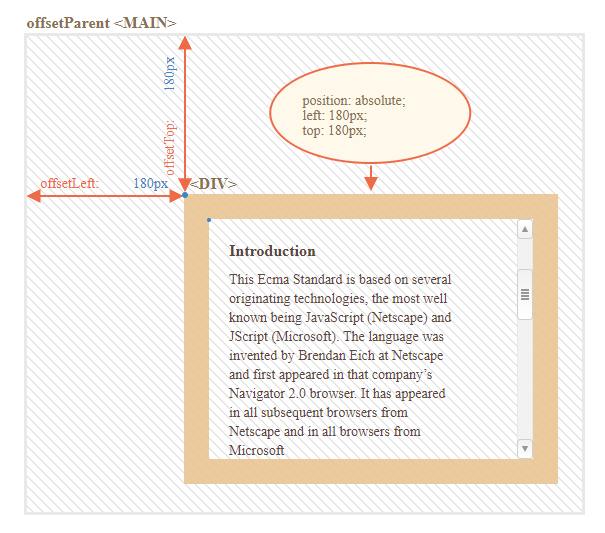
В свойстве offsetParent находится ближайший предок, который удовлетворяет условиям:

-    CSS-свойство position равно absolute, relative, fixed или sticky (по умолчанию это position: static);

-    или <td>, <th>, <table>;

-    или <body>.

Свойства offsetLeft/offsetTop содержат координаты x/y относительно верхнего левого угла offsetParent:



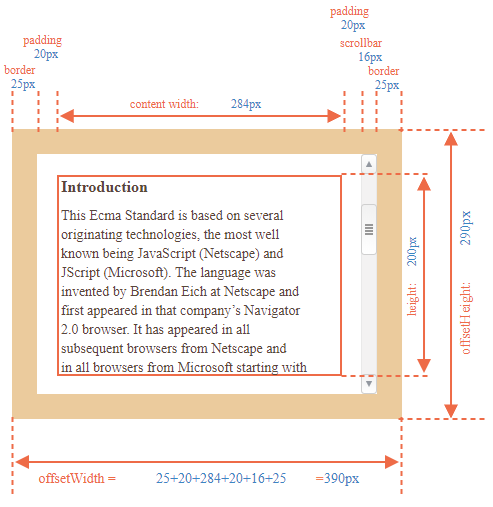
OffsetParent равно null:

1.  Для скрытых элементов (с CSS-свойством display:none или когда его нет в документе).

2.  Для элементов <body> и <html>.

3.  Для элементов с position:fixed.

Свойства [offsetWidth/Height](https://learn.javascript.ru/size-and-scroll#offsetwidth-height) содержат полный размер элемента, включая рамки.

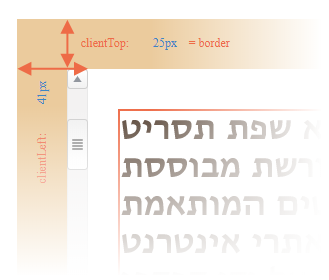
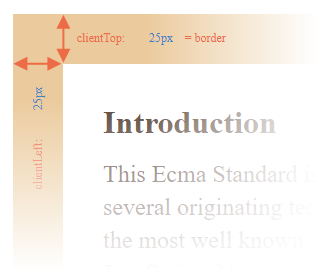


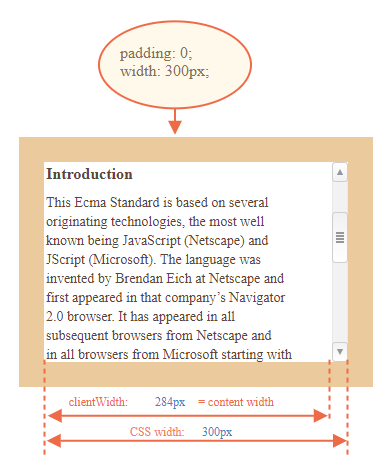
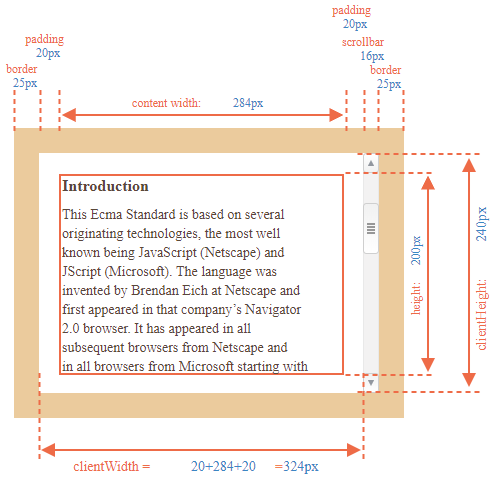
Если элемент (или любой его родитель) имеет display:none или отсутствует в документе, то все его метрики равны нулю (или null, если это offsetParent).

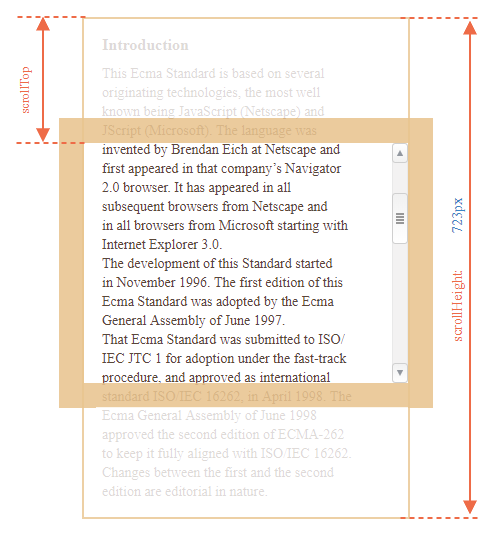
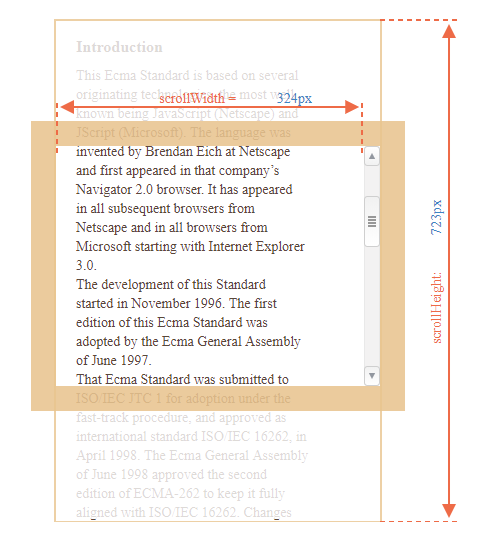
Функция isHidden также вернёт true для элементов, которые в принципе показываются, но их размеры равны нулю (например, пустые <div>).

Внутри элемента есть рамки (border). Для них есть свойства-метрики clientTop и clientLeft.

На самом деле эти свойства – не ширины рамок, а отступы внутренней части элемента от внешней. Когда документ располагается справа налево, полоса прокрутки находится слева, и тогда clientLeft включает и ширину полосы прокрутки. Пример на иврите:







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.

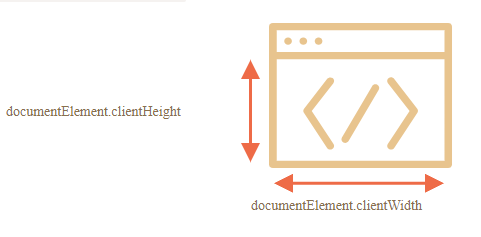
54. Размеры и прокрутка окна. Координаты. (Т6.10, Т6.11)

Резюмируя все, что будет ниже. Js-свойства для получение размеров/координат окна/страницы:

* **сlientWidth**/**clientHeight** (ширина/высота окна без полосы прокрутки; document.documentElement.clientWidth);
* **innerWidth**/**innerHeight (**ширина/высота окна с полосой прокрутки; window.innerWidth);
* body.**scrollHeight,** documentElement.scrollHeight, body.**offsetHeight,** documentElement.offsetHeight, body.**clientHeight,** documentElement.clientHeight (что-то из етова для полной высоты документы (см. ниже); document.body.scrollHeight);
* **scrollLeft/Top** (кол-во пикселей, на которое контент страницы/элемента прокручен влево/вверх; documentElement.ScrollLeft);
* **pageXOffset/pageYOffset** (кол-во пикселей, на которое прокручен документ по горизонтали/вертикали; только для чтения. отличие от предыдущего в обращении походу: window.pageXOffset);
* **scrollBy(x,y)** (метод прокручивает страницу на указанные величины; window.scrollBy);
* **scrollTo(pageX,pageY)** (метод прокручивает документ до указанных координат; window.scrollTo);
* **scrollIntoView(top)** (прокручивает страницу, чтобы элемент оказался вверху(top=true)/внизу; elem.scrollIntoView);
* **getBoundingClientRect()** и его св-ва x/y, width/height, top/bottom, left/right  
  (возвращает координаты в контексте окна для минимального по размеру прямоугольника, который заключает в себе элемент elem; elem.getBoundingClientRect);
* **elementFromPoint(x, y)** (возвращает самый глубоко вложенный элемент в окне, находящийся по координатам (x, y), которые должны быть видимыми; document.elementFromPoint);
* pageY = clientY + высота вертикально прокрученной части документа  
  pageX = clientX + ширина горизонтально прокрученной части документа  
  (связь координат относительно окна и относительно документа).

[Ширина/высота окна](https://learn.javascript.ru/size-and-scroll-window#shirina-vysota-okna)

**ClientWidth**/**clientHeight** - ширина/высота окна без полосы прокрутки в document.documentElement:



window.**innerWidth**/**innerHeight** ширина/высота окна с полосой прокрутки.

Геометрические свойства верхнего уровня могут работать немного иначе, если в HTML нет <!DOCTYPE HTML>. Поэтому в современном HTML всегда надо указывать DOCTYPE.

[Ширина/высота документа](https://learn.javascript.ru/size-and-scroll-window#shirina-vysota-dokumenta)

Чтобы надёжно получить полную высоту документа, следует взять максимальное из этих свойств (т.к. св-ва страницы могут работать не так как ожидается):

let scrollHeight = Math.max(

  document.body.**scrollHeight**,

  document.documentElement.scrollHeight,

  document.body.**offsetHeight**,

  document.documentElement.offsetHeight,

  document.body.**clientHeight**,

  document.documentElement.clientHeight

);

alert('Полная высота документа с прокручиваемой частью: ' + scrollHeight);

Чтобы получить текущее состояние прокрутки страницы в большинстве браузеров можно обратиться к **documentElement.scrollLeft/Top**, за исключением основанных на старом WebKit (Safari), там нужно использовать document.body вместо document.documentElement.

Текущую прокрутку можно прочитать из свойств **window.pageXOffset/pageYOffset**:

alert('Текущая прокрутка сверху: ' + window.pageYOffset);

alert('Текущая прокрутка слева: ' + window.pageXOffset);

 Эти свойства доступны только для чтения.

Для прокрутки страницы из JavaScript её DOM должен быть полностью построен.

Обычные элементы можно прокручивать, изменяя scrollTop/scrollLeft. Можно сделать то же самое для страницы в целом, используя document.documentElement.scrollTop/Left (кроме основанных на старом WebKit (Safari), где, как сказано выше, document.body.scrollTop/Left).

Метод **scrollBy(x,y)** прокручивает страницу относительно её текущего положения. Например, scrollBy(0,10)прокручивает страницу на 10px вниз.

Метод **scrollTo(pageX,pageY)** прокручивает страницу на абсолютные координаты (pageX,pageY). То есть, чтобы левый-верхний угол видимой части страницы имел данные координаты относительно левого верхнего угла документа. Это всё равно, что поставить scrollLeft/scrollTop. Для прокрутки в самое начало можно использовать scrollTo(0,0).

Вызов **elem.scrollIntoView(top)** прокручивает страницу, чтобы elem оказался вверху. У него есть один аргумент:

-    если top=true (по умолчанию), то страница будет прокручена, чтобы elem появился в верхней части окна. Верхний край элемента совмещён с верхней частью окна.

-    если top=false, то страница будет прокручена, чтобы elem появился внизу. Нижний край элемента будет совмещён с нижним краем окна.

Чтобы запретить прокрутку страницы установите **document.body.style.overflow = "hidden"**. Восстановление: document.body.style.overflow = "".

Недостатком этого способа является то, что сама полоса прокрутки исчезает. Если она занимала некоторую ширину, то теперь эта ширина освободится, и содержимое страницы расширится, текст «прыгнет», заняв освободившееся место. Это выглядит немного странно, но это можно обойти, если сравнить clientWidth до и после остановки, и если clientWidth увеличится (значит полоса прокрутки исчезла), то добавить padding в document.body вместо полосы прокрутки, чтобы оставить ширину содержимого прежней.

**Координаты.**

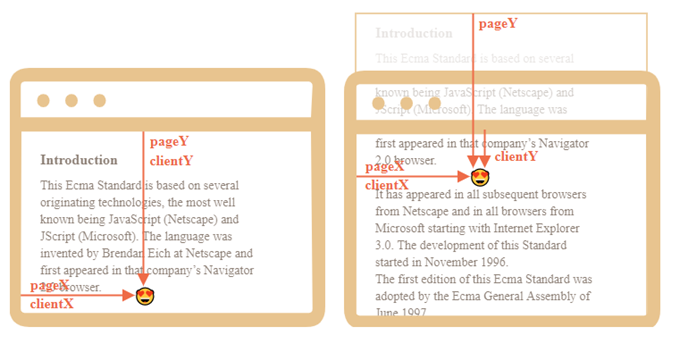
Большинство методов JS работают в одной из систем координат:

1.  Относительно окна браузера – как position:fixed, отсчёт идёт от верхнего левого угла окна. Далее будем обозначать эти координаты как clientX/clientY.

2.  Относительно документа – как position:absolute на уровне документа, отсчёт идёт от верхнего левого угла документа. Далее будем обозначать эти координаты как pageX/pageY.

Когда страница полностью прокручена в самое начало, то верхний левый угол окна совпадает с левым верхним углом документа, при этом обе этих системы координат тоже совпадают. Но если происходит прокрутка, то координаты элементов в контексте окна меняются, так как они двигаются, но в то же время их координаты относительно документа остаются такими же.

На картинке ниже показаны координат точки до прокрутки (слева) и после (справа):



Метод **elem.getBoundingClientRect()** возвращает координаты в контексте окна для минимального по размеру прямоугольника, который заключает в себе элемент elem, в виде объекта встроенного класса [DOMRect](https://www.w3.org/TR/geometry-1/#domrect).

Основные свойства объекта типа DOMRect:

-    **x/y** – X/Y-координаты начала прямоугольника относительно окна,

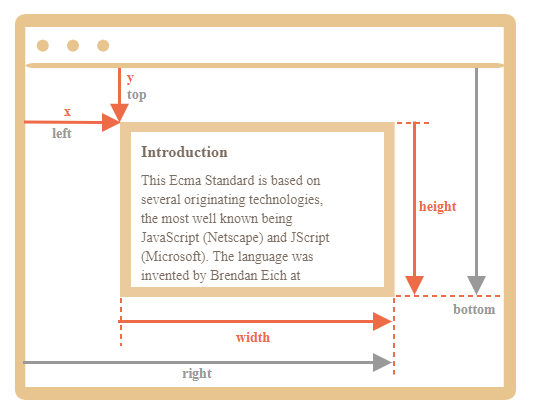
-    **width/height** – ширина/высота прямоугольника (могут быть отрицательными).

Дополнительные, «зависимые», свойства:

-    **top/bottom** – Y-координата верхней/нижней границы прямоугольника,

-    **left/right** – X-координата левой/правой границы прямоугольника.

Картинка с результатами вызова elem.getBoundingClientRect():



Internet Explorer и Edge не поддерживают свойства x/y. Таким образом, можно либо сделать полифил (добавив соответствующие геттеры в DomRect.prototype), либо использовать top/left, так как это всегда одно и то же при положительных width/height, в частности – в результате вызова elem.getBoundingClientRect().

Вызов **document.elementFromPoint(x, y)** возвращает самый глубоко вложенный элемент в окне, находящийся по координатам (x, y). Метод работает, только если координаты (x,y) относятся к видимой части содержимого окна. Если любая из координат представляет собой отрицательное число или превышает размеры окна, то возвращается null.

Чаще всего нужны координаты для позиционирования чего-либо. Чтобы показать что-то около нужного элемента, можно вызвать getBoundingClientRect, чтобы получить его координаты элемента, а затем использовать CSS-свойство position вместе с left/top (или right/bottom).

[Координаты относительно документа](https://learn.javascript.ru/coordinates#getCoords)

В такой системе координат отсчёт ведётся от левого верхнего угла документа, не окна. В CSS координаты относительно окна браузера соответствуют свойству position:fixed, а координаты относительно документа – свойству position:absolute на самом верхнем уровне вложенности.

Можно воспользоваться свойствами position:absolute и top/left, чтобы привязать что-нибудь к конкретному месту в документе. При этом прокрутка страницы не имеет значения. Но сначала нужно получить верные координаты.

Не существует стандартного метода, который возвращал бы координаты элемента относительно документа, но можно написать его сами.

Две системы координат связаны следующими формулами:

-    pageY = clientY + высота вертикально прокрученной части документа,

-    pageX = clientX + ширина горизонтально прокрученной части документа.

55. Браузерные события. (Т7.1)

Событие – это сигнал от браузера о том, что что-то произошло. Все DOM-узлы подают такие сигналы.

События мыши:

-    **click** – происходит, когда кликнули на элемент левой кнопкой мыши (на устройствах с сенсорными экранами оно происходит при касании);

-    **contextmenu** – происходит, когда кликнули на элемент правой кнопкой мыши;

-    **mouseover** / **mouseout** – когда мышь наводится на / покидает элемент;

-    **mousedown** / **mouseup** – когда нажали / отжали кнопку мыши на элементе;

-    **mousemove** – при движении мыши.

События на элементах управления:

-    **submit** – пользователь отправил форму <form>;

-    **focus** – пользователь фокусируется на элементе, например, нажимает на <input>.

Клавиатурные события:

-    **keydown** и **keyup** – когда пользователь нажимает / отпускает клавишу.

События документа:

-    **DOMContentLoaded** – когда HTML загружен и обработан, DOM документа полностью построен и доступен.

CSS events:

-    **transitionend** – когда CSS-анимация завершена.

Существует множество других событий.

[Обработчики событий](https://learn.javascript.ru/introduction-browser-events#obrabotchiki-sobytiy)

Событию можно назначить обработчик, то есть **функцию, которая сработает, как только событие произошло**. Именно благодаря обработчикам JavaScript-код может реагировать на действия пользователя. Есть несколько способов назначить событию обработчик.

Обработчик может быть назначен прямо в разметке, в атрибуте, который называется on<событие>:

<input value="Нажми меня" onclick="alert('Клик!')" type="button">

Но лучше создать отдельную JavaScript-функцию и вызвать её там. Следующий пример по клику запускает функцию countRabbits():8

<script>

  function countRabbits() { ... }

</script>

<input type="button" onclick="countRabbits()" value="Считать кроликов!">

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() { … };

</script>

Так как у элемента DOM может быть только одно свойство с именем onclick, то назначить более одного обработчика так нельзя.

Обратите внимание:

button.onclick = sayThanks; // правильно

button.onclick = sayThanks(); // неправильно

Добавлять несколько обработчиков позволяют методы **addEventListener** и **removeEventListener**.

Синтаксис добавления обработчика:

**element.addEventListener(event, handler[, options]);**

-    **event** – имя события, например "click";

-    **handler** – ссылка на функцию-обработчик;

-    **options** – дополнительный объект со свойствами:

·    **once:** если true, тогда обработчик будет автоматически удалён после выполнения;

·   **capture**: фаза, на которой должен сработать обработчик (подробнее об этом будет рассказано в вопросе о [всплытии и погружени](https://learn.javascript.ru/bubbling-and-capturing)и событий). options может быть false/true, это тоже самое, что {capture: false/true}.

·    **passive**: если true, то указывает, что обработчик никогда не вызовет preventDefault() (подробнее это будет рассматриваться в вопросе о [действии браузера по умолчанию](https://learn.javascript.ru/default-browser-action)).

Для удаления обработчика следует использовать removeEventListener:

element.removeEventListener(event, handler[, options]);

 Для удаления обработчика нужно передать именно ту функцию-обработчик которая была назначена.

[Объект события](https://learn.javascript.ru/introduction-browser-events#obekt-sobytiya)

Узнать детали события можно с помощью объекта **event**. Его свойства:

-   **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.

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.Метод handleEvent не обязательно должен выполнять всю работу сам. Он может вызывать другие методы, которые заточены под обработку конкретных типов событий.

56. Всплытие и погружение событий. (Т7.2)

Рассмотрим пример. Этот обработчик для <div> сработает, если вы кликните по любому из вложенных тегов, будь то <em> или <code>:

<div onclick="alert('Обработчик!')">

  <em>Если вы кликните на <code>EM</code>, сработает обработчик на <code>DIV</code></em>

</div>

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

Принцип всплытия очень простой. Когда на элементе происходит событие, **обработчики сначала срабатывают на нём, потом на его родителе, затем выше и так далее, вверх по цепочке предков.** Например, есть 3 вложенных элемента FORM > DIV > P с обработчиком на каждом:

<style>

  body \* {

margin: 10px;

border: 1px solid blue;

  }

</style>

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

  <div onclick="alert('div')">DIV

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

  </div>

</form>

Клик по внутреннему <p> вызовет обработчик onclick:

1.  Сначала на самом <p>.

2.  Потом на внешнем <div>.

3.  Затем на внешнем <form>.

4.  И так далее вверх по цепочке до самого document.

Поэтому если кликнуть на <p>, то появятся три оповещения: p → div → form. Этот процесс называется «всплытием», потому что события «всплывают» от внутреннего элемента вверх через родителей подобно тому, как всплывает пузырёк воздуха в воде. Не все события всплывают. Например, событие focus не всплывает. Однако, это исключение, всё-таки большинство событий всплывают.

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

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

-    **event.target – это «целевой» элемент**, на котором произошло событие, в процессе всплытия он неизменен.

-    **this – это «текущий»** элемент, до которого дошло всплытие, на нём сейчас выполняется обработчик.

Например, если стоит только один обработчик form.onclick, то он «поймает» все клики внутри формы. Где бы ни был клик внутри – он всплывёт до элемента <form>, на котором сработает обработчик. При этом внутри обработчика form.onclick:

-    this (=event.currentTarget) всегда будет элемент <form>, так как обработчик сработал на ней.

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

Результат

// script.js

form.onclick = function(event) {

  event.target.style.backgroundColor = 'yellow';

  // браузеру нужно некоторое время, чтобы зарисовать всё жёлтым

  setTimeout(() => {

alert("target = " + event.target.tagName + ", this=" + this.tagName);

event.target.style.backgroundColor = ''

  }, 0);

};

// example.css

form {

  background-color: green;

  position: relative;

  width: 150px;

  height: 150px;

  text-align: center;

  cursor: pointer;

}

div {

  background-color: blue;

  position: absolute;

  top: 25px;

  left: 25px;

  width: 100px;

  height: 100px;

}

p {

  background-color: red;

  position: absolute;

  top: 25px;

  left: 25px;

  width: 50px;

  height: 50px;

  line-height: 50px;

  margin: 0;

}

body {

  line-height: 25px;

  font-size: 16px;

}

// index.html

<!DOCTYPE HTML>

<html>

<head>

  <meta charset="utf-8">

  <link rel="stylesheet" href="example.css">

</head>

<body>

  Клик покажет оба: и <code>event.target</code>, и <code>this</code> для сравнения:

  <form id="form">FORM

<div>DIV

   <p>P</p>

</div>

  </form>

  <script src="script.js"></script>

</body>

</html>

Возможна и ситуация, когда event.target и this – один и тот же элемент, например, если клик был непосредственно на самом элементе <form>, не на его подэлементе.

[**Прекращение всплытия**](https://learn.javascript.ru/bubbling-and-capturing#prekraschenie-vsplytiya)

Всплытие идёт с «целевого» элемента прямо наверх. По умолчанию событие будет всплывать до элемента <html>, а затем до объекта document, а иногда даже до window, вызывая все обработчики на своём пути. Но любой промежуточный обработчик может решить, что событие полностью обработано, и остановить всплытие. **Для этого нужно вызвать метод event.stopPropagation().** Например, здесь при клике на кнопку <button> обработчик body.onclick не сработает:

<body onclick="alert(`сюда всплытие не дойдёт`)">

  <button onclick="event.stopPropagation()">Кликни меня</button>

</body>

Если у элемента есть несколько обработчиков на одно событие, то даже при прекращении всплытия все они будут выполнены. То есть, **event.stopPropagation() препятствует продвижению события дальше**, но на текущем элементе все обработчики будут вызваны.

Для того, **чтобы полностью остановить обработку, существует метод event.stopImmediatePropagation()**. Он не только предотвращает всплытие, но и останавливает обработку событий на текущем элементе.

Не прекращайте всплытие без необходимости. Зачастую прекращение всплытия через event.stopPropagation() имеет свои подводные камни, которые со временем могут стать проблемами. Например:

1.  При создании вложенного меню, каждое подменю обрабатывает клики на своих элементах и делает для них stopPropagation, чтобы не срабатывало внешнее меню.

2.  Позже было решено отслеживать все клики в окне для какой-то своей функциональности, к примеру, для статистики – где вообще кликают пользователи. Обычно используют document.addEventListener('click'…), чтобы отлавливать все клики.

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

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

Существует ещё одна фаза из жизненного цикла события – «погружение» (иногда её называют «перехват»). Она очень редко используется в реальном коде, однако тоже может быть полезной. Стандарт [DOM Events](http://www.w3.org/TR/DOM-Level-3-Events/) описывает 3 фазы прохода события:

1.  Фаза погружения (capturing phase) – событие сначала идёт сверху вниз.

2.  Фаза цели (target phase) – событие достигло целевого(исходного) элемента.

3.  Фаза всплытия (bubbling stage) – событие начинает всплывать.

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

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

Обработчики, добавленные через on<event>-свойство или через HTML-атрибуты, или через addEventListener(event, handler) с двумя аргументами, ничего не знают о фазе погружения, а работают только на 2-ой и 3-ей фазах. Чтобы поймать событие на стадии погружения, нужно использовать третий аргумент capture вот так:

elem.addEventListener(..., {capture: true})

// или просто "true", как сокращение для {capture: true}

elem.addEventListener(..., true)

Существуют два варианта значений опции capture:

-    Если аргумент false (по умолчанию), то событие будет поймано при всплытии.

-    Если аргумент true, то событие будет перехвачено при погружении.

Обратите внимание, что хоть и формально существует 3 фазы, 2-ую фазу («фазу цели»: событие достигло элемента) нельзя обработать отдельно, при её достижении вызываются все обработчики: и на всплытие, и на погружение. Посмотрим и всплытие и погружение в действии:

<style>

  body \* {

margin: 10px;

border: 1px solid blue;

  }

</style>

<form>FORM

  <div>DIV

<p>P</p>

  </div>

</form>

<script>

  for(let elem of document.querySelectorAll('\*')) {

elem.addEventListener("click", e => alert(`Погружение: ${elem.tagName}`), true);

elem.addEventListener("click", e => alert(`Всплытие: ${elem.tagName}`));

  }

</script>

Здесь обработчики назначаются каждому элементу в документе, чтобы увидеть в каком порядке они вызываются по мере прохода события. Если кликнуть по <p>, то последовательность следующая:

1.  HTML → BODY → FORM → DIV (фаза погружения, первый обработчик).

2.  P (фаза цели, срабатывают обработчики, установленные и на погружение, и на всплытие, так что выведется два раза).

3.  DIV → FORM → BODY → HTML (фаза всплытия, второй обработчик).

Существует свойство event.eventPhase, содержащее номер фазы, на которой событие было поймано. Но оно используется редко, обычно это и так известно в обработчике.

Если обработчик добавлен с помощью addEventListener(..., true), то надо передать то же значение аргумента capture в removeEventListener(..., true), когда снимаем обработчик.

Если есть несколько обработчиков одного события, назначенных addEventListener на один элемент, в рамках одной фазы, то их порядок срабатывания – тот же, в котором они установлены:

elem.addEventListener("click", e => alert(1)); // всегда сработает перед следующим

elem.addEventListener("click", e => alert(2));

57. Делегирование событий. Действия браузера по умолчанию. (Т7.3 (кроме: прием проектирования «поведения»), Т7.4)

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.

58. Генерация событий. (Т7.5)

Можно не только назначать обработчики, но и генерировать события из JavaScript-кода. Пользовательские события могут быть использованы при создании графических компонентов. Например, корневой элемент меню может генерировать события, относящиеся к этому меню: open (меню раскрыто), select (выбран пункт меню) и т.п. Также **можно генерировать встроенные события, такие как click, mousedown** и другие, что бывает полезно для автоматического тестирования.

**Конструктор Event**

Встроенные классы для событий формируют иерархию, аналогично классам для DOM-элементов. Её корнем является встроенный класс Event. Событие встроенного класса Event можно создать так:

**let event = new Event(type[, options]);**

Где:

**type** – тип события, строка, например, "click" или другая;

**options** – объект с двумя необязательными свойствами:

• **bubbles**: true/false – если true, тогда событие всплывает;

• **cancelable**: true/false – если true, тогда можно отменить действие по умолчанию.

По умолчанию оба свойства установлены в false: {bubbles: false, cancelable: false}.

После того, как объект события создан, надо запустить событие на элементе, вызвав метод elem.dispatchEvent(event). Затем обработчики отреагируют на него, как будто это обычное встроенное событие. Если при создании указан флаг bubbles, то оно будет всплывать. В примере ниже событие click инициируется JavaScript-кодом так же, как если бы кликнули по кнопке:

<button id="elem" onclick="alert('Клик!');">Автоклик</button>

<script>

  let event = new Event("click");

  elem.dispatchEvent(event);

</script>

В то же время можно легко отличить «настоящее» событие от сгенерированного кодом. Свойство event.isTrusted принимает значение true для событий, порождаемых реальными действиями пользователя, и false для генерируемых кодом.

Можно создать всплывающее событие с именем "hello" и поймать его на document. Всё, что нужно сделать – это установить флаг bubbles в true:

<h1 id="elem">Привет из кода!</h1>

<script>

  document.addEventListener("hello", function(event) { // (1)

    alert("Привет от " + event.target.tagName); // Привет от H1

  });

  // запуск события на элементе

  let event = new Event("hello", {bubbles: true}); // (2)

  elem.dispatchEvent(event);

</script>

Надо использовать **addEventListener для собственных событий**, т.к. on<event>-свойства существуют только для встроенных событий, то есть document.onhello не сработает. Надо **передавать флаг bubbles:true, иначе событие не будет всплывать.**

Механизм всплытия идентичен как для встроенного события (click), так и для пользовательского события (hello). Также одинакова работа фаз всплытия и погружения.

**Конструкторы MouseEvent, KeyboardEvent и другие**

Для некоторых конкретных типов событий есть свои специфические конструкторы. Вот небольшой список конструкторов для различных событий пользовательского интерфейса, которые можно найти в спецификации UI Event: UIEvent, FocusEvent, MouseEvent, WheelEvent, KeyboardEvent.

Стоит использовать их вместо new Event, если надо создавать такие события. К примеру, new MouseEvent("click"). Специфический конструктор позволяет указать стандартные свойства для данного типа события. Например, clientX/clientY для события мыши:

let event = new MouseEvent("click", {

  bubbles: true,

  cancelable: true,

  clientX: 100,

  clientY: 100

});

alert(event.clientX); // 100

Обратите внимание: это нельзя было бы сделать с обычным конструктором Event. Проверим:

let event = new Event("click", {

  bubbles: true, // только свойства bubbles и cancelable

  cancelable: true, // работают в конструкторе Event

  clientX: 100,

  clientY: 100

});

alert(event.clientX); // undefined, неизвестное свойство проигнорировано

Впрочем, использование конкретного конструктора не является обязательным, можно обойтись Event, а свойства записать в объект отдельно, после создания, вот так: event.clientX=100. Здесь это скорее вопрос удобства и желания следовать правилам. События, которые генерирует браузер, всегда имеют правильный тип. Полный список свойств по типам событий можно найти в спецификации, например для MouseEvent.

**Пользовательские события**

Для генерации пользовательских событий, таких как "hello", следует использовать конструктор new CustomEvent. Технически CustomEvent абсолютно идентичен Event за исключением одной небольшой детали. У второго аргумента-объекта есть дополнительное свойство detail, в котором можно указывать информацию для передачи в событие. Например:

<h1 id="elem">Привет для Васи!</h1>

<script>

  elem.addEventListener("hello", function(event) {

    alert(event.detail.name);

  });

  elem.dispatchEvent(new CustomEvent("hello", {

    detail: { name: "Вася" }

  }));

</script>

Свойство detail может содержать любые данные. Надо сказать, что никто не мешает и в обычное new Event записать любые свойства. Но CustomEvent предоставляет специальное поле detail во избежание конфликтов с другими свойствами события. Класс события содержит информацию о том, что это за событие, и если оно не браузерное, а пользовательское, то стоит всё-таки использовать CustomEvent.

На сгенерированном событии обработчик может вызвать метод **event.preventDefault()**, если задан флаг cancelable:true. Для пользовательских событий, названия которых браузеру неизвестны, соответственно, нет никаких действий браузера по умолчанию. Но код, который генерирует событие, может предусматривать какие-то ещё действия после dispatchEvent.

Вызов event.preventDefault() является возможностью для обработчика события сообщить в сгенерировавший событие код, **что эти действия надо отменить.** Тогда вызов elem.dispatchEvent(event) возвратит false. И код, сгенерировавший событие, приостановит выполнение.

В примере ниже есть функция hide(), которая при вызове генерирует событие "hide" на элементе #rabbit, уведомляя всех интересующихся, что кролик собирается спрятаться. Любой обработчик может узнать об этом, подписавшись на событие hide через rabbit.addEventListener('hide',...) и, при желании, отменить действие по умолчанию через event.preventDefault(). Тогда кролик не исчезнет:

<pre id="rabbit">

  |\   /|

   \|\_|/

   /. .\

  =\\_Y\_/=

   {>o<}

</pre>

<script>

  function hide() {

    let event = new CustomEvent("hide", {

      cancelable: true

    });

    if (!rabbit.dispatchEvent(event)) {

      alert('действие отменено обработчиком');

    } else {

      rabbit.hidden = true;

    }

  }

  rabbit.addEventListener('hide', function(event) {

    if (confirm("Вызвать preventDefault?")) {

      event.preventDefault();

    }

  });

  setTimeout(hide, 2000);

</script>

Обычно события обрабатываются асинхронно. То есть, если браузер обрабатывает onclick и в процессе этого произойдёт новое событие, то оно ждёт, пока закончится обработка onclick. Исключением является ситуация, когда событие инициировано из обработчика другого события. Тогда управление сначала переходит в обработчик вложенного события и уже после этого возвращается назад. В примере ниже событие menu-open обрабатывается синхронно во время обработки onclick:

<button id="menu">Меню (нажми меня)</button>

<script>

  menu.onclick = function() {

    alert(1);

    menu.dispatchEvent(new CustomEvent("menu-open", {

      bubbles: true

    }));

    alert(2);

  };

  document.addEventListener('menu-open', () => alert('вложенное событие'))

</script>

Обратите внимание, что вложенное событие menu-open всплывает и обрабатывается на document. Обработка вложенного события полностью завершается до того, как управление возвращается во внешний код (onclick). Это справедливо не только для dispatchEvent, но и для других случаев. JavaScript в обработчике события может вызвать другие методы, которые приведут к другим событиям – они тоже обрабатываются синхронно.

Также можно либо поместить dispatchEvent (или любой другой код инициирующий события) в конец обработчика onclick, либо обернуть такой код в setTimeout(..., 0):

<button id="menu">Меню (нажми меня)</button>

<script>

    menu.onclick = function() {

    alert(1);

    setTimeout(() => menu.dispatchEvent(new CustomEvent("menu-open", {

      bubbles: true

    })));

    alert(2);

  };

  document.addEventListener('menu-open', () => alert('вложенное событие'))

</script>

Теперь dispatchEvent запускается асинхронно после исполнения текущего кода, включая mouse.onclick, поэтому обработчики полностью независимы.

59. События мыши. События mouseover/out, mouseenter/leave. (Т7.6, Т7.7)

События мыши происходят не только от манипуляций мышью, но и эмулируются на сенсорных устройствах, чтобы сделать их совместимыми.

[**Типы событий мыши**](https://learn.javascript.ru/mouse-events-basics#tipy-sobytiy-myshi)

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

[Простые события](https://learn.javascript.ru/mouse-events-basics#prostye-sobytiya) (наиболее часто используемые):

-  mousedown/mouseup – кнопка мыши нажата/отпущена над элементом;

- mouseover/mouseout – курсор мыши появляется над элементом и уходит с него;

-  mousemove – каждое движение мыши над элементом генерирует это событие.

[Комплексные события](https://learn.javascript.ru/mouse-events-basics#kompleksnye-sobytiya):

-    click – вызывается при mousedown , а затем mouseup над одним и тем же элементом, если использовалась левая кнопка мыши;

-    contextmenu – вызывается при mousedown правой кнопкой мыши;

-    dblclick – вызывается двойным кликом на элементе.

Комплексные события состоят из простых.  С ними удобнее работать.

[**Порядок событий**](https://learn.javascript.ru/mouse-events-basics#poryadok-sobytiy)

Одно действие может вызвать несколько событий. Например, клик мышью вначале вызывает mousedown, когда кнопка нажата, затем mouseup и click, когда она отпущена. В случае, когда одно действие инициирует несколько событий, порядок их выполнения фиксирован. То есть обработчики событий вызываются в следующем порядке: mousedown → mouseup → click. События обрабатываются в той же последовательности: onmouseup завершается до того, как запускается onclick.

События, связанные с кликом, всегда имеют **свойство which**, которое **позволяет определить нажатую кнопку мыши**. Это свойство не используется для событий click и contextmenu, поскольку первое происходит только при нажатии левой кнопкой мыши, а второе – правой. События mousedown и mouseup срабатывают на любой кнопке и свойство which позволяет различать между собой «нажатие правой кнопки» и «нажатие левой кнопки» мыши.

Есть три возможных значения:

-    event.which == 1 – левая кнопка;

-    event.which == 2 – средняя кнопка;

-    event.which == 3 – правая кнопка.

[**Модификаторы: shift, alt, ctrl и meta**](https://learn.javascript.ru/mouse-events-basics#modifikatory-shift-alt-ctrl-i-meta)

Все события мыши включают в себя информацию о нажатых клавишах-модификаторах. Их свойства: shiftKey, altKey, ctrlKey, metaKey (Cmd для Mac). Например, кнопка внизу работает только при комбинации Alt+Shift+клик:

<button id="button">Нажми Alt+Shift+Click на мне!</button>

<script>

  button.onclick = function(event) {

if (event.altKey && event.shiftKey) {

   alert('Ура!');

}

  };

</script>

В Windows и Linux клавишами-модификаторами являются Alt, Shift и Ctrl. На Mac есть ещё одна: Cmd, она соответствует свойству metaKey. В большинстве случаев, когда в Windows/Linux используется Ctrl, на Mac пользователи используют Cmd. Поэтому, когда пользователь Windows нажимает Ctrl+Enter и Ctrl+A, пользователь Mac нажимает Cmd+Enter или Cmd+A, и так далее, большинство приложений используют Cmd вместо Ctrl. Поэтому, если надо поддерживать такие комбинации, как Ctrl+клик, то для Mac имеет смысл использовать Cmd+клик. Это удобней для пользователей Mac.

Левый клик в сочетании с Ctrl интерпретируется как правый клик на Mac и генерирует событие contextmenu, а не click как на Windows/Linux. Поэтому, если надо, чтобы пользователям всех операционных систем было удобно, то вместе с ctrlKey нужно использовать metaKey. Для JS-кода это означает, что надо проверить if (event.ctrlKey || event.metaKey).

Все события мыши имеют координаты двух видов: относительно окна (event.clientX и event.clientY) и относительно документа (event.pageX и event.pageY).

Клики мышью имеют побочный эффект, который может быть неудобен в некоторых интерфейсах: двойной клик мышью выделяет текст.  Это действие браузера по умолчанию при наступлении события mousedown. Поэтому альтернативным решением проблемы будет обработать событие mousedown и предотвратить его:

<b ondblclick="alert('Клик!')" onmousedown="return false">

  Сделайте двойной клик на мне

</b>

Теперь выделенный жирным элемент не выделяется при двойном клике. Текст внутри него по-прежнему можно выделить. Однако, выделение должно начаться не на самом тексте, а до него или после.

Вместо предотвращения выделения, можно отменить его «постфактум» в обработчике событий. Например, так:

<b ondblclick="getSelection().removeAllRanges()">

  Сделайте двойной клик на мне

</b>

При двойном клике на элементе, выделенном жирным шрифтом, выделение появится и тут же будет немедленно снято. Выглядит это не очень красиво.

Если надо отключить выделение для защиты контента от копирования, то можно использовать другое событие: oncopy.

<div oncopy="alert('Копирование запрещено!');return false">

  Уважаемый пользователь,

  Копирование информации запрещено.

</div>

Скопировать текст в <div> не получится, потому что срабатывание события oncopy по умолчанию запрещено.

Мышь: mouseover/out, mouseenter/leave

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

[**Mouseover/mouseout, relatedTarget**](https://learn.javascript.ru/mousemove-mouseover-mouseout-mouseenter-mouseleave#mouseover-mouseout-relatedtarget)

Событие mouseover происходит в момент, когда курсор оказывается над элементом, а событие mouseout – в момент, когда курсор уходит с элемента.

Эти события являются особенными, потому что у них имеется свойство relatedTarget. Оно дополняет target. Когда мышь переходит с одного элемента на другой, то один из них будет храниться в target, а другой в relatedTarget.

Для события mouseover:

-    event.target – это элемент, на который курсор перешёл;

-    event.relatedTarget – это элемент, с которого курсор ушёл.

Для события mouseout наоборот:

-    event.target – это элемент, с которого курсор ушёл;

-    event.relatedTarget – это элемент, на который курсор перешёл.

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

**Событие mousemove происходит при движении мыши**. Однако, это не означает, что указанное событие генерируется при прохождении каждого пикселя. Браузер периодически проверяет позицию курсора и, заметив изменения, генерирует события mousemove. Это означает, что если пользователь двигает мышкой очень быстро, то некоторые DOM-элементы могут быть пропущены:

Если курсор мыши двигается очень быстро с #FROM на #TO элемент, как это показано выше, то лежащие между ними элементы <div> (или некоторые из них) могут быть пропущены. Событие mouseout может запуститься на элементе #FROM и затем сразу же сгенерируется mouseover на элементе #TO.

**Представьте ситуацию** – курсор мыши перешёл на элемент. Сгенерировано событие mouseover. Затем курсор перешёл на дочерний элемент, и сгенерировано mouseout. То есть курсор всё ещё на элементе, но событие mouseout.

По логике браузера, курсор мыши может быть только над одним элементом в любой момент времени – над самым глубоко вложенным (и верхним по z-index). Таким образом, если курсор переходит на другой элемент (пусть даже дочерний), то он покидает предыдущий.

[**События mouseenter и mouseleave**](https://learn.javascript.ru/mousemove-mouseover-mouseout-mouseenter-mouseleave#sobytiya-mouseenter-i-mouseleave)

События **mouseenter/mouseleave похожи на mouseover/mouseout.** Они тоже генерируются, когда курсор мыши переходит на элемент или покидает его. Но есть и пара важных отличий:

1.  **Переходы внутри элемента по дочерним элементам не считаются.**

2.  События mouseenter/mouseleave не всплывают.

Когда курсор становится над элементом – генерируется mouseenter, и не имеет значения, где именно находится курсор внутри элемента. Событие mouseleave происходит, когда курсор покидает элемент.

События mouseenter/leave простые и легкие в использовании, но они не всплывают, а это значит что их нельзя делегировать.

60. Drag'n'Drop с событиями мыши. События клавиатуры: keyup, keydown. Прокрутка: событие scroll. (Т7.8, Т7.9, Т7.10)

События **Drag’n’Drop генерируются, когда пользователь захватывает элемент мышкой и перетаскивает его**.  Основной алгоритм Drag’n’Drop выглядит так:

1.Отслеживание события **mousedown** на переносимом элементе.

2.При нажатии – подготовка элемента к перемещению (например, создание его копии).

3.При наступлении **mousemove** перемещение элемента на новые координаты путём смены left/top и position:absolute.

4.На **mouseup** (при отпускании кнопки мыши) – остановка переноса элемента.

Браузер имеет свой собственный Drag’n’Drop, который автоматически запускается и вступает в конфликт с нашим. Это происходит именно для картинок и некоторых других элементов. Его нужно отключить:

ball.ondragstart = function() {

  return false;

};

Итоговый код

ball.onmousedown = function(event) {

  let shiftX = event.clientX - ball.getBoundingClientRect().left;

  let shiftY = event.clientY - ball.getBoundingClientRect().top;

  ball.style.position = 'absolute';

  ball.style.zIndex = 1000;

  document.body.append(ball);

  moveAt(event.pageX, event.pageY);

  function moveAt(pageX, pageY) {

ball.style.left = pageX - shiftX + 'px';

ball.style.top = pageY - shiftY + 'px';

  }

  function onMouseMove(event) {

moveAt(event.pageX, event.pageY);

  }

  document.addEventListener('mousemove', onMouseMove);

  ball.onmouseup = function() {

    document.removeEventListener('mousemove', onMouseMove);

ball.onmouseup = null;

  };

};

ball.ondragstart = function() {

  return false;

};

События прокрутки позволяют реагировать на прокрутку страницы или элемента. Есть много хороших вещей, которые можно сделать при этом.

Например:

* Показать/скрыть дополнительные элементы управления или информацию, основываясь на том, в какой части документа находится пользователь.
* Загрузить больше данных, когда пользователь прокручивает страницу вниз до конца.

Вот небольшая функция для отображения текущей прокрутки:

window.addEventListener('scroll', function() {

  document.getElementById('showScroll').innerHTML = pageYOffset + 'px';

});

События клавиатуры должны использоваться, если необходимо обрабатывать взаимодействие пользователя с клавиатурой (в том числе виртуальной). К примеру, если нужно реагировать на стрелочные клавиши Up и Down или горячие клавиши (включая комбинации клавиш).

Событие **keydown** происходит при нажатии клавиши, а **keyup** – при отпускании.

61. Свойства и методы формы. (Т8.1)

Формы и элементы управления, такие как <input>, имеют множество специальных свойств и событий.

Формы в документе входят в специальную коллекцию document.forms. Это – так называемая «**именованная» коллекция**: можно использовать для получения формы как её имя, так и порядковый номер в документе.

**document.forms.my - форма с именем** "my" (name="my")

**document.forms[0]** - первая форма в документе

Когда форма уже получена, любой элемент доступен в именованной коллекции **form.elements**. Например:

<form name="my">

  <input name="one" value="1">

  <input name="two" value="2">

</form>

<script>

  // получаем форму

  let form = document.forms.my; // <form name="my"> element

  // получаем элемент

  let elem = form.elements.one; // <input name="one"> element

  alert(elem.value); // 1

</script>

Может быть несколько элементов с одним и тем же именем, это часто бывает с кнопками-переключателями radio. В этом случае form.elements[name] является коллекцией, например:

<form>

  <input type="radio" name="age" value="10">

  <input type="radio" name="age" value="20">

</form>

<script>

let form = document.forms[0];

let ageElems = form.elements.age;

alert(ageElems[0].value); // 10, the first input value

</script>

Эти навигационные свойства не зависят от структуры тегов внутри формы. Все элементы, как бы глубоко они ни находились в форме, доступны в коллекции form.elements.

Форма может содержать один или несколько элементов <fieldset> внутри себя. Они также поддерживают свойство elements. Например:

<body>

  <form id="form">

<fieldset name="userFields">

   <legend>info</legend>

   <input name="login" type="text">

</fieldset>

  </form>

  <script>

alert(form.elements.login); // <input name="login">

let fieldset = form.elements.userFields;

alert(fieldset); // HTMLFieldSetElement

// можно получить информацию как из формы, так и из fieldset

alert(fieldset.elements.login == form.elements.login); // true

  </script>

</body>

Есть более короткая запись: можно получить доступ к элементу через form[index/name]. Вместо form.elements.login можно написать form.login. Это также работает, но есть небольшая проблема: если надо получить элемент, а затем менять его свойство name, то он всё ещё будет доступен под старым именем (также, как и под новым). В этом легче разобраться на примере:

<form id="form">

  <input name="login">

</form>

<script>

  alert(form.elements.login == form.login); // true, ведь это одинаковые <input>

  form.login.name = "username"; // изменяем свойство name у элемента input

  // form.elements обновили свои имена:

  alert(form.elements.login); // undefined

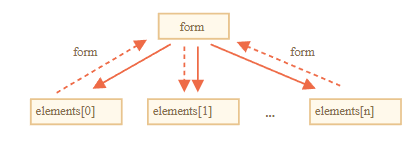
  alert(form.elements.username); // input

  // теперь для прямого доступа можно использовать оба имени: новое и старое

  alert(form.username == form.login); // true

</script>

Для любого элемента форма доступна через element.form. Так что форма ссылается на все элементы, а эти элементы ссылаются на форму. Вот иллюстрация:



Пример:

<form id="form">

  <input type="text" name="login">

</form>

<script>

  // form -> element

  let login = form.login;

  // element -> form

  alert(login.form); // HTMLFormElement

</script>

Рассмотрим элементы управления, используемые в формах, обращая внимание на их особенности.

[**input и textarea**](https://learn.javascript.ru/form-elements#input-i-textarea)

К их значению можно получить доступ через свойство input.value (строка) или input.checked (булево значение) для чекбоксов. Вот так:

input.value = "Новое значение";

textarea.value = "Новый текст";

input.checked = true; // для чекбоксов и переключателей

Обратите внимание: хоть <textarea>...</textarea> и хранит значение как вложенный HTML, не следует использовать textarea.innerHTML. Там хранится только тот HTML, который был изначально на странице, а не текущее значение.

[**select и option**](https://learn.javascript.ru/form-elements#select-i-option)

Элемент <select> имеет 3 важных свойства:

1.   select.options – коллекция из элементов <option>,

2.   select.value – значение выбранного в данный момент <option>,

3.   select.selectedIndex – номер выбранного <option>.

Имеется три способа задать значение для <select>:

1.   Найти необходимый <option> и установить в option.selected значение true.

2.   Установить в select.value значение нужного нам <option>.

3.   Установить в select.selectedIndex номер <option>.

Первый способ наиболее понятный, но (2) и (3) являются более удобными при работе. Вот эти способы на примере:

<select id="select">

  <option value="apple">Яблоко</option>

  <option value="pear">Груша</option>

  <option value="banana">Банан</option>

</select>

<script>

  // все три строки делают одно и то же

  select.options[2].selected = true;

  select.selectedIndex = 2;

  select.value = 'banana';

</script>

В отличие от большинства других элементов управления, <select multiple> позволяет выбрать несколько вариантов. В этом случае необходимо пройтись по select.options, чтобы получить все выбранные значения. Например так:

<select id="select" multiple>

  <option value="blues" selected>Блюз</option>

  <option value="rock" selected>Рок</option>

  <option value="classic">Классика</option>

</select>

<script>

  // получаем все выбранные значения из списка множественного выбора

  let selected = Array.from(select.options)

.filter(option => option.selected)

.map(option => option.value);

  alert(selected); // Блюз,Рок

</script>

Полное описание элемента <select> доступно в спецификации<https://html.spec.whatwg.org/multipage/forms.html#the-select-element>.

Элемент <option> редко используется сам по себе. В описании [элемента option](https://html.spec.whatwg.org/multipage/forms.html#the-option-element) есть короткий синтаксис для создания элемента:

option = new Option(text, value, defaultSelected, selected);

Параметры:

-     text – текст внутри,

-     value – значение,

- defaultSelected – если true, то ставится HTML-атрибут selected,

-     selected – если true, то элемент <option> будет выбранным.

Пример:

let option = new Option("Текст", "value");

// создаст <option value="value">Текст</option>

Тот же элемент, но выбранный:

let option = new Option("Текст", "value", true, true);

Элементы <option> имеют дополнительные свойства:

-     selected – выбрана ли опция,

-     index – номер опции среди других в списке <select>,

-     text – содержимое опции (то, что видит посетитель).

62. Изменение значений элемента формы. Формы: отправка, событие и метод submit. (Т8.3, Т8.4)

Рассмотрим различные события, сопутствующие обновлению данных.

[**Событие: change**](https://learn.javascript.ru/events-change-input#sobytie-change)

Событие change срабатывает по окончании изменения элемента. Для текстовых <input> это означает, что событие происходит при потере фокуса. Пока пользователь печатает в текстовом поле в примере ниже, событие не происходит. Но когда перемещает фокус в другое место, например, нажимая на кнопку, то произойдёт событие change:

<input type="text" onchange="alert(this.value)">

<input type="button" value="Button">

https://lh5.googleusercontent.com/uEKOp90CypK9BR7bI0vSdECTXn4-ntbNEPgsIGySlR9ZsM_z2wA6oA0AiTotV0aRNKDbT5wFzysW7icBVeZLo1o6eDwCVLIrN4dtzjNhW2kdz9TNWcKwY13a5VNanWWPQA

https://lh5.googleusercontent.com/4LZm7sffsShXDRaEu1jTFVU03jnqOt_vHRK3jLjm7RfUX5PCY0McvTx-PUSJCI6bw5sa6tr9BlE6O6U2iz34JCrf39x_8WsUWncLr1vYuRpuQtfRgzvuo-AqBHToMH0kGg

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>

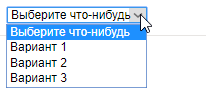
  <option value="1">Вариант 1</option>

  <option value="2">Вариант 2</option>

  <option value="3">Вариант 3</option>

</select>

https://lh4.googleusercontent.com/Pws0DNa9pyw-ilcgdXfSsF31ocs0JgKO_yb4JEjePqda5EliseIVYbS3jq-5wwDE0cwkaPLbEdLYjmsfN3-Jckt19dIVcMTLQTJLuADSzkIClE8jfpUwVkl7_bsu4I-FQQ

****

[**Событие: input**](https://learn.javascript.ru/events-change-input#sobytie-input)

Событие input срабатывает каждый раз при изменении значения. В отличие от событий клавиатуры, оно работает при любых изменениях значений, даже если они не связаны с клавиатурными действиями: вставка с помощью мыши или распознавание речи при диктовке текста. Например:

<input type="text" id="input"> oninput: <span id="result"></span>

<script>

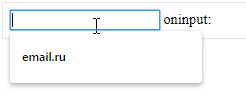
  input.oninput = function() {

result.innerHTML = input.value;

  };

</script>

https://lh6.googleusercontent.com/nTtELquUZuDB7fIAKPM63BARJs5VkSdwUS3sBNPqf1vt-Bj0MldXR8eIsrjkIm7zlFiqa9hU6vR392knW3Ln-bXv5y_6r7pWYdU7jc_pLCdRexz_92OFQWdEwpCi_Hb6oA



Если надо обрабатывать каждое изменение в <input>, то это событие является лучшим выбором. С другой стороны, событие input не происходит при вводе с клавиатуры или иных действиях, если при этом не меняется значение в текстовом поле, т.е. нажатия клавиш ⇦, ⇨ и подобных при фокусе на текстовом поле не вызовут это событие.

Событие input происходит после изменения значения. Поэтому нельзя использовать event.preventDefault() там – будет уже слишком поздно, никакого эффекта не будет.

[**События: cut, copy, paste**](https://learn.javascript.ru/events-change-input#sobytiya-cut-copy-paste)

Эти события происходят при вырезании/копировании/вставке данных. Они относятся к классу [ClipboardEvent](https://www.w3.org/TR/clipboard-apis/#clipboard-event-interfaces) и обеспечивают доступ к копируемым/вставляемым данным. Также можно использовать event.preventDefault() для предотвращения действия по умолчанию, и в итоге ничего не скопируется/не вставится. Например, код, приведённый ниже, предотвращает все подобные события и показывает, что надо вырезать/копировать/вставить:

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

<script>

  input.oncut = input.oncopy = input.onpaste = function(event) {

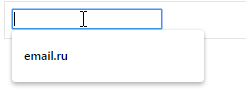
alert(event.type + ' - ' + event.clipboardData.getData('text/plain'));

return false;

  };

</script>

https://lh5.googleusercontent.com/tW07611cJ_uy-VQxQVGFqI2_ZMVeVHgepAoVKWtD_j5IYhulkPdtx4EnOJ-PH7xIecsc6VO9sdfgsCbY5FdVfqJqOnzz_LYjPtI2BP_YRMcdv74Y3opEJpgrYYkXsWCreA



Технически, можно скопировать/вставить всё. Например, можно скопировать файл из файловой системы и вставить его. Существует список методов [в спецификации](https://www.w3.org/TR/clipboard-apis/#dfn-datatransfer) для работы с различными типами данных, чтения/записи в буфер обмена. Но обратите внимание, что буфер обмена работает глобально, на уровне ОС. Большинство браузеров в целях безопасности разрешают доступ на чтение/запись в буфер обмена только в рамках определённых действий пользователя, к примеру, в обработчиках событий onclick.

Также запрещается генерировать «пользовательские» события буфера обмена при помощи dispatchEvent во всех браузерах, кроме Firefox.

**Формы: отправка, событие и метод submit.**

При отправке формы срабатывает событие submit, оно обычно используется для проверки (валидации) формы перед её отправкой на сервер или для предотвращения отправки и обработки её с помощью JavaScript.

Метод **form.submit() позволяет инициировать отправку формы** из JavaScript. Можно использовать его для динамического создания и отправки наших собственных форм на сервер. Рассмотрим их подробнее.

[**Событие: submit**](https://learn.javascript.ru/forms-submit#sobytie-submit)

Есть **два основных способа отправить форму:**

1.   Первый – нажать кнопку <input type="submit"> или <input type="image">.

2.   Второй – нажать Enter, находясь на каком-нибудь поле.

Оба действия сгенерируют событие submit на форме. Обработчик может проверить данные, и если есть ошибки, показать их и вызвать event.preventDefault(), тогда форма не будет отправлена на сервер.

В примере ниже оба действия показывают alert и форма не отправится благодаря return false:

<form onsubmit="alert('submit!');return false">

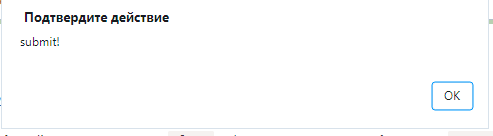
  Первый пример: нажмите Enter: <input type="text" value="Текст"><br>

  Второй пример: нажмите на кнопку "Отправить": <input type="submit" value="Отправить">

</form>

https://lh6.googleusercontent.com/KjLrqWbj7-dDHGc1qobhwfMuuYLomv-li0O7VigPmPi86bHTQYZjhK23-GWMiQ70zrWmxTom5i8anO_QmKVdqWXFO9LgFDC_-fJAYL8wwBVO5F-mMitEMY3fpMNPbf2gMQ

Первый и второй примеры, результат одинаковый:



При отправке формы по нажатию Enter в текстовом поле, генерируется событие click на кнопке <input type="submit">. При этом никакого клика на самом деле не было. Пример:

<form onsubmit="alert('submit!');return false">

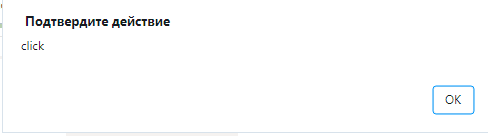
 <input type="text" size="30" value="Установите фокус здесь и нажмите Enter">

 <input type="submit" value="Отправить" onclick="alert('click')">

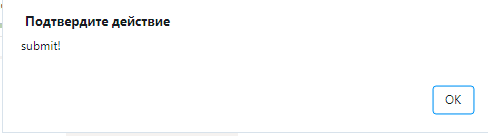
</form>

https://lh4.googleusercontent.com/RZ1dHGLJsL5nEgCBuMftyXRSr90Ugi8kTmuSYm7yNkXk-VxIXBsWxxHBW9n8Jg1jgxkW9ty4LuqjxpTw8VIb7v6alOznmpciTNBq2Nk8iNU8RPMYE4A0VzyGy9m1miyzAQ

Появилось окно:



После клика на ок:



[**Метод: submit**](https://learn.javascript.ru/forms-submit#metod-submit)

Чтобы отправить форму на сервер вручную, можно вызвать метод form.submit(). При этом событие submit не генерируется. Предполагается, что если программист вызывает метод form.submit(), то он уже выполнил всю соответствующую обработку. Иногда это используют для генерации формы и отправки её вручную, например так:

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

form.action = 'https://google.com/search';

form.method = 'GET';

form.innerHTML = '<input name="q" value="test">';

// перед отправкой формы, её нужно вставить в документ

document.body.append(form);

form.submit();

63. JavaScript-анимация. (Т8.5)

С помощью JavaScript-анимаций можно делать вещи, которые нельзя реализовать на CSS. Например, движение по сложному пути с временной функцией, отличной от кривой Безье, или canvas-анимации.

**Использование setInterval**

Анимация реализуется через последовательность кадров, каждый из которых немного меняет HTML/CSS-свойства. Например, изменение style.left от 0px до 100px – двигает элемент. И если делать это с помощью setInterval, изменяя на 2px с небольшими интервалами времени, например, 50 раз в секунду, тогда изменения будут выглядеть плавными. Принцип такой же, как в кино: 24 кадров в секунду достаточно, чтобы создать эффект плавности. Код мог бы выглядеть так:

let timer = setInterval(function() {

  if (animation complete) clearInterval(timer);

  else increase style.left by 2px

}, 20); // изменять на 2px каждые 20ms, это около 50 кадров в секунду

Более детальная реализация этой анимации:

let start = Date.now(); // запомнить время начала

let timer = setInterval(function() {

  let timePassed = Date.now() - start;

  if (timePassed >= 2000) {

    clearInterval(timer); // закончить анимацию через 2 секунды

    return;

  }

  // отрисовать анимацию

  draw(timePassed);

}, 20);

// в то время как timePassed изменяется от 0 до 2000

// left изменяет значение от 0px до 400px

function draw(timePassed) {

  train.style.left = timePassed / 5 + 'px';

}

**Использование requestAnimationFrame**

Допустим есть несколько анимаций, работающих одновременно. Если запустить их независимо с помощью setInterval(..., 20), тогда браузеру будет необходимо выполнять отрисовку гораздо чаще, чем раз в 20ms.

Это происходит из-за того, что каждая анимация имеет своё собственное время старта и «каждые 20 миллисекунд» для разных анимаций – разные. Интервалы не выравнены и будет несколько независимых срабатываний в течение 20ms. Другими словами:

setInterval(function() {

  animate1();

  animate2();

  animate3();

}, 20)

меньше нагружают систему, чем три независимых функции:

setInterval(animate1, 20); // независимые анимации

setInterval(animate2, 20); // в разных местах кода

setInterval(animate3, 20);

Эти независимые анимации лучше сгруппировать вместе, тогда они будут легче для браузера, а значит – не грузить процессор и более плавно выглядеть.

Также стоит учитывать следующее: когда CPU перегружен или есть другие причины делать перерисовку реже (например, когда вкладка браузера скрыта), не следует делать её каждые 20ms.

Спецификация Animation timing описывает функцию requestAnimationFrame, которая решает все описанные проблемы и делает даже больше. Синтаксис:

let requestId = requestAnimationFrame(callback)

Такой вызов планирует запуск функции callback на ближайшее время, когда браузер сочтёт возможным осуществить анимацию. Если в callback происходит изменение элемента, тогда оно будет сгруппировано с другими requestAnimationFrame и CSS-анимациями. Таким образом браузер выполнит один геометрический пересчёт и отрисовку, вместо нескольких. Значение requestId может быть использовано для отмены анимации:

// отмена запланированного запуска callback

cancelAnimationFrame(requestId);

Функция callback имеет один аргумент – время, прошедшее с момента начала загрузки страницы в миллисекундах. Это значение может быть получено с помощью вызова performance.now(). Как правило, callback запускается очень быстро, если только не перегружен CPU или не разряжена батарея ноутбука, или у браузера нет какой-то ещё причины замедлиться.

Код ниже показывает время между первыми 10 запусками requestAnimationFrame. Обычно оно 10-20 мс:

<script>

  let prev = performance.now();

  let times = 0;

  requestAnimationFrame(function measure(time) {

    document.body.insertAdjacentHTML("beforeEnd", Math.floor(time - prev) + " ");

    prev = time;

    if (times++ < 10) requestAnimationFrame(measure);

  })

</script>

**Структура анимации**

Теперь можно создать более сложную функцию анимации с помощью requestAnimationFrame:

function animate({timing, draw, duration}) {

  let start = performance.now();

**requestAnimationFrame(function animate(time) {**

    // timeFraction изменяется от 0 до 1

    let timeFraction = (time - start) / duration;

    if (timeFraction > 1) timeFraction = 1;

    // вычисление текущего состояния анимации

    let progress = timing(timeFraction);

    draw(progress); // отрисовать её

    if (timeFraction < 1) {

      requestAnimationFrame(animate);

    }

  });

}

Функция animate имеет три аргумента, которые описывают анимацию:

1. **duration** – продолжительность анимации. Например, 1000.

2. **timing(timeFraction)** – функция расчёта времени, как CSS-свойство transition-timing-function, которая будет вычислять прогресс анимации (как ось y у кривой Безье) в зависимости от прошедшего времени (0 в начале, 1 в конце).

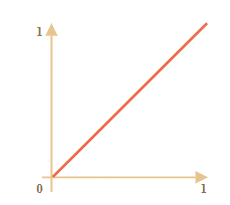
Например, линейная функция значит, что анимация идёт с одной и той же скоростью:

function linear(timeFraction) {

  return timeFraction;

}

График функции:



Это похоже на значение linear для transition-timing-function.

3. draw(progress) – функция отрисовки, которая получает аргументом значение прогресса анимации и отрисовывает его. Значение progress=0 означает что анимация находится в начале, и значение progress=1 – в конце.

Эта та функция, которая рисует анимацию. Пример функции для сдвига элемента:

function draw(progress) {

  train.style.left = progress + 'px';

}

В отличие от CSS-анимаций, можно создать любую функцию расчёта времени и любую функцию отрисовки. Функция расчёта времени не будет ограничена только кривой Безье, а функция draw может менять не только свойства, но и создавать новые элементы (например, для создания анимации фейерверка).

**Функции расчёта времени**

Самый простой пример линейной функции расчёта времени рассматривался выше. Рассмотрим другие.

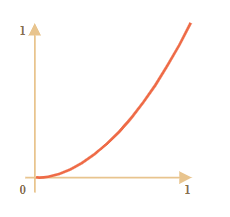
*Степень n*. Если надо ускорить анимацию, то можно возвести progress в степень n. Например, параболическая кривая:

function quad(timeFraction) {

  return Math.pow(timeFraction, 2)

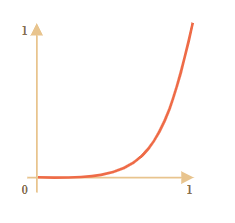
}

График:



Повышение степени увеличивает скорость анимации. Вот график для функции progress в степени 5:

*Дуга.* Функция:

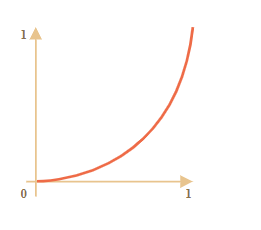


function circ(timeFraction) {

  return 1 - Math.sin(Math.acos(timeFraction));

}

График:



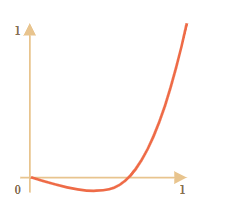
*Обратно: выстрел из лука*. Эта функция совершает «выстрел из лука». В начале «натягивается тетива», а затем «выстрел». В отличие от предыдущей функции, теперь всё зависит от дополнительного параметра x – «коэффициента эластичности». Он определяет силу «натяжения тетивы». Код:

function back(x, timeFraction) {

  return Math.pow(timeFraction, 2) \* ((x + 1) \* timeFraction - x)

}

График для x = 1.5:



*Отскоки.* Представьте, что бросили мяч вниз. Он падает, ударяется о землю, подскакивает несколько раз и останавливается. Функции bounce делает то же самое, но в обратном порядке: «отскоки» начинаются сразу. Для этого заданы специальные коэффициенты:

function bounce(timeFraction) {

  for (let a = 0, b = 1, result; 1; a += b, b /= 2) {

    if (timeFraction >= (7 - 4 \* a) / 11) {

      return -Math.pow((11 - 6 \* a - 11 \* timeFraction) / 4, 2) + Math.pow(b, 2)

    }

  }

}

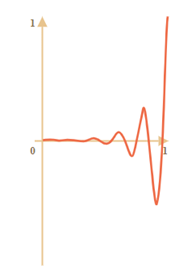
*Эластичная анимация*. Ещё одна «эластичная» функция, которая принимает дополнительный параметр x для «начального отрезка».

function elastic(x, timeFraction) {

  return Math.pow(2, 10 \* (timeFraction - 1)) \* Math.cos(20 \* Math.PI \* x / 3 \* timeFraction)

}

График для x = 1.5:



**Реверсивные функции: ease\***

Прямое использование функций расчёта времени называется «easeIn». Иногда нужно показать анимацию в обратном режиме. Преобразование функции, которое даёт такой эффект, называется «easeOut».

В режиме «easeOut» timing функции оборачиваются функцией timingEaseOut:

timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction)

Другими словами, имеется функция «преобразования» – makeEaseOut, которая берет «обычную» функцию расчёта времени и возвращает обёртку над ней (преобразованный вариант):

function makeEaseOut(timing) {

  return function(timeFraction) {

    return 1 - timing(1 - timeFraction);

  }

}

Например, можно взять функцию bounce описанную выше:

let bounceEaseOut = makeEaseOut(bounce);

Таким образом, отскоки будут не в начале функции, а в конце:

    function makeEaseOut(timing) {

      return function(timeFraction) {

        return 1 - timing(1 - timeFraction);

      }

    }

    function bounce(timeFraction) {

      for (let a = 0, b = 1, result; 1; a += b, b /= 2) {

        if (timeFraction >= (7 - 4 \* a) / 11) {

          return -Math.pow((11 - 6 \* a - 11 \* timeFraction) / 4, 2) + Math.pow(b, 2)

        }

      }

    }

    let bounceEaseOut = makeEaseOut(bounce);

    brick.onclick = function() {

      animate({

        duration: 3000,

        timing: bounceEaseOut,

        draw: function(progress) {

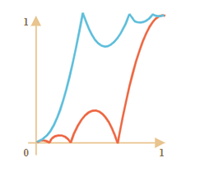
          brick.style.left = progress \* 500 + 'px';

        }

      });

    };

Ниже можно увидеть, как трансформации изменяют поведение функции:



Если раньше анимационный эффект, такой как отскоки, был в начале, то после трансформации он будет показан в конце. На графике выше красным цветом обозначена обычная функция и синим – после easeOut. Обычный скачок – объект сначала медленно скачет внизу, а затем резко подпрыгивает вверх. Обратный easeOut – объект вначале прыгает вверх, и затем скачет там.

Можно применить эффект дважды – в начале и конце анимации. Такая трансформация называется «easeInOut». Для функции расчёта времени, анимация будет вычисляться следующим образом:

if (timeFraction <= 0.5) { // первая половина анимации

  return timing(2 \* timeFraction) / 2;

} else { // вторая половина анимации

  return (2 - timing(2 \* (1 - timeFraction))) / 2;

}

Код функции-обёртки:

function makeEaseInOut(timing) {

  return function(timeFraction) {

    if (timeFraction < .5)

      return timing(2 \* timeFraction) / 2;

    else

      return (2 - timing(2 \* (1 - timeFraction))) / 2;

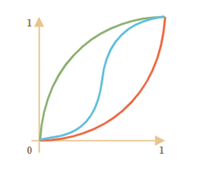
  }

}

bounceEaseInOut = makeEaseInOut(bounce);

Функция «easeInOut» объединяет два графика в один: easeIn (обычный) для первой половины анимации and easeOut (обратный) – для второй половины.

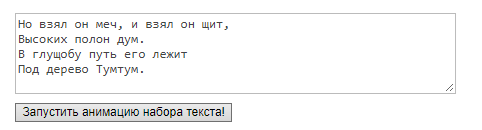
Разница хорошо заметна, если сравнивать графики easeIn, easeOut и easeInOut для функции circ. Красный – обычный вариант circ (easeIn), зелёный – easeOut, синий – easeInOut:



Как видно, график первой половины анимации представляет собой уменьшенный easeIn, а второй – уменьшенный easeOut. В результате, анимация начинается и заканчивается одинаковым эффектом.

Вместо передвижения элемента можно делать что-нибудь ещё. Всё, что нужно – это правильно написать функцию draw.

Вот пример «скачущей» анимации набирающегося текста:



// style.css

textarea {

  display: block;

  border: 1px solid #BBB;

  color: #444;

  font-size: 110%;

}

button {

  margin-top: 10px;

}

// index.html

<!DOCTYPE HTML>

<html>

<head>

  <meta charset="utf-8">

  <link rel="stylesheet" href="style.css">

  <script src="https://js.cx/libs/animate.js"></script>

</head>

<body>

  <textarea id="textExample" rows="5" cols="60">Но взял он меч, и взял он щит,

Высоких полон дyм.

В глyщобy пyть его лежит

Под дерево Тyмтyм.

  </textarea>

  <button onclick="animateText(textExample)">Запустить анимацию набора текста!</button>

  <script>

    function animateText(textArea) {

      let text = textArea.value;

      let to = text.length,

        from = 0;

      animate({

        duration: 5000,

        timing: bounce,

        draw: function(progress) {

          let result = (to - from) \* progress + from;

          textArea.value = text.substr(0, Math.ceil(result))

        }

      });

    }

    function bounce(timeFraction) {

      for (let a = 0, b = 1, result; 1; a += b, b /= 2) {

        if (timeFraction >= (7 - 4 \* a) / 11) {

          return -Math.pow((11 - 6 \* a - 11 \* timeFraction) / 4, 2) + Math.pow(b, 2)

        }

      }

    }

  </script>

</body>

</html>