# Примитивные типы данных

[Смотреть материал на видео](https://www.youtube.com/watch?v=w41UBEqv6ZA&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/primitivnye-tipy-dannyh>

Пришло время познакомиться с примитивными типами данных в JavaScript. В самом простом случае, когда мы объявляем переменную вот так:

let a = 2;

то переменная принимает числовой тип (number). Этот пример показывает, что тип переменной определяется типом данных, которые мы ей присваиваем. Если переменной присвоить строку:

let b = "Строка";

то переменная b принимает строковый тип. В JavaScript существует семь основных типов данных. Рассмотрим их по порядку.

**Числовой тип (number)**

Числовой тип данных представляет как целочисленные значения, так и числа с плавающей точкой. Например, обе эти переменные будут относиться к числовому типу:

let a = 5;

let b = 7.23;

JavaScript «знает» пять основных типов числовых литералов:

* десятичный – целое число (например, 0, 5, -10, -100, 56);
* приближенные к вещественным – приблизительные вещественные числа (например, 6.7, 8.54, -10.34);
* экспоненциальный (научный) – с использованием буквы ‘e’ (например, 10 = 1e1, 20 = 2e1, 25000 = 25e3, 8700 = 8.7e3);
* восьмиричная: 0o777;
* шестнадцатиричная: 0xff24f.

Однако, на практике чаще всего используются целые и приближенные к вещественным.

**Оператор typeof**

Чтобы узнать тип переменной используется оператор typeof, который имеет такой синтаксис:

typeof <переменная>;

или

typeof(<переменная>);

его можно вызывать как со скобками, так и без скобок. Обычно используют первый вариант. Выведем тип переменной a:

console.log(**typeof** a);

Мы видим значение number – числовой тип.

**Infinity и NaN**

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому же типу данных:

Infinity, -Infinity и NaN.

**Infinity** представляет собой математическую бесконечность ∞. Это особое значение, которое больше любого числа. Мы можем получить его в результате деления на ноль:

let c = 1/0;

или задать его явно:

let d = **Infinity**;

let d2 = -**Infinity**;

Мы можем также получить бесконечность, если попытаемся записать в переменную очень большое число, которое выходит за пределы представления чисел в JavaScript. Например, такое:

let inf = 1.0e1000;

Следующее значение NaN (от англ. Not a Number – не число) означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:

let c = "строка"/2;

Значение NaN «прилипчиво». Любая операция с NaN возвращает NaN:

let c = "строка"/2 + 2;

Значения Infinity и NaN хотя и относятся к числовым типам, но скорее являются индикаторами, говорящими разработчику скрипта что пошло не так.

**Строковый тип (string)**

Как мы говорили на предыдущем занятии, строки можно задавать через такие кавычки:

let msg1 = "строка 1";

let msg2 = 'строка 2';

let msg3 = `строка 3`;

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript. Обратные кавычки (апострофы) имеют «расширенный функционал» и были введены в ES6. Они позволяют встраивать выражения в строку, заключая их в ${…}. Например, так:

let msg3 = `Значение a = ${a}`;

Здесь вместо ${a} будет подставлено значение переменной a и в консоле увидим «Значение a = 5». Вместо переменной можно записать любое выражение языка JavaScript:

let msg3 = `Значение a = ${1+2}`;

Получим строку «Значение a = 3». Все это работает только с обратными кавычками. Если их заменить на простые, то никакой подстановки не произойдет:

let msg3 = 'Значение a = ${a}';

Тем, кто знаком с другими языками программирования следует знать, что в JavaScript нет типа данных для отдельного символа (типа char). Здесь есть только строки и даже один символ – это строка из одного символа.

А как быть, если мы хотим в строке записать кавычки? Например, так:

let msg1 = "строка "привет"";

Есть несколько способов это сделать. Первый – заменить двойные кавычки всей строки на одинарные:

let msg1 = 'строка "привет" ';

Второй способ – использовать так называемое экранирование символов:

let msg1 = "строка **\"**привет**\"**";

мы здесь перед каждой кавычкой внутри строки поставили обратный слеш. Это называется экранированием символов. На практике используются оба способа, так что выбирайте любой удобный для себя. Кстати, если нужно просто отобразить обратный слеш, то его следует записать так:

let msg1 = "строка **\\**";

**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: true значит «да, правильно», а false значит «нет, не правильно». Например:

let isWin = **true**, isCheckedField = **false**;

Булевые значения также могут быть получены из результатов сравнения:

let isGreater = 4 > 1;

и часто применяются в условных операторах, о которых мы будем говорить на последующих занятиях.

**Значение «null»**

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

let idProcess = **null**;

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно».

**Значение «undefined»**

Это свой тип данных, который имеет только одно значение undefined, означающее, что значение не было присвоено. Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined:

let arg;

По идее мы можем присвоить значение undefined любой переменной:

let a = **undefined**;

Но так делать не рекомендуется. Если нам нужно отметить, что переменная не содержит данных, то для этого используется значение null, а undefined – только для проверок: была ли переменная инициализирована или по каким-то причинам ей «забыли» присвоить значение.

**Тип Symbol**

Это новый тип данных (symbol), который впервые появился в спецификации ES6. Он используется для создания уникальных идентификаторов. Например, так:

let id = Symbol();

И в большей степени он удобен для дальнейшего развития JavaScript. Чтобы можно было добавлять новые идентификаторы, не меняя код, в котором он уже мог использоваться.

Символы создаются конструктором Symbol() и могут дополняться необязательным описанием:

let id2 = Symbol("id2");

Их уникальность можно проверить оператором идентичности, о котором мы будем говорить на следующих занятиях. Здесь приведу лишь пример:

console.log(id === id2);

В некоторой степени они похожи на объекты, но относятся к примитивным типам.

Если сейчас вам не совсем понятно, что такое Symbol, просто пропустите этот момент. Когда придет время для его использования, все встанет на свои места.

# Приведение типов, оператор присваивания, функции alert, prompt, confirm

[Смотреть материал на видео](https://www.youtube.com/watch?v=fXvT-eTdLvQ&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/privedenie-tipov-operator-prisvaivaniya-funkcii-alert-prompt-confirm>

На предыдущем занятии мы с вами познакомились с примитивными типами данных. И видели, как переменные автоматически приводились к нужному типу – типу присваиваемых данных. Однако, бывают случаи когда необходимо явно преобразовать выражение к некоторому типу.

Рассмотрим сначала строковые преобразования. Предположим, у нас имеется булевая переменная:

let a = **true**;

console.log(a);

console.log(**typeof** a);

Далее, мы укажем, что хотим преобразовать ее в строковый тип. Это делается так:

a = String(a);

console.log(a);

console.log(**typeof** a);

Вот эта операция String(<выражение>) называется операцией приведения типов. В данном случае булевый тип приводится к строковому и true становится строкой.

По аналогии делается численное преобразование. Пусть у нас есть строка:

let a = **"123"**;

ее можно преобразовать в число вот так:

a = Number(a);

в итоге переменная a будет содержать не строку, а число 123. Кстати, когда явно видно, что данные должны быть преобразованы в числа, то виртуальная JavaScript-машина это делает автоматически. Например, при таком делении двух строк:

console.log(**"6"** / **"3"**);

в консоле увидим значение 2 – результат деления числа 6 на число 3. Здесь приведение к числовому типу было сделано автоматически. Практически во всех математических операциях делается приведение данных к числовому типу, если это возможно, кроме операции сложения. Складывая две строки:

console.log(**"6"** + **"3"**);

мы получим строку «63», а не число 9. Дело в том, что именно этот оператор + выполняет соединение двух строк и к строке 6 была присоединена строка 3, в результате получили строку 63. Причем строки будут соединяться, если хотя бы один из операндов имеет строковый тип. Например, в этих случаях также получим строки:

console.log(**"6"** + 3);

console.log(6 + **"3"**);

И только когда оба операнда числа, результатом будет число 9:

console.log(6 + 3);

Так что складывая две переменные как числа, следует быть уверенными, что они числового типа. Для этого их можно явно привести к числовым значениям, а затем, сложить:

let a = **"6"**, b = **"3"**;

console.log(Number(a) + Number(b));

Если же по каким-то причинам строку нельзя привести к числу:

let a = **"no digit"**;

a = Number(a);

то получим значение NaN. Общие правила численного преобразования такие:

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

console.log( Number(**"   123   "**) ); *// 123*

console.log( Number(**"123z"**) );      *// NaN (ошибка чтения числа в "z")*

console.log( Number(**true**) );        *// 1*

console.log( Number(**false**) );       *// 0*

## Приведение к булевому типу

Для этого используется функция

Boolean(<выражение>)

которая преобразует:

значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined и NaN, в **false**;

все остальные значения в **true**.

Например:

console.log( Boolean(1) ); *// true*

console.log( Boolean(0) ); *// false*

console.log( Boolean(**"0"**) ); *// true*

console.log( Boolean(**"Привет!"**) ); *// true*

console.log( Boolean(**""**) ); *// false*

Обратите внимание, что строчка с нулем «0» преобразуется в true, так как это обычная не пустая строка.

## Оператор присваивания

Теперь подробнее рассмотрим оператор присваивания. Его не стоит путать со знаком = из математики. Он выполняет присваивание переменной, записанной слева от него, значения (или переменной), записанной справа. Например:

let a = 4;

Этот оператор имеет один из самых низких приоритетов (равный 3). Именно поэтому, когда переменной что-либо присваивают, например, x = 2 \* 2 + 1, то сначала выполнится арифметика, а уже затем происходит присваивание.

Также возможно присваивание по цепочке:

let a, b, c;

a = b = c = 2 + 2;

console.log( a, b, c );

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

Далее, оператор = возвращает значение. Например, при присваивании x=value значение value сначала записывается в x, а затем возвращается. В частности, именно поэтому возможна такая запись:

let a, b = 1;

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

console.log( a, b, c );

Здесь сначала переменной a будет присвоено значение 1+1=2, далее оно возвращается оператором присваивания и получаем вычисление разности 3-2=1. В итоге имеем значения a=2, b=1, c=1.

Этот пример приведен, чтобы вы понимали, как это работает, так как иногда это можно увидеть в JavaScript-библиотеках, но писать самим в таком стиле не рекомендуется. Такие трюки не сделают ваш код более понятным или читабельным.

## Функции alert, prompt, confirm

В заключение этого занятия познакомимся с функциями alert, prompt, confirm, которые реализуются в JavaScript движках, работающих в браузерах. То есть, вызывая эти функции, мы предполагаем, что наш скрипт запущен именно в браузере, а не какой-то другой среде.

Первая функция alert() отображает окно с сообщением и приостанавливает дальнейшее исполнение скрипта. Синтаксис функции такой:

alert(<сообщение>);

Например:

alert(**"Hello"**);

выведет на экран модальное окно с сообщением «Hello». *Модальное* означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы до тех пор, пока окно открыто. Для продолжения работы скрипта пользователь должен нажать кнопку «OK». Окно будет закрыто и программа продолжит свою работу.

Следующая функция prompt() отображает модальное окно для ввода каких-либо данных. Синтаксис этой функции такой:

result = prompt(title, [default]);

здесь title – текст для отображения в окне; default – необязательный аргумент, устанавливающий некое значение по умолчанию. Пример:

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

console.log(age);

Выведет окно и попросит ввести ваш возраст. Если пользователь нажмет кнопку OK, то переменная age будет содержать введенный возраст, а при нажатии на «Отмена» (или нажав клавишу ESC) – значение null.

**Заметка:** для браузера IE лучше всегда указывать второй параметр, иначе значение по умолчанию будет равно undefined.

Третья функция confirm() отображает модальное окно с текстом вопроса и имеет такой синтаксис:

result = confirm(question);

здесь question – текст вопроса; result – результат ответа: принимает значение true, если пользователь нажал на кнопку OK и false – в других случаях. Пример:

let isCar = confirm(**"У тебя есть машина?"**);

console.log(isCar);

Все эти функции нам понадобятся в дальнейшем при изучении JavaScript.

# Арифметические операции

[Смотреть материал на видео](https://www.youtube.com/watch?v=VbdKqwLfuG0&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/arifmeticheskie-operacii>

На этом занятии рассмотрим работу операторов (- + \* / % ++ --) в JavaScript. Но вначале немного терминологии, чтобы мы понимали друг друга:

***Операнд****– то, к чему применяется оператор.*

Например, в умножении 5 \* 2 есть два операнда: левый операнд равен 5, а правый операнд равен 2. Иногда их называют «аргументами» вместо «операндов».

***Унарным****называется оператор, который применяется к одному операнду.*

Например, оператор унарный минус "-" меняет знак числа на противоположный:

let a = 1;

a=-a;   *//унарный минус*

***Бинарным****называется оператор, который применяется к двум операндам.*

Тот же минус существует и в бинарной форме:

let a = 1, b = 2;

b-a;    *//бинарный минус*

Из этих примеров видно, что унарный минус и бинарный минус – это два совершенно разных оператора: в первом случае он меняет знак, а во втором – реализует вычитание.

Раз мы заговорили об операторе -, начнем с него. Итак, унарный минус и вычитание для чисел работает очевидным образом по математическим правилам:

let x = 2.8, y = 7.3;

console.log(x-y);

Но, что если один из операндов не будет числом, например, строкой:

let x = **"2.8"**, y = 7.3;

В этом случае он сначала приводится к числу, а затем, применяется данный оператор:

let x = **"2.8"**, y = 7.3;

console.log(x-y);

И в этом случае:

let arg = -x;

console.log(**typeof** arg);

Как видите, тип переменной стал number. Если вместо строки взять любой другой тип, то он также будет приводиться к number:

let x = **true**, y = **null**, z = **undefined**;

console.log(-x);    *// -1*

console.log(-y);    *// -0*

console.log(-z);     *//NaN*

console.log(**typeof** -x, **typeof** -y, **typeof** -z);

Следующий оператор + работает несколько иначе. Для чисел все то же самое:

let x = 4, y = 7;

let sum = x+y;

console.log(sum);

Унарный плюс тоже все пытается перевести в числа, если это возможно:

let x = **"4"**, y = **"не число"**, z = **true**, t = **null**, u = **undefined**;

console.log(**typeof** +x, **typeof** +y, **typeof** +z, **typeof** +t);

console.log(+x, +y, +z, +t, +u);

Все будет приведено к числовому типу с результатом:

4 NaN 1 0 NaN

Здесь строка «не число» и значение undefined преобразовались в NaN, остальные величины в соответствующие числа.

Учитывая, что унарный + не искажает чисел, его часто используют для преобразования выражения к числовому типу вместо Number, то есть:

+"46"  написать гораздо быстрее, чем Number("46")

А вот при сложении двух операндов + работает не только как арифметическая операция. С его помощью можно делать соединение двух строк (или как еще говорят: конкатенацию строк). Например,

let str1 = **"Hello"**, str2 = **"Wordl!"**;

console.log(str1+str2);

Получим строку «HelloWorld!». Добавим пробел между словами:

console.log(str1+**" "**+str2);

получим «Hello World!». То есть, бинарный оператор либо объединяет строки, либо складывает два числа. Но как понять: когда он будет складывать, а когда объединять? Например, вот в таких вариантах:

console.log(5+**"6"**);

console.log(**"6"**+10);

console.log(**"6.46"**+10);

console.log(-2+**"1.3"**);

Здесь везде будет происходить соединение двух строк. Почему так? Правило очень простое. Если один из операндов бинарного оператора + является строкой, то оба операнда преобразуются в строки и объединяются в единую строку. Если же оба операнда – числа, то выполняется их математическое сложение. Например, вот здесь везде значения будут складываться как числа:

console.log(5+2);

console.log(**true**+10);

console.log(**null**+10);

console.log(-2 + +**"1.3"**);

console.log(+**"6"** + +**"3"**);

Обратите внимание на две последние операции. Здесь строки сначала приводятся к числовому типу с помощью унарного +, а затем, выполняется сложение двух чисел.

Но, что если записать сложение вот так:

console.log(3+2+**"2"**);

Результатом будет строка «52», а не «322». Почему? Потому что операторы выполняются слева направо и сначала сложатся два числа 3+2=5, а затем, число 5 плюс строка «2» получим строку «52».

Все следующие операторы \* / % \*\* ++ -- относятся к арифметическим и выполняют автоматическое преобразование выражения к числу.

Операторы \* и / выполняют умножение и деление двух чисел, например,

let x=**"2"**, y = 5;

console.log(x/y);   *//0.4*

Здесь строка «2» автоматически была приведена к числу. И, если кто знаком с языком С++ и ему подобными, обратите внимание, что при делении двух целых чисел результат получается дробным, а не целым. Здесь все работает строго по математическим правилам.

Умножение работает аналогично:

console.log(x\*y);

Эти операторы можно объединять и записывать, например, так:

console.log(x\*y+1.3/6-10);

Соответственно порядок их выполнения определяется приоритетами. Как и в математике приоритет у операций \* и / выше, чем у + и -, поэтому сначала делается умножение и деление, а затем, сложение и вычитание. Если нужно изменить приоритеты, то также как и в математике, используются круглые скобки:

console.log(x\*(y+1.3)/6-10);

Здесь сначала делается сложение в скобках, а затем, все остальные операции в порядке их приоритетов.

Следующая операция % вычисляет остаток от деления одного числа на другое, например,

console.log( 5 % 2 ); *// 1, остаток от деления 5 на 2*

console.log( 8 % 3 ); *// 2, остаток от деления 8 на 3*

console.log( 6 % 3 ); *// 0, остаток от деления 6 на 3*

Причем, в отличие от многих других языков программирования, здесь можно записывать и вещественные числа:

console.log( 5.2 % 2.3 ); *// примерно 0.6*

Но все-таки ее обычно применяют именно к двум целым числам.

Следующая операция \*\* возведение в степень. Например:

console.log( 2 \*\* 2 ); *// 4  (2 \* 2)*

console.log( 2 \*\* 3 ); *// 8  (2 \* 2 \* 2)*

console.log( 4 \*\* 2 ); *// 16  (4 \* 4)*

Она также может работать и с дробными значениями:

console.log( 4 \*\* (1/2) ); *// 2 (степень 1/2 эквивалентна взятию квадратного корня)*

console.log( 8 \*\* (1/3) ); *// 2 (степень 1/3 эквивалентна взятию кубического корня)*

Последние две операции ++ (инкремент) и -- (декремент) увеличивают и уменьшают значение переменной на 1:

let counter = 2, cnt = 5;

counter++; *// работает как counter = counter + 1*

cnt--; *// работает как cnt = cnt - 1*

console.log( counter, cnt ); *// 3, 4*

Операции ++ и -- не только записываются короче, но и быстрее исполняются JavaScript-машиной, чем аналогичные операции

counter = counter + 1  и cnt = cnt - 1

Причем, считается, что для увеличения производительности предпочтительнее использовать префиксную запись:

++counter и --cnt

Но эти две операции (префиксная и постпрефиксная) отличаются не только скоростью исполнения. У них есть другой нюанс. Покажем его на таком примере:

let a, b, c = 10, d = 10;

a = c++;

b = ++d;

console.log( a, b, c, d );

Получим результат: a=10, b=11, c=11, d=11. Обратите внимание, что значение переменной a равно 10, а не 11. Это из-за того, что при постпрефиксной записи сначала выполняется присваивание значение a=c и только потом переменная c увеличивается на 1. При префиксной записи, сначала значение переменной d увеличивается на 1, а затем, число 11 присваивается переменной b. Отсюда и получаются такие результаты. Вот это нужно запомнить и иметь в виду при написании скриптов.

Приоритет у операций ++ и -- выше, чем у операций \* и /, поэтому они корректно будут работать при такой записи:

let a = 4;

console.log(2 \* ++a);

В программировании часто приходится менять значение переменной на определенную величину. Например, увеличить на 5, или уменьшить на 2, или увеличить в 10 раз, уменьшить в 4 раза и так далее. Все это можно выполнить такими короткими операторами:

let a = 0, b = 1;

a += 5;        *// эквивалент: a = a + 5*

b -= 2;        *// эквивалент: b = b - 2*

console.log( a, b );

a \*= 10; *// эквивалент: a = a \* 10*

b /= 4;         *// эквивалент: b = b / 4*

console.log( a, b );

# Условные операторы if и switch

[Смотреть материал на видео](https://www.youtube.com/watch?v=9B9gnKfMj34&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/uslovnye-operatory-if-i-switch>

На этом занятии поговорим об условных операторах. Что это такое? Представьте, что вам нужно вычислить модуль числа, хранящегося в переменной x. Как это сделать? Очевидно, нужно реализовать такой алгоритм.

И в этом алгоритме есть вот такое ветвление программы: при x<0 меняется знак на противоположный, а при других x это не делается. В результате получаем модуль числа в переменной x.

Так вот, чтобы реализовать проверку таких условий в JavaScript имеется два условных оператора: if и switch. И начнем с рассмотрения первого оператора if. В самом простом случае его синтаксис такой:

if(<выражение>) оператор;

Если выражение в круглых скобках истинно, то выполняется оператор, записанный в if. Иначе этот оператор не выполняется. Используя этот оператор, запишем программу для вычисления модуля числа:

let x = -5;

**if**(x < 0) x = -x;

console.log(**"|x| = "** + x);

Здесь операция изменения знака переменной x будет выполняться только для отрицательных величин, а положительные просто выводиться в консоль, минуя эту операцию.

Какие операторы сравнения существуют в JavaScript и как они работают? Многие из них нам известны из школьного курса математики, это:

|  |  |
| --- | --- |
| a > b | Истинно, если a больше b |
| a < b | Истинно, если a меньше b |
| a >= b | Истинно, если a больше или равно b |
| a <= b | Истинно, если a меньше или равно b |
| a == b | Истинно, если a равно b (обратите внимание, для сравнения используется двойной знак равенства) |
| a != b | Истинно, если a не равно b |

Все эти операторы при сравнении возвращают булевое значение: true – истина или false – ложь. Например:

console.log( 2 > 1 );  *// true (верно)*

console.log( 2 == 1 ); *// false (неверно)*

console.log( 2 != 1 ); *// true (верно)*

Результат сравнения можно присвоить переменной, как и любое значение:

let result = 7 > 5; *// результат сравнения присваивается переменной result*

console.log( result ); *// true*

## Сравнение строк

Как вы видите, сравнение двух числовых значений выполняется вполне очевидным образом. Но можно ли, например, сравнивать строки между собой? Оказывается да, можно. Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок. Другими словами, строки сравниваются посимвольно. Например:

console.log( **'Я'** > **'А'** ); *// true*

console.log( **'Кот'** > **'Код'** ); *// true*

console.log( **'Сонный'** > **'Сон'** ); *// true*

Алгоритм сравнения двух строк довольно прост:

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.

Сравнение продолжается, пока не закончится одна из строк. Если обе строки заканчиваются одновременно, и все их соответствующие символы равны между собой, то строки считаются равными. Иначе, большей считается более длинная строка.

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

1. К равна К.
2. о равна о.
3. т больше чем д.

На этом сравнение заканчивается. Первая строка больше.

## Сравнение разных типов данных

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

console.log( **'2'** > 1 ); *// true, строка '2' становится числом 2*

console.log( **'01'** == 1 ); *// true, строка '01' становится числом 1*

При сравнении булевых значений:

console.log( **true** == 1 ); *// true (true приводится к 1)*

console.log( **false** == 0 ); *// true (false приводится к 0)*

console.log( **null** == **undefined** ); *// true (обе величины приводятся к 0)*

Однако, при использовании сравнения с помощью операторов <, >, <=, >= всюду получим false:

console.log( **null** >= **undefined** );  *// false (null – к нулю, undefined – к NaN)*

console.log( **null** <= **undefined** );  *// false (null – к нулю, undefined – к NaN)*

Внимание! Странное сравнение null с нулем:

console.log( **null** > 0 );  *// (1) false*

console.log( **null** == 0 ); *// (2) false*

console.log( **null** >= 0 ); *// (3) true*

Это просто нужно иметь в виду при программировании логики скриптов.

## Строгое сравнение

В JavaScript можно выполнять сравнение двух значений с учетом их типа данных. Для этого используется оператор === (три равно). Применяя его к предыдущему примеру, получим:

console.log( **true** === 1 ); *//false*

console.log( **false** === 0 ); *//false*

Оба результата дадут ложь – false, так как значения, имеющие разные типы считаются не равными друг другу. Та же ситуация будет и при таком строгом сравнении:

console.log( **"5"** === 5 ); *//false*

console.log( **"7"** === 7 ); *//false*

console.log( **null** === **undefined** ); *// false*

Здесь также имеем разные типы данных, поэтому результат – false. А вот если взять строгое неравенство, то получим истину во всех случаях:

console.log( **"5"** !== 5 ); *//true*

console.log( **null** !== **undefined** ); *// true*

Вот так можно сравнивать между собой значения с учетом их типов.

Теперь, когда мы знаем как сравниваются между собой величины, вернемся к нашему условному оператору if. И предположим, что хотим определить знак числа в переменной x. Конечно, проверку можно записать вот так:

let x = -5;

**if**(x < 0) console.log(**"x отрицательное число"**);

**if**(x >= 0) console.log(**"x неотрицательное число"**);

Но можно сделать лучше. Смотрите, мы здесь имеем дело со взаимоисключающими условиями, то есть, они не могут произойти одновременно: либо первое, либо второе. Для таких ситуаций можно использовать ключевое слово else – иначе, чтобы ускорить процесс проверки:

**if**(x < 0) console.log(**"x отрицательное число"**);

**else** console.log(**"x неотрицательное число"**);

Теперь, у нас здесь всего одно условие. Если оно истинно, то выполнится первый console.log, а иначе – второй console.log. Такая программа будет работать быстрее.

В общем случае, синтаксис оператора if else следующий:

if(<выражение>) оператор 1;  
else оператор 2;

И так как после else можно ставить любой оператор, то предыдущую программу можно записать и так:

**if**(x < 0) console.log(**"x отрицательное число"**);

**else** **if**(x > 0) console.log(**"x положительное число"**);

**else** console.log(**"x равен 0"**);

И вообще таких конструкций if else может быть много. Далее, обратим внимание на такой факт: во всех наших примерах по условию шел один оператор – console.log. Но что если нужно выполнить несколько операторов по некоторому условию? Для этого их нужно заключать в фигурные скобки. Например:

let x = -10, sgn = 0;

**if**(x < 0) {

         sgn = -1;

         console.log(**"x отрицательное число"**, sgn);

}

**else** **if**(x > 0) {

         sgn = 1;

         console.log(**"x положительное число"**, sgn);

}

**else** console.log(**"x равен 0"**, sgn);

Здесь по первым двум условиям выполняется два оператора: присвоение значения переменной sgn и вывод результата в консоль. В последнем else записан только один оператор – вывод в консоль, поэтому здесь фигурные скобки ставить не обязательно.

В ряде случаев конструкцию if else удобнее записывать через тернарный условный оператор, который имеет такой синтаксис:

let result = условие ? значение1 : значение2;

Сначала вычисляется условие: если оно истинно, то возвращается значение1, в противном случае – значение2. Например:

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

Получим true, если возраст (age) больше 18, иначе – false. Кстати, проверку из данного примера можно сделать короче, просто прописав

let accessAllowed = age > 18;

здесь оператор > вернет true при возрасте больше 18 и false – в противном случае.

Теперь, когда мы разобрались с базовыми моментами проверки условий, сделаем следующий шаг и попробуем реализовать проверку попадания переменной x в диапазон [2; 7], то есть, условие должно быть истинным, когда x принимает значения в этом диапазоне чисел. Очевидно, что здесь должно быть две проверки: первая – мы проверяем, что x >= 2 и вторая – проверяем, что x <= 7. Если оба этих условия выполняются одновременной, то x попадает в наш диапазон.

Реализовать такую проверку на JavaScript можно так:

let x = 4;

**if**(x >= 2 && x <= 7) console.log(**"x попадает в [2; 7]"**);

**else** console.log(**"x не попадает в [2; 7]"**);

Смотрите, здесь записано два условия, объединенных по И (два амперсанда – это И). В результате, общее составное условие будет считаться истинным, если истинно и первое и второе условие. Если хотя бы одно из этих условий ложно, то ложно и все составное условие. В результате мы корректно реализуем проверку на вхождение значения переменной в диапазон [2; 7].

А теперь давайте реализуем противоположное условие, что x не принадлежит диапазону [2; 7]. Условие будет таким:

let x = 40;

**if**(x < 2 || x > 7) console.log(**"x не попадает в [2; 7]"**);

**else** console.log(**"x попадает в [2; 7]"**);

Здесь в составном условии используется связка по ИЛИ (две вертикальные черты – это ИЛИ) и оно будет истинно, если истинно или первое, или второе условие. То есть, в нашем случае, если x < 2 или x > 7, то делается вывод о невхождении переменной x в указанный диапазон.

Итак, запомните следующие правила:

* условие x >= 2 && x <= 7 истинно, если истинно каждое из подусловий (x>=2 и x <= 7) и ложно, если ложно хотя бы одно из них;
* условие x < 2 || x > 7 истинно, если истинно хотя бы одно из подусловий (x < 2 или x > 7) и ложно, когда оба ложны.

Вот так можно записывать более сложные условия в условном операторе if. Причем они могут комбинироваться в любом сочетании, например:

let x = 4, y = -2;

**if**(x >= 2 && x <= 7 && (y < 0 || y > 5))

         console.log(**"x попадает в [2; 7], y не попадает в [0; 5]"**);

Здесь реализована проверка, что x должно принадлежать [2; 7], а y не принадлежать [0; 5]. И обратите внимание вот на эти круглые скобки. Дело в том, что приоритет у операции && больше, чем у ||, поэтому без скобок у нас бы получилась вот такая проверка:

**if**( (x >= 2 && x <= 7 && y < 0) || (y > 5) )

то есть, мы проверяли бы, что x принадлежит [2; 7] и y меньше нуля ИЛИ y больше 5. Как вы понимаете – это уже совсем другая проверка. Поэтому учитывайте приоритет этих операций при формировании составного условия. Если нужно изменить приоритет – используйте круглые скобки.

## Одиночные проверки

Внутри условия можно прописывать и такие одиночные выражения:

let x = 4, y = **true**, z = **false**;

**if**(x) console.log(**"x = "** + x + **" дает true"**);

**if**(!0) console.log(**"0 дает false"**);

**if**(**"0"**) console.log(**"строка 0 дает true"**);

**if**(!**""**) console.log(**"пустая строка дает false"**);

**if**(y) console.log(**"y = true дает true"**);

**if**(!z) console.log(**"z = false дает false"**);

Вот этот восклицательный знак – это отрицание – НЕ, то есть, чтобы проверить, что 0 – это false мы преобразовываем его в противоположное состояние с помощью оператора отрицания НЕ в true и условие срабатывает. Аналогично и с переменной z, которая равна false.

Из этих примеров можно сделать такие выводы:

1. Любое число, отличное от нуля, дает true. Число 0 преобразуется в false.
2. Пустая строка – это false, любая другая строка с символами – это true.
3. С помощью оператора НЕ можно менять условие на противоположное (в частности, false превращать в true).

Итак, в условиях мы можем использовать три оператора: &&, || и !. Самый высокий приоритет у операции НЕ, следующий приоритет имеет операция И и самый маленький приоритет у операции ИЛИ.

Вот так работает оператор if в JavaScript.

## Оператор switch

Второй условный оператор switch используется, когда из множества возможных вариантов нужно выбрать какой-то один. Например, мы хотим проверить: принимает ли некая переменная одно из значений 1, 2, 3 или 4. Это будет выглядеть так:

let item = 3;

**switch**(item) {

**case** 1: console.log(**"item = 1"**);

**case** 2: console.log(**"item = 2"**);

**case** 3: console.log(**"item = 3"**);

**case** 4: console.log(**"item = 4"**);

**default**: console.log(**"item другое значение"**);

}

И в консоле увидим, что item=3, 4 и другое значение. Почему так и как это все работает? В круглых скобках оператора switch записывается переменная, которая сравнивается со значениями (константами), указанными в case. Далее, через двоеточие пишутся операторы, которые выполняются при совпадении значения переменной с константой.  В нашем случае item=3, срабатывает case 3 и выполняется console.log("item = 3"). Но, оператор switch в JavaScript реализован так, что все последующие операторы тоже будут выполняться. В ряде случаев это бывает полезно и нужно. Если же мы хотим прервать выполнение оператора switch после успешной проверки, нужно записать оператор break вот так:

**switch**(item) {

**case** 1: console.log(**"item = 1"**);**break**;

**case** 2: console.log(**"item = 2"**);**break**;

**case** 3: console.log(**"item = 3"**);**break**;

**case** 4: console.log(**"item = 4"**);**break**;

**default**: console.log(**"item другое значение"**);

}

Теперь в консоле мы видим только строчку item=3. Операторы, стоящие после default выполняются, если не сработала ни одна из проверок. Это необязательное ключевое слово: switch может быть записан и без него, но тогда мы не увидим «item другое значение», если переменная не будет равна 1, 2, 3 или 4.

Так когда же использовать условный оператор if, а когда switch? Правило простое: если нужно проверить переменную на равенство множеству значений, то лучше использовать switch – он сработает быстрее, чем if. Во всех других случаях применяется оператор if.

В практике программирования в 99% случаев используется if и лишь для некоторых частных проверок – оператор switch.

# Операторы циклов for, while, do while

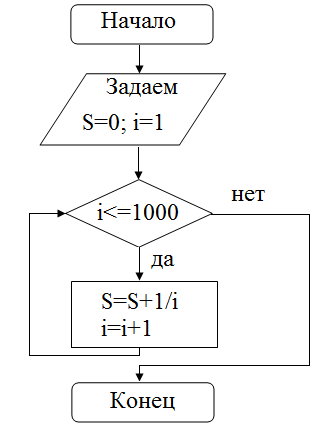
[Смотреть материал на видео](https://www.youtube.com/watch?v=oc6HYRgY5uw&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/operatory-ciklov-for-while-do-while>

Ни одна сколь-нибудь серьезная программа на JavaScript не обходится без циклов. Что такое циклы? Представьте, что вам необходимо вычислить вот такую вот сумму:

C:\Users\Andrey_User\Downloads\image001.png

Расписывать все это через тысячу слагаемых не очень то удобно. И к тому же число слагаемых может зависеть от значения переменной и быть неопределенным. В таких задачах без циклов не обойтись. Представим сначала эту задачу в виде вот такой блок-схемы.



Здесь мы сначала инициализируем переменные S=0 и i=1, а дальше идет такая конструкция. Сначала проверяем условие, если i <=1000, то к предыдущему значению S прибавляем текущее слагаемое 1/i, а затем, i увеличиваем на 1. И снова переходим на проверку: является ли i <= 1000. Если так, то вычисление суммы продолжается, а иначе завершаем работу алгоритма.

## Оператор цикла while

Эта блок схема показывает принцип работы цикла и отсюда хорошо видно, что цикл состоит из двух составляющих: условия цикла и тела цикла, то есть, операторов, которые выполняются внутри цикла. В частности, такую конструкцию можно реализовать с помощью оператора цикла while, по следующему синтаксису:

while(<выражение>) {

      [тело цикла]

}

И программа будет выглядеть так:

let S=0, i=1;

while(i <= 1000) {

         S += 1/i;

         ++i;

}

console.log(S);

В качестве выражения в цикле while можно писать все те же самые условия, что и в условном операторе if. Например, можно вычислять сумму S пока либо i<=1000, либо S < 5. Такое условие запишется так:

while(i <= 1000 && S < 5)

здесь цикл будет работать пока i<=1000 и S<5 как только одно из подусловий станет ложным, все составное условие становится ложным и цикл завершит свою работу.

Рассмотрим еще один пример. Будем вычислять сумму

C:\Users\Andrey_User\Downloads\image003.png

и запишем все вот в такой краткой форме:

let S=0, i=1;

while((S += i++) < 100 );

console.log(S);

Здесь сумма вычисляется непосредственно внутри выражения, а тело цикла отсутствует – после while стоит точка с запятой. Так тоже можно делать, но не рекомендуется, т.к. это усложняет понимание работы программы другими программистами. Лучше все расписывать в несколько строк.

## Оператор цикла for

Самый часто используемый оператор цикла – это оператор for, который имеет такой синтаксис:

for(<инициализация счетчика>;<условие>;<изменение счетчика>) {  
     [тело цикла]  
}

Давайте перепишем нашу программу подсчета суммы

с помощью цикла for, получим:

let S=0;

**for**(let i=1;i <= 1000; ++i) S += 1/i;

console.log(S);

Здесь весь цикл записан буквально в одну строчку, а тело цикла состоит из одного оператора – подсчета суммы ряда. Именно поэтому здесь не стоят фигурные скобки – для одного оператора они не обязательны.

Вначале идет инициализация счетчика let i=1, через точку с запятой – условие цикла и далее через точку с запятой правило изменения счетчика: увеличение на 1 на каждой итерации. Такая запись цикла выглядит гораздо короче и нагляднее, поэтому оператор for так часто используется на практике.

Вторым примером рассмотрим задачу вычисления значений линейной функции

C:\Users\Andrey_User\Downloads\image004.png

Скрипт будет выглядеть так:

let f, k = 0.5, b = 2;

**for**(let x=0;x <= 1; x += 0.1) {

         f = k\*x+b;

         console.log(f);

}

Здесь все достаточно очевидно, только теперь счетчик x у нас меняется от 0 до 1 с шагом 0,1. Далее идет тело цикла из двух операторов, поэтому они заключены в фигурные скобки.

Цикл for можно записывать и в таком виде:

let x=0;

**for**(;x <= 1; x += 0.1) { ... }

или так:

let x=0;

**for**(;x <= 1; ) {

         f = k\*x+b;

         console.log(f);

         x += 0.1;

}

или, даже так:

let x=0;

**for**(;;) {

**if**(x > 1) **break**;

         f = k\*x+b;

         console.log(f);

         x += 0.1;

}

В последнем случае мы не прописали никакого условия, поэтому цикл for работал бы вечно. Поэтому в теле цикла добавлена проверка: если x>1, то выполняется оператор break, который прерывает работу цикла. Забегая вперед, скажу, что оператор break прерывает работу любого цикла, не только for.

## Оператор цикла do while

Последний оператор цикла, который мы рассмотрим – это цикл do while. Он имеет следующий синтаксис:

do {  
      [тело цикла]  
}  
while(<выражение>);

Отличие этого цикла от двух предыдущих в том, что он сначала выполняет тело цикла и только потом проверяет условие цикла: если оно истинно, то цикл продолжается, если ложно, то завершается.

Когда может использоваться такая конструкция? Например, если пользователь вводит с клавиатуры некий пароль и мы должны проверить: правильно ли он был введен. Если правильно, то цикл завершается, иначе просим пользователя ввести пароль повторно. Такую программу можно реализовать так:

**const** PSW = **"password"**;

let psw = **null**;

**do** {

         psw = prompt(**"Введите пароль"**, **""**);

}

while(psw != PSW);

console.log(**"Вы вошли в систему!"**);

## Вложенные циклы

Итак, мы с вами рассмотрели три цикла: while, for и do while. Все эти циклы можно комбинировать друг с другом. То есть, создавать вложенные циклы. Когда это необходимо? Например, для подсчета вот такой двойной суммы ряда

C:\Users\Andrey_User\Downloads\image005.png

Программа для ее вычисления будет выглядеть так:

let S=0, M=10, N=5;

**for**(let i=1;i <= N; ++i)

**for**(let j=1; j <= M; ++j)

                   S += i\*j;

console.log(**"S = "** + S);

Мы здесь сначала пробегаем все значения j от 1 до M при фиксированном i=1, затем, значение i увеличивается на 1, становится 2 и при этом i снова пробегаются значения j от 1 до M. И так пока i не превысит значение N. То есть, второй цикл вложен вот в этот первый. И таких вложений можно делать сколько угодно.

## Операторы break и continue

Всеми тремя циклами можно управлять с помощью двух специальных операторов: break и continue. Об операторе break мы уже говорили – он прерывает работу любого цикла. Но только одного. Например, здесь при вложенных циклах:

**for**(let i=1;i <= N; ++i)

**for**(let j=1; j <= M; ++j) {

**if**(j == 5) **break**;

                   S += i\*j;

         }

будет остановлен только один внутренний цикл и управление будет передано внешнему циклу. Чтобы прервать оба цикла можно использовать оператор break с меткой вот так:

all: **for**(let i=1;i <= N; ++i)

**for**(let j=1; j <= M; ++j) {

**if**(j == 5) **break** all;

                   S += i\*j;

         }

Мы здесь явно указываем внешний цикл с меткой all, который следует прервать. В результате будут остановлены оба цикла. Здесь следует иметь в виду, что в данном случае метки применяются именно к операторам циклов. Если метку записать выше или ниже, она либо не сработает, либо применится к ближайшему циклу, содержащему оператор break.

Следующий оператор **continue** позволяет пропускать тело цикла и перейти к следующей итерации, не прерывая работу самого цикла. Например, мы хотим перебрать все целые значения от -5 до 5, исключая значение 0. Такую программу можно реализовать так:

**for**(let i=-5;i <= 5; ++i) {

**if**(i == 0) **continue**;

         console.log(**"i = "** + i);

}

При выполнении этой программы увидим, что в консоль выведены все значения кроме нуля. Так как при i=0 срабатывает условие и выполняется оператор continue. Все что находится после этого оператора пропускается и цикл продолжается уже со значением i=1.

Оператор continue также можно использовать с меткой, подобно оператору break.

Вот мы с вами рассмотрели базовые реализации операторов циклов. Забегая вперед, скажу, что в JavaScript имеются и другие реализации для циклов, но о них речь пойдет когда возникнет необходимость в их применении.

# Объявление функций, аргументы по умолчанию

[Смотреть материал на видео](https://www.youtube.com/watch?v=ms8nJXsXflk&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/obyavlenie-funkciy-argumenty-po-umolchaniyu>

Начиная с этого занятия, мы будем рассматривать способы объявления и вызовов функций. Что вообще такое функции в JavaScript. В действительности, это такие объекты специального вида, которые можно многократно вызывать в нужных местах скрипта, выполняя какое-то частое действие. Например, можно задать функцию для вычисления модуля числа, или функцию для вычисления площади или периметра прямоугольника и так далее. То есть, если в программе требуется многократно выполнять какое-либо действие, то его лучше реализовать в виде функции, а не писать каждый раз заново.

Примеры некоторых функций вы уже знаете – это рассмотренные ранее alert(), prompt() и confirm(). Но мы можем объявить и свои собственные, используя такой синтаксис:

function имя\_функции([аргументы]) {

          [тело функции]

}

Здесь имя функции придумывается самим программистом также как и для имен переменных. Но, если имя переменной должно отвечать на вопрос кто, что, то функция – это действие, а значит, ее имя должно соответствовать глаголу, например,

showText, getString, out\_log, calcSum, checkForm и т.д.

Приведем простейшее объявление функции, которая выводит в консоль сообщение «Вызов функции»:

**function** out\_log() {

         console.log(**"Вызов функции"**);

}

Теперь, чтобы вызвать эту функцию, нужно записать ее имя и после нее поставить круглые скобки:

out\_log();

Если мы напишем еще один вызов:

out\_log();

то функция будет вызвана дважды. Вот так задаются и вызываются функции в JavaScript. Причем вызывать функции можно до их определения. Если записать строчки out\_log() до объявления функции, то так тоже будет работать. Почему это работает? В отличие от обычных переменных, которые нужно сначала объявлять и только потом использовать, JavaScript-машина все функции, объявленные явным образом, анализирует и определяет до выполнения скрипта. Поэтому, где бы мы ни объявили глобальную функцию, она становится доступной в любой точке скрипта.

Как правило, функции имеют входные параметры. Например, чтобы вывести в консоль произвольную строчку, следует переписать нашу функцию так:

out\_log(**"1-й вызов"**);

out\_log(**"2-й вызов"**);

**function** out\_log(msg) {

         console.log(msg);

}

Здесь указан один параметр msg, который и выводится в консоль. Соответственно, при вызове такой функции следует передать ей этот параметр, например, так как это сделано у нас. В общем случае ей могут быть переданы любые данные:

out\_log(1);

out\_log(**true**);

и так далее, мы не ограничены никакими типами. Если функция принимает более одного аргумента, то они разделяются запятыми. Например:

**function** showMessage(from, text) {

         let msg = from + **": "** + text;

         console.log(msg);

}

Со следующими вызовами:

showMessage(**"Аня"**, **"Привет!"**);

showMessage(**"Аня"**, **"Как дела?"**);

Внутри функции showMessage() объявлена переменная msg. Она будет видна только внутри функции и отсутствует за ее пределами. Это сделует учитывать: все переменные объявленные внутри функций существуют только внутри них и не существую за их пределами. Это называется областью видимости переменных. В данном случае область видимости ограничивается функцией.

Но из функции можно обращаться к переменным, объявленным за ее пределами непосредственно в скрипте. Например:

**function** showMessage(from, text) {

         let msg = from + **" "** + email + **": "** + text;

         console.log(msg);

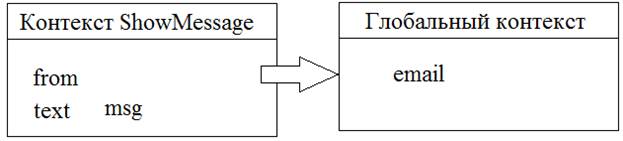
}

let email = **"anna@m.ru"**;

showMessage(**"from"**, **"Привет!"**);

showMessage(**"from"**, **"Как дела?"**);

Мы здесь объявили переменную email после объявления функции, но до ее вызова. Поэтому, когда функция будет вызываться, она сначала просмотрит свою область видимости (свой контекст) и не найдя эту переменную, обратится к контексту следующего уровня, в данном случае глобальной области видимости, где у нас и определена эта переменная. Соответственно, ее значение и будет использовано при вызове. Графически это можно представить так:



Причем, если мы изменим значение email внутри функции, то это приведет к ее изменению и в глобальной области, например:

**function** showMessage(from, text) {

         email += **"+"**;

         let msg = from + **" "** + email + **": "** + text;

         console.log(msg);

}

При повторном вызове функции у нас email будет отображаться уже два плюса. То есть, переменная была изменена при первом вызове функции. Но, если объявить такую же переменную внутри функции:

**function** showMessage(from, text) {

         let email = **"+"**;

         let msg = from + **" "** + email + **": "** + text;

         console.log(msg);

}

То она будет найдена во внутреннем контексте и обращения к глобальной области происходить не будет, то есть, здесь мы будем работать исключительно с локальной переменной email.

А что будет, если вызвать showMessage с одним агрументом:

showMessage(**"from"**);

Функция будет вызвана, но второй параметр text станет равным undefined. Если же мы хотим, чтобы вместо undefined использовалось какое-либо значение по умолчанию, то его следует явно прописать при объявлении функции:

**function** showMessage(from, text=**"Как дела?"**) {

         let msg = from + **" "** + email + **": "** + text;

         console.log(msg);

}

Обратите внимание, что это будет работать только в новых версиях стандарта JavaScript, в старых версиях приходилось бы делать так:

**function** showMessage(from, text) {

**if**(text === **undefined**) text =**"Как дела?"**;

         let msg = from + **" "** + email + **": "** + text;

         console.log(msg);

}

Рассмотренные выше варианты функций что-то делали, но ничего не возвращали. Но часто бывают задачи, когда функция должна возвращать некоторое значение. Например, при вычислении модуля числа функция возвращает значение модуля. Это реализуется следующим образом:

**function** abs(x) {

**if**(x < 0) x = -x;

**return** x;

}

let res = abs(-5);

console.log(res);

Мы в этой функции сначала вычисляем модуль переменной x, а затем, с помощью оператора return возвращаем это значение. То есть, оператор return показывает, что должна возвратить функция. Например, если переписать функцию вот так:

**function** abs(x) {

**if**(x < 0) x = -x;

**return** 0;

}

то она станет все время возвращать 0 вне зависимости от значения x. Но нам в данной реализации нужно вернуть именно значение x, поэтому пишем return x.

В операторе return допускается записывать целые выражения, например, вот такие реализации функций также будут корректно работать:

**function** abs(x) {

**return** (x < 0) ? -x : x;

}

**function** sum(a, b) {

**return** a+b;

}

Одной из частых ошибок, которые делают начинающие программисты – располагают возвращаемое значение на следующей строчке после оператора return:

**function** abs(x) {

**return**

                   (x < 0) ? -x : x;

}

Если все записать так, то интерпретатор JavaScript автоматически поставит точку с запятой после return и у нас в функции будет два независимых оператора. Причем, функция будет возвращать уже не значение модуля, а значение undefined, т.к. в операторе return уже ничего не будет записано.

При необходимости, возвращаемые значения можно записать в несколько строчек, записав их в круглых скобках:

**function** abs(x) {

**return** (

                   (x < 0) ? -x : x

         );

}

Следует отметить, что функция завершает свою работу как только встречается оператор return. Например, в следующей реализации вызова console.log() не будет, так как он стоит после оператора return:

**function** abs(x) {

**return** (x < 0) ? -x : x;

         console.log(x);

}

Это свойство удобно использовать для досрочного завершения функции, например при проверке деления на ноль:

**function** div(a, b) {

**if**(b == 0) **return** **Infinity**;

**return** a/b;

}

Здесь при b=0 деление выполняться не будет, а сразу вернется значение Infinity.

В JavaScript функции всегда возвращают значения: вне зависимости есть или нет в них оператора return. Если оператора нет, то функция возвратит значение undefined:

**function** out\_log() {

         console.log(**"Вызов функции"**);

}

let res = out\_log();

console.log(res);

Или, если просто написать return, будет то же самое:

**function** out\_log() {

**return**;

         console.log(**"Вызов функции"**);

}

В правильно структурированной программе функции должны выполнять какое-то одно простое действие. Не стоит их перегружать задачами. Например, приведенную ниже функцию нахождения простых чисел

**function** showPrimes(n) {

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

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

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

    }

    console.log( i ); *// простое*

  }

}

Лучше записать в виде двух функций:

**function** showPrimes(n) {

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

**if** (!isPrime(i)) **continue**;

    console.log(i);  *// простое*

  }

}

**function** isPrime(n) {

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

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

  }

**return** **true**;

}

Очевидно, что второй вариант легче для понимания: вместо куска кода мы видим название действия (isPrime). Иногда разработчики называют такой код **самодокументируемым**. Таким образом, допустимо создавать функции, даже если мы не планируем повторно использовать их. Такие функции структурируют код и делают его более понятным.

Вот мы с вами рассмотрели базовые принципы создания собственных функций в JavaScript.

# Function Expression, анонимные и callback-функции

[Смотреть материал на видео](https://www.youtube.com/watch?v=VvmVeELTVoE&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/function-expression-anonimnye-i-callback-funkcii>

Синтаксис, который мы использовали на предыдущем занятии для объявления функций, называется Function Declaration (объявление функции):

**function** showMsg() {

         console.log(**"Hello!"**);

}

Существует ещё один синтаксис создания функций, который называется Function Expression (функциональное выражение). Оно выглядит вот так:

let showMsg = **function**() {

         console.log(**"Hello!"**);

};

Здесь мы создаем функцию и явно присваиваем ее переменной showMsg. На самом деле в предыдущем примере происходит то же самое, только записано в неявном виде. Обратите внимание, в конце следует ставить точку с запятой, т.к. мы инициализируем переменную showMsg по синтаксису:

let <переменная> = <значение>;

Значение этой переменной можно вывести в консоль и посмотреть чему она равна:

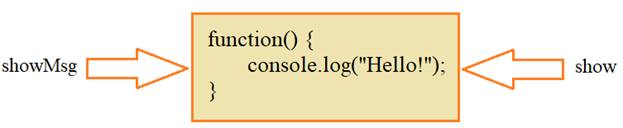
console.log( showMsg );

console.log( **typeof** showMsg );

Да, переменная showMsg содержит искомую функцию, точнее ссылку на функцию. А вот тип переменной – это специальный тип данных, означающий функцию. На самом деле – это объект JavaScript специального вида. Если мы вернем запись функции по синтаксису Function Declaration, то при выводе в консоль ничего не изменится. Это как раз и говорит, что оба объявления подобны. Подобны, но не идентичны! Об их различиях речь пойдет позже в этом занятии.

Вернем запись в виде Function Expression. Здесь у нас имеется переменная showMsg, которая ссылается на функцию. А раз так, то можем ли мы ее скопировать в другую переменную и получить копию этой функции? Почти, только мы скопируем не саму функцию, а ссылку на нее, в результате обе переменные будут обращаться к одной и той же функции:

let show = showMsg;



Теперь вызовем функцию через эти переменные. Для этого после их имени нужно поставить круглые скобки:

show(); *// 1-й вызов функции*

showMsg(); *//2-й вызов функции*

Запомните этот момент! Без круглых скобок мы работаем с переменными show и showMsg, а с круглыми скобками – вызываем соответствующие функции. Этот синтаксис следует правильно понимать и помнить.

Далее, все это будет работать, если функцию объявить и как Function Declaration, так как обе записи – это, по сути, одно и тоже: создается переменная showMsg, которая ссылается на объект-функцию.

Ну, хорошо, разобрались. Но зачем нужно объявление по Function Expression? Для ответа на этот вопрос приведем такой пример. Предположим, у нас есть функция, которая запрашивает у пользователя согласие на использование cookies в браузере. И реализуем ее так:

**function** agreeCookies(question, yes, no) {

**if** (confirm(question)) yes();

**else** no();

}

**function** agreeYes() {

   console.log(**"Вы приняли соглашение о cookies"**);

}

**function** agreeNo() {

  console.log(**"Вы отказались от использования cookies"**);

}

agreeCookies(**"Наш сайт использует cookies. Нам нужно ваше согласие"**, agreeYes, agreeNo);

Смотрите, у функции agreeCookies три параметра: первый отвечает за текст сообщения, а два других – содержат ссылки на функции, которые вызываются при согласии или несогласии с использованием cookies. В результате мы получили довольно гибкий код: вся логика по обработке вынесена в отдельные функции, которые могут обрабатывать запрос самыми разными способами. При этом сама функция agreeCookies остается неизменной. Аргументы yes, no, которые содержат ссылки на функции, еще называют callback-функциями (от англ. call back – обратный вызов), то есть, отложенный вызов, если в нем будет необходимость.

Данный пример можно записать и короче, вот так:

**function** agreeCookies(question, yes, no) {

**if** (confirm(question)) yes();

**else** no();

}

agreeCookies(

**"Наш сайт использует cookies. Нам нужно ваше согласие"**,

**function** () { console.log(**"Вы приняли соглашение о cookies"**); },

**function** () { console.log(**"Вы отказались от использования cookies"**); }

);

Мы здесь непосредственно при вызове функции agreeCookies передаем ей все необходимые параметры: строку запроса и две функции, которые объявлены прямо внутри вызова agreeCookies. У них нет имен, поэтому такие функции называются анонимными. Соответственно, эти функции доступны и существуют только в момент вызова agreeCookies и не существуют за ее пределами.

Программисты на JavaScript очень часто используют такой подход при написании скриптов. И это считается хорошим стилем в программировании.

Итак, можно заключить, что имя функции – это переменная, которая содержит специальные данные – функцию. Эти данные можно воспринимать не как обычные статичные значения, а как некое действие, которое может быть выполнено в любом месте скрипта.

Теперь посмотрим на отличия между двумя объявлениями функций по Function Declaration и Function Expression. Как мы говорили на предыдущем занятии, функции объявленные по Function Declaration:

**function** agreeCookies(question) {

**if** (confirm(question)) console.log(**"Вы приняли соглашение о cookies"**);

**else** console.log(**"Вы отказались от использования cookies"**);

}

Могут вызываться и до и после их объявления, то есть вот так:

agreeCookies(**"Наш сайт использует cookies. Нам нужно ваше согласие"**);

**function** agreeCookies(question) {

**if** (confirm(question)) console.log(**"Вы приняли соглашение о cookies"**);

**else** console.log(**"Вы отказались от использования cookies"**);

}

agreeCookies(**"Наш сайт использует cookies. Нам нужно ваше согласие"**);

Если же мы объявим эту же функцию как Function Expression:

let agreeCookies = **function**(question) {

**if** (confirm(question)) console.log(**"Вы приняли соглашение о cookies"**);

**else** console.log(**"Вы отказались от использования cookies"**);

};

То вызвать ее до объявления переменной agreeCookies будет невозможно, так как она не будет существовать. А вот после объявления вызовется без проблем.

Далее, если у нас включен строгий режим "use strict" (а мы договорились его использовать), то объявления по Function Declaration видны только внутри блока кода, в котором они объявлены. Например, вот в такой программе:

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

**if**(age < 18) {

**function** setAccess() {

   console.log(**"Доступ запрещен: вы слишком молоды"**);

}

}

**else** {

**function** setAccess() {

         console.log(**"Доступ разрешен"**);

   }

}

setAccess();

В блоках if else объявляется функция setAccess, которую предполагается вызвать и определить: разрешать доступ или нет. Но здесь возникнет ошибка, так как функция не будет существовать за пределами блоков if else. Это ограничение можно обойти, используя Function Expression:

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

let setAccess = **null**;

**if**(age < 18) {

setAccess = **function** () {

console.log(**"Доступ запрещен: вы слишком молоды"**);

};

}

**else** {

setAccess = **function** () {

console.log(**"Доступ разрешен"**);

};

}

setAccess();

Теперь переменная setAccess будет содержать ссылку на функцию, в зависимости от указанного возраста и после блока if else вызывается без проблем. Кстати, эту же программу можно записать гораздо короче, вот так:

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

let setAccess = (age < 18) ?

**function** () {console.log(**"Доступ запрещен: вы слишком молоды"**); } :

**function** () { console.log(**"Доступ разрешен"**); };

setAccess();

Здесь переменной setAccess будет присваивается анонимная функция в зависимости от указанного возраста.

Так как же понимать: когда использовать объявления по Function Declaration, а когда по Function Expression? Обычно, поступают так. В первую очередь следует рассматривать синтаксис Function Declaration. Он дает больше свободы в организации кода: функции, объявленные таким образом, можно вызывать и до и после их объявления. К тому же такие функции проще воспринимаются в тексте скрипта.

Объявления по Function Expression следует использовать только тогда, когда по каким-то причинам нам не подходит объявление по Function Declaration. Например, при реализации callback-функций. Или вызова функции за пределами блока их объявления. Все эти примеры мы с вами только что рассматривали.

# Анонимные и стрелочные функции

[Смотреть материал на видео](https://www.youtube.com/watch?v=NjTTfljy_SU&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/anonimnye-i-strelochnye-funkcii>

Давайте поближе познакомимся с анонимными функциями, то есть, с функциями, у которых нет имени:

**function** () {

console.log("это анонимная функция");

};

Но просто объявить такую функцию нельзя, возникнет ошибка выполнения такого кода. Это легко поправить, если присвоить эту функцию какой-либо переменной:

let anonym = **function**() {

console.log("это анонимная функция");

};

Теперь ошибки синтаксиса возникать не будет. Мало того, мы теперь можем даже вызвать эту функцию через переменную anonym:

anonym();

Вспоминаем этот синтаксис: чтобы вызвать какую-либо функцию, после ее имени нужно записать круглые скобки. После выполнения мы в консоле увидим сообщение «это анонимная функция».

Этот же синтаксис прекрасно сработает и с именованной функцией, например, так:

let anonym = **function** func() {

  console.log("это анонимная функция");

};

anonym();

Правда, если мы попробуем вызвать функцию по ее второму имени:

func();

то возникнет ошибка, что func не определена. Дело в том, что по приоритетам в глобальной области видимости (то есть, вне функции) будет существовать только переменная anonym, но не переменная func. Имя func существует только внутри самой функции, за ее пределами такого имени уже нет.

Вернемся к анонимной функции. Можно ли ее вызвать, не присваивая ее переменной? Оказывается, можно. Для этого поместим функцию в круглые скобки и в конце поставим еще одни круглые скобки для ее вызова:

"use strict";

(**function** () {

console.log("это анонимная функция");

})();

Обратите внимание, чтобы этот код сработал, после директивы “use strict” нужно поставить точку с запятой. Без точки с запятой интерпретатор будет полагать, что мы прописываем аргумент у строки “use strict” и получим синтаксическую ошибку.

Теперь наша функция вызывается непосредственно в том месте, где объявлена. Это называется функциональным выражением. То есть, когда интерпретатор встречает такую запись, он немедленно выполняет код, находящийся внутри анонимной функции. Функция вызывается благодаря круглым скобкам, записанным после ее объявления. Как пример, такие функции часто используют как аргументы callback-функций, о которых мы немного говорили на прошлом занятии. Приведу здесь еще один пример. Вызовем анонимную функцию через секунду после запуска скрипта. Это делается с помощью стандартной JavaScript-функции setTimeout:

setTimeout(**function** () {

console.log("это анонимная функция");

}, 1000);

Здесь первый агрумент – это callback-функция, а второй – задержка исполнения в миллисекундах (1000 мс = 1 сек).

## Стрелочные функции

В новой спецификации стандарта языка JavaScript – ES6 вводится понятие стрелочных функций. Это такое развитие или, скорее, упрощение анонимных функций. Что это такое? Представим, что у нас вот такая анонимная функция:

let anonym = **function** () {

console.log("это анонимная функция");

};

Преобразуем ее в стрелочную. Уберем ключевое слово function, а вместо фигурных скобок запишем вот такие символы => - очень похоже на стрелку.

let anonym = () => console.log("это анонимная функция");

anonym();

Предыдущую функцию закомментируем. Все, получили стрелочную функцию. Ее можно вызвать через переменную anonym и она будет работать аналогично закомментированной анонимной функции.

Возможно, такая запись функции выглядит несколько необычно? Но это дело привычки. В дальнейшем она будет также хорошо восприниматься как и обычные функции.

Но особенности стрелочных функций на этом не заканчиваются. Предположим, мы хотим, чтобы такая функция возвращала какой-либо результат. В обычных функциях для этого используется оператор return. Здесь это делается несколько иначе. Давайте возвратим стрелочной функцией строку, которая выводилась в консоль. Для этого ее нужно записать так:

let anonym = () => "это анонимная функция";

console.log (anonym() );

Смотрите как элегантно это выглядит. Если мы хотим что-то вернуть, то просто пишем возвращаемое выражение после стрелки. И никакого оператора return! Мало того, его здесь даже нельзя прописать – будет синтаксическая ошибка. Это все работает только в таком виде. То есть, когда телом стрелочной функции выступает какое-либо одно выражение (в данном случае – это строковый литерал), то записывая его без фигурных скобок, оператор return подразумевается.

Как пример можно записать и такое выражение:

let anonym = () => 48+30;

Увидим в консоле результат суммы – число 78. И так далее. Но, если мы поместим тело функции в фигурные скобки, то возврат уже работать не будет:

let anonym = () => {48+30};

Мы получим значение undefined, как правило, означающее, что функция ничего не возвращает. Чтобы все работало, оператор return здесь обязателен:

let anonym = () => {**return** 48+30};

Фигурные скобки в стрелочных функциях используются, если тело функции содержит более одного выражения:

let anonym = () => {

let a = 48, b = 30;

**return** a+b;

};

Здесь фигурные скобки обязательны и без них код работать уже не будет.

Далее, стрелочным функциям можно передавать аргументы, как и обычным функциям. И, как вы наверное уже догадались, они прописываются внутри круглых скобок, например, так:

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

console.log ( sum(3, 4) );

Смотрите, как просто можно записать стрелочную функцию для сложения двух значений. Кстати, если функция принимает один параметр, то круглые скобки можно не прописывать:

let show = message => "Сообщение: "+message;

console.log ( show("привет мир!") );

Во всех других случаях круглые скобки обязательны:

let show = (from, msg) => {

let text = from+": "+msg;

**return** text;

}

console.log ( show("Самообразование", "Привет мир!") );

Стрелочные функции, несмотря на всю их простоту, имеют свои особенности. Например, у них нет своего контекста при их вызове, как у обычных функций. И они чаще всего используются в ООП. Но обо всем этом пока не задумывайтесь, на данном этапе нужно в целом понять как они записываются и вызываются.

# Рекурсивные функции

[Смотреть материал на видео](https://www.youtube.com/watch?v=9nO4dpxpb84&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/rekursivnye-funkcii>

Начиная с этого занятия, мы немного глубже рассмотрим работу с функциями в JavaScript и начнем с понятия рекурсии.

*Рекурсивные функции – это функции, которые вызывают сами себя.*

Понять работу рекурсии проще всего на примере. Предположим, мы хотим вычислить значение числа x в степени n. Для простоты степень n у нас будет натуральным числом: n = 1, 2, 3,… Один из вариантов реализации может быть такой:

**function** pow(x, n) {

let res = 1;

**for**(let i = 0;i < n;++i)

   res \*= x;

**return** res;

}

Вызываем функцию, проверяем как она работает:

console.log( pow(2, 2) );

Реализуем эту же функцию через рекурсию. Для этого заметим, что

C:\Users\Andrey_User\Downloads\image001.gif

то есть, для вычисления значения на текущем n-м шаге достаточно взять значение на предыдущем n-1-м шаге и умножить его на x. Эта формула, по сути, отражает принцип рекурсии. В нашем случае она будет выглядеть так:

x\*pow(x,n-1)

Получаем такую реализацию:

**function** pow(x, n) {

**if**(n <= 0) **return** 1;

**else** **return** x\*pow(x, n-1);

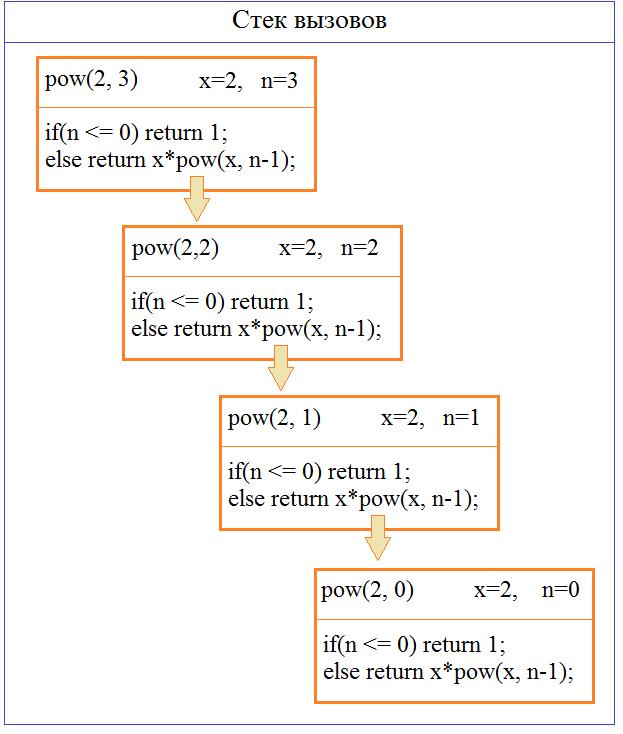
}

let res = pow(2, 3);

Как это работает? Сначала вызывается функция pow(2, 3). Она помещается в стек вызова функций, в котором хранится контекст выполнения (execution context) текущей функции pow(2, 3). Далее, выполняется тело функции. Проверяется первое условие. Оно оказывается ложным, так как 3 <= 0 дает false. Поэтому идет переход на else и прежде чем выполнить оператор return, снова вызывается та же функция pow(2, 2).

Выполнение функции pow(2, 3) останавливается, в стек помещается контекст выполнения второй функции pow(2, 2) и она запускается. Снова проверяется первое условие, оно ложно, переходим по else к оператору return и опять вызываем функцию pow(2, 1).

Здесь снова все повторяется, в стек помещается очередной вызов функции и по условию вызывается следующая функция pow(2, 0).



Теперь первое условие становится истинным и функция pow(2,0) возвращает значение 1 и рекурсия не идет дальше вглубь – она останавливается. Функция pow(2.0) завершается, ее контекст выполнения удаляется из стека вызовов и управление передается функции pow(2, 1). Но она частично уже выполнена. Поэтому, берется значение 1 от pow(2, 0), результат умножается на x=2 и величина 2 возвращается функцией pow(2, 1).

Контекст выполнения функции pow(2,1) удаляется из стека, управление переходит к вышестоящей функции pow(2,2) и здесь мы уже имеем результат умножения x\*x, то есть, 2\*2 = 4. Далее, возвращаемся к самой верхней функции pow(2,3), здесь 2\*4=8 и этот результат становится результатом вычисления рекурсивной функции.

Рекурсию очень удобно применять для обхода свойств объекта сложной структуры. Представьте, что у нас есть компания, состоящая из нескольких отделов и в каждом отделе работают свои сотрудники:

let company = {

sales: [{name: 'Иван', salary: 1000}, {name: 'Михаил', salary: 600}],

development: {

sites: [{name: 'Евгений', salary: 2000}, {name: 'Алексей', salary: 1800}],

  internals: [{name: 'Федор', salary: 1300}]

}

};

И нужно написать функцию вычисления суммарной зарплаты для всей компании и для каждого из отделов. Это можно реализовать так:

**function** sumSalary(department) {

**if**(Array.isArray(department)) {

**return** department.reduce((prev, current) => prev + current.salary, 0);

}

**else** {

   let sum = 0;

**for**(let prop **in** department) {

    sum += sumSalary(department[prop]);

   }

**return** sum;

}

}

И вызывать для любого отдела или компании в целом:

console.log( sumSalary(company) );

console.log( sumSalary(company.sales) );

console.log( sumSalary(company.development) );

Вот что из себя представляют рекурсивные функции. У них есть только одно важное ограничение: стек вызова не бесконечный – нельзя рекурсию делать слишком глубокой. Разумное значение – не более 1000 рекурсивных итераций. Если требуется больше, то алгоритм лучше подкорректировать.

# Остаточные аргументы и оператор расширения

[Смотреть материал на видео](https://www.youtube.com/watch?v=9nO4dpxpb84&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/ostatochnye-argumenty-i-operator-rasshireniya>

Далее, несколько слов об остаточных параметрах функций. В JavaScript можно задавать функции, принимающие неограниченное число аргументов. Например, стандартные функции Math.max и Math.min могут работать с произвольным числом входных параметров. Как объявлять такие функции? Это делается с помощью такого синтаксиса:

**function** sumAll(...args) {

let sum = 0;

**for**(let val of args)

   sum += val;

**return** sum;

}

Здесь мы тремя точками говорим буквально следующее: нужно взять все переданные аргументы и поместить их в массив args. Далее уже идет работа с самим массивом.

console.log( sumAll(1, 2) );

console.log( sumAll(1, 2, 3, 4) );

Кстати, вспоминая работу с массивами, мы можем записать эту функцию гораздо короче:

**function** sumAll(...args) {

**return** args.reduce((prevVal, value) => prevVal += value, 0);

}

Вот она сила и мощь языка JavaScript! Но почему массив этих аргументов называется остаточными параметрами? Дело в том, что мы можем комбинировать обычные аргументы и такой массив, например:

**function** sumAll(arg1, arg2, ...args) {

**return** arg1 + arg2 + args.reduce((prevVal, value) => prevVal += value, 0);

}

и остаточные аргументы всегда должны быть записаны последними – в массив args собирается все, что мы передаем помимо первых двух аргументов. Например, при таком вызове:

console.log( sumAll(1, 2) );

массив args будет пустым. А при таком:

console.log( sumAll(1, 2, 3, 4) );

содержать значения 3 и 4 – остаточные параметры. Вот так это работает.

В противоположность остаточным аргументам, когда множество переданных значений помещаются в массив, в JavaScript существует оператор расширения, который разворачивает массив в набор отдельных значений. Как он записывается и зачем он нужен? Давайте опять обратимся к стандартной функции Math.max. Она находит максимальное значение из переданных значений. Но, что если наши значения хранятся в массиве?

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

При вызове

let max = Math.max(items);

console.log( max );

получим значение NaN. Конечно, мы могли бы передать каждый элемент массива вот так:

let max = Math.max(items[0], items[1], ..., items[4]);

но это крайне неудобно и теряется гибкость кода. Гораздо удобнее воспользоваться оператором расширения, который все это сделает автоматически:

let max = Math.max(...items);

И теперь мы видим верное значение 5. Мы даже можем записать так сразу два массива:

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

let digs = [-1, 0, 6, 10, 101];

let max = Math.max(...items, ...digs);

и найти максимум среди всех этих элементов. Или, комбинировать их с обычными значениями:

let max = Math.max(...items, 1000, ...digs, 0);

Кстати, используя этот механизм, можно делать объединение массивов:

let comp = [...items, ...digs];

console.log( comp );

А также вписывать туда отдельные значения:

let comp = [...items, -1, -2, -3, ...digs];

Как видите – это очень гибкий механизм. И он работает не только с массивами, но с любыми итерируемыми (перебираемыми) данными, например, со строками:

let letters = [..."Привет"];

console.log( letters );

Вам может показаться, что один и тот же оператор … используется как для остаточных параметров, так и для расширения. Но, в действительности, здесь есть четкое разделение:

* если оператор … записан при объявлении функции в списке его аргументов, то это для сбора остаточных аргументов;
* во всех остальных случаях … это оператор расширения итерируемых объектов.

# Замыкания, вложенные функции

[Смотреть материал на видео](https://www.youtube.com/watch?v=srWBcOJxM54&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/zamykaniya-vlozhennye-funkcii>

На этом занятии подробно разберем области видимости данных в JavaScript и такое важное понятие как замыкание.

Для начала представим, что у нас имеется вот такая программа:

let name = "Иван";

getName("Привет!");

**function** getName(say) {

console.log(name + ": " + say);

}

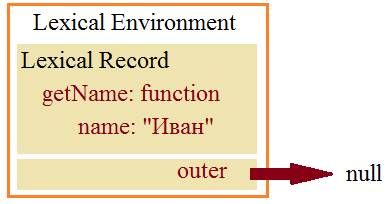
Как это работает в деталях. Как мы уже говорили, все функции, объявленные как Function Declaration, создаются JavaScript-движком в первую очередь. Но что значит создаются? В действительности, при запуске любого скрипта создается специальный объект (на уровне JavaScript-машины и недоступный программисту), называемый глобальным лексическим окружением (Lexical Envirnoment). Он состоит из двух частей:

1. Environment Record – объект, в котором хранятся локальные данные как свойства этого объекта;
2. Ссылка на внешнее лексическое окружение (если его нет, то эта ссылка равна null).

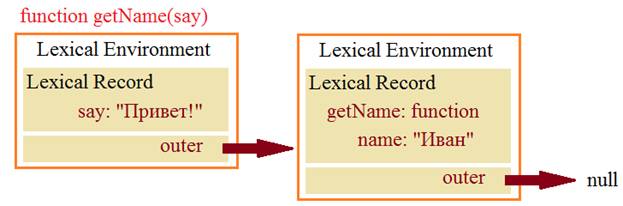
Так вот, объявление функции getName – это не что иное, как создание свойства getName в глобальном лексическом окружении:



Далее, начинает выполняться скрипт. Создается переменная name. Значит, в лексическом окружении появляется еще одно свойство – name:



Затем идет вызов функции getName("Привет!"). В лексическом окружении ищется свойство getName со ссылкой на функцию. Так как оно там уже есть, то функция успешно вызывается. Но, каждый вызов функции создает свое лексическое окружение и картина будет выглядеть так:



Здесь аргументы функции становятся свойствами лексического окружения функции. Далее, выполняется строчка

console.log(name + ": " + say);

Переменная name сначала ищется в лексическом окружении функции. Не находит ее, тогда процесс поиска повторяется на следующем уровне, на который ссылается outer. В глобальном окружении переменная name существует, берется ее значение для вывода в консоль. То же самое происходит и с переменной say. Но она имеется в лексическом окружении самой функции, поэтому поиск здесь и завершается, не переходя к глобальному пространству. В результате мы видим сообщение «Иван: Привет!».

Как только функция завершила свою работу, ее лексическое окружение автоматически уничтожается с помощью встроенного в JavaScript механизма, называемый «**сборщик мусора**».

Из этого примера следуют такие важные моменты:

1. Внешнее лексическое окружение не имеет доступа к данным внутренних окружений.
2. Поиск переменных начинается с текущего окружения и при необходимости последовательно переходит к внешним окружениям. Останавливается, как только переменная найдена.
3. Лексическое окружение автоматически уничтожается, когда в нем более нет необходимости (на него нет внешних ссылок).

Здесь нужно лишь добавить: если переменная не находится, то при включенном режиме "use strict" она принимает значение undefined. Иначе, будет ссылаться на глобальный объект (в браузере – это window).

Исходя из всего сказанного, легко понять, что выведет в консоль следующий скрипт:

let name = "Иван";

getName("Привет!");

**function** getName(say) {

let name = "Федор";

console.log(name + ": " + say);

}

Да, мы видим строчку: «Федор: Привет!». То есть, переменная name была найдена в локальном окружении и взято значение «Федор», а не «Иван».

А вот, если мы выведем что-то в глобальном лексическом окружении:

console.log(name);

console.log(say);

то увидим «Иван», а переменная say не будет найдена. Вот так это работает. Причем, обратите внимание, свои лексические окружения создают не только вызываемые функции, но и условные операторы:

if(...) {

// внутреннее лексическое окружение

}

операторы циклов:

while, for, do while(...) {

// внутреннее лексическое окружение

}

И в современных браузерах даже просто проставленные фигурные скобки:

{

// внутреннее лексическое окружение

}

Однако, старые браузеры блок кода внутри фигурных скобок помещают в текущее лексическое окружение, не создавая нового. Поэтому, в старых программах для изоляции кода от внешнего скрипта, часто применялся такой прием с анонимными функциями:

(**function**() {

*// код внутри изолирован от внешнего скрипта*

let name = "Григорий";

console.log(name);

})();

Причем, он исполнялся в момент, когда интерпретатор JavaScript-машины доходил до этого места программы. Но теперь, в этом приеме нет необходимости и можно просто использовать фигурные скобки.

Еще раз обращу ваше внимание, что внутреннее лексическое окружение создается каждый раз при вызове функции. Если она вызывается несколько раз, то столько же и будет создано лексических окружений. Все лексические окружения автоматически удаляются при отсутствии внешних ссылок на них. Это важный момент!

Мы можем даже создавать функции внутри других функций. Они называются вложенными функциями. В JavaScript – это обычная практика:

**function** getName(say) {

**function** getSay() {

**return** ": " + say;

}

console.log(name + getSay());

}

Здесь функция getSay создана для удобства – вывода сообщения по определенному шаблону. Ее лексическое окружение будет вложено в лексическое окружение функции getName и, конечно же, она будет иметь доступ ко всем внутренним переменным функции getName.

Теперь, когда мы со всем этим разобрались, посмотрим на работу вот такой программы:

**function** createCounter() {

let count = 0;

**return** **function**() {

**return** count++;

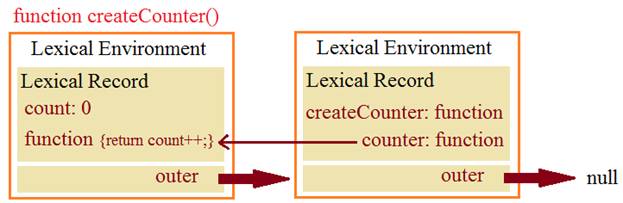
};

}

let counter = createCounter();

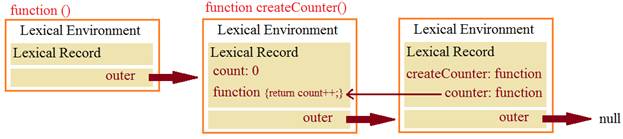
console.log( counter() );

Вначале здесь объявляется функция createCounter, то есть, в глобальном окружении появляется такое свойство. Далее, эта функция вызывается. Создается ее локальное лексическое окружение с переменной count и анонимной функцией:



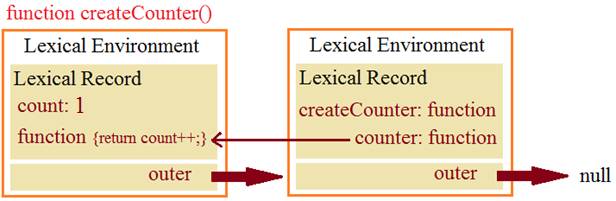
На эту анонимную функцию ссылается переменная counter из глобального окружения. В результате, после завершения выполнения функции createCounter() ее локальное окружение не уничтожается, так как все еще остается ссылка counter.

Далее, в программе идет вызов анонимной функции. Появляется еще одно лексическое пространство этой функции и картина выглядит уже вот так:



Анонимная функция берет переменную count из внешнего лексического пространства, возвращает ее и увеличивает на 1. Вот такое обращение в теле функции к переменной из внешнего лексического окружения в программировании называют **замыканием**.

При завершении, лексическое пространство анонимной функции уничтожается и остается два пространства:



Но значение count теперь равно 1. Если мы вызовем counter() еще раз, то получим значение 1 и переменная count станет равной 2. И так далее. Вот так работают замыкания в JavaScript. Причем, обратите внимание, если мы создадим еще один счетчик:

let counter2 = createCounter();

то будет создано еще одно лексическое пространство и оба счетчика будут работать независимо друг от друга:

console.log( counter2() );

Я постарался предельно просто изложить материал о лексических окружениях и замыканиях, опуская некоторые несущественные на мой взгляд детали. Если вы хотите научиться грамотно писать код на JavaScript, то этот материал следует хорошо знать. Особенно, часто на собеседованиях соискателя любят спрашивать о замыканиях и особенностях работы лексических окружений. И теперь, вы все это знаете.

# Создание функций "на лету"

[Смотреть материал на видео](https://www.youtube.com/watch?v=n4NT7whZZws&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/sozdanie-funkciy-new-function>

В JavaScript существует возможность буквально создавать самим скриптом новые функции. Для этого используется такой синтаксис:

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

Функция создаётся с заданными аргументами arg1...argN и телом functionBody. Например, создадим функцию для вычисления суммы двух аргументов:

let sumTwo = **new** Function('a', 'b', 'return a+b;');

Здесь sumTwo ссылается на созданный объект-функцию, принимающий два аргумента с телом функции «return a+b». Запустим ее и посмотрим как она будет работать:

console.log( sumTwo(1, 2) );

Да, видим, что действительно вычисляется сумма от двух аргументов. Получается, что используя new Function() можно передавать строки с именами аргументов и строку с телом функции, а на выходе получать готовую функцию. То есть, JavaScript может создавать сам себя в процессе выполнения. И это иногда используется на практике.

Если нужно создать функцию без аргументов, то сразу прописывается тело функции:

let hello = **new** Function('console.log("hello");');

hello();

Надо сказать, что такое создание функций в процессе исполнения скрипта используется крайне редко. Один из примеров – прием сложных данных с сервера вместе с функцией, которая их распаковывает.

Также эти функции иначе ведут себя при замыканиях. Создадим внутри функции createMsg функцию с помощью new Function:

**function** createMsg() {

let msg = "Hello";

**return** **new** Function('console.log(msg);');

}

Теперь, при вызове функции createMsg() переменная hello ссылается на новую созданную функцию, которая выводит в консоль переменную msg:

let hello = createMsg();

hello();

Но в момент вызова возникает ошибка, связанная с отсутствие переменной msg. Почему так? Почему замыкание здесь не сработало? Дело в том, что когда функция создаётся с использованием new Function, она всегда ссылается на глобальное внешнее окружение, а не на родительское от createMsg. Поэтому такая функция имеет доступ только к глобальным переменным. И, так как глобальной переменной msg не существует, то и возникает такая ошибка.

Если в нашу программу добавить эту глобальную переменную:

let msg="global hello";

то эта строчка и будет отображена в консоли.

# Функции setTimeout, setInterval и clearInterval

[Смотреть материал на видео](https://www.youtube.com/watch?v=n4NT7whZZws&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/funkcii-settimeout-setinterval-clearinterval>

Рассмотрим две часто используемые в JavaScript функции: setTimeout и setInterval. Первая позволяет выполнять произвольную функцию через определенный период времени и останавливается, а вторая – многократно выполняет функцию через указанный интервал.

Функция setTimeout имеет такой синтаксис:

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

* func|code – ссылка на функцию или строка кода, которая должна выполниться;
* delay – задержка перед вызовом в миллисекундах (1000 мс = 1 сек);
* arg1, arg2, … - возможные аргументы для запускаемой функции (не поддерживаются в старых браузерах IE9-).

Например, запустим функцию

**function** createMsg() {

let msg = "Hello";

console.log(msg);

}

через две секунды:

setTimeout(createMsg, 2000);

Обратите внимание, мы передаем в setTimeout ссылку на функцию, поэтому круглые скобки после ее имени не пишем. Далее указан интервал запуска 2000 мс = 2 сек.

Если функции нужно передать какие-либо аргументы, то это делается так:

**function** createMsg(msg) {

console.log(msg);

}

setTimeout(createMsg, 2000, "timeout hello");

Во всех примерах функция запускалась только один раз спустя 2 сек. Поэтому, setTimeout в таких случаях можно не останавливать – она автоматически завершится после однократного запуска функции. Но, бывают случаи, когда setTimeout нужно завершить до ее запуска. Например, с сервера загружаются сообщения: они могут скачаться быстро, если составляют небольшой размер, но могут скачиваться долго, в случае передачи фотографии или видео. Скрипт на стороне клиента не может заранее знать об объеме принимаемых данных, поэтому для подстраховки запускается отложенный вызов функции, которая на экране отображает анимированное изображение, показывающее процесс загрузки. Почему вызов отложенный? Потому что при быстрой загрузке ничего показывать не нужно, пользователь этого не заметит. И в этом случае нужно прервать функцию setTimeout, чтобы на экране ничего не отображалось. Это может выглядеть так:

**function** downloadMsg() {

let idLoading = setTimeout(**function**() {

   console.log("Идет загрузка данных...");

}, 500);

setTimeout(**function**() {

   clearTimeout(idLoading); *//останавливаем setTimeout после загрузки данных*

   console.log("Данные загружены");

}, 2000);     *//задержка: имитация загрузки с сервера*

}

downloadMsg();

В setTimeout также можно передавать строки с выражением, которое нужно выполнить:

setTimeout("alert('Привет')", 1000);

Но вместо них лучше использовать функции, в том числе и стрелочные:

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

Если в setTimeout не указывается второй аргумент, то задержка принимается равной 0 и функция, переданная в setTimeout запускается настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода. Например:

setTimeout(() => console.log("Мир"));

console.log("Привет");

Увидим сообщения: «Привет» и «Мир».

## Функция setInterval

Вторая функция setInterval имеет такой же синтаксис, что и функция setTimeout, но выполняет переданную ей функцию многократно с заданным интервалом, пока не будет остановлена:

**function** createClock(seconds) {

let sec = seconds;

**return** **function**() {

   sec++;

   console.log("Прошло " + sec + " секунд(а)");

}

}

let clock = createClock(0);

setInterval(clock, 1000);

В этом примере мы, используя замыкание, создали функцию clock и далее, запускаем ее с интервалом в 1 секунду. Она будет выполняться до тех пор, пока мы ее не остановим:

let idClock = setInterval(clock, 1000);

setTimeout(**function**() {

clearInterval(idClock)

}, 5000);

Остановку делаем с помощью отложенного вызова анонимной функции через 5 секунд, в которой вызываем clearInterval с указанием idClock.

Во всех рассмотренных примерах на передаваемые функции в setTimeout или setInterval автоматически создавались ссылки и хранились в планировщике. Это предотвращало удалений функций сборщиком мусора, даже если на них не было явных внешних ссылок в скрипте. Например:

setTimeout(() => console.log("Мир"));

На стрелочную функцию ссылается планировщик до момента ее выполнения. А в

let idInterval = setInterval(**function**() {}, 1000);

анонимная функция остается в памяти до тех пор, пока не будет вызван

clearInterval(idInterval);

Здесь стоит иметь в виду, что передаваемая функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные тоже существуют. Они могут занимать больше памяти, чем сама функция. Поэтому, если регулярный вызов функции больше не нужен, то лучше отменить его.

## Потеря this

Одной из проблем использования setTimeout и setInterval является потеря this при вызове методов объектов. Например:

let car = {

model: "bmw",

showModel() {

   console.log( **this**.model );

}

};

setTimeout(car.showModel, 1000);

В консоле мы увидим undefined. Почему? Дело в том, что здесь теряется контекст при вызове функции. Это эквивалентно вот такому вызову:

let show = car.showModel;

show();

И, так как в JavaScript this вычисляется динамически при каждом вызове функции, то здесь JavaScript-машина просто не может связать функцию show с объектом car.

Исправить ситуацию можно несколькими способами. Первый:

setTimeout(**function**() {car.showModel();}, 1000);

вызвать car.showModel через анонимную функцию-обертку. Здесь мы при вызове явно указываем объект car, поэтому this будет определен корректно. Второй способ – использовать метод bind, о котором мы говорили на предыдущем занятии:

let show = car.showModel.bind(car);

setTimeout(show, 1000);

Этот способ предпочтительнее использовать на практике, так как после вызова bind он не зависит от значения переменной car. Если она будет изменена, то функция show все равно корректно вызовется, а вот в первом случае мы бы получили ошибку. Вот это следует иметь в виду при реализации отложенных вызовов.

И в заключение занятия пару слов об особенностях работы стрелочных функций. Важно знать, что у стрелочных функций нет своего контекста выполнения (лексического окружения), а значит, нет и своего this. В некоторых случаях этот момент имеет ключевое значение, например:

let group = {

title: "ТКбд-11",

students: ["Иванов", "Петров", "Сидоров"],

showList() {

**this**.students.forEach(

    student => console.log(**this**.title + ': ' + student)

   );

}

};

group.showList();

Здесь метод forEach при вызове обычной функции устанавливает контекст this=undefined, но при использовании стрелочной функции контекст неизбежно берется от функции showList и this=group. Например, если заменить стрелочную функцию на обычную, то получим ошибку:

**function**(student) {console.log(**this**.title + ': ' + student);}

с сообщением undefined не имеет свойства title.

# Деструктурирующее присваивание

[Смотреть материал на видео](https://www.youtube.com/watch?v=yk7ZLmempoI&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/destrukturiruyushchee-prisvaivanie>

В JavaScript встроен достаточно гибкий механизм присваивания значений переменным по данным массивов и объектов. Например, у нас есть массив марок машин:

let cars = ["yaguar", "porshe", "mercedes"];

и мы хотим каждый элемент присвоить отдельной переменной. Это можно сделать так:

let [car1, car2, car3] = cars;

Здесь у нас формируются три переменные с соответствующими значениями:

console.log(car1, car2, car3);

Это называется **деструктурирующим присваиванием**. Если, например, нам нужно пропустить второе значение, то это можно реализовать так:

let [car1, , car2] = ["yaguar", "porshe", "mercedes"];

Как видите, механизм очень гибкий. То есть, мы можем элементы массива с легкостью присваивать нужным переменным. Один такой практический пример выглядит следующим образом:

let [firstName, middleName, lastName] = "Иван Иванович Иванов".split(" ");

console.log(firstName, middleName, lastName);

Мы здесь разбиваем строку на составляющие имени и сразу присваиваем их переменным. Быстро, просто и очень удобно.

Если у нас достаточно длинный массив и мы хотим выбрать из него, например, только первые два элемента, а остальные поместить в массив меньшей длины, то это делается так:

let [fr1, fr2, ...last] = ["Груша", "Слива", "Яблоко", "Персик", "Виноград"];

console.log(fr1, fr2, last);

Здесь fr1 = "Груша", fr2 = "Слива", а last – массив из оставшихся элементов. То есть, чтобы нам «собрать» остаточные значения, нужно поставить троеточие и указать имя массива, в который мы их сложим.

Все это будет работать, если число переменных превышает число элементов массива:

let [fr1, fr2, fr3, fr4] = ["Груша", "Слива"];

console.log(fr1, fr2, fr3, fr4);

И, чтобы значения не были undefined, можно прописать значения по умолчанию:

let [fr1, fr2, fr3 = "Яблоко", fr4 = "Персик"] = ["Груша", "Слива"];

Значения по умолчанию могут быть гораздо более сложными выражениями или даже функциями. Они выполняются, только если значения отсутствуют.

let [name = prompt('name?'), lastname = prompt('lastname?')] = ["Иван"];

console.log(name, lastname);

Здесь prompt запустится только для lastname, так как для него будет использовано значение по умолчанию. Первой переменной просто присваивается значение из массива и prompt будет пропущен.

Деструктурирующее присваивание работает с любым перебираемым объектом, например, с обычной строкой:

let [a1, a2, a3, a4] = "Иван";

console.log(a1, a2, a3, a4);

Здесь каждой переменной присваивается отдельный символ строки. Или, с набором:

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

console.log(one, two, three);

Очень удобно использовать деструктурирующее присваивание для перебора свойств объекта:

**for**(let [key, value] of Object.entries(car))

    console.log(`car[${key}] = ${value}`);

Здесь метод entries преобразует объект в двумерный массив по формату «ключ-значение».

Для перебора свойств коллекции Map можно использовать такой же подход:

let car = **new** Map();

car.**set**("model", "toyota").

**set**("color", 0xaf).

**set**("price", 1000);

**for**(let [key, value] of car)

console.log(`car[${key}] = ${value}`);

Деструктурирующее присваивание можно выполнять и с обычными объектами. Например, имеется объект

let args = {

width: 100,

height: 200,

tag: "div",

class: "div-id"

};

И далее, указываем свойства, которые хотим выбрать из этого объекта:

let {width, tag, height} = args;

console.log(width, tag, height);

Причем, эти свойства можно записывать не по порядку, механизм деструктурирующего присваивания отработает корректно. В результате будут созданы отдельные переменные width, tag, height с соответствующими значениями.

Но, что если мы хотим именовать переменные не так как именованы свойства? Для этого используется такой синтаксис:

let {width: w, tag, height: h} = args;

console.log(w, tag, h);

Теперь, вместо переменной width будет создана переменная w и то же самое для переменной height. Переменная tag остается без изменения.

Здесь также как и в случае с массивами можно использовать аргументы по умолчанию:

let {width: w = 0, tag, height: h = 0} = {tag: "span"};

console.log(w, tag, h);

И, если в объекте не будет таких свойств, то переменным будут присвоены эти значения, иначе они были бы равны undefined.

Если мы выбираем только часть свойств из объекта, то оставшиеся можно поместить в объект, используя для его объявления троеточие:

let {width, height, ...last} = args;

console.log(width, height, last);

Во всех приведенных примерах деструктурирующего присваивания мы объявляли переменные в момент присваивания им значений. А можно ли использовать уже существующие переменные? Например, так:

let width, height;

{width, height} = args;

Здесь JavaScript-машина выдаст нам сообщение об ошибке. Дело в том, что вот эти фигурные скобки у нее будут восприниматься не как элемент деструктурирующего присваивания, а как блок кода:

{

width, height

}

 = args;

С этой точки зрения мы действительно имеем какую-то абракадабру, а не текст программы. Чтобы поправить положение, следует заключить всю строчку в круглые скобки:

let width, height;

({width, height} = args);

console.log(width, height);

И тогда все заработает.

Наконец, если объект или массив имеют более сложную структуру, например, такую:

let args = {

tag: "div",

class: "div-class",

size: {width: 100, height: 200}

};

То для деструктурирующего присваивания можно использовать более сложные шаблоны:

let {

   size: {width, height},

   class: cl

} = args;

console.log(width, height, cl);

Мы здесь говорим, что переменные width и height должны браться из вложенного объекта size, а свойство class должно быть скопировано в переменную cl. В итоге, получаем три переменные width, height, cl с соответствующими значениями.

Механизм деструктурирующего присваивания часто используют для передачи параметров аргументам функций. Есть ситуации, когда функция имеет много параметров, большинство из которых не обязательны. Например, при создании пользовательских интерфейсов. Представьте себе функцию, которая создаёт меню. Ей передаются значения ширины, высоты, заголовок, список элементов и так далее:

**function** showMenu(title = "Untitled", width = 200, height = 100, items = []) {

*// ...*

}

Но такой функцией пользоваться неудобно – нужно наизусть помнить порядок аргументов. Это неудобно. Исправить ситуацию можно с помощью деструктурирующего присваивания, определяя аргументы как результат развертывания некоторого объекта:

**function** showMenu({title = "Untitled", width = 200, height = 100, items = []}) {

console.log(title);

console.log(width, height);

console.log(items);

}

А сами аргументы описать с помощью объекта:

let args = {

title: "Заголовок меню",

items: ["Пункт 1", "Пункт 2", "Пункт 3"]

};

Теперь можно вызвать функцию с этими параметрами:

showMenu(args);

Смотрите, как теперь все выглядит изящно и красиво. Если какие-то свойства не указываем, то им присваиваются значения по умолчанию. Например, здесь width = 200, height = 100. А что, если мы не будем передавать функции никаких аргументов и вызовем ее вот так:

showMenu();

У нас возникнет ошибка, так как предполагается деструктирующее присваивание, а никакого объекта (даже пустого) передано не было. Ситуацию можно было бы исправить вот так:

showMenu({});

Но это выглядит некрасиво и непрактично. Лучше обозначить пустой объект как значение по умолчанию в самой функции:

**function** showMenu({title = "Untitled", width = 200, height = 100, items = []} = {}) {

*//...*

}

И теперь ее можно свободно вызывать без аргументов:

showMenu();

Вот что из себя представляет механизм деструктурирующего присваивания и как он может использоваться на практике.

# Свойства name, length и методы call, apply, bind

[Смотреть материал на видео](https://www.youtube.com/watch?v=-T2LHJpV8d4&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/svoystva-name-length-i-metody-call-apply-bind>

Ранее, мы неоднократно говорили, что функция в JavaScript – это объект – особый объект, но, тем не менее, объект. А раз – это объект, то есть ли у него встроенные свойства и можно ли создавать свои? Разберемся с этим.

Первое: у объекта-функции есть, по крайней мере, два встроенных свойства (которые нельзя изменять) – это name и length. Например, у нас есть  какая-то функция:

**function** showMessage(msg) {

console.log(msg);

}

Мы можем обратиться к свойству name и вывести его в консоль:

console.log(showMessage.name);

Обратите внимание, мы здесь не выполняем функцию, а лишь обращаемся к объекту-функции – к его свойству name. Это свойство корректно работает почти во всех случаях. Если, к примеру, взять такое объявление:

let func = **function** (msg) {

console.log(msg);

}

То мы увидим имя «func» - той переменной, что на нее ссылается:

console.log(func.name);

Или, при объявлении методов внутри объекта:

let car = {

go() {},

stop: **function**() {}

};

console.log(car.go.name);

console.log(car.stop.name);

Все работает как и ожидается. И только в редких случаях, как например, здесь:

let ar = [**function**() {}];

console.log(ar[0].name);

это свойство будет содержать пустую строку.

Следующее встроенное свойство «length» содержит количество параметров функции в её объявлении. Например:

**function** func1(a) {}

**function** func2(a, b) {}

**function** other(a, b, ...more) {}

console.log(func1.length); *// 1*

console.log(func2.length); *// 2*

console.log(other.length); *// 2*

Обратите внимание, что в length содержатся явно заданные аргументы, остаточные аргументы, записанные в массив more – игнорируются.

Второй вопрос: можно ли у функций создавать свои свойства? Имеет положительный ответ. Например, объявим функцию:

**function** funcCount() {

console.log("вызов функции: " + ++funcCount.counter);

}

И добавим ей свойство:

funcCount.counter = 0;

Каждый раз при вызове этой функции мы будем увеличивать этот счетчик на единицу:

funcCount();

funcCount();

В результате, мы получили счетчик числа вызовов функции. Другой пример. Возьмем функцию из темы «замыкания»:

**function** createCounter() {

let count = 0;

**return** **function**() {

**return** count++;

};

}

let counter = createCounter();

console.log( counter() );

И перепишем ее вот так:

**function** createCounter() {

**function** counter() {

**return** counter.count++;

}

counter.count = 0;

**return** counter;

}

Здесь мы убрали замыкание и используем добавленное свойство count вложенной функции counter. При создании счетчика это свойство устанавливается в 0 и, далее, при каждом вызове counter() увеличивается на 1.

В чем может быть преимущество такого подхода? Например, теперь мы можем сбрасывать значение счетчика прямо в глобальном окружении:

counter.count = 0;

console.log( counter() );

Иногда это может быть полезным, а иногда – источником дополнительных ошибок. Так что как именно реализовывать функции со свойствами или переменными – это уже решает сам программист, исходя из конкретной задачи.

## Методы call и apply

Наконец, у объекта-функции есть несколько полезных методов. Первые два – это call и apply. Они позволяют вызывать функции с указанием их контекста выполнения в соответствии с синтаксисом:

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

здесь args – это массив или массивоподобный объект. Рассмотрим пример, когда целесообразно применение этих методов. Предположим, имеется  объект:

let car = {

model: "mercedes",

getModel(model) {

**if**(model) console.log( model );

**else** console.log( **this**.model );

}

};

И мы определяем глобальную ссылку на его метод:

let func = car.getModel;

Тогда, при его вызове получим ошибку:

func();

связанную с потерей контекста выполнения этого метода: он начинает выполняться в глобальном контексте, в котором ссылка this = undefined. Но, с помощью встроенного метода call это можно поправить:

func.call(car);

Здесь мы первым аргументом указываем верное значение this=car и функция выполняется как задумано.

Соответственно, можно вызвать метод и с набором аргументов, например:

func.call(car, "opel");

В этом случае будет использован передаваемый аргумент со значением «opel». Число этих аргументов может быть любым.

Аналогично работает и метод apply, только вместо набора аргументов он принимает массив значений, либо массивоподобный объект:

let myMath = {

nameObj: "myMath",

sum(...args) {

**return** **this**.nameObj+": "+args.reduce((val, prevVal) => prevVal += val, 0);

}

};

let sum = myMath.sum;

console.log( sum.apply(myMath, [1, 2, 3, 4]) );

## Метод bind

Вообще, на практике чаще используется метод apply, так как он немного более универсальный, чем call.

Еще один применяемый метод на практике – это bind:

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

он неявно связывает контекст (this=context) до вызова самой функции. Например, для нашего случая с myMath его можно использовать так:

let sum = myMath.sum.bind(myMath);

console.log( sum(1, 2, 3, 4) );

Такой вызов выглядит естественнее и, кроме того, не привязан к переменной myMath, которая может быть изменена.

Интересно, что метод bind также позволяет связывать аргументы с функцией, вот так:

let sum = myMath.sum.bind(myMath, 1, 2);

console.log( sum(3, 4) );

Результат будет тот же – 10, так как первые два аргумента 1 и 2 уже привязаны к функции и их указывать не нужно. Однако, на практике этой возможностью почти не пользуются.

Следующий момент, который мы рассмотрим на этом занятии – это имя для функций, заданных синтаксисом Function Expression. Они еще называются Named Function Expression (NFE):

let getName = **function** Name(name) {

**if**(name) **return** name;

**else** **return** Name("none");

}

Несмотря на то, что у функции как бы два имени, из вне она доступна только по ссылке getName:

console.log( getName("Иван") );

console.log( getName() );

Если мы попытаемся ее вызвать так:

Name();

то получим ошибку, указывающую, что такой переменной не существует. И действительно, Name существует только в теле функции, то есть, внутри лексического окружения этой функции. Но зачем это все нужно? Почему бы нам не делать так (вместо Name прописать getName):

let getName = **function** Name(name) {

**if**(name) **return** name;

**else** **return** getName("none");

}

И все тоже будет работать. Дело в том, что такой код потенциально критичен: если переменная getName будет изменена, например, так:

let getName2 = getName;

getName = **null**;

то, вызывая эту функцию через ссылку getName2, получим ошибку:

console.log( getName2() );

А вот первый вариант записи функции через ее внутреннее имя будет работать в обоих случаях.

Обратите внимание, что такой трюк с «внутренним» именем работает только для Function Expression и не работает для Function Declaration. Так как для Function Declaration синтаксис не предусматривает возможность объявить дополнительное «внутреннее» имя. Если в программе нам нужно надёжное «внутреннее» имя, стоит переписать Function Declaration в стиле, о котором мы только что говорили – Named Function Expression.

# Создание объектов и цикл for in

[Смотреть материал на видео](https://www.youtube.com/watch?v=p7_MMqJ7pHM&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/sozdanie-obektov-cikl-for-in>

Практически все программы на JavaScript используют объекты. Как мы говорили, даже функции являются объектами определенного рода. Поэтому, прежде чем углубляться в изучение этого языка, нам нужно знать основы работы с объектами.

Зачем вообще нужны объекты в JavaScript? Предположим, вы делаете сайт по продаже книг. Будет разумно каждый товар (книгу) представить в виде объекта с набором следующих свойств:

* название;
* автор;
* число страниц;
* цена.

Создадим объект. Для этого используется такой синтаксис:

[переменная =] {

ключ1: значение1,

ключ2: значение2,

…

ключN: значениеN

};

В нашем случае все можно определить так:

let book = {

title: "название",

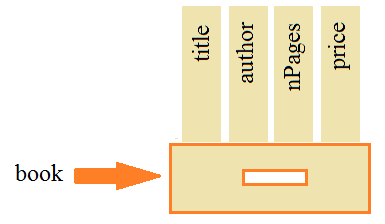
author: "автор",

nPages: 0,

price: 0

};

Этот объект можно воспринимать как некий ящик, в котором 4 секции с именами title, author, nPage, price, которые называются ключами, и соответствующими значениями.



На этот ящик имеется ссылка book, через которую мы к нему обращаемся. Таким образом, данные в объектах сохраняются по принципу

ключ: значение

записанных через запятую. Причем, ключ может быть только строкой, а вот значение – любым типом данных. Пара ключ-значение называется свойством объекта, иногда – полями объекта. К каждому свойству можно обратиться по его ключу. Для этого сначала пишется имя переменной (book), ставится точка и далее следует название ключа:

console.log(book.title);

console.log(book.price);

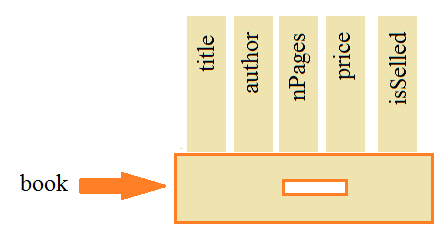
и так далее. Так как значение ключа может быть любым типом данных, добавим к нашему объекту еще одно такое свойство:

book.isSelled = **false**;

Работает эта строчка так: если в объекте нет ключа isSelled, то оно добавляется и ему присваивается значение false. Если же такой ключ уже существует, то его значение переписывается:

book.isSelled = **true**;

Здесь мы изменили значение ранее добавленного ключа с false на true. Теперь наш объект выглядит так.



Для удаления свойств из объекта используется оператор delete, например:

**delete** book.nPages;

удалит свойство nPages из объекта book. В этом можно убедиться. Если мы попытаемся вывести его в консоль

console.log(book.nPages);

то увидим значение undefined, которое, как правило, соответствует несуществующему свойству объекта (если, конечно, значение undefined явно не присвоено какому-либо свойству).

Если мы хотим точно убедиться есть ли какое-либо свойство в объекте, можно использовать оператор in:

console.log("nPages" **in** book); *//вернет false*

console.log("author" **in** book); *//вернет true*

Данный оператор возвратит false, если такого свойства нет и true, если есть. Однако, его очень редко используют на практике, обычно для проверки применяют строгое сравнение со значением udefined:

**if**(book.nPages === **undefined**) console.log("nPages не существует");

Далее, из объекта можно удалить все свойства и получить пустой объект:



Это эквивалентно созданию вот такого объекта:

let book = {};

или же, его можно создать так:

let book = **new** Object();

В обоих случаях результат будет одинаковый. Но, на практике, обычно используют вариант с фигурными скобками {...}. Такое объявление называют *литералом объекта* или *литеральной нотацией*.

Имя свойства объекта может быть какой-нибудь фразой. В этом случае ее записывают явно как строку в кавычках:

let book = {

title: "название",

author: "автор",

nPages: 0,

"size book": {height: 100, width: 20}

};

Здесь значением свойства является объект с двумя полями: height и width. Но, обратиться к такому полю через точку будет уже невозможно:

book.size book = {height: 100, width: 50};

так как в названии ключа имеется пробел и мы получим синтаксическую ошибку. При обращении к свойству через точку требуется, чтобы ключ именовался по тем же правилам, что и имена переменных. Поэтому для доступа к ключу с фразой нужно применять другой синтаксис – через квадратные скобки:

book["size book"] = {height: 100, width: 50};

console.log(book["size book"]);

Тогда все сработает и данное свойство будет изменено на новую строку. Удаление такого свойства работает аналогично:

**delete** book["size book"];

Квадратные скобки также позволяют обратиться к свойству через переменную, например, так:

let keyName = "size book";

console.log(book[keyName]);

Это очень удобный подход, например, если нужно обратиться к произвольному полю объекта, в зависимости от выбора пользователя:

let keyName = prompt("Что вы хотите узнать о книге?", "title");

console.log(book[keyName]);

Запись через точку такого не позволяет. Используя квадратные скобки, можно даже создавать новое свойство через переменную, следующим образом:

let newKey = "color";

let car = {

model: "toyota",

[newKey]: "black",

};

console.log(car[newKey]);

Или, даже так:

let car = {

model: "toyota",

[newKey+"Car"]: "black",

};

console.log(car["colorCar"]);

Здесь мы буквально создаем новое имя ключа непосредственно при объявлении объекта.

Подведём небольшой итог: в большинстве случаев, когда имена свойств известны и просты, используется запись через точку. Если же нам нужно что-то более сложное, то мы используем квадратные скобки.

Далее, давайте посмотрим на такой пример создания объекта внутри функции:

let car = createCar("toyota", "black");

**function** createCar(model, color) {

**return** {

model: model,

color: color

};

}

Ссылка на созданный объект присваивается глобальной переменной car и через нее мы уже можем работать с этим объектом:

console.log(car.model);

Но, кроме того, обратите внимание, что в момент создания название ключа совпадает с названием переменной, которая ему присваивается. Это довольно частая ситуация в JavaScript, поэтому для инициализации таких объектов был придуман упрощенный синтаксис:

**function** createCar(model, color) {

**return** {

model, *//то же самое, что и model:model*

color *//то же самое, что и color:color*

};

}

Эта запись полностью эквивалентна предыдущей. Но использовать ее удобнее.

Рассмотрим теперь такую задачу. Предположим, у нас имеется объект:

let book = {

title: "Муму",

author: "Тургенев",

price: 100,

nPages: 282

};

И мы хотим вывести все его свойства в консоль. Это можно сделать с помощью специального оператора цикла for in, следующим образом:

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

    console.log(key+": "+book[key]);

}

Мы здесь внутри цикла объявили переменную key, которая на каждой итерации цикла будет принимать значение очередного ключа. Далее через оператор in мы указываем объект, из которого будут читаться ключи и затем, в теле цикла выводим сам ключ и через двоеточие – его значение.

Вот так можно перебирать свойства любого объекта в JavaScript.

Используя этот цикл for, мы можем провести небольшое исследование и посмотреть как выстраиваются поля в объектах. Например:

let phoneCodes = {

"7": "Россия",

"49": "Германия",

"41": "Швейцария",

"1": "США"

};

**for** (let code **in** phoneCodes) {

console.log(code);

}

Смотрите, при выводе свойства упорядочились по номерам с 1 до 49. Несколько неожиданно, да? Дело в том, что применительно к объектам существует такое правило: все целочисленные имена ключей упорядочиваются по возрастанию значений. Всех другие ключи сохраняются в порядке их написания. Например, если поставить перед каждой цифрой символ +:

let phoneCodes = {

"+7": "Россия",

"+49": "Германия",

"+41": "Швейцария",

"+1": "США"

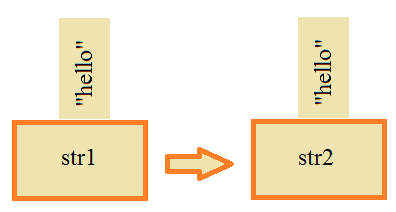
};

Они перестают быть целочисленными и потому остаются на своих местах. Вот это следует иметь в виду.

Далее, рассмотрим особенности копирования объектов. В отличие от примитивных типов данных, которые буквально копируются по значению. Например, здесь мы получим две совершенно независимые строки:

let str1 = "hello";

let str2 = str1;



Объекты копируются по ссылке, то есть, имея ссылку на какой-либо объект, например:

let book = {

title: "название",

author: "автор",

nPages: 0,

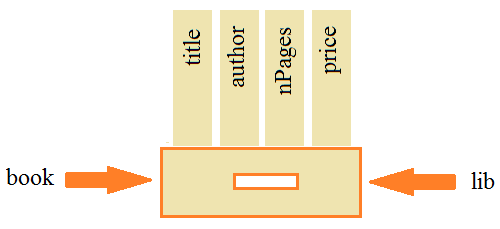
price: 0

};

И выполняя присваивание:

let lib = book;

будет скопирована ссылка на объект, а не сам объект.



То есть, и book и lib будут указывать на один и тот же объект. И мы можем дальше работать с объектом через любую переменную:

lib.title = "Муму";

console.log(book.title);

Здесь у вас может возникнуть вопрос: а как же можно создать клон объекта или делать его структурные копии? Об этом мы подробно поговорим на последующих занятиях, чтобы не перегружать информацией этот урок.

Пока вернемся к ссылкам на объект. Они приводят к некоторым особенностям взаимодействия с объектами. Например, задавая константную переменную:

**const** book = {…};

мы тем не менее можем свободно менять свойства объекта:

book.nPages = 80;

Почему? Дело в том, что мы объявляем константой саму ссылку, а не сам объект, то есть, объект можно свободно менять, на ссылку это никак не влияет, но вот если мы захотим изменить ссылку:

book = **null**;

то получим сообщение об ошибке.

Следующий момент. Сравнивая между собой два объекта:

console.log(book == lib);

console.log(book === lib);

мы фактически сравниваем значения ссылок. И эти равенства будут истинны только в том случае, если указывают на один и тот же объект. А вот в этом случае:

let obj1 = {};

let obj2 = {};

console.log(obj1 == obj2);

console.log(obj1 === obj2);

Получим false, так как значения ссылок не равны между собой – они указывают на разные объекты.

При других сравнениях типа obj1 > obj2 или с примитивом obj == 5 объекты преобразуются к примитивным типам данных, а потом сравниваются: либо как строки, либо как числа, либо как булевы значения и так далее.

# Методы объектов, ключевое слово this

[Смотреть материал на видео](https://www.youtube.com/watch?v=FRH0e2Is9_M&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/metody-obektov-klyuchevoe-slovo-this>

На предыдущем занятии мы с вами познакомились с объектами и их свойствами. Например, объект автомобиль:

let car = {

model: "toyota",

color: "black",

};

Здесь заданы два свойства (model, color) для характеристики модели и цвета машины. То есть, свойства – это данные. Но автомобиль обладает не только какими-то характеристиками (максимальной скоростью, типом трансмиссии, типом кузова и т.п.) он способен к действию! И самое распространенное его действие – ехать. Как описать действие? Помните, когда мы говорили о функциях, то упоминали, что они реализуют определенные действия? Так вот, ровно такие же функции можно помещать в объекты. Зададим свойство go объекта car, которое будет являться ссылкой на функцию:

let car = {

model: "toyota",

color: "black",

go: **function**() {

       console.log("машина едет");

}

};

Такое свойство называется методом объекта. То есть, любое свойство, которое является ссылкой на функцию, называют методом объекта. А название метода должно отвечать на вопрос «что делает метод?» и быть глаголом. В нашем случае как раз так и есть: go – это глагол, означающий идти, ехать и все в таком духе.

Но, чтобы наша виртуальная машина поехала, нужно еще вызвать этот метод. Это делается так:

car.go();

Опять же, вспоминая занятия по функциям, мы говорили, что для вызова функции нужно указать ее имя и круглые скобки. Здесь ровно так все и записано. Отлично, теперь наша машина едет!

Давайте, теперь остановим ее. Добавим в объект car еще один метод stop, который будет это делать. Пропишем его следующим образом:

car.stop = **function** () {

console.log("машина остановлена");

}

Смотрите, мы здесь используем объявление функции по синтаксису Function Experssion. В результате в car добавляется новый метод stop. Вызовем его:

car.stop();

Теперь наш автомобиль умеет еще и останавливаться.

Методы, как и любая функция могут принимать аргументы. Например, добавим аргумент методу go, передавая ему имя водителя:

let car = {

model: "toyota",

color: "black",

go: **function** (driverName) {

       console.log("Водитель " + driverName + ": машина едет");

},

stop: **function** () {

       console.log("машина остановлена");

}

};

И вызовем этот метод так:

car.go("Федор");

Теперь мы знаем не только, что машина едет, но и кто ее ведет. Кстати, новый стандарт ES6 допускает записывать методы в более краткой форме, вот так:

let car = {

model: "toyota",

color: "black",

go(driverName) {

       console.log("Водитель " + driverName + ": машина едет");

},

stop() {

       console.log("машина остановлена");

}

};

Мы здесь убрали двоеточие и слово function, оставили только круглые скобки. Такая запись смотрится естественнее и по смыслу абсолютно такая же, что и в предыдущем случае.

## Ключевое слово this

Предположим, что мы хотим добавить еще один метод, который бы возвращал марку машины. Если записать его вот так:

getModel() {

**return** model;

}

то при вызове этого метода:

console.log( car.getModel() );

возникнет ошибка: переменная model не существует. И действительно, такой переменной нет, есть свойство model. Чтобы обратиться к этому свойству внутри объекта, используется ключевое слово this:

getModel() {

**return** **this**.model;

}

Фактически, this – это ссылка на текущий объект. И через эту ссылку можно обращаться к любому свойству и методу внутри этого объекта. Теперь, если мы запустим скрипт, то убедимся, что все работает.

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. И для примера мы посмотрим на вот такую программу:

let car1 = {model: "toyota"};

let car2 = {model: "opel"};

car1.getModel = getModel;

car2.getModel = getModel;

console.log( car1.getModel() );

console.log( car2.getModel() );

**function** getModel() {

**return** **this**.model;

}

Смотрите, здесь два разных объекта car1 и car2, в которых есть свойство model. В низу прописана функция getModel, которая возвращает это свойство. Обратите внимание, в ее теле присутствует ключевое слово this. Казалось бы, к чему оно должно относиться? Ведь функция не прописана в объекте? Но в JavaScript ссылка this динамически вычисляется в момент выполнения скрипта, в данном случае this будет вычисляться только в момент вызова этой функции. Далее, в объекты car1 и car2 добавляем методы getModel, которые являются ссылками на одну и ту же функцию. Теперь, когда идет вызов функции getModel через ссылку первого объекта car1, this указывает на объект car1. Когда функция вызывается через ссылку из объекта car2, this указывает на этот второй объект. Вот такое нетривиальное поведение ссылки this существует в JavaScript.

Конечно, если мы просто попытаемся вызвать функцию

getModel();

то получим ошибку выполнения, так как this в ней будет иметь значение undefined (при включенном режиме "use strict") и никакого доступа к свойству model мы здесь не получим. Если у вас не включен строгий режим, то в браузерах ссылка this будет указывать на глобальный объект window. Но, обычно, если программист задает такую функцию, то ожидается, что this будет ссылаться на какой-либо объект из контекста вызова этого метода.

Если вы до этого изучали другие языки программирования, то, скорее всего, привыкли к идее "фиксированного this" – когда методы, определённые внутри объекта, всегда сохраняют в качестве значения this ссылку на свой объект (в котором был определён метод).

В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Эта идея 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. Например, посмотрите на такую программу:

let car = {

model: "toyota",

getModel() {

**return** **this**.model;

}

};

let **get** = car.getModel();

console.log( car.getModel() );

console.log( **get**() );

На первый взгляд может показаться, что здесь все в порядке. Но на самом деле, вот этот последний вызов get() завершится ошибкой. Почему? Дело в том, что мы создали ссылку get на функцию getModel объекта car. И когда вызывается get(), то это эквивалентно вызову getModel вне объекта и this становится равным undefined. Из-за этого возникает ошибка. Вот на такие моменты при работе с ключевым словом this следует обращать внимание.

# Клонирование объектов и функции конструкторы

[Смотреть материал на видео](https://www.youtube.com/watch?v=gMXR44HGYyM&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/klonirovanie-obektov-funkcii-konstruktory>

Итак, пришло время разобраться: как создать клон объекта – точную его копию в памяти компьютера? Предположим, у нас имеется некий объект book:

let book = {

title: "название",

author: "автор",

nPages: 0,

price: 0,

};

Для его клонирования, создадим пустой объект и скопируем в него все примитивные свойства другого объекта:

let lib = {};

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

lib[key] = book[key];

console.log(key + ": " + lib[key]);

}

Но здесь есть одна проблема: так можно скопировать только примитивные данные. А что если внутри объекта book будет еще один объект? Например, так:

let book = {

title: "название",

author: "автор",

nPages: 0,

price: 0,

size: {height: 100, width: 50}

};

тут свойство size указывает на объект. Тогда при его копировании мы скопируем ссылку на него, а не сам объект. Чтобы исправить это, мы должны в цикле клонирования делать проверку: является ли значение book[key] объектом, и если это так – копируем и его структуру тоже. Это называется «глубокое клонирование». Я приведу здесь лишь пример реализации глубокого копирования. На практике его использовать нельзя, так как он не учитывает всех разновидностей полей объекта и в ряде случаев будет давать неверные результаты:

**function** cloneDeepObj(dest, obj) {

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

**if**((**typeof** obj[key]) == "object") {

  dest[key] = cloneDeepObj({}, obj[key]);

}

**else** {

  dest[key] = obj[key];

*//console.log(key + ": " + dest[key]);*

}

}

**return** dest;

}

let lib = cloneDeepObj({}, book);

book.size.height = 0;

console.log(lib);

Я реализовал все через функцию, которая вызывает саму себя, если встречается свойство, содержащее объект. Здесь это ключевой момент. Когда встречается объект, то запускается такой же алгоритм для создания его клона, затем, результат возвращается и присваивается свойству dest[key]. Так реализуется копирование вложенных объектов. Причем глубина вложенности может быть любой (но, конечно, в разумных пределах).

Далее, мы вызываем эту функцию. В качестве первого аргумента передаем пустой объект, а вторым – клонируемый объект. Результат клонирования сохраняется по ссылке в переменной lib. Затем, для проверки, мы меняем значение вложенного объекта book, выводим в консоль объект lib и видим, что изменения в book его не коснулись, то есть, все было скопировано корректно.

Существует стандартный алгоритм глубокого клонирования, Structured cloning algorithm (<http://w3c.github.io/html/infrastructure.html#safe-passing-of-structured-data>). Он решает описанную выше задачу, а также более сложные задачи. Чтобы не изобретать велосипед, мы можем использовать реализацию этого алгоритма из JavaScript-библиотеки lodash ([https://lodash.com](https://lodash.com/)), в частности, метод \_.cloneDeep(obj).

## Функция-конструктор для объектов

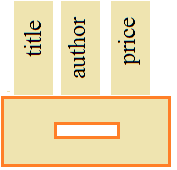
Вообще, на практике редко встречается необходимость клонировать объекты и если вы с этим столкнулись, то проанализируйте свой алгоритм, скорее всего вы делаете что-то не так. И только при жесткой необходимости прибегайте к подобным мерам. Обычно, на практике создание множества однотипных объектов выполняется с помощью функции-конструктора совместно с оператором new. Рассмотрим это подробнее.

Что такое функция-конструктор? Это обычная функция, но ее имя должно начинаться с заглавной буквы. Например:

**function** Book() {}

**function** Car() {}

и так далее. Предположим, что мы хотим создавать объекты для книг с такой структурой:



Запишем функцию-конструктор Book в виде:

**function** Book(title, author) {

**this**.title = title;

**this**.author = author;

**this**.price = 10;

}

Теперь можем создать объект с помощью оператора new:

let book = **new** Book("Муму", "Тургенев");

и вывести объект в консоль:

console.log( book );

Когда функция вызывается как new Book(…), происходит буквально следующее:

1. Создаётся новый пустой объект, на который указывает this.
2. Выполняется тело функции, добавляя в объект новые свойства.
3. Возвращается значение this.

Обратите внимание, что пункт 1 и 3 мы нигде явно не прописывали, движок JavaScript добавляет эти конструкции автоматически при вызове функции через new. В результате мы получаем такое выполнение:

**function** Book(title, author) {

*// this = {};  (неявно)*

**this**.title = title;

**this**.author = author;

**this**.price = 10;

*// return this;  (неявно)*

}

Преимущество такого подхода перед обычным созданием объекта в том, что мы можем создать еще одну книгу (еще один объект), используя эту же функцию-конструктор:

let book2 = **new** Book("Онегин", "Пушкин");

и теперь мы имеем два разных объекта:

console.log( book );

console.log( book2 );

Как видите, так создавать новые объекты бывает гораздо удобнее, чем каждый раз прописывать их через литерал {}. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

В JavaScript допускается использовать анонимные функции в качестве конструкторов, например, так:

let car = **new** **function**() {

**this**.model = "reno";

**this**.go = **function**() {

console.log("машина едет");

}

}

И далее, уже работать с объектом:

car.go();

console.log( car );

Конечно такой конструктор вызывается единожды в том месте, где прописан, так как анонимная функция не может быть вызвана где-либо еще. Тогда зачем это нужно? Не проще ли в этом случае создать объект через литерал {}? Иногда нет. Например, создавая сложный объект, мы можем формировать его структуру по некому алгоритму, на основе локальных или даже глобальных переменных. В этом случае прописать логику внутри функции оказывается гораздо удобнее.

Если последний момент вас ставит в тупик, просто пропустите его. Когда придет время использовать нечто подобное, вы об этом вспомните.

Давайте вернемся к функции Book и заметим такую особенность: она может быть вызвана и без оператора new, вот так:

let book = Book();

и тогда мы получим ошибку выполнения, так как ссылка this становится равной undefined. Поправим код внутри функции так, чтобы он корректно работал в обоих случаях. Для этого мы будем использовать свойство target оператора new, который равен undefined, если функция вызвана без new. Получим:

**function** Book(title, author) {

**if**(**new**.target == **undefined**) *//если вызвали без new*

**return** **new** Book(title, author); *//добавим new автоматически*

**this**.title = title;

**this**.author = author;

**this**.price = 10;

}

Теперь можно вызывать функцию как с new, так и без new – результат будет тот же. Часто такую подстановку делают в библиотечных конструкторах, чтобы облегчить использование таких функций. Но делать это в своем собственном скрипте не рекомендуется, так как отсутствие new может ввести разработчика в заблуждение. С оператором new мы точно знаем, что в итоге будет создан новый объект.

Наконец, в конструкторе мы можем явно прописать оператор return, к ошибке это не приведет. Но есть такие особенности:

* При вызове return с объектом, будет возвращён объект, а не this.
* При вызове return с примитивным значением, примитивное значение будет отброшено.

Например, в этом конструкторе будет создан и возвращен объект, указанный в операторе return:

**function** User() {

**this**.name = "user1";

**return** {name: "user2"};

}

console.log( **new** User().name );

А если записать так:

**function** User() {

**this**.name = "user1";

**return**;

}

То будет создан объект с именем «user1». Но, обычно, return все-таки опускают и не записывают.

# Введение в массивы, цикл for of

[Смотреть материал на видео](https://www.youtube.com/watch?v=-M_10IJV1wk&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/vvedenie-v-massivy>

В жизни мы часто сталкиваемся со списками: список студентов, марок машин, значений функции, элементов HTML-документа и т.п. Так вот, для хранения упорядоченных коллекций существует особая структура данных, которая называется массивом (по англ. array).

Обычно массивы в JavaScript задаются с помощью литерала – квадратных скобок:

let ar = [];  *//пустой массив*

let ar = [1, 30, 0, -2]; *//одномерный массив чисел*

Каждый элемент разделяется запятой. В результате мы здесь объявили массив из 4-х чисел. Или, так:

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

Здесь имеем массив из трех строковых элементов. Для обращения к отдельному элементу массива используется такой синтаксис:

<имя массива>[индекс элемента]

Например, выведем значение первого элемента массива ar в консоль:

console.log( ar[0] );

Обратите внимание, индекс первого элемента массива равен 0, не 1, как мы привыкли в повседневной жизни, а 0! Запомним это. Соответственно, второй элемент имеет индекс 1 и так далее:

console.log( ar[1] );

console.log( ar[2] );

Любой элемент массива можно изменить, используя тот же синтаксис:

ar[1] = 'Груша';

Теперь наш массив содержит элементы: "Яблоко", "Груша", "Слива". Вот так можно обращаться к элементам массива для считывания или изменения их значений.

В любой массив можно добавлять новые элементы, например, так:

ar[3] = 'Лимон';

Здесь мы указываем индекс элемента, в который хотим записать значение «Лимон» и, так как такого элемента нет, то он добавляется. В результате получаем массив из четырех строк:

"Яблоко", "Груша", "Слива", "Лимон"

У массива есть свойство length, хранящее значение равное максимальному индексу плюс один:

Array.length = maxIndex+1

Используя это свойство, можно через цикл вывести значения всех элементов массива:

**for**(let i=0;i < ar.length;++i)

console.log( ar[i] );

Но можно это сделать и так:

console.log( ar );

Обратите внимание, length – это не число элементов массива в привычном нам виде. Например, если в наш массив добавить еще один элемент с большим индексом:

ar[999] = 'Виноград';

то length = 1000, но в массиве будет всего пять элементов:

"Яблоко", "Груша", "Слива", "Лимон" и "Виноград" (с индексом 999)

Далее, при переборе такого массива через цикл, несуществующие элементы будут принимать значения undefined и ошибки никакой не возникнет.

В JavaScript свойство length можно задавать вручную. Представим, имеется вот такой массив из чисел:

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

далее, мы уменьшаем его длину:

dig.length = 3;

теперь наш массив содержит только первые три элемента. Если попытаться восстановить длину снова до 5:

dig.length = 5;

то можем убедиться, что последние два элемента уже не существуют:

console.log( dig );

Вот так через свойство length можно обрезать длину массивов.

Язык JavaScript организован таким образом, что в массиве могут храниться элементы любого типа. Например:

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

И выведем в консоль свойство name второго элемента:

console.log( ar[1].name );

Причем, массив не обязательно записывать на одной строке, можно и так:

let ar = [ 'Яблоко', { name: 'Джон' },

**true**,

**function**() { alert('привет'); } ];

Массивы в JavaScript – это особого вида объекты, которые предназначены для работы с упорядоченными коллекциями данных и оптимизированы для работы с ними. Здесь ключевое слово – упорядоченные. Дело в том, что движок JavaScript старается хранить элементы массива в непрерывной области памяти, один за другим и это один из способов оптимизации. Но мы можем все это легко нарушить, если не будем рассматривать массивы именно как массивы. Например, можно создать пустой массив:

let digits = [];

а, затем, в очень дальний индекс запишем какое-либо значение:

digits[99999] = 1;

Технически так все будет работать, но оптимизации уже не будет. То есть, работать будет медленнее. Или, можно сделать даже так:

digits.zero = **null**;

Так как массив – это объект, то мы просто создаем свойство zero со значением null. Оптимизации здесь тоже не будет. Наконец, заполняя массив с конца:

digits[100] = 100;

digits[99] = 99;

…

мы тоже теряем оптимизацию. Во всех этих случаях движок JavaScript будет полагать, что мы работаем с массивом как с обычным объектом и оптимизация будет отключена. Поэтому, если уж вы решили использовать массив, то используйте его по назначению, заполняйте однотипными значениями по порядку и используйте встроенные методы для обработки его элементов.

## Перебор элементов массива

Один из распространенных способов перебора элементов массива мы только что видели – это цикл for:

Например, для массива:

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

перебрать его элементы можно так:

**for**(let i=0;i < fruits.length; ++i)

console.log( fruits[i] );

мы увидим все значения элементов. Но есть и второй новый способ перебора с помощью цикла for of:

**for**(let value of fruits)

console.log( value );

Эта запись короче, но имеет свои особенности: значения массива копируются в переменную value, то есть, внутри цикла мы имеем дело не с самими значениями массива fruits, а с их копиями. Например, из-за этого мы не можем менять значения элементов массива внутри такого цикла:

**for**(let value of fruits) {

    value = "none";

}

console.log(fruits);

В консоле мы увидим прежний массив. А вот через первый цикл так делать можно:

**for**(let i=0;i < fruits.length; ++i)

fruits[i] = "none";

Преимуществом цикла for of является его оптимизация под массивы. В частности, он работает в 10-100 раз быстрее цикла for in, который мы рассматривали ранее, для перебора свойств объекта. Формально, мы могли бы использовать и его:

**for**(let key **in** fruits)

console.log( fruits[key] );

Но это будет медленнее и к тому же там мы будем перебирать все публичные ключи массива, а не только целочисленные, которые являются индексами элементов массива. В общем, вывод такой. Нужно перебрать массив, используйте или обычный цикл for или цикл for of.

# Методы push/pop, shift/unshift и многомерные массивы

[Смотреть материал на видео](https://www.youtube.com/watch?v=-M_10IJV1wk&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/metody-push-pop-shift-unshift-mnogomernye-massivy>

Массивы имеют встроенные методы для обработки элементов. Первые два метода, которые мы рассмотрим – это push и pop. Они позволяют добавлять и удалять последний элемент массива. Например, имеется массив:

let ar = [ "Яблоко", "Слива" ];

и мы хотим добавить следующий элемент. Конечно, это можно сделать уже знакомым нам способом:

ar[2] = "Груша";

но здесь приходится высчитывать нужный индекс следующего элемента. Хорошо, можно тогда сделать так:

ar[ar.length] = "Груша";

но такая запись не очень хорошо воспринимается. Часто для добавления последнего элемента используют метод push:

ar.push("Груша");

Смотрите, это выглядит кратко и понятно. Для удаления последнего элемента из массива применяется метод pop:

ar.pop();

причем, этот метод еще и возвращает удаленный элемент, то есть, этот метод можно записать и так:

let delElem = ar.pop();

console.log( delElem );

В консоль будет выведена строка «Груша».

## Методы shift и unshift

Аналогичные методы имеются и для добавления и удаления первых элементов массива – это shift и unshift. Метод shift удаляет первый элемент массива:

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

let delElem = fruits.shift();

console.log( fruits );

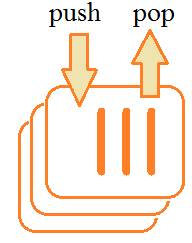
Он также как и метод pop возвращает значение удаляемого элемента. Следующий метод unshift добавляет элемент в начало массива:

fruits.unshift('Персик');

С помощью этих четырех рассмотренных методов на базе массивов в JavaScript часто реализуют очереди двух типов:

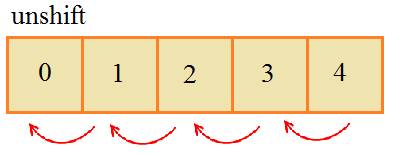


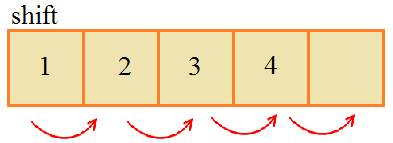
Первым вошел, первым вышел (по англ. FIFO), например, данные поступающие во входной буфер, или события, приходящие в приложение.



Вторая очередь – это своего рода стек, организованная по принципу первым вошел, последним вышел. Здесь добавление и удаление элемента из очереди всегда выполняется с хвоста массива. Например, так организован стек вызова функций.

Причем, вот эта вторая очередь будет работать быстрее, чем первая. Почему? Дело в том, что методы push и pop, работающие с последними элементами, выполняются весьма эффективно. Все что им нужно сделать – это либо добавить, либо удалить последний элемент. А вот функции shift и unshift работают, как правило, заметно дольше, так как при удалении первого элемента, необходимо сдвигать все оставшиеся элементы вправо на место удаляемого, а при добавлении первого элемента – сдвигать элементы массива влево.





Таким образом, чем длиннее массив, тем дольше работают эти два метода.

## Многомерные массивы

Так как массивы могут содержать самые разные типы данных, то этим можно воспользоваться для организации двумерных массивов. В двумерных массивах каждый элемент является одномерным массивом, например:

let matrix = [

  [1, 2, 3],

  [4, 5, 6],

  [7, 8, 9]

];

console.log( matrix );

Имеем матрицу 3х3 и как видим в консоле каждый элемент является одномерным массивом, длиной 3 элемента. Для доступа к отдельному элементу такого массива, используется запись:

matrix[1][1] = 0;

В результате центральный элемент изменит свое значение с 5 на 0. Выведем этот массив в консоль с помощью вложенных циклов:

**for**(let row of matrix) {

let cols = "";

**for**(let val of row)

cols += val + " ";

console.log(cols);

}

Или же это можно сделать так:

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

let cols = "";

**for**(let j = 0; j < matrix[i].length;++j)

cols += matrix[i][j] + " ";

console.log(cols);

}

По аналогии можно создавать многомерные массивы самой разной конфигурации.

# Методы массивов

[Смотреть материал на видео](https://www.youtube.com/watch?v=--fQdPVwyLQ&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/metody-massivov>

На этом занятии рассмотрим основные методы массивов. Некоторые из них мы уже рассмотрели на предыдущем занятии – это push/pop и shift/unshift. Здесь мы продолжим эту тему и чтобы было проще воспринимать материал, все методы разбиты на группы и начнем с группы добавления/удаления элементов массива. В нее входят рассмотренные нами методы push/pop и shift/unshift и следующий метод – это splice.

## splice

Метод splice() – это универсальный «швейцарский нож» для работы с массивами. Умеет всё: добавлять, удалять и заменять элементы. Его синтаксис такой:

Array.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. Этот метод проще всего понять, рассмотрев примеры. Начнем с удаления. Предположим, имеется массив:

let ar = ["Я", "смотрю", "этот", "обучающий", "урок"];

Удалим 3-й и 4-й элементы «этот» и «обучающий»:

ar.splice(2, 2);

мы здесь указали индекс элемента, с которого происходит удаление и число удаляемых элементов. Выведем результат в консоль:

console.log(ar);

Видим, в массиве остались строки «я», «смотрю», «урок».

В следующем примере мы удалим первые три элемента и добавим два других:

let delElem = ar.splice(0, 3, "Это", "классный");

получаем массив ar:

«Это», «классный», «обучающий», «урок»

и массив delElem, состоящий из удаленных элементов. Этот пример также показывает, что метод splice возвращает массив из удаленных величин.

С помощью метода splice можно вставлять элементы, не удаляя существующие. Для этого аргумент deleteCount устанавливается в 0:

ar.splice(3, 0, "интересный");

Получим массив:

"Я", "смотрю", "этот", "интересный", "обучающий", "урок"

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

ar.splice(-3, 3, "это", "обучающее", "видео");

Здесь удаляются последние 3 элемента и вместо них вставляются новые строчки.

## slice

Метод slice имеет синтаксис:

Array.slice([start], [end])

возвращает массив, в который копирует элементы, начиная с индекса start и заканчивая индексом end-1. Например:

let ar = ["Я", "смотрю", "этот", "обучающий", "урок"];

let res1 = ar.slice(2, 4);   *//этот, обучающий*

let res2 = ar.slice(3);      *//обучающий, урок*

let res3 = ar.slice(-3);     *//этот, обучающий, урок*

В res1 копируются элементы с индексами 2 и 3, в res2 с индекса 3 и до конца массива, а в res3 от 3-го индекса с конца и до конца массива. Если вызвать просто

let copyArr = ar.slice();

console.log(copyArr);

то будет скопирован массив целиком. Этим часто пользуются, если нужно создать копию массива.

## concat

Метод concat возвращает новый массив, состоящий из элементов текущего массива, плюс элементы, указанные в качестве аргументов:

Array.concat(arg1, arg2...)

Здесь arg1, arg2 могут быть как примитивными данными (строки, числа), так и массивами. Например:

let ar = [1, 2];

let res1 = ar.concat([3, 4]); *// 1,2,3,4*

console.log( res1 );

let res2 = ar.concat([3, 4], [5, 6]); *// 1,2,3,4,5,6*

console.log( res2 );

console.log( ar.concat([3, 4], 5, 6) ); *// 1,2,3,4,5,6*

По идее, этот метод работает и с объектами, вот так:

let obj = {name: "guest"};

let ar = [1, 2];

let res = ar.concat(obj);

console.log( res );

но здесь копируется лишь ссылка на объект, а не он сам.

## forEach

Данный метод перебирает элементы массива и при этом позволяет с ними выполнить какие-либо действия. Имеет следующий синтаксис:

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

  // ... делать что-то с item

});

Например, здесь выводятся элементы массива в консоль:

let ar = ["Я", "смотрю", "этот", "обучающий", "урок"];

ar.forEach(**function**(item) {

         console.log(item);

});

Обратите внимание, что нам нет необходимости указывать все аргументы функции, достаточно лишь те, что необходимы. В реализацию метода forEach очень хорошо вписываются стрелочные функции, о которых мы говорили ранее, например так:

let dig = [1, 2, 3, 4, 5, 6, 7];

dig.forEach( (item) => console.log(item) );

или так для вывода только четных значений:

dig.forEach( (item, index) => {

**if**(item % 2 == 0) console.log(`${item} с индексом ${index}`);

});

А вот так все нечетные элементы можно заменить на 1:

dig.forEach( (item, index, array) => {

**if**(item % 2 != 0) array[index] = 1;

});

Далее мы рассмотрим группу методов для поиска элементов в массиве.

## indexOf, lastIndexOf и includes

Данные методы имеют одинаковый синтаксис и делают по сути одно и то же с небольшими отличиями. Их синтаксис следующий:

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

Например:

let ar = ["Я", "смотрю", "этот", "обучающий", "урок", 0, **false**, **null**];

let res1 = ar.indexOf("смотрю", 0);           *// 1*

let res2 = ar.lastIndexOf(**null**, 0);             *// -1*

let res3 = ar.includes(0, 3);                *// true*

Получим индекс, равный 1 для элемента «смотрю», -1 для null и true для нуля. Почему метод lastIndexOf вернул значение -1 (то есть, элемент null не найден), хотя он присутствует в массиве? Дело в том, что аргумент from – это индекс с которого мы просматриваем массив в обратном направлении. Так как здесь from=0, то мы идет назад с нулевого элемента, т.е. мы просматриваем только 1-й элемент. Чтобы просмотреть весь массив, этот индекс можно либо не указывать, либо указать индекс последнего элемента:

let res2 = ar.lastIndexOf(**null**);          *// 7*

Теперь значение null находится как надо. Также обратите внимание, что эти методы используют строгое сравнение ===. То есть, если мы ищем false, они находят именно false, а не ноль (числовой эквивалент false).

## find и findIndex

Метод find позволяет найти элемент массива по какому-либо критерию (условию). Она имеет следующий синтаксис:

let result = ar.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.

Предположим, у нас имеется массив объектов:

let cars = [

{model: "toyota", price: 1000},

{model: "opel", price: 800},

{model: "reno", price: 1200}

];

Найдем в нем первую машину со стоимостью меньше 1000 единиц:

let res = cars.find(item => item.price < 1000);

console.log(res);

Результатом будет элемент opel. Так как на практике в JavaScript часто применяются массивы из объектов, то метод find бывает весьма полезным для поиска нужного элемента. Также обратите внимание, что в примере мы в find передаем стрелочной функции только один аргумент – item. Так тоже можно делать и это довольно типичная ситуация.

Метод ar.findIndex – по сути, то же самое, что и find, но возвращает индекс, по которому элемент был найден, и -1 в противном случае:

let res = cars.findIndex(item => item.price < 1000);

## filter

Рассмотренные методы find и findIndex ищут первый подходящий элемент. Если же нужно найти все элементы по заданному критерию (условию), то следует использовать метод filter:

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

  // если true – элемент добавляется к результату, и перебор продолжается

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

});

Перепишем наш пример:

let res = cars.filter(item => item.price <= 1000);

Получаем массив из двух найденных элементов.

## Методы преобразования массива: map

Метод map довольно часто используется на практике и позволяет получить результаты обработки элементов массива. Его синтаксис похож на предыдущие функции:

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

  // возвращается новое значение вместо элемента

});

И может быть использован так:

let cars = ["toyota", "opel", "reno"];

let res = cars.map(**function** (item) {

**return** item.length;

});

Мы здесь получаем массив длин строк массива cars.

## sort

Данный метод сортирует массив по тому критерию, который указывается в ее необязательной callback-функции:

ar.sort(function(a, b) {

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

if (a == b) return 0; // если равны

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

})

Сортировка выполняется непосредственно внутри массива ar, но функция также и возвращает отсортированный массив, правда это возвращаемое значение, обычно игнорируется. Например:

let dig = [4, 25, 2];

dig.sort();

console.log( dig );

И получим неожиданный результат: 2, 25, 4. Дело в том, что по умолчанию метод sort рассматривает значения элементов массива как строки и сортирует их в лексикографическом порядке. В результате, строка «2» < «4» и «25» < «4», отсюда и результат. Для указания другого критерия сортировки, мы должны записать свою callback-функцию:

dig.sort(**function**(a, b) {

**if**(a > b) **return** 1;

**else** **if**(a < b) **return** -1;

**else** **return** 0;

});

Теперь сортировка с числами проходит так как нам нужно. Кстати, чтобы изменить направление сортировки (с возрастания на убывание), достаточно поменять знаки больше и меньше на противоположные. И второй момент: callback-функция не обязательно должна возвращать именно 1 и -1, можно вернуть любое положительное, если a>b и отрицательное при a<b. В частности, это позволяет переписать приведенный выше пример вот в такой краткой форме:

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

По аналогии можно формировать и более сложные алгоритмы сортировки самых разных типов данных: строк, чисел, объектов, булевых переменных и так далее.

## reverse

Данный метод прост, он меняет порядок следования элементов на обратный. Например:

let dig = [4, 25, 2, 1];

dig.reverse();

Получим этот же массив со значениями: 1, 2, 25, 4.

## split, join

Метод split применяется к строкам и позволяет разбить строку по указанному разделителю на массив строк. Например, мы в строке указываем email адреса, на которые хотим отправить письма:

let emailsTo = "alex12@m.ru; m2@m.com; pp@g.com; upr@g.ru";

Но в программе нам было бы удобнее оперировать массивами из этих адресов. Чтобы разбить строку по email мы вызовем для строки метод split и укажем разделитель «; »:

let arEmails = emailsTo.split("; ");

**for**( let email of arEmails)

console.log( email );

Все, теперь можно использовать массив arEmails вместо строки с email адресами.

У данного метода есть второй необязательный аргумент, указывающий максимальное число элементов в выходном массиве. Обычно, он опускается, но если, например, указать вот так:

let arEmails = emailsTo.split("; ", 2);

то в выходном массиве будет только два первых элемента.

Метод join работает в точности наоборот: он из массива строк формирует единую строку, например:

let strEmails = arEmails.join(", ");

console.log( strEmails );

Получаем строку из email-адресов через запятую.

## reduce и reduceRight

Если нам нужно перебрать массив – мы можем использовать forEach, for или for..of. Если нужно перебрать массив и вернуть данные для каждого элемента – мы используем map.

Методы reduce и reduceRight похожи на методы выше, но они немного сложнее и, как правило, используются для вычисления какого-нибудь единого значения на основе всего массива.

Синтаксис:

let value = ar.reduce(function(previousValue, item, index, array) {  
  // ...  
}, [initial]);

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

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

Например, требуется вычислить сумму значений элементов массива. Это очень легко реализовать этим методом, например, так:

let digs = [1, -2, 100, 3, 9, 54];

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

console.log(sum);

Здесь значение sum при первом вызове будет равно 0, так как мы вторым аргументом метода reduce указали 0 – это начальное значение previousValue (то есть sum). Затем, на каждой итерации мы будем иметь ранее вычисленное значение sum, к которому прибавляем значение текущего элемента – current. Так и подсчитывается сумма.

А вот примеры вычисления произведения элементов массива:

let pr = digs.reduce((pr, current) => pr\*current, 1);

console.log(pr);

Здесь мы уже указываем начальное значение 1, иначе бы все произведение было бы равно нулю.

Если начальное значение не указано, то в качестве previousValue берется первый элемент массива и функция стартует сразу со второго элемента. Поэтому во всех наших примерах второй аргумент можно было бы и не указывать. Но такое использование требует крайней осторожности. Если массив пуст, то вызов reduce без начального значения выдаст ошибку:

let digs = [];

let pr = digs.reduce((pr, current) => pr\*current);

Поэтому, лучше использовать начальное значение.

Метод reduceRight работает аналогично, но проходит по массиву справа налево.

## Array.isArray

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

console.log(**typeof** {}); *// object*

console.log (**typeof** []); *// тоже object*

Но массивы используются настолько часто, что для этого придумали специальный метод: Array.isArray(value). Он возвращает true, если value массив, и false, если нет.

console.log(Array.isArray({})); *// false*

console.log(Array.isArray([])); *// true*

Подведем итоги по рассмотренным методам массивов. У нас получился следующий список:

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

# Методы числовых объектов

[Смотреть материал на видео](https://www.youtube.com/watch?v=8_1c2DAkNII&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/metody-chislovyh-obektov>

Теперь, когда мы с вами подробно познакомились с функциями, объектами и методами вернемся к примитивным типам данных и посмотрим, какими полезными методами они обладают. Да, все верно, JavaScript так устроен, что примитивные типы тоже имеют свои методы подобно объектам. И начнем рассмотрение с числового типа Number.

## Форматы записи чисел

Ранее мы уже говорили, что в JavaScript числа можно задавать следующим образом:

let a1 = 1e9;   *// 1 000 000 000*

let a2 = 0.5e3; *// 0.5\*1000 = 500*

let a3 = 1e-6;  *// 0.000001*

let a4 = 0xff;  *// шестнадцатиричная запись (ff = 255)*

let a5 = 0b11111111; *// двоичная запись десятичного числа 255*

let a6 = 0o377; *// восьмеричная форма записи числа 255*

Можно использовать и другие системы счисления, но они почти не встречаются на практике.

## Особенности числовой арифметики

В JavaScript число представлено в виде 64-битного формата IEEE-754. Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 из них для хранения положения десятичной точки (если число целое, то хранится 0), и один бит отведён на хранение знака. Если число слишком большое, оно переполнит 64-битное хранилище, JavaScript вернёт бесконечность:

console.log( 1e500 ); *// Infinity*

Наиболее часто встречающаяся ошибка при работе с числами в JavaScript – это потеря точности. Например, сумма двух таких вещественных чисел:

let res = 0.1+0.2;

console.log( res );

не дает точное значение 0,3. Потому, например, при таком сравнении:

**if**(res == 0.3) console.log( **true** );

мы получим значение false и оператор после if выполнен не будет. Почему так происходит? Не вдаваясь в подробности, можно сказать так: вещественные числа 0.1 и 0.3 в двоичной системе счисления (именно так они представляются в памяти ЭВМ) являются бесконечной дробью. Отсюда и потеря точности.

Как обойти эту проблему и корректно работать с такими числами? Если это имеет значение, то в общем случае нужно придумывать свои подходы в каждой конкретной ситуации. Например, если разрабатывается интернет-магазин, то все вычисления можно делать целыми числами, выраженные в копейках и только на экране браузера результат отображать в рублях и копейках. И так далее. Здесь каждый сам решает как ему обойти эту проблему.

## toString(base)

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

let num = 255;

console.log( num.toString(16) );  *// ff*

console.log( num.toString(2) );   *// 11111111*

Параметр base может принимать целые значения из диапазона от 2 до 36 (по умолчанию 10). Интересной особенностью записью методов у чисел служит такой синтаксис:

let res = 127..toString(16);

console.log( res );

После любого числа можно поставить две точки и вызвать любой числовой метод.

## Методы округления чисел

В JavaScript имеется встроенный объект Math, предоставляющий различные математические методы. Здесь мы рассмотрим функции, связанные с округлением чисел. Они следующие:

* Math.floor – округление в меньшую сторону;
* Math.ceil – округление в большую сторону;
* Math.round – округление до ближайшего целого.

Например:

let dig = 1.5;

let res = Math.floor(dig);

console.log( res );        *// 1*

console.log( Math.ceil(dig) );       *// 2*

console.log( Math.round(1.4) );      *// 1*

console.log( Math.round(1.5) );      *// 2*

А что если мы хотим округлить число до второго знака после запятой (до сотых)? Это можно сделать, например, так:

let dig = 1.23456;

let res = Math.round(dig\*100)/100;

console.log( res );

Но для такой операции в JavaScript имеется встроенный числовой метод toFixed(n), где n – число знаков после запятой. Наш пример можно переписать так:

console.log( dig.toFixed(2) );

## isFinite и isNaN

Если по каким-то причинам результатом вычислений становятся значения Infinite или NaN, то было бы полезно иметь функции, которые бы умели «узнавать» такие выражения. Этими функциями как раз и являются isFinite и isNaN. Например:

console.log( isNaN(**NaN**) );       *// true*

console.log( isNaN("1") );     *// false*

console.log( isNaN(2) );          *// false*

console.log( isNaN("abc") );    *// true*

По аналогии работает и вторая функция isFinit:

console.log( isFinite("15") ); *// true*

console.log( isFinite("str") ); *// false, потому что специальное значение: NaN*

console.log( isFinite(**Infinity**) ); *// false*

Как видим, она возвращает true, если это число и оно ограничено (не бесконечно) и false в других случаях.

Кстати, иногда isFinite используется для проверки: содержится ли в строке число:

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

**if**( isFinite(num) ) console.log("это число");

**else** console.log("это не число");

Правда, если num будет пустой строкой, то эта функция решит, что это число 0 (так как пустая строка приводится к числу 0). Поэтому, более точная проверка будет такой:

**if**( num.length > 0 && isFinite(num) ) console.log("это число");

**else** console.log("это не число");

## parseInt и parseFloat

Данные методы позволяют выделить число из строки, даже если в строке имеются другие не числовые символы. Например, при работе с CSS часто встречаются такие выражения: «12pt», «100%», «340px» и прочее. Если мы попытаемся получить значения с помощью уже знакомых нам подходов:

let arg1 = "12pt";

let arg2 = "100%";

let arg3 = "340px";

console.log( Number(arg1) );

console.log( +arg2 );

console.log( +arg3 );

То всюду получим значение NaN. Но, функии parseInt и parseFloat справляются с такими задачами. Они преобразовывают строку в число до тех пор, пока либо не дойдут до конца, либо не встретится ошибка преобразования. В данном случае, получим всюду правильные числовые значения:

console.log( parseInt(arg1) );

console.log( parseInt(arg2) );

console.log( parseInt(arg3) );

Если же в строке предполагается вещественное число, то применяется вторая функция:

console.log( parseFloat("12.5pt") );

console.log( parseFloat("  90.5% ") );

console.log( parseFloat("+30.5px") );

Функция parseInt имеет второй необязательный аргумент, в котором можно указать систему, в которой представлено число, например:

console.log( parseInt('0xff', 16) ); *// 255*

console.log( parseInt('aa', 16) ); *// 170, можно и без 0x*

console.log( parseInt('11011', 2) ); *// 27*

## Другие математические функции

Math.random() возвращает псевдослучайное число в диапазоне (0; 1], например:

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

   console.log( Math.random() );

Причем, при каждом запуске будем получать разные последовательности чисел.

Math.max(a, b, c...) / Math.min(a, b, c...) возвращают наибольшее/наименьшее число из переданных аргументов:

let max = Math.max(1, 2, 0, -10, 5, 7);

let min = Math.min(1, 2, -10, 5);

console.log( max );        *// 7*

console.log( min );        *// -10*

Число аргументов может быть любым.

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

console.log( Math.pow(2, 10) ); *// 2 в степени 10 = 1024*

Объект Math содержит множество других функций (cos, sin) и констант, например, PI, которыми удобно оперировать в JavaScript.

# Методы и свойства строк

[Смотреть материал на видео](https://www.youtube.com/watch?v=Rvn5x8vGGGg&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/metody-i-svoystva-strok>

По аналогии с числами у строк тоже есть свой набор методов, упрощающих работу с ними. И, как вы уже догадались, на этом занятии мы будем рассматривать методы работы со строками. Но в начале отметим, что все строки в JavaScript представлены в кодировке UTF-16, даже если в браузере HTML-документ имеет другую кодировку.

Итак, поехали. Мы с вами уже говорили о разнообразии кавычек для представления строковых литералов, например:

let str1 = "Hello World!";

let str2 = 'Hello World!';

let str3 = `Hello World!`;

Все это одинаковые строки, то есть, кавычки «работают» одинаково, но последние предоставляют нам более широкий функционал и позволяют вписывать значения переменных непосредственно в строку:

let name = "Java", age = 18;

let str = `Меня зовут ${name}, мне ${age} лет`;

console.log( str );

Обо всем это мы с вами уже говорили. Продолжим. В строках можно указывать специальные управляющие символы (они называются спецсимволами). Наиболее распространенных несколько:

* \n – спецсимвол перевода на новую строку;
* \r – возврат каретки (используется в ОС Windows совместно с символом \n);
* \t – спецсимвол табуляции;
* \uXXXX – символ в кодировке UTF-16;
* \" и \' – символы кавычек (внутри строки);
* \\ - символ обратного слеша.

Существуют и другие спецсимволы, но они используются довольно редко. Вот примеры использования этих символов:

let str = 'Hello!**\n**I**\'**m Javascript.**\n**Вот мой символ **\t** табуляции, обратный слеш **\\** и символ **\u**00A9 копирайта';

console.log( str );

Далее рассмотрим частые свойства и методы, которые имеются у строковых переменных и литералов.

## length

Свойство length содержит длину строки, например:

let str = 'Hello!**\n**I**\'**m Javascript.';

console.log( str.length );

Увидим в консоле длину строки str. Обратите внимание, что length – это свойство, а не метод, поэтому вот такое обращение str.length() работать не будет.

## [] и charAt

Из любой строки можно взять отдельный символ. Это делается с помощью оператора [], в которых указывается индекс нужного символа. Например:

let str = 'Hello!**\n**I**\'**m Javascript.';

let ch1 = str[0];

let ch2 = str[7];

console.log(ch1, ch2);

console.log(**typeof** ch1);

Обратите внимание, что так как в JavaScript нет символьного типа, то возвращаемое значение – это строка, состоящая из одного символа.

Ту же самую операцию выделения символа можно выполнить и с помощью метода charAt(pos), но он менее удобен и существует, скорее по историческим причинам для совместимости со старыми скриптами:

let ch1 = str.charAt(0);

let ch2 = str.charAt(7);

Интересной особенностью JavaScript является возможность перебрать строку посимвольно с помощью цикла for of, используемого для массивов:

**for**(let ch of "Hello")

   console.log(ch);

Обратите внимание, что строки в JavaScript изменять нельзя. Например, нельзя выполнить такую операцию:

str[0] = "h";

получим ошибку исполнения. Если нужно изменить строку, то создается новая измененная строка.

## toLowerCase() и toUpperCase()

Данные методы возвращают строку в нижнем и верхнем регистрах. Например:

let str = 'Hello!';

let low = str.toLowerCase();

let hi = "string".toUpperCase();

console.log(low, hi);

Обратите внимание, что мы можем вызывать методы непосредственно у строковых литералов, а не только у переменных.

## infexOf и lastIndexOf

Данный метод выполняет поиск подстроки substr, начиная с позиции pos:

str.indexOf(substr[, pos])

возвращает позицию, на которой располагается совпадение, либо -1 если совпадений не найдено.

let str = '<span class="clock">12:34</span>';

let indx1 = str.indexOf("clock");        *// 13*

let indx2 = str.indexOf("span", 2);      *// 27*

let indx3 = str.indexOf("div"); *// -1*

console.log(indx1, indx2, indx3);

Обратите внимание, данный метод находит только одно первое совпадение, дальше поиск не продолжается. Если нужно найти все совпадения, то можно реализовать такой простой алгоритм:

let indx = -1;

while(**true**) {

indx = str.indexOf("span", indx+1);

**if**(indx == -1) **break**;

console.log(indx);

}

Другой похожий метод

str.lastIndexOf(substr, position)

ищет подстроку с конца строки к началу. Он используется тогда, когда нужно получить самое последнее вхождение:

let indx = str.lastIndexOf("span");

console.log(indx);

## includes, startsWith, endsWith

Следующие три метода позволяют проверять: есть ли в строке искомая подстрока. Первый метод имеет ожидаемый синтаксис:

str.includes(substr[, pos])

он возвращает true, если подстрока substr была найдена в строке str и false в противном случае. Второй необязательный параметр pos указывает начальный индекс для поиска. Вот примеры использования данного метода:

let str = '<span class="clock">12:34</span>';

console.log( str.includes("span") );

console.log( str.includes("<span>") );

console.log( str.includes("clock", 20) );

Следующие два метода startsWith и endsWith проверяют, соответственно, начинается ли и заканчивается ли строка определённой строкой:

console.log( str.startsWith("span") );    *//false*

console.log( str.startsWith("<span") );        *//true*

console.log( str.endsWith("span>") );   *//true*

## slice

Есть три основных метода для выделения подстрок из строки – это substring, substr и slice. Метод slice имеет следующий синтаксис:

str.slice(start [, end])

и возвращает часть строки от start до end (не включая его). Например:

console.log( str.slice(0, 5) );       *//<span*

console.log( str.slice(6, 11) );   *//class*

console.log( str.slice(12) );         *//"clock"...*

console.log( str.slice(-7, -1) );     *//</span*

Следующий метод

str.substring(start [, end])

работает практически также как и slice, но здесь аргумент start может быть больше, чем end, например:

console.log( str.substring(6, 11) );     *//class*

console.log( str.substring(11, 6) );     *//class*

Но отрицательные значения записывать нельзя, они будут трактоваться как 0.

Последний метод

str.substr(start [, length])

Возвращает часть строки, начиная с индекса start и длиной в length символов. В противоположность предыдущим методам, здесь указывается длина вместо конечной позиции:

console.log( str.substr(6, 13) );   *//class = "clock"*

console.log( str.substr(12) );     *//"clock">12:34</span>*

При отрицательном значении первого аргумента позиция отсчитывается с конца строки.

Какой из этих трех методов выбирать для выделения строк? По большому счету без разницы. Они все работают эффективно, так что это дело предпочтения программиста.

## Нюансы сравнения строк

Если мы проверяем строки на равенство, то никаких особых проблем в JavaScript это не вызывает, например:

**if**("abc" == "abc") console.log( "строки равны" );

**if**("abc" != "ABC") console.log( "строки не равны" );

Но, когда мы используем знаки больше/меньше, то строки сравниваются в лексикографическом порядке. То есть:

1. Если код текущего символа одной строки больше кода текущего символа другой строки, то первая строка больше второй (остальные символы не имеют значения), например:

console.log( "z" > "Za" );        *//true*

2. Если код текущего символа одной строки меньше кода текущего символа другой строки, то первая строка меньше второй:

console.log( "B" < "a" ); *//true*

3. При равенстве символов больше та строка, которая содержит больше символов:

console.log( "abc" < "abcd" ); *//true*

4. В остальных случаях строки равны:

console.log( "abc" == "abc" ); *//true*

Но в этом алгоритме есть один нюанс. Например, вот такое сравнение:

console.log( "Америка" > "Japan" ); *//true*

Дает значение true, так как русская буква A имеет больший код, чем латинская буква J. В этом легко убедиться, воспользовавшись методом

str.codePointAt(pos)

который возвращает код символа, стоящего в позиции pos:

console.log( "А".codePointAt(0), "J".codePointAt(0) );

Сморите, у буквы А код равен 1040, а у буквы J – 74. Напомню, что строки в JavaScript хранятся в кодировке UTF-16. К чему может привести такой результат сравнения? Например, при сортировке мы получим на первом месте страну «Japan», а потом «Америка». Возможно, это не то, что нам бы хотелось? И здесь на помощь приходит специальный метод для корректного сравнения таких строк:

str.localeCompare(compareStr)

он возвращает отрицательное число, если str < compareStr, положительное при str > compareStr и 0 если строки равны. Например:

console.log( "Америка".localeCompare("Japan") );      *// -1*

возвращает -1 как и должно быть с учетом языкового сравнения. У этого метода есть два дополнительных аргумента, которые указаны в документации JavaScript. Первый позволяет указать язык (по умолчанию берётся из окружения) — от него зависит порядок букв. Второй позволяет определять дополнительные правила, например, чувствительность к регистру.

## Некоторые другие методы

У строковых переменных есть еще пара полезных и часто используемых методов, это:

str.trim()

убирает пробелы в начале и конце строки:

let str = "   string   ";

console.log( str.trim() );

и

str.repeat(n)

для повторения строки n раз:

let str = "Abc";

console.log( str.repeat(5) );

Это, конечно, не все методы строк. По мере использования JavaScript вы познакомитесь со многими другими, но для начала этого будет достаточно. Также отдельно стоит тема регулярных выражений – мощнейший инструмент для поиска, замены и проверки различных строковых шаблонов, но об этом мы будем говорить на отдельном занятии.

# Коллекции Map и Set

[Смотреть материал на видео](https://www.youtube.com/watch?v=Ggmk21ZNSKQ&list=PLA0M1Bcd0w8x9TltCzZDhw0SatK1d10yy" \t "_blank)

<https://proproprogs.ru/javascript/kollekcii-map-i-set>

Итак, мы с вами рассмотрели примитивные типы, объекты и массивы. Для полноты картины не хватает еще двух структур данных – это карты (Map) и наборы (Set). И на этом занятии мы с ними познакомимся. Начнем с карт Map.

## Map

Помните, когда мы говорили об объектах, то отмечали, что ключи (имена свойств) являются строками – всегда строки. Так вот, в отличие от объектов в Map в качестве ключа могут выступать не только строки, но вообще любые типы данных.

У Map есть такие 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.

Рассмотрим их. Например:

let m = **new** Map();

m.**set**("string", "строка");

m.**set**(7, "простое число");

m.**set**(**true**, {descr: "boolean", value: **true**});

console.log( m.**get**("string") );

console.log( m.**get**(7) );

console.log( m.**get**(**true**) );

В качестве ключей можно использовать и объекты, например:

let user = {

name: "JavaScript",

type: "ES6"

};

let m = **new** Map();

m.**set**(user, "объект user");

console.log( m.**get**(user) );

Часто коллекцию Map используют благодаря этому свойству – возможности использования объектов в качестве ключей. Ни одна другая структура в JavaScript не обладает таким свойством.

Чтобы сравнивать ключи, объект Map использует алгоритм SameValueZero. Это почти такое же сравнение, что и ===, с той лишь разницей, что NaN считается равным NaN. Так что NaN также может использоваться в качестве ключа. Причем этот встроенный алгоритм сравнения не может быть заменён или модифицирован.

Создать объект Map также можно на основе двумерного массива:

let car = **new** Map([

   ["model", "opel"],

   ["color", 0xff],

   ["price", 1000]

]);

Здесь каждый элемент массива интерпретируется как «ключ – значение». В итоге коллекция car содержит три элемента с ключами model, color, price и соответствующими значениями.

Вообще, объект Map можно создать на основе любого итерируемого объекта, представляющего данные в формате «ключ - значение». Например, можно взять некий объект:

let book = {

author: "Пушкин",

title: "Онегин",

pages: 100,

price: 80

};

И создать из него карту:

let lib = **new** Map(Object.entries(book));

Здесь метод объекта entries преобразует объект book в массив с элементами «ключ – значение», на основе которого и формируется lib.

Есть также противоположный метод Object.fromEntries, который из двумерного массива по формату «ключ-значение», формирует объект:

let prices = Object.fromEntries([

['banana', 1],

['orange', 2],

['meat', 4]

]);

В частности, этот принцип используется для преобразования Map в Object:

let objLib = Object.fromEntries(lib.entries());

console.log(objLib);

Здесь метод entries коллекции Map, также как и у Object, преобразует содержимое в двумерный массив по формату «ключ-значение». Затем, с помощью метода fromEntries этот двумерный массив преобразуется в объект. Вот так можно Map превратить в Object.

Но, вернемся к коллекции car, переберем его содержимое. Для этого, обычно, используется цикл for of следующим образом:

**for** (let value of car) {

console.log(value);

}

В результате value принимает вид массива [ключ, значение] и выводится в консоль.

Если нам нужно выбрать только ключи из коллекции car, то можно сделать так:

**for** (let key of car.keys()) {

console.log(key);

}

Здесь мы в цикл передаем итерируемый (перебираемый) объект по ключам и в цикле выводим ключи в консоль.

Для отображения значений соответствующих ключей, передается другой итерируемый объект:

**for** (let value of car.values()) {

console.log(value);

}

В отличие от обычных объектов Object, в Map перебор происходит в том же порядке, в каком происходило добавление элементов.

Еще одним вариантом перебора служит цикл forEach, встроенный в объект Map:

Map.forEach( function(value, key, map) {

// что-то делаем

});

Например, можно вывести в консоль информацию объекта Map следующим образом:

car.forEach((value, key) => {

console.log( `car[${key}] = ${value}` );

});

Здесь используется стрелочная функция с двумя входными аргументами value и key.

## Set

Коллекция Set формируется из уникальных данных (без ключей – только данные). Уникальность означает, что одно и то же значение не может быть добавлено дважды – оно просто проигнорируется.

Данный объект имеет следующие методы и свойства:

* new Set(iterable) – создаёт Set, если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в Set;
* set.add(value) – добавляет значение (если оно уже есть, то ничего не происходит), возвращает тот же объект set;
* set.delete(value) – удаляет значение, возвращает true если value было найдено и удалено, иначе false;
* set.has(value) – возвращает true, если значение присутствует в коллекции, иначе false;
* set.clear() – удаляет все значения из набора;
* set.size – возвращает количество элементов в наборе.

Например, мы ожидаем посетителей, и нам необходимо составить их список. Но повторные визиты не должны приводить к дубликатам. Каждый посетитель должен появиться в списке только один раз. Здесь как раз и подойдет множество Set:

let guests = **new** Set();

let alex = { name: "Alexey", old: 25 };

let oleg = { name: "Oleg", old: 32 };

let masha = { name: "Masha", old: 18 };

guests.add(alex);

guests.add(oleg);

guests.add(masha);

guests.add(alex);

guests.add(masha);

Выведем в консоль получившийся набор:

**for** (let guest of guests) {

console.log(guest.name);

}

Получим только троих гостей. Все так, как и должно быть.

Перебрать объект Set также можно и с помощью встроенного метода forEach. Он имеет следующий синтаксис:

Set.forEach( function(value, valueAgain, set) {

// что-то делаем

});

Здесь второй аргумент valueAgain добавлен для синтаксической совместимости данного метода с аналогичным методом из Map. Это бывает полезно, если программист решит вместо Set использовать Map. Тогда ему достаточно будет поменять Set на Map и все заработает уже с новым объектом.

Выполним перебор элементов с помощью forEach:

guests.forEach( (item) => {

console.log(item.name+": "+item.old);

});

Как видите, все достаточно просто. Перебор Map и Set всегда осуществляется в порядке добавления элементов, так что нельзя сказать, что это – неупорядоченные коллекции, но поменять порядок элементов или получить элемент напрямую по его номеру здесь не получится.