### 1. Общие сведения о JavaScript, назначение, особенности. - Хорунжая

JavaScript - это относительно простой объектно-ориентированный язык, предназначенный для создания небольших клиентских и серверных приложений для Internet.

Назначение

**JavaScript** обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам.

Программы на этом языке называются *скриптами*. Они могут встраиваться в HTML и выполняться автоматически при загрузке веб-страницы.

Скрипты распространяются и выполняются, как простой текст. Им не нужна специальная подготовка или компиляция для запуска.

Сегодня JavaScript может выполняться не только в браузере, но и на сервере или на любом другом устройстве, которое имеет специальную программу, называющуюся [«движком» JavaScript](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%B8%D0%B6%D0%BE%D0%BA_JavaScript).

У браузера есть собственный движок, который иногда называют «виртуальная машина JavaScript».

Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что изначально был создан для браузеров, не требующих этого.

Возможности JavaScript сильно зависят от окружения, в котором он работает. Например, [Node.JS](https://ru.wikipedia.org/wiki/Node.js) поддерживает функции чтения/записи произвольных файлов, выполнения сетевых запросов и т.д.

В браузере для JavaScript доступно всё, что связано с манипулированием веб-страницами, взаимодействием с пользователем и веб-сервером.

Например, в браузере JavaScript может:

* Добавлять новый HTML-код на страницу, изменять существующее содержимое, модифицировать стили.
* Реагировать на действия пользователя, щелчки мыши, перемещения указателя, нажатия клавиш.
* Отправлять сетевые запросы на удалённые сервера, скачивать и загружать файлы (технологии [AJAX](https://ru.wikipedia.org/wiki/AJAX) и [COMET](https://ru.wikipedia.org/wiki/Comet_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))).
* Получать и устанавливать куки, задавать вопросы посетителю, показывать сообщения.
* Запоминать данные на стороне клиента («local storage»).

Особенности:

* Полная интеграция с HTML/CSS.
* Простые вещи делаются просто.
* Поддерживается всеми основными браузерами и включён по умолчанию.

### 2. Переменные и константы, области видимости; глобальные переменные - Карпешина

Когда в программе необходимо сохранить значение, чтобы использовать его позже, это значение присваивается переменной. Переменная – это просто символьное имя для значения, которое обеспечивает возможность получить значение по имени, то есть, когда в программе указывается имя переменной вместо неё подставляется значение.

Константа – это просто символьное имя для значения. Константа даёт возможность обратиться к значению по имени, это означает, что, когда в программе указывается имя константы, вместо неё подставляется значение. Константы используются для хранения данных, которые не должны изменяться во время выполнения программы. Прежде чем использовать константу, её необходимо объявить. Константы объявляются с помощью ключевого слова const, за которым следует имя константы.

Переменные let:

* Видны только после объявления и только в текущем блоке.
* Нельзя переобъявлять (в том же блоке).
* При объявлении переменной в цикле for(let …) – она видна только в этом цикле. Причём каждой итерации соответствует своя переменная let.

Переменная const – это константа, в остальном – как let.

Область видимости переменной – это участок исходного кода программы, в котором переменные и функции видны и их можно использовать.

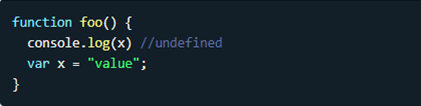
Переменная, объявленная вне функции или блока, называется глобальной.

Переменная, объявленная внутри функции, называется локальной. Локальная переменная доступна в любом месте внутри тела функции, в которой она была объявлена. Локальная переменная создаётся каждый раз заново при вызове функции и уничтожается при выходе из неё. Локальная переменная имеет преимущество перед глобальной переменной с тем же именем, это означает, что внутри функции будет использоваться локальная переменная, а не глобальная.

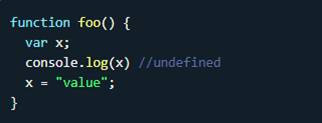
### 3. «Всплытие» (hoisting) объявлений - Ожерельева

Hoisting (поднятие) – это механизм, используемый в JavaScript для передвижения переменных и объявлений функций вверх своей области видимости (до того, как код будет выполнен). Не имеет значения, где были объявлены функция или переменные, все они передвигаются вверх своей области видимости, вне зависимости от того локальная она или же глобальная.

Стоит отметить то, что механизм “поднятия” передвигает только объявления функции или переменной. Назначения переменным остаются на своих местах. Так, например, это не сработает с директивой let и const.



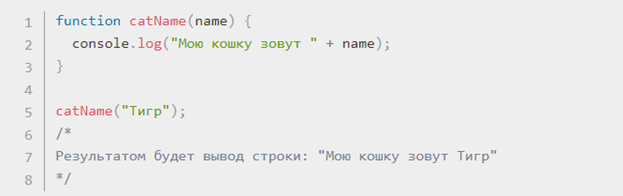
Пример всплытия выглядит так:



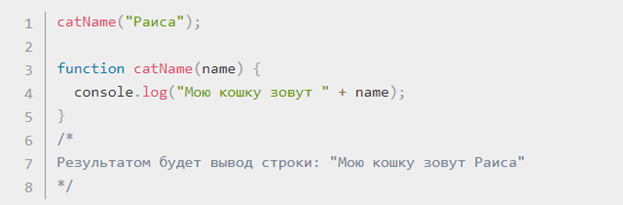
«Underfind», т.к. JavaScript «поднял» объявление переменной. Во избежание underfind, нужно объявлять и инициализировать переменные (запустить их в работу и присвоить начальное значение) перед их использованием.

Одним из преимуществ помещения в память объявлений функций до выполнения кода то, что можно использовать функцию до ее объявления. Это происходит благодаря тому, как работает контекст выполнения в JavaScript.

Пример обычного (ожидаемого) рабочего кода:



Пример вызывания функции в коде до ее объявления (hoisting):



Всплытие позволяет запускать объявленные функции выше, чем они объявлены в контексте, тем самым является понятным способом объяснения происходящего.

(На самом же деле, объявления переменных и функций попадают в память в процессе фазы компиляции, но остаются в коде на том месте, где вы их объявили.)

### 4. Типы данных JavaScript. Примитивные типы и тип Object.

Стандарт ECMAScript® определяет следующие типы данных:

* **Простые** (их также называют примитивными) типы:
  + логический (англ. Boolean)
  + нулевой (англ. Null)
  + неопределённый (англ. Undefined)
  + числовой (англ. Number)
  + строковый (англ. String)
  + символ (англ. Symbol​)
* **Объектный** (англ. Object) – это коллекция именованных значений, которые обычно называют свойствами (properties) объекта.

**Булевый (логический) тип «boolean»**

Логические, или булевы значения, могут иметь лишь одно из двух значений: true (истина) или false (ложь). Значения true или false обычно появляются в операциях сравнения или логических операциях. В качестве сравнительного выражения можно использовать любое выражение. Любое выражение, которое возвращает значение 0, null, undefined или пустую строку, интерпретируется как false. Выражение, определяющее какое-либо другое значение, интерпретируется как true.

**Тип данных Number**

Для представления чисел в JavaScript используется 64-битный формат, определяемый стандартом [IEEE-754](http://en.wikipedia.org/wiki/IEEE_754-1985). Число, находящееся непосредственно в коде программы, называется числовым литералом. Помимо десятичных целых литералов JavaScript распознает шестнадцатеричные значения. Числа в шестнадцатиричном формате могут включать любую последовательность цифр от 0 до 9 и буквы от a до f, которая обязательно начинается с последовательности символов «0x».

**Тип данных String**

Строковый тип (string) - это неизменяемая, упорядоченная последовательность 16-битных значений, каждое из которых представляет символ Unicode (буквы, цифры, знаки пунктуации, специальные символы и пробелы). Строки могут быть пустыми либо состоять из одного и более символов. Строки создаются при помощи двойных (") или одинарных (') кавычек. В строке, ограниченной парой одинарных кавычек, можно использовать двойные кавычки, и наоборот – одинарные кавычки можно использовать в строке, заключенной в пару двойных кавычек.

**Тип данных Null**

Нулевой тип (null) содержит единственное специальное значение – null. Ключевое слово null невозможно использовать в качестве имени функции или переменной. Значение null является ссылкой на «пустой» объект и имеет специальное назначение – обычно оно используется для инициализации переменной, которой впоследствии будет присвоено значение.

**Тип данных Undefined**

Неопределенный тип (undefined) образует свой собственный тип, который содержит единственное специальное значение – undefined. Такое значение имеет переменная, объявленная с помощью оператора var, но не инициализированная. Значение undefined возвращается при обращении к переменной, которой никогда не присваивалось значение, а также к несуществующему свойству объекта или элементу массива.

**Тип данных Символ (Symbol)**

Символ (symbol) является нововведением JavaScript начиная с ECMAScript версии 6. Символ – это уникальное, неизменяемое, примитивное значение, которое служит для создания уникальных идентификаторов.

Важной особенностью символа также является то, что его значение уникально. Символ можно использовать как имя свойства, которое гарантированно не будет повторяться с любым другим свойством. Даже, если у двух символов одинаковое имя, то это не значит, что они равны.

**Тип данных Object**

В дополнение к рассмотренным выше примитивным типам данных JavaScript поддерживает составной тип данных – объект (object). Объект состоит из многих значений (либо элементарных, таких как числа и строки, либо сложных, например других объектов), которые называются свойствами объекта. Доступ к свойствам осуществляется по имени свойства («по ключу»). Имя свойства является строкой, поэтому можно считать, что объекты связывают строки со значе­ниями. Вместе эти фрагменты информации образуют пары «ключ-значение».

Создание объекта с помощью объектного литерала начинается с определения обыч­ной переменной. В правой части этой инструкции записывается литерал объекта – это заключенный в фигурные скобки {} список разделенных запятой пар *"имя-значение"*, заключенный в фигурные скобки. Имя свойства и значение отделены друг от друга двоеточием. Второй способ создания объектов связан с использованием конструктора Object(). При этом сначала используется выражение new Object(), а затем определяются и инициализируются свойства полученного объекта.

### 5. Приведение типов, явное и неявное. - Юля Силова

Вообще приведение типов - это операция приведения значения одного типа в значение другого типа. Различают явное приведение типов и неявное приведение типов.

Для JavaScript применимы следующие определения:

***Явное приведение*** *задаётся программистом в тексте программы с помощью средств языка, либо специальных функций, которые принимают значение одного типа и возвращают значение другого типа.*

***Неявное приведение*** *выполняется транслятором (компилятором или интерпретатором) по правилам, описанным в стандарте языка.*

## Зачем нужны приведения типов

Во-первых, эта операция успешно применяется, когда мы используем в своём коде условные операторы. Приведение в этом случае осуществляется автоматически. Для того, чтобы это контролировать нужно иметь представление, какое значение в какое преобразуется, чтобы избежать нежелательного поведения программы в будущем.

javascript

|  |  |
| --- | --- |
| 1  2  3  4  5 | let a = 5;  if (a) {  console.log('a автоматически привелось к true');  } |

Во-вторых, приведение очень помогает, когда, например, нам пришла строка с данными, но по факту в этой строке находится число и оно нам нужно для сложения. В этом случае, без явного приведения будет совсем грустно.

javascript

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1  2  3  4  5  6 | let data = '25';  *// Явное приведение*  let res2 = parseInt(data) + 25; *// 50*  let res1 = data + 25; *// "2525"*  *Таблица явных приведений типов*   |  |  | | --- | --- | | *Логическое* | *!!'0' === true* | | *!!0 === false* | | *!!'5' === true* | | *!!'' === false* | | *!!undefined === false* | | *!!null === false* | | *!!NaN === false* | | *Boolean([]) === true* | | *В число* | *+'132' === 132* | | *'132' \* 1 === 132* | | *'132' - 0 === 132* | | *Number('132') === 132* | | *Number('132.132') === 132.132* | | *Number('132abc') === NaN* | | *Number('132.132abc') === NaN* | | *parseInt('132abc') === 132* | | *parseInt('132.132abc') === 132.132* | | *parseFloat('132abc') === 132* | | *parseFloat('132.132') === 132.132* | | *parseFloat('132.132abc') === 132.132* | | *В строку* | *True + '' === 'True'* | | *String(132) === '132'* | | *String(null) === 'null'* |  *Таблица неявных преобразований* *На примере alert() и if (value) { }*   |  |  | | --- | --- | | *Логическое* | *if (undefined) === false* | | *if (null) === false* | | *if (132) === true* | | *if (0) === false* | | *if (NaN) === false* | | *if ('false') === true* | | *if ('') === false* | | *В число* | *alert( +true ) === 1* | | *alert( +false ) === 0* | | *В строку* | *alert( true ) === 'true'* | |

### 6. Литералы (literals) JavaScript - Назар Данилыч

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Grammar_and_types> нужно пролистать страничку вниз, там будет отлично написано с примерами(но там оч много), поэтому, берем

### **Понятие литерала(**https://true-coder.ru/js-dlya-nachinayushhix/js-dlya-nachinayushhix-urok-1-literaly.html

### **)**

**Литерал** – это любое значение указанное явным образом в коде. В качестве литералов в js могут выступать числа, строки (текстовые значения), логические значения. Функциональные литералы, массивы и объекты мы рассмотрим позже.

### **Числовые литералы**

В js дробные и целые числа в десятичной системе счисления записываются в виде последовательности цифр. Дробная часть десятичной дроби отделяется точкой. Если опустить целую часть десятичной дроби, то она будет считаться равной нулю. Например, литерал .8 описывает значение 0.8.

Примеры числовых литералов: 0, 2, 4, 8, 3.14, .2.

### **Экспоненциальная форма записи числовых литералов**

Числовые литералы могут представляться в экспоненциальной форме записи. Согласно этой записи вначале (необязательно, если есть дробная часть) идут цифры, затем дробная часть (необязательно, если есть целая часть), отделённая точкой, затем (необязательно) знак «+» или минус и буква *e* (E).

Такая запись обозначает число, умноженное на 10 в степени, определяемой числом, которое стоит после буквы *e* (E). Например, 6.2e10 — это 6.2 умножить на 10 в 10-ой степени, 1.2E-2 – это 1.2 умножить на 10 в минус второй степени, .1e1 — это единица, -.2e3 – это -200 и т.д.

### **Шестнадцатеричные литералы**

Такие литералы задают числа в шестнадцатеричной системе счисления, они начинаются с символов «0x» или «0X», а затем следует запись числа в шестнадцатеричной форме. Такая запись содержит числа от 0 до 9 и большие или маленькие буквы a, b, c, d, e, f.

Примеры шестнадцатеричных литералов: 0xFF, 0Xbb, 0x123, 0x0.

Для перевода числа из десятеричной системы счисления в шестнадцатиричную необходимо делить нацело (без остатка) на 16 пока не получится ноль, записывая остатки от деления в обратном порядке, при этом остаток 10 запишется как A, 11 как B и т.д.

Переведём число 123 из десятичной, в шестнадцатеричную систему счисления. Символ «/» будет означать деление нацело.

123 / 16 = 7, остаток 11 (запишется как B вторым символом)

7 / 16 = 0, остаток 7

В итоге шестнадцатеричное представление числа 123 – 7B, а литерал, описывающий это значение – 0X7B.

Для перевода числа из шестнадцатеричной формы представления в десятичную необходимо начиная с последних брать цифры этого числа, умножать на 16 в степени номер цифры считая с нуля и с конца числа, при этом заменяя цифры A, B, C, D, E, F на 10, 11, 12, 13, 14 и 15. Например, 7B = B \* 16^0 + 7 \* 16^1 = 11 \* 1 + 7 \* 16 = 11 + 112 = 123.

Шестнадцатеричная форма записи используется, как более краткое представление числа в текстовом виде. С помощью этой формы записи задаются, например, цвета в СSS-свойствах и кодируются кириллические символы в get-запросе.

### **Восьмеричные литералы**

Такие литералы задают числа в восьмеричной системе счисления, они начинаются с нуля, а затем следует запись числа в восьмеричной форме. Такая запись содержит числа от 0 до 7.

Перевод числа из восьмеричной системы счисления в десятичную и назад выполняется аналогично случаю шестнадцатеричных чисел, только делить и умножать надо на 8.

Примеры восьмеричных литералов: , 012, 0123, 02.

Если после нуля стоит последовательность цифр, содержащая 8 или 9, то такой числовой литерал описывает число в десятичной системе счисления, состоящее из цифр, стоящих после нуля.

### **Строковые литералы**

Строковый литерал (строка) представляет собой последовательность символов, заключённых в двойные («) или одинарные (‘) кавычки. Строковый литерал необходимо записывать в одну строку, в нём могут содержаться одинарные кавычки, если строка ограничена двойными кавычками и двойные кавычки, если строка ограничена одинарными кавычками.

Примеры строковых литералов: ‘Привет мир!’, «site=’true-coder.ru'», », ‘123’.

Некорректные строковые литералы: ‘mess=’hello ‘, «двойные («) кавычки».

### **Логические значения**

Логический тип данных имеет всего два возможных значения: true, false. Логическое значение говорит об истинности чего-либо. Возможно, вам проще будет рассматривать эти значения, как «да» (true) или «нет» (false).

### 7. Блоки инструкций (block statements), условные инструкции (if..else, switch)

if (*условие*)

*инструкция1*

[else

*инструкция2*]

Несколько команд if ... else могут быть вложены для создания условия else if. Обратите внимание, что в JavaScript нет ключевого слова elseif (в одно слово).

if (*условие1*)

*инструкция1*

else if (*условие2*)

*инструкция2*

else if (*условие3*)

*инструкция3*

...

else

*инструкцияN*

Чтобы увидеть, как это работает, ниже представлен пример правильного вложения с отступами:

if (*условие1*)

*инструкция1*

else

if (*условие2*)

*инструкция2*

else

if (*условие3*)

...

Чтобы выполнить несколько инструкций в условии, используйте блочный оператор ({...}) для группирования этих инструкций. В общем, хорошей практикой всегда является использование блочных операторов, особенно в коде, включающем вложенные операторы if:

if (*условие*) {

*инструкции1*

} else {

*инструкции2*

}

### 8. Обработка исключений - Пояркова Лена

Оператор обработки исключений try...catch обычно применяется в следующих ситуациях:

* Для некоторого участка кода, в котором может возникнуть ошибка;
* Для валидации (проверки) данных перед отправкой на сервер;
* Для использования в сложных случаях при написании кроссбраузерных приложений.

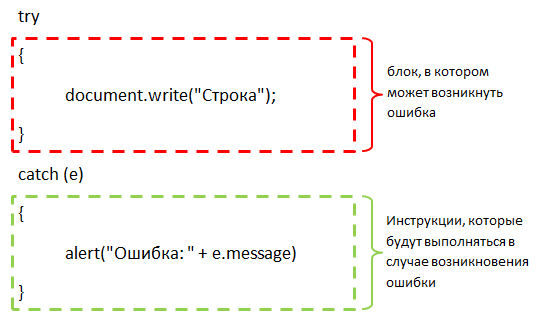
Теперь давайте рассмотрим, что же произойдет в сценарии на JavaScript при возникновении ошибки и для чего нужен оператор try...catch.

Если в сценарии на языке JavaScript появится ошибка, то дальнейшее выполнение программы будет прекращено. Т.е. инструкции (операторы), которые идут после ошибки выполняться не будут. Но если нам необходимо выполнение программы не прерывать, а переходить к следующей инструкции после той, где может возникнуть ошибка, то эту инструкцию необходимо заключить в оператор "try...catch".

## Синтаксис оператора try...catch

Принцип работы с оператором try...catch заключается в следующем:

Блок, в котором могут возникнуть ошибки, мы обрамляем фигурными скобками, и перед ним пишем ключевое слово try (с англ. попробовать). После этого блока пишем ключевое слово catch (с англ. поймать, ловить) и в круглых скобках указываем переменную, в которой будем хранить информацию об ошибке. Далее фигурными скобками обрамляем блок, предназначенный для обработки ошибок.



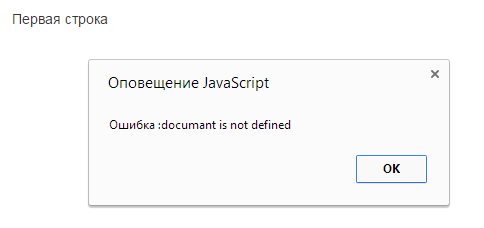
## Принцип работы оператора try...catch

Оператор обработки исключений работает следующим образом:

Сначала он пытается выполнить все инструкции (операторы), указанные в блоке try. Если внутри блока try ошибки не возникает, то блок catch просто игнорируется, т.е. он не выполняется. В том случае если внутри блока try возникает ошибка, то оставшиеся операторы в этом блоке будут проигнорированы (т.е. они выполняться не будут) и сразу же начнут выполняться операторы внутри блока catch. Самое главное то, что инструкции, идущие после оператора try...catch, продолжат выполняться и работа программы не остановится.

1. try
2. {
3. document.write("Первая строка");
4. documant.write("Вторая строка");
5. document.write("Третья строка");
6. }
7. catch (e)
8. {
9. alert("Ошибка :" + e.message);
10. }

В результате выполнения кода в 4 строчке произойдет ошибка, т.к. оператор documant браузером будет не определён. В этом случае 5 строчка браузером выполнена не будет, т.к. начнут выполняться операторы в блоке catch. В качестве параметра блока catch определим переменную e, в которой будет хранить информацию об ошибке. Данную переменную мы будем использовать в операторе alert для вывода сообщения об ошибке.



## Блок finally

У оператора обработки исключений есть ещё один блок, который называется finally. Данный блок является не обязательным, и его можно использовать только при необходимости. Он не имеет параметров и в отличие от блока catch выполняется всегда, вне зависимости от того возникла ошибка или нет.

Но так как инструкции, идущие после оператора try...catch тоже выполняются всегда, то возникает вопрос: "Зачем нужен блок finally?"

На самом деле, инструкции, идущие после try...catch, выполняются не всегда. Это может произойти только в том случае, если произойдет ошибка в блоке catch. Когда это произойдёт, программа приостановит своё выполнение и операторы, идущие после конструкции try...catch выполняться не будут. Единственный блок инструкций (операторов), который будет выполняться – это блок finally.

1. try
2. {
3. document.write("Первая строка
4. ");
5. documant.write("Вторая строка
6. ");
7. document.write("Третья строка
8. ");
9. }
10. catch (e)
11. {
12. alrt("Ошибка :" + e.message);
13. }
14. finally
15. {
16. document.write("Завершающие действия");
17. }
18. alert("Операторы после try...catch");

В результате выполнения кода оператор alert, идущий после конструкции try...catch так и не будет выполнен.

### 9. Циклы for - Руслан Солнышко

Цикл for используется для выполнения итераций по элементам массивов или объектов, напоминающих массивы, таких как arguments и HTMLCollection. Условие проверяется перед каждой итерацией цикла. В случае успешной проверки выполняется код внутри цикла, в противном случае код внутри цикла не выполняется и программа продолжает работу с первой строки, следующей непосредственно после цикла.

Цикл for состоит из трёх разных операций:

Шаг 1. инициализация var i = 0; — объявление переменной-счётчика, которая будет проверяться во время выполнения цикла. Эта переменная инициализируется со значением 0. Чаще всего в качестве счётчиков цикла выступают переменные с именами i, j и k.

Шаг 2. проверка условия i < 5; — условное выражение, если оно возвращает true, тело цикла (инструкция в фигурных скобках) будет выполнено. В данном примере проверка условия идёт до тех пор, пока значение счётчика меньше 5.

Шаг 3. завершающая операция i++ — операция приращения счётчика, увеличивает значение переменной var i на единицу. Вместо операции инкремента также может использоваться операция декремента.

По завершении цикла в переменной var i сохраняется значение 1. Следующий виток цикла выполняется для for (var i = 1; i < 5; i++) { }. Условное выражение вычисляется снова, чтобы проверить, является ли значение счётчика i всё ещё меньше 5. Если это так, операторы в теле цикла выполняются ещё раз. Завершающая операция снова увеличивает значение переменной на единицу. Шаги 2 и 3 повторяются до тех пор, пока условие i < 5; возвращает true.

Чтобы вывести значения массива с помощью цикла for, нужно задействовать свойство массива length. Это поможет определить количество элементов в массиве и выполнить цикл такое же количество раз.

Если значение свойства length не изменяется в ходе выполнения цикла, можно сохранить его в локальной переменной, а затем использовать эту переменную в условном выражении. Таким образом можно повысить скорость выполнения цикла, так как значение свойства length будет извлекаться всего один раз за всё время работы цикла.

### 10. Циклы do..while - Алайская

Выражение **do...while** создает цикл, который выполняет указанное выражение до тех пор, пока условие не станет ложным. Условие проверяется после выполнения выражения, то есть выражение выполнится как минимум один раз.

## **Синтаксис**

do

*выражение*

while (*условие*);

**выражение**

Выражение, которое выполняется как минимум один раз и выполняется на каждом шаге цикла, пока условие истинно. Выражение может содержать несколько строк, для этого необходимо сгруппировать код в [блок](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/block) ({ ... }).

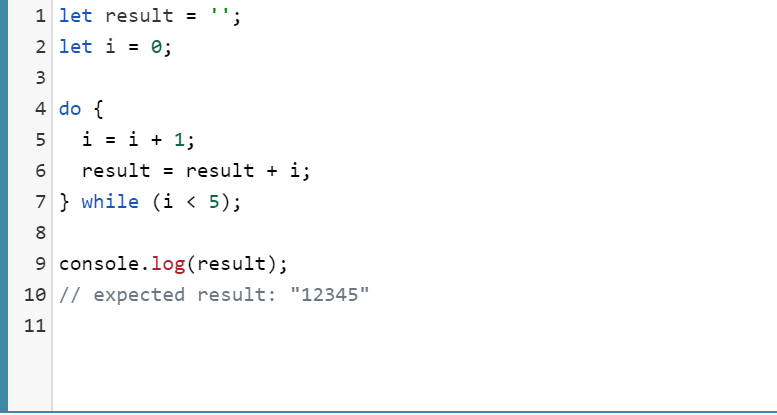
**условие**

Выражение, которое вычисляется после каждого шага цикла. Если условие истинно, то выражение выполняется еще раз. Когда условие ложно, выполняется выражение, следующее после do...while.

## **Примеры**

Использование do...while

В примере, цикл do...while выполняется до тех пор, пока i не перестанет быть меньше 5.



### 11. Циклы for..in - Алайская

**Цикл for...in** проходит через перечисляемые свойства объекта. Он пройдёт по каждому отдельному элементу.

## **Синтаксис**

for (variable in object) {*...*

}

**variable**

Другое (очередное) имя свойства назначается переменной на каждой итерации.

**object**

Объект, по чьим свойствам мы проходим

## **Описание**

Цикл for...in проходит только по перечисляемым свойствам. Объекты, созданные встроенными конструкторами, такими как Array и Object имеют неперечисляемые свойства от Object.prototype и String.prototype, например, от [String](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String)-это [indexOf()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf), а от [Object](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object) - метод [toString()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/toString). Цикл пройдёт по всем перечисляемым свойствам объекта, а также тем, что он унаследует от конструктора прототипа (свойства объекта в цепи прототипа).

## **Примеры**

Следующее выражение берёт аргументом объект. Затем проходит по всем перечислимым свойствам объекта и возвращает строку содержащую имена свойств и их значения.

var obj = {a:1, b:2, c:3}; for (var prop in obj) { console.log("obj." + prop + " = " + obj[prop]); } // Выведет: // "obj.a = 1" // "obj.b = 2" // "obj.c = 3"

Проход по массиву и for...in

**Замечание:** for...in не следует использовать для [Array](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array), где важен порядок индексов.

Индексы массива - это перечисляемые свойства с целочисленными именами, в остальном они аналогичны свойствам объектов. Нет гарантии, что for...in будет возвращать индексы в конкретном порядке. Цикл for...in возвращает все перечисляемые свойства, включая имеющие нецелочислиненные имена и наследуемые.

### 

### 12. c - Джикаева Тамара

### for…of

Оператор for...of относится к типу оператора for, который циклически повторяет итерируемые объекты ( iterable objects)), пока не достигнет конца строки.

Рассмотрим базовый пример:

let arr = [2,4,6,8,10]

for(let a of arr) {

log(a)

}

// It logs:

// 2

// 4

// 6

// 8

// 10

Цикл for...of через массив arr выполнен с меньшим количеством кода, чем при использовании цикла for.

let myname = "Nnamdi Chidume"

for (let a of myname) {

log(a)

}

// It logs:

// N

// n

// a

// m

// d

// i

//

// C

// h

// i

// d

// u

// m

// e

При использовании цикла for приходится задействовать математику и логику, чтобы рассчитать момент достижения конца myname и прекратить процесс. Однако с циклом for...of можно забыть о лишней головной боли.

У цикла for...of есть одно общее определение:

for ( variable of iterable) { //...}

variable хранит значение всех свойств итерируемого объекта на каждой итерации. iterable — итерируемый объект.

### 13. Управление циклами: операторы label, break, continue - Доманский Павел

При работе с циклами в JavaScript существую операторы, которые помогают управлять циклом. Такими операторами выступают break, continue и label.  
Оператор label: Данный оператор представляет из себя метку, к которой можно вернуться с помощью команд break и continue. Иными словами данная метка представляет из себя альтернативный способ выхода из цикла.  
Синтаксис: Данная операция представляет из себя блочное выражение, которое выступает в качестве ссылки.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var counter = 0;  outerloop:  for (i=3; i<10; i++){  if (counter =3){  break outerloop // Если counter = 3 - прекращает выполнение  цикла с label outerloop:  }else{  counter++ // В ином случае продолжает выполнение цикла  }}  console.log(counter)  //expected output: 3 |

label:  
statement

где label - любое слово, которое не является командой  
statement - инструкция которое включается в себя взаимодействие с label (операции break, continue и тд)

Оператор break: Оператор break прерывает выполнение текущего цикла или цикла с меткой label. После чего переносит выполнение программы к следующему оператору, следующим за циклом (прерванным оператором)  
Синтаксис:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | var counter = 0;  outerloop:  for (i=3; i<10; i++){  if (counter =3){  break outerloop // Если counter = 3 - прекращает  выполнение цикла с label outerloop:  }else{  counter++ // В ином случае продолжает выполнение цикла  }}  console.log(counter)  //expected output: 3 |

break; или же break label;

где break - оператор прекращения выполнения операции.  
label - опционально. С помощью данной метки можно отменить выполнение не того цикла, в котором находится break, а именно того цикла на который ссылается label.  
Оператор continue:  
Инструкция continue прерывает выполнение текущей итерации текущего или отмеченного цикла. После чего продолжает его выполнение на следующей итерации.  
Синтаксис:

continue label; или continue;  
где label - необязательная метка цикла.

|  |  |
| --- | --- |
| 1  2  3  4  5  6    7  8  9  10 | var counter = 0;  var counter2= 0  outerloop: for(i=0; i<10; i++){  if (counter<=5){  counter++ // Когда первый счетчик меньше или равен 5, мы прибавляем к первому счетчику +1 и прекращаем выполнение цикла outerloop.  continue outerloop}  Когда же счетчик равен 6, мы прибавляем ко второму счетчику +1 и заканчиваем цикл outerloop  counter2++  break outerloop}  console.log(counter, counter2)  //Expected output: 6, 1 |

Важно!  
По сравнению с инструкцией [break](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/break), continue прерывает выполнение цикла не полностью, вместо этого:

* В цикле [while](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/while) оно переносит поток выполнения к условию.
* В цикле [for](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/for) оно переносит поток выполнения к финальному выражению в описании цикла.

Инструкция continue может использоваться вместе с необязательной меткой, которая будет начинать следующую итерацию отмеченного цикла, а не текущего. В этом случае, continue должен находиться внутри отмеченного блока, который соответствует метке.

### 14. Функции: способы объявления и способы вызова; стрелочные функции; возвращаемое значение - Соня Морозова

**Существует три способа объявления функции: Function Declaration, Function Expression и Named Function Expression.**

**Function Declaration** (сокращённо FD) – это "классическое" объявление функции. В JavaScript функции объявляются с помощью литерала функции. Синтаксис объявления FD:

|  |  |
| --- | --- |
| 1 | function идентификатор (параметры) { инструкции } |

Литерал функции состоит из следующих четырёх частей:

1. Ключевое слово function.

2. Обязательный идентификатор, определяющий имя функции. В качестве имени функции обычно выбирают глагол, т. к. функция выполняет действие.

3. Пара круглых скобок вокруг списка из нуля или более идентификаторов, разделяемых запятыми. Данные идентификаторы называются параметрами функции.

4. Тело функции, состоящее из пары фигурных скобок, внутри которых располагаются инструкции. Тело функции может быть пустым, но фигурные скобки должны быть указаны всегда.

Простой пример:

|  |  |
| --- | --- |
| 1  2  3 | function sayHi() {  alert("Hello");  } |

Встречая ключевое слово function интерпретатор создаёт функцию и затем присваивает ссылку на неё переменной с именем sayHi (переменная с данным именем создаётся интерпретатором автоматически).

**Function Expression** (сокращённо FE) – это объявление функции, которое является частью какого-либо выражения (например присваивания). Синтаксис объявления FE:

|  |  |
| --- | --- |
| 1 | function (параметры) { инструкции } |

Простой пример:

|  |  |
| --- | --- |
| 1  2  3 | var sayHi = function () {  alert("Hello");  }; |

Функцию FE иначе ещё называют "

**Named Function Expression** (сокращённо NFE) – это объявление функции, которое является частью какого-либо выражения (например присваивания). Синтаксис объявления NFE:

|  |  |
| --- | --- |
| 1 | function идентификатор (параметры) { инструкции } |

Простой пример:

|  |  |
| --- | --- |
| 1  2  3 | var sayHi = function foo() {  alert("Hello");  }; |

Объявления FE и NFE обрабатываются интерпретатором точно так же, как и объявление

Программный код, расположенный в теле функции, выполняется не в момент объявления функции, а в момент её вызова. как FE или

**Пять способов вызвать функцию**

Механизмы вызова функций в JavaScript имеют ряд важных отличий, и незнание их может вылиться в ошибки, которые будет непросто найти.

Давайте напишем простую функцию, которая возвращает массив из трех элементов — текущего значения this и двух аргументов, переданных в функцию.

function makeArray(arg1, arg2){

return [ this, arg1, arg2 ];

}

### **Самый распространенный способ: глобальный вызов**

Новички часто объявляют функции так, как показано в примере выше. Вызвать эту функцию не составляет труда:

makeArray('one', 'two'); // => [ window, 'one', 'two' ]

**Любой код в нашем скрипте, не «привязанный» к чему-либо (т.е. находящийся вне объявления объекта) на самом деле находится в контексте глобального объекта.** В нашем случае, makeArray — не просто функция. На самом деле, **makeArray — метод глобального объекта (в случае исполнения кода в браузере) window.**

**Правило вызова функций №1: Если функция вызывается напрямую, без указания объекта (например, myFunction()), значением this будет глобальный объект (window в случае исполнения кода в браузере).**

### **Вызов метода**

**Правило вызова функций №2: В функции, вызванной с использованием синтаксиса вызова метода, например, obj.myFunction() или obj[&#39;myFunction&#39;](), this будет иметь значение obj.**

### **Еще два способа: apply() и call()**

Логично, что чем чаще вы используете функции, тем чаще вам приходится передавать их и вызывать в разных контекстах. **Зачастую возникает необходимость переопределить значение this.** Если вы помните, функции в JavaScript являются объектами. На практике это означает, что у функций есть предопределенные методы. apply() и call() — два из них. Они позволяют переопределять значение this:

var car = { year: 2008, model: 'Dodge Bailout' };

makeArray.apply( car, [ 'one', 'two' ] ); // => [ car, 'one', 'two' ]

makeArray.call( car, 'one', 'two' ); // => [ car, 'one', 'two' ]

Эти два метода очень похожи. Первый параметр переопределяет this. Различия между ними заключаются в последющих аргументах: Function.apply() принимает массив значений, которые будут переданы функции, а Function.call() принимает аргументы раздельно.

**Правило вызова функций №3: Если требуется переопределить значение this, не копируя функцию в другой объект, можно использовать myFunction.apply( obj ) или myFunction.call( obj ).**

### **Конструкторы**

В JavaScript нет классов, а любой пользовательский тип нуждается в конструкторе. **Кроме того, методы пользовательского типа лучше объявлять через prototype, который является свойством фукции-конструктора.**

**Правило вызова функций №4: При вызове функции с оператором new, значением this будет новый объект, созданный средой исполнения JavaScript. Если эта функция не возвращает какой-либо объект явно, будет неявно возвращен this.**

**Стрелочные функции**

**Выражения стрелочных функций** имеют более короткий синтаксис по сравнению с [функциональными выражениями](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/function) и лексически привязаны к значению [this](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this) (но не привязаны к собственному [this](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/arguments), [super](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/super), или [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target)). Выражение стрелочных функций не позволяют задавать имя, поэтому стрелочные функции [анонимны](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/name), если их ни к чему не присвоить.

## **Синтаксис**

### **Базовый синтаксис**

(param1, param2, …, paramN) => { statements }

(param1, param2, …, paramN) => expression

// эквивалентно: (param1, param2, …, paramN) => { return expression; }

// Круглые скобки не обязательны для единственного параметра:

(singleParam) => { statements }

singleParam => { statements }

// Функция без параметров нуждается в круглых скобках:

() => { statements }

() => expression

// Эквивалентно: () => { return expression; }

### **Расширенный синтаксис**

// Когда возвращаете литеральное выражение объекта, заключите тело в скобки

params => ({foo: bar})

// [Rest параметры](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Rest_parameters) и [параметры по умолчанию](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/Default_parameters) поддерживаются

(param1, param2, ...rest) => { statements }

(param1 = defaultValue1, param2, …, paramN = defaultValueN) => { statements }

// [Деструктуризация](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment) тоже поддерживается

var f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;

f(); // 6

Два фактора повлияли на появление стрелочных функции: более короткий синтаксис и лексика this.Стрелочные функции не содержат собственный контекст [this](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this), а используют значение [this](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this) окружающего контекста.

**Строгий режим исполнения**

Поскольку значение this определяется лексикой, правила строгого режима ([strict mode](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Strict_mode)) относительно this игнорируются:

var f = () => { 'use strict'; return this; };

f() === window; // или глобальный объект

Все остальные правила строгого режима применяются как обычно.

Так как значение this определяется лексикой, вызов стрелочных функций с помощью методов call() или apply(), даже если передать аргументы в эти методы, не влияет на значение this.

Стрелочные функции не имеют собственного объекта arguments, поэтому в теле стрелочных функций arguments будет ссылаться на переменную в окружающей области.

### **Использование стрелочных функций как методов**

Как показано ранее, стрелочные функции лучше всего подходят для функций без методов. Посмотрим, что будет, когда мы попробуем их использовать как методы:

'use strict';

var obj = {

i: 10,

b: () => console.log(this.i, this),

c: function() {

console.log(this.i, this);

}

}

obj.b(); // prints undefined, Window {...} (или глобальный объект)

obj.c(); // prints 10, Object {...}

Стрелочные функции не объявляют привязку ("bind") их контекста this.

Стрелочные функции не могут быть использованы как конструктор и вызовут ошибку при использовании с new.

Ключевое слово [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield) не может быть использовано в теле стрелочной функции (за исключением случаев, когда разрешается использовать в функциях, вложенных в тело стрелочной функции). Как следствие стрелочные функции не могут быть использованы как генераторы.

**Тело функции**

Тело стрелочной функции может иметь краткую (concise body) или блочную (block body) форму.

Блочная форма не возвращает значение, необходимо явно вернуть значение.

var func = x => x \* x; // краткий синтаксис,

// неявно возвращает результат

var func = (x, y) => { return x + y; }; // блочный синтаксис,

// явно возвращает результат

Не забывайте оборачивать скобками объектные строки.

var func = () => ({ foo: 1 });

Стрелочная функция не может содержать разрывы строк между параметрами и стрелкой.

Поскольку стрелка в стрелочной функции не является оператором, то стрелочные функции имеют специальные правила разбора (парсинга), которые взаимодействуют с [приоритетами операторов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence/) иначе, чем в обычных функциях.

let callback;

callback = callback || function() {}; // ok

callback = callback || () => {};

// SyntaxError: invalid arrow-function arguments

callback = callback || (() => {}); // ok

**Возвращаемые значения** — это на самом деле просто значения, которые функция возвращает после своего завершения. Вы уже неоднократно встречали возвращаемые значения, хотя, возможно, и не осознавали этого. Напишем небольшой код:

var myText = 'I am a string';

var newString = myText.replace('string', 'sausage');

console.log(newString);

// функция replace() принимает строку,

// заменяет одну подстроку другой и возвращает

// новую строку со сделанными заменами

Мы вызываем функцию [replace()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace) на строке myText и передаем ей 2 параметра — заменяемую подстроку и подстроку, которой будем заменять. Когда функция завершит выполнение, она вернет значение, которым является новая строка со сделанными в ней заменами. В коде выше мы сохраняем это возвращаемое значение как значение переменной newString.

Некоторые функции не возвращают значения.

В основном, возвращаемое значение используется там, где функция является чем-то вроде вспомогательного звена при вычислениях. Вы хотите получить результат, который включает в себя некоторые значения. Эти значения вычисляются функцией, которая возвращает результат так, что он может быть использован в следующих стадиях вычисления.

### 15. Функции: области видимости и замыкания; сборка мусора - Соня Морозова

## **Область видимости**

Область видимости в JavaScript определяет, какие переменные доступны вам. Существуют два типа областей видимости: **глобальная** и **локальная**.

### **Глобальная область видимости**

Если переменная объявлена вне всех функций или фигурных скобок ({}), то считается, что она определена в **глобальной области видимости**.

const globalVariable = 'some value';

Как только происходит объявление глобальной переменной, можно использовать эту переменную везде в коде, даже в функциях.

const hello = 'Hello CSS-Tricks Reader!';

function sayHello () {

console.log(hello);

}

console.log(hello); // 'Hello CSS-Tricks Reader!'

sayHello(); // 'Hello CSS-Tricks Reader!'

Хотя можно объявлять переменные в глобальной области видимости, но не рекомендуется это делать. Всё из-за того, что существует вероятность пересечения имен, когда двум или более переменным присваивают одинаковое имя. Если переменные объявляются через const или let, то каждый раз, когда будет происходить пересечение имён, будет показываться сообщение об ошибке. Такое поведение нежелательно.

Если объявлять переменные через var, то вторая переменная после объявления перепишет первую. Такое поведение тоже нежелательно, т.к. код усложняется в отладке.

Итак, следует всегда объявлять локальные переменные, а не глобальные.

**Локальная область видимости**

Переменные, которые используются только в определенной части кода, считаются помещенными в локальную область видимости. Такие переменные называются **локальными**.

В JavaScript выделяют два типа локальных областей видимости:

* **область видимости функции**
* **и область видимости блока**.

*Сначала рассмотрим* ***область видимости функции***

#### **Область видимости функции**

Переменная, объявленная внутри функции, доступна только внутри функции. Код снаружи функции не имеет к ней доступа.

В примере ниже, переменная hello находится внутри области видимости функции sayHello:

function sayHello () {

const hello = 'Hello CSS-Tricks Reader!';

console.log(hello);

}

sayHello(); // 'Hello CSS-Tricks Reader!'

console.log(hello); // Ошибка, hello не определена

#### **Область видимости блока**

Переменная, объявленная внутри фигурных скобок {} через const или let, доступна только внутри фигурных скобок.

**Блочная область видимости** является частным случаем **области видимости функции**, т.к. функции объявляются с фигурными скобками (кроме случаев использования стрелочных функций с неявным возвращением значения).

**У функций нет доступа к областям видимости других функций**

Функции не имеют доступа к областям видимости других функций, когда они объявляются раздельно, даже если одна функция используется в другой.

**Вложенные области видимости**

Когда функция объявляется в другой функции, то внутренняя функция имеет доступ к переменным внешней функции. Такой поведение называется разграничением **лексических областей видимости**.В тоже время внешняя функция не имеет доступа к переменным внутренней функции.

**Замыкания**

Всякий раз, когда вы вызываете функцию внутри другой функции, вы создаете замыкание. Говорят, что внутренняя функция является замыканием. Результатом замыкания обычно является то, что в дальнейшем становятся доступными переменные внешней функции.

function outerFunction () {

const outer = `I see the outer variable!`;

function innerFunction() {

console.log(outer);

}

return innerFunction;

}

outerFunction()(); // I see the outer variable!

Так как внутренняя функция является возвращаемым значением внешней функции, то можно немного сократить код, совместив возврат значения с объявлением функции.

function outerFunction () {

const outer = `I see the outer variable!`;

return function innerFunction() {

console.log(outer);

}

}

outerFunction()(); // I see the outer variable!

**Благодаря замыканиям появляется доступ к внешней функции, поэтому они обычно используются для двух целей:**

**1. контроля побочных эффектов;**

**2. создания приватных переменных.**

### **Контроль побочных эффектов с помощью замыканий**

Побочные эффекты появляются, когда производятся какие-то дополнительные действия помимо возврата значения после вызова функции. Множество вещей может быть побочным эффектом, например, Ajax-запрос, таймер или даже console.log:

**Приватные переменные с замыканиями**

Как вы теперь знаете, переменные, созданные внутри функции, не могут быть доступны снаружи. Из-за того, что они не доступны, их также называют **приватными переменными**.Однако иногда требуется доступ к такой приватной переменной, и для этого используются замыкания.

**Сборка мусора**

Как уже упоминалось выше, проблема точного определения, когда какая-либо часть памяти "более не нужна" - однозначно неразрешима. В результате сборщики мусора решают поставленную задачу лишь частично.

**Сборка мусора на основе подсчёта ссылок**

Это наиболее примитивный алгоритм сборки мусора, сужающий понятие "объект более не нужен" до "для данного объекта более нет ни одного объекта, ссылающегося на него". Объект считается подлежащим уничтожению сборщиком мусора, если количество ссылок на него равно нулю.

#### **Пример**

var o = {

a: {

b:2

}

}; // создано 2 объекта. Один ссылается на другой как на одно из своих полей.

// Второй имеет виртуальную ссылку, поскольку присвоен в качестве значения переменной 'o'.

// Очевидно, что ни один из них не подлежит сборке мусора.

#### **Ограничение : циклические ссылки**

Основное ограничение данного наивного алгоритма заключается в том, что если два объекта ссылаются друг на друга (создавая таким образом циклическую ссылку), они не могут быть уничтожены сборщиком мусора, даже если "более не нужны".

Создаётся два ссылающихся друг на друга объекта, что порождает циклическую ссылку. Они не будут удалены из области видимости функции после завершения работы этой функции, таким образом, сборщик мусора не сможет их удалить, несмотря на их очевидную ненужность. **Так как сборщик мусора считает, что, раз на каждый из объектов существует как минимум одна ссылка, то уничтожать их нельзя.**

### **Алгоритм "Mark-and-sweep"**

Данный алгоритм сужает понятие "объект более не нужен" до "объект недоступен".

Основывается на понятии о наборе объектов, называемых *roots* (в JavaScript root'ом является глобальный объект). Сборщик мусора периодически запускается из этих roots, сначала находя все объекты, на которые есть ссылки из roots, затем все объекты, на которые есть ссылки из найденных и так далее. Стартуя из roots, сборщик мусора, таким образом, находит все *доступные* объекты и уничтожает недоступные.

Данный алгоритм лучше предыдущего, поскольку "ноль ссылок на объект" всегда входит в понятие "объект недоступен". Обратное же - неверно, как мы только что видели выше на примере циклических ссылок.

Все усовершенствования в области сборки мусора в интерпретаторах JavaScript за последние несколько лет представляют собой усовершенствования данного алгоритма, но не новые алгоритмы сборки мусора, поскольку дальнейшее сужение понятия "объект более не нужен" не представляется возможным.

16. Функции: объект this - Господин Роман

Значение this – это объект «перед точкой», который использовался для вызова метода. Когда функция вызывается как метод объекта, используемое в этой функции ключевое слово this принимает значение объекта, по отношению к которому вызван метод.

let user = {

### name: "Джон",

### age: 30,

### sayHi() {

### // this == это "текущий объект" == user

### console.log(‘Hello ’ + this.name);

### }

### };

### 

### user.sayHi(); // Output: Hello Джон

### Возможно получить доступ к объекту без слова this, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект)

### Но такой код будет ненадёжным. Если мы решим скопировать ссылку на объект user в другую переменную, например, user1 = user, и перезапишем переменную user чем-то другим, тогда будет осуществлён доступ к неправильному объекту при вызове метода из user1. et user = {

### name: "Джон",

### age: 30,

### sayHi() {

### alert( user.name ); // приведёт к ошибке

### }

### };

### 

### 

### let user1 = user;

### user = null; // обнулим переменную, не хранит ссылку на объект.

### 

### user1.sayHi(); // Ошибка! Внутри sayHi() используется user, которая больше не ссылается на объект!

### 17. Объект Function- Шульцева Арина

**Синтаксис**

*new Function([arg1[, arg2[, ...argN]],] functionBody)*

**Параметры**

*arg1, arg2, ... argN*

Имена, используемые функцией в качестве имён формальных аргументов. Каждое имя должно быть строкой, представляющий допустимый идентификатор JavaScript, либо списком таких строк, разделённых запятой; например "x", "theValue" или "a,b".

*functionBody*

Строка, содержащая инструкции JavaScript, составляющие определение функции.

**Описание**

Объекты Function, созданные констуктором Function, разбираются при создании функции. Это менее эффективно определения функции при помощи выражения function или инструкции function и вызова её внутри кода, поскольку такие функции разбираются вместе с остальным кодом.

Все аргументы, переданные в функцию, трактуются как имена идентификаторов параметров создаваемой функции, и имеют тот же порядок следования, что и при их передаче в конструктор функции.

Вызов констуктора Function как функции (без использования оператора new) имеет тот же самый эффект, что и вызов его как констуктора.

**Свойства и методы объекта Function**

Глобальный объект Function не имеет собственных методов или свойств, однако, поскольку он сам является функцией, он наследует некоторые методы и свойства через цепочку прототипов объекта Function.prototype.

**Пример: указание аргументов в конструкторе Function**

Следующий код создаёт объект Function, который принимает два аргумента.

// Создаём функцию, принимающую два аргумента, и возвращающую их сумму

var adder = new Function('a', 'b', 'return a + b');

// Вызываем функцию

adder(2, 6);

// > 8

Аргуметы "a" и "b" являются именами формальных аргументов, которые используются в теле функции, "return a + b".

### 18. Выражения и операторы - Дереза Виктория

**Выражения в JavaScript** представляют собой комбинации *операндов* и *операторов*.

*Операции* в выражениях выполняются последовательно в соответствии со значением приоритета (чем больше значение приоритета, тем он выше). Возвращаемый результат не всегда имеет значение того же типа, что и тип обрабатываемых данных. Например, в операциях сравнения участвуют операнды различных типов, но возвращаемый результат всегда будет логического типа.

РИС. 1. СТРУКТУРА ВЫРАЖЕНИЯ В JAVASCRIPT

**Операнды** — это данные, обрабатываемые сценарием JavaScript. В качестве операндов могут быть как простые типы данных, так и сложные, а также другие выражения.

**Операторы** — это символы языка, выполняющие различные операции с данными. Операторы могут записываться с помощью символов пунктуации или ключевых слов.

В зависимости от количества операндов различают следующие типы операторов:

**унарный** — в операции участвует один операнд;

**бинарный** — в операции участвуют два операнда;

**тернарный** — комбинирует три операнда.

Простейшая форма выражения — **литерал** — нечто, вычисляемое само в себя, например, число 100, строка "Hellow world". Переменная тоже может быть выражением, так как она вычисляется в присвоенное ей значение.

**3.14** //Числовой литерал

"Привет, МИР!" //Строковый литерал

false //Логический литерал

null //Литерал значения null

/java/ //Литерал регулярного выражения

{x:100, y:100} //Объектный литерал

[10, 20, 30, 40] //Литерал массива

function(x){return x+2;} //Функциональный литерал

**Res** //Переменная Res

Значение выражения-литерала – это просто значение самого литерала.

Значение выражения-переменной – это значение, содержащееся в переменной, или значение, на которое переменная ссылается.

Эти простые выражения-литералы можно объединять в более сложные выражения. **Операторы языка JavaScript**

**краткий обзор операторов в JavaScript в таблице, а полный обзор по ссылке если надо будет что-то дополнить** <https://html5book.ru/vyrazheniya-v-javascript/>

**Операторы языка JavaScript**

|  |  |  |  |
| --- | --- | --- | --- |
| **Приоритет** | **Оператор** | **Виды операндов** | **Выполняемые действия (операция)** |
| 15 | . | Объект, идентификатор | Обращение к свойству |
|  | [] | Массив, целое число | Индексация массива |
|  | () | Функция, аргументы | Вызов функции |
|  | new | Вызов конструктора | Создание нового объекта |
| 14 | ++ | Левостороннее выражение | Префиксный или постфиксный инкремент (унарный) |
|  | -- | Левостороннее выражение | Префиксный или постфиксный декремент (унарный) |
|  | - | Число | Унарный минус (смена знака) |
|  | + | Число | Унарный плюс (нет операции) |
|  | ~ | Целое число | Поразрядное дополнение (унарный) |
|  | ! | Логическое значение | Логическое дополнение (унарный) |
|  | delete | Левостороннее значение | Аннулирование определения свойства (унарный) |
|  | typeof | Любой | Возвращает тип данных (унарный) |
|  | void | Любой | Возвращает неопределенное значение (унарный) |
| 13 | \*, /, % | Числа | Умножение, деление, остаток |
| 12 | +, - | Числа | Сложение, вычитание |
|  | + | Строки | Объединение (конкатенация) строк |
| 11 | << | Целые числа | Сдвиг влево |
|  | >> | Целые числа | Сдвиг вправо с расширением знакового разряда |
|  | >>> | Целые числа | Сдвиг вправо с дополнением нулями |
| 10 | <, <= | Числа или строки | Меньше чем, меньше или равно |
|  | >, >= | Числа или строки | Больше чем, больше или равно |
|  | instanceof | Объект, конструктор | Проверка типа объекта |
|  | in | Строка, объект | Проверка наличия свойства |
| 9 | == | Любой | Проверка на равенство |
|  | != | Любой | Проверка на неравенство |
|  | === | Любой | Проверка на идентичность |
|  | !== | Любой | Проверка на неидентичность |
| 8 | & | Целые числа | Поразрядная операция И |
| 7 | ^ | Целые числа | Поразрядная операция исключающего ИЛИ |
| 6 | | | Целые числа | Поразрядная операция ИЛИ |
| 5 | && | Логические значения | Логическое И |
| 4 | || | Логические значения | Логическое ИЛИ |
| 3 | ?: | Логическое значение, любое, любое | Условный трехместный оператор |
| 2 | = | Левостороннее значение, любое | Присваивание |
|  | \*=, /=, %=, +=, =, <<=, >>=, >>>=, &=, ^=, |= | Левостороннее значение, любое | Присваивание с операцией |
| 1 | , | Любой | Множественное вычисление |

Большинство операторов, таких как +, -, /, \* и т.п. являются двухместными, то есть требуют наличия двух операндов (если кто не знает, что такое операнд, то см. [**здесь**](https://info-master.su/programming/kurs/expressions.php)).

Также имеются и унарные операторы, например, знак - может быть как бинарным (двухместным), так и унарным. Здесь:

**-7**

с помощью унарного оператора **-** мы меняем знак числа **7**.

Кроме того, в JavaScript есть один трёхместный оператор ?: (он объединяет в одно значение три выражения).

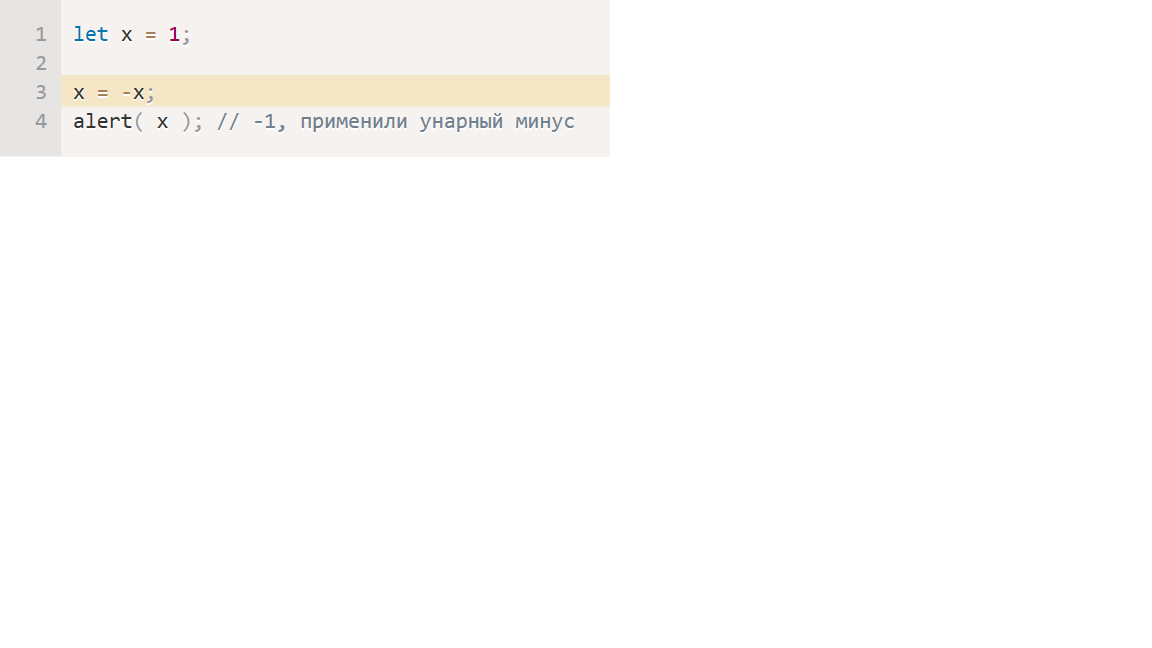
### 

### 19. Унарные, бинарные и тернарный операторы. Вагнер Сима

**Унарным** называется оператор, который применяется к одному операнду. Унарные операции выполняются справа налево.

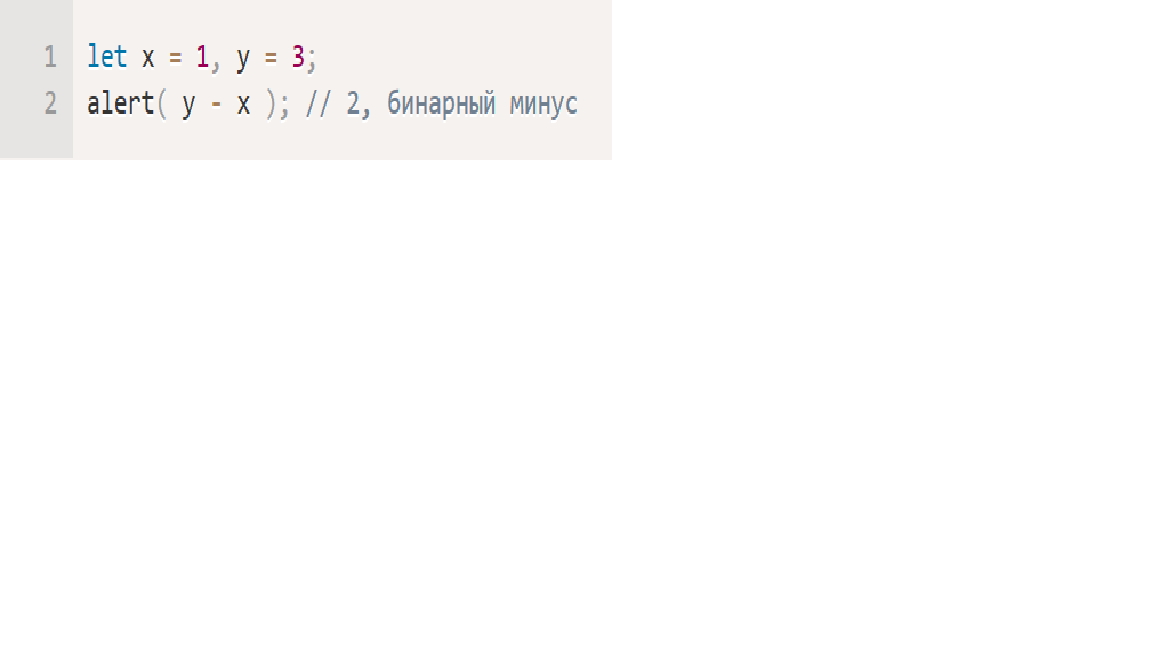
Операции увеличения и уменьшения увеличивают или уменьшают значение операнда на единицу и могут быть записаны как справа так и слева от операнда. Если знак операции записан перед операндом (префиксная форма), то изменение операнда происходит до его использования в выражении. Если знак операции записан после операнда (постфиксная форма), то операнд вначале используется в выражении, а затем происходит его изменение.

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



**Бинарным** называется оператор, который применяется к двум операндам. Бинарные операции выполняются слева направо.

Тот же минус существует и в бинарной форме:



**Тернарным** оператором в JavaScript называется символ - ? (вопрос). Такое название он получил от латинского "ternarius" - тройной. Он имеет 3 аргумента: условие - сравнение 1 - сравнение 2. А у обычных операторов аргументов всего 1 или 2. Вот поэтому он и называется: "тернарный оператор".

Тернарной условной операцией называют использование двух символов - ? и : (вопрос и двоеточие).

Символ двоеточия : в JavaScript называется "оператором сравнения", т.к. он ставится после условия, между 2 аргументами, один из которых будет выбран в качестве значения переменной, если он соответствует условию, конечно (Это альтернатива конструкции if - else).

Тернарный оператор работает только в паре, т.е. поставив оператор ?, нужно обязательно(!) использовать и оператор :. Иначе скрипт работать НЕ будет!

При вычислении выражений тип каждого операнда может быть преобразован к другому типу. Преобразования типов могут быть неявными, при выполнении операций и вызовов функций, или явными, при выполнении операций приведения типов.

### 20. Операторы присваивания

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

let x = 2 \* 2 + 1;

alert( x ); // 5

Возможно присваивание по цепочке:

let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4

alert( b ); // 4

alert( c ); // 4

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

**Оператор "=" возвращает значение**

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

let a = 1;

let b = 2;

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

alert( a ); // 3

alert( c ); // 0

В примере выше результатом (a = b + 1) будет значение, которое присваивается в a (то есть 3). Потом оно используется для дальнейших вычислений.

Забавное применение присваивания, не так ли? Нам нужно понимать, как это работает, потому что 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.

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

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

Например:

let n = 2;

n = n + 5;

n = n \* 2;

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

let n = 2;

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

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

alert( n ); // 14

Подобные краткие формы записи существуют для всех арифметических и побитовых операторов: /=, -= и так далее.

Вызов с присваиванием имеет в точности такой же приоритет, как обычное присваивание, то есть выполнится после большинства других операций:

let n = 2;

n \*= 3 + 5;

alert( n ); // 16 (сначала выполнится правая часть, превратив выражение в n \*= 8)

### 21. Деструктурирующее присваивание, spread-синтаксис, оставшиеся параметры (rest)

# **Деструктурирующее присваивание**

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array.

Объекты позволяют нам создавать одну сущность, которая хранит элементы данных по ключам, а массивы – хранить упорядоченные коллекции данных.

Но когда мы передаём их в функцию, то ей может понадобиться не объект/массив целиком, а элементы по отдельности.

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

# **spread-синтаксис**

Spread syntax позволяет расширить доступные для итерации элементы (например, массивы или строки) в местах:

* для функций: где ожидаемое количество аргументов для вызовов функций равно нулю или больше нуля
* для элементов (литералов массива)
* для выражений объектов: в местах, где количество пар "ключ-значение" должно быть равно нулю или больше (для объектных литералов)

function sum(x, y, z) {

return x + y + z;

}

const numbers = [1, 2, 3];

console.log(sum(...numbers)); // expected output: 6

console.log(sum.apply(null, numbers)); // expected output: 6

Spread syntax на самом деле переходит лишь на один уровень глубже при копировании массива. Таким образом, он может не подходить для копирования многоразмерных массивов, как показывает следующий пример: (также как и [Object.assign()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign))

# **оставшиеся параметры (rest -** [**«…»**](https://learn.javascript.ru/destructuring-assignment#ostatochnye-parametry)**)**

Если мы хотим не просто получить первые значения, но и собрать все остальные, то мы можем добавить ещё один параметр, который получает остальные значения, используя оператор «остаточные параметры» – троеточие ("..."):

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

alert(name1); // Julius

alert(name2); // Caesar

// Обратите внимание, что `rest` является массивом.

alert(rest[0]); // Consul

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

alert(rest.length); // 2

Переменная rest является массивом из оставшихся элементов. Вместо rest можно использовать любое другое название переменной, просто убедитесь, что перед переменной есть три точки и она стоит на последнем месте в деструктурирующем присваивании.

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

Что если в объекте больше свойств, чем у нас переменных? Можем ли мы взять необходимые нам, а остальные присвоить куда-нибудь?

Можно использовать троеточие, как и для массивов. В некоторых старых браузерах (IE) это не поддерживается, используйте Babel для полифила.

Выглядит так:

let options = {

title: "Menu",

height: 200,

width: 100

};

// title = свойство с именем title

// rest = объект с остальными свойствами

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

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

alert(rest.height); // 200

alert(rest.width); // 100

**итоги:**

* Деструктуризация позволяет разбивать объект или массив на переменные при присвоении.
* Полный синтаксис для объекта:  
   let {prop : varName = default, ...rest} = object
* Свойства, которые не были упомянуты, копируются в объект rest.  
  Полный синтаксис для массива:  
   let [item1 = default, item2, ...rest] = array
* Первый элемент отправляется в item1; второй отправляется в item2, все остальные элементы попадают в массив rest.  
  Можно извлекать данные из вложенных объектов и массивов, для этого левая сторона должна иметь ту же структуру, что и правая.

### 22. Арифметические операторы JavaScript.

**Арифметические операции** принимают в качестве операндов числовые значения (это может быть и литерал и переменная) и возвращают результат в виде одного числового значения. Стандартными арифметическими операциями являются сложение (+), вычитание (-), умножение (\*) и деление (/).

**Сложение (+).** Оператор сложения возвращает сумму числовых операндов или объединяет строки. **Operator:** x + y

**Вычитание (-).** Оператор вычитания вычитает один операнд из другого и возвращает разницу. **Operator:** x - y

**Деление (/).** Оператор деления производит деление его операндов, где левый операнд - делимый, а правый - делитель. **Operator: x / y**

**Умножение (\*).** Оператор умножения возвращает произведение операндов. **Operator: x \* y**

**Остаток от деления (%).** Оператор возвращает целый остаток от деления левого операнда на правый. Возвращаемое значение всегда получает знак делимого, а не делителя. Он использует встроенную функцию modulo, для получения результата, которая является целочисленным остатком деления var1 на var2 — например— var1 modulo var2. **Оператор:** var1 % var2

**Возведение в степень (\*\*).** Оператор возведения в степень возвращает результат первого операнда в степень. это, var1var2, в предыдущем выражении, где var1 и var2 - переменные. Он право ассоциативен. a \*\* b \*\* c равно a \*\* (b \*\* c). **Оператор:** var1 \*\* var2

**Инкремент (++).** Оператор инкремента увеличивает на единицу(инкрементирует) операнд и возвращает значение.

* Если операция используется как постфикс, с оператором после операнда (например, x++), значение операнда возвращается, а затем увеличивается на единицу.
* Если используется префиксная форма с оператором перед операндом (например, ++x), значение операнда возвращается увеличенным на единицу.

**Оператор:** x++ или ++x

**Декремент (--).** Операция декремента уменьшает на 1 (отнимает единицу) свой операнд и возвращает значение.

* Если операция используется как постфикс (например, x--), значение операнда возвращается, а затем уменьшается на единицу.
* Если используется префиксная форма (например, --x), значение операнда возвращается уменьшенным на единицу.

**Оператор:** x-- или --x

### 23. Логические операторы JavaScript.

Логические операторы используются, как правило, с примитивами [Boolean](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Boolean) (логического) типа. В этом случае результатом работы оператора является значение типа Boolean. Между тем операторы && и || возвращают, вообще говоря, значение одного из операнда, потому при использовании в качестве аргументов этих операторов величин, тип которых отличен от Boolean, тип возвращаемого значения может быть отличным от Boolean.

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Использование** | **Описание** |
| Логическое И (&&) | *expr1* && *expr2* | Возвращает значение expr1, если оно может быть преобразовано в false; иначе возвращает значение expr2. Таким образом, при использовании с величинами типа Boolean оператор && вернет true, если оба операнда могут быть преобразованы в true; иначе оператор && вернет false. |
| Логическое ИЛИ (||) | *expr1* || *expr2* | Возвращает значение expr1, если оно может быть преобразовано в true; иначе возвращает значение expr2. Таким образом, при использовании с величинами типа Boolean оператор || вернет true если хоть один из них равен true; в других случаях вернет false. |
| Логическое НЕ (!) | !*expr* | Возвращает false если значение expr можно привести к true; в противоположном случае возвращает true. |

Примеры значений выражений, которые могут быть преобразованы в false:

* null;
* NaN;
* 0;
* пустая строка ("");
* undefined.

Хоть операторы && и || могут использовать операнды с не булевыми значениями, но они всё равно рассматриваются, как булевы операторы, т.к. их возвращаемые ими значения всегда могут быть сконвертированы в булевы значения.

### 24. Операторы сравнения JavaScript.

В JavaScript имеются как строгие сравнения, так и сравнения с преобразованием типа операндов. Строгие сравнения (к примеру, ===) истинны только в том случае, если типы сравниваемых значений являются одинаковыми (к примеру: string-string, number-number). Однако, чаще используются сравнения с преобразованием типов (к примеру, ==). Такой тип сравнения, перед тем как непосредственно выполнить сравнение, приводит операнды к одному типу. В случае же, абстрактного реляционного сравнения, операнды сперва преобразуются в примитивы, затем приводятся к одному типу, и только после этого сравниваются.

Особенности сравнений:

* Две строки строго равны только в том случае, если они имеют одинаковую длину, и те же символы в одинаковой последовательности и соответствующих позициях.
* Два числа строго равны в том случае, если они численно равны. [NaN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/NaN) не равно ничему, в том числе и NaN. Нули с положительным и отрицательным знаком равны.
* Два логических значения (boolean) равны только в том случае, если они оба истинны (true) или ложны (false).
* Два различных объекта никогда не равны как в строгих, так и в абстрактных сравнениях.
* Сравнение объекта истинно лишь в том случае, если оба операнда ссылаются на один и тот же объект в памяти.
* Виды null и undefined равны себе как в строгом сравнении, так и в абстрактном.

**Операторы равенства**

**Равно (==).** *Оператор равно* сначала приводит операнды к одному типу, и затем применяет строгое сравнение. Если оба операнда являются объектами, то JavaScript сравнивает внутренние ссылки, которые равны в том случае, если они ссылаются на один и тот же объект в памяти.

x == y

**Не равно (!=).** Оператор не равно возвращает true в том случае, если операнды не равны.Он аналогичен оператору равенства, перед сравнением приводит операнды к одному типу. В случае если оба операнда являются объектами, JavaScript сравнивает внутренние ссылки, которые не равны в том случае, если относятся к разным объектам в памяти.

x != y

**Строго равно (===).** Оператор возвращает истину в том случае, если операнды строго равны (см. выше). В отличие от оператора равно, данный оператор не приводит операнды к одному типу.

x === y

**Строго не равно (!==).** Оператор строго не равно возвращает истину в том случае, если операнды не равны, или их типы отличаются друг от друга.

x !== y

**Операторы сравнения**

**Больше (>).** Оператор больше возвращает истину в том случае, если значение левого операнда больше, чем правого.

x > y

**Больше или равно (>=).** Оператор больше или равно, возвращает истину в том случае, если значение операнда слева больше или равно значению операнда справа.

x >= y

**Меньше(<).** Оператор меньше, возвращает истину в том случае, если значение операнда слева меньше, чем значение операнда справа.

x < y

**Меньше или равно (<=).** Оператор меньше или равно, возвращает истину в том случае, если значение операнда слева меньше, или равно значению операнда справа.

x <= y

### 25. Массивы. Индексированные и ассоциативные массивы

Массивы JavaScript используются для хранения множества значений в одной переменной. Самый простой способ создать массив в JavaScript это определить переменную-массив, присвоив ей нужные значения в виде константы-массива:

var **имя\_массива** = [*элемент1, элемент2, ...*];

var cars = ["Saab", "Volvo", "BMW"];

Массив — это особая переменная, которая может хранить более одного значения за раз.

Если у вас есть список неких значений (например, список марок автомобилей), то сохранение их в отдельных переменных будет выглядеть следующим образом:

var car1 = "Saab";

var car2 = "Volvo";

var car3 = "BMW";

Массив позволяет хранить множество значений под одним именем, и вы можете получить доступ к значению по его индексу.

**Индексирование массивов**

Чтобы получить доступ к элементу массива, необходимо обратиться к нему по его индексу.

В следующем выражении извлекается значение первого элемента массива **cars**:

var name = cars[0];

В следующем выражении изменяется первый элемент массива **cars**:

cars[0] = "Opel";

Первый элемент массива имеет индекс [0], второй [1] и т.д. Индексация массивов всегда начинается с 0.

Кроме этого JavaScript допускает использование всего массива, обратившись к нему по его имени:

var cars = ["Saab", "Volvo", "BMW"];

document.getElementById("demo").innerHTML = cars;

Добавление элементов с большими индексами может создать в массиве "дыры" со значением undefined.

В следующем примере в массиве fruits будут созданы "дыры" по индексам [4] и [5]:

var fruits = ["Банан", "Апельсин", "Яблоко", "Манго"];

fruits[6] = "Лимон"; // добавим новый элемент в массив fruits

### **Ассоциативные массивы**

Во многих языках программирования существует особый тип массивов, в которых в качестве индексов используются имена. Такие массивы называются ассоциативными.

JavaScript не поддерживает ассоциативные массивы.

В JavaScript массивы всегда используют цифровые индексы.

var person = [];

person[0] = "Иван";

person[1] = "Петров";

person[2] = 46;

var x = person.length; // person.length вернет 3

var y = person[0]; // person[0] вернет "Иван"

Если вы в качестве индексов укажете имена, то JavaScript переопределит такой массив в стандартный объект. После этого свойства и методы массива будут возвращать неверный результат.

В JavaScript массивы для доступа к элементам используют цифровые индексы. Объекты используют индексы-имена.

Таким образом, массивы — это особый тип объектов с цифровой индексацией элементов.

### 26. Объекты JavaScript - Калинина

Объекты используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка. Поэтому мы должны понять их, прежде чем углубляться куда-либо ещё.

Объект — это набор свойств, и каждое свойство состоит из имени и значения, ассоциированного с этим именем. Значением свойства может быть функция, которую можно назвать *методом* объекта. В дополнение к встроенным в браузер объектам, вы можете определить свои собственные объекты.

Объект может быть создан с помощью фигурных скобок {…} с необязательным списком *свойств*. Свойство – это пара «ключ: значение», где ключ – это строка (также называемая «именем свойства»), а значение может быть чем угодно.

Мы можем представить объект в виде ящика с подписанными папками. Каждый элемент данных хранится в своей папке, на которой написан ключ. По ключу папку легко найти, удалить или добавить в неё что-либо.

Пустой объект («пустой ящик») можно создать, используя один из двух вариантов синтаксиса:

let user = new Object(); // синтаксис "конструктор объекта"

let user = {}; // синтаксис "литерал объекта"

Обычно используют вариант с фигурными скобками {...}. Такое объявление называют *литералом объекта* или *литеральной нотацией*.

[Литералы и свойства](https://learn.javascript.ru/object#literaly-i-svoystva)

При использовании литерального синтаксиса {...} мы сразу можем поместить в объект несколько свойств в виде пар «ключ: значение»:

let user = { // объект

name: "John", // под ключом "name" хранится значение "John"

age: 30 // под ключом "age" хранится значение 30

};

Свойства объекта также иногда называют *полями объекта*.

У каждого свойства есть ключ (также называемый «имя» или «идентификатор»). После имени свойства следует двоеточие ":", и затем указывается значение свойства. Если в объекте несколько свойств, то они перечисляются через запятую.

В объекте user сейчас находятся два свойства:

1. Первое свойство с именем "name" и значением "John".
2. Второе свойство с именем "age" и значением 30.

Для обращения к свойствам используется запись «через точку»:

// получаем свойства объекта:

alert( user.name ); // John

alert( user.age ); // 30

Значение может быть любого типа. Давайте добавим свойство с логическим значением:

user.isAdmin = true;

Для удаления свойства мы можем использовать оператор delete:

delete user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки:

let user = {

name: "John",

age: 30,

"likes birds": true // имя свойства из нескольких слов должно быть в кавычках

};

Последнее свойство объекта может заканчиваться запятой:

let user = {

name: "John",

age: 30,

}

Это называется «висячая запятая». Такой подход упрощает добавление, удаление и перемещение свойств, так как все строки объекта становятся одинаковыми.

### [Создание объекта](https://javascript.ru/tutorial/object/intro#sozdanie-obekta)

Следующие два варианта создания объекта эквивалентны:

|  |
| --- |
| // эквивалентные записи |

|  |
| --- |
| var o = new Object() |

|  |
| --- |
| var o = {} |

### [Добавление свойств](https://javascript.ru/tutorial/object/intro#dobavlenie-svoystv)

Есть два синтаксиса добавления свойств в объект. Первый - точка, второй - квадратные скобки:

|  |
| --- |
| // эквивалентные записи |

|  |
| --- |
| o.test = 5 |

|  |
| --- |
| o["test"] = 5 |

Квадратные скобки используются в основном, когда название свойства находится в переменной:

|  |
| --- |
| var name = 'test' |

|  |
| --- |
| o[name] = 5 |

Здесь имя свойства "test" является ключом в ассоциативном массиве, по которому лежит значение 5.

лежит значение 5.

### [Доступ к свойствам](https://javascript.ru/tutorial/object/intro#dostup-k-svoystvam)

Доступ к свойству осуществляется точно так же:

|  |
| --- |
| alert(o.test) |

|  |
| --- |
| alert(o['test']) |

Если у объекта нет такого свойства, то результат будет 'undefined'

|  |
| --- |
| var o = {} |

|  |
| --- |
| alert(o.nosuchkey) // => undefined |

Никакой ошибки при обращении по несуществующему свойству не будет, просто вернется специальное значение [undefined](https://javascript.ru/undefined).

### [Удаление свойств](https://javascript.ru/tutorial/object/intro#udalenie-svoystv)

Удаляет свойство оператор [delete](https://javascript.ru/delete):

|  |
| --- |
| o.test = 5 |

|  |
| --- |
| delete o.test |

|  |
| --- |
| o['bla'] = true |

### [Расширенное создание](https://javascript.ru/tutorial/object/intro#rasshirennoe-sozdanie)

Свойства можно указывать непосредственно при создании объекта, через список в фигурных скобках вида {..., ключ : значение, ...}:

|  |  |
| --- | --- |
| 1 | var o = { |

|  |  |
| --- | --- |
| 2 | test:5, |

|  |  |
| --- | --- |
| 3 | bla:true |

|  |  |
| --- | --- |
| 4 | } |

Получившийся объект можно изобразить так:

javascript object

## [Методы объектов](https://javascript.ru/tutorial/object/intro#metody-obektov)

### [Добавление метода](https://javascript.ru/tutorial/object/intro#dobavlenie-metoda)

Как и в других языках, у объектов javascript есть методы.

Например, создадим объект rabbit с методом run

|  |  |
| --- | --- |
| 1 | var rabbit = {} |

|  |  |
| --- | --- |
| 2 | rabbit.run = function(n) { |

|  |  |
| --- | --- |
| 3 | alert("Пробежал "+n+" метров!") |

|  |  |
| --- | --- |
| 4 | } |

Добавление метода в объект - просто присвоение функции function(n) { ... } свойству rabbit.run.

Теперь можно запускать

|  |  |
| --- | --- |
| 1 | var rabbit = {} |

|  |  |
| --- | --- |
| 2 | rabbit.run = function(n) { |

|  |  |
| --- | --- |
| 3 | alert("Пробежал "+n+" метров!") |

|  |  |
| --- | --- |
| 4 | } |

|  |  |
| --- | --- |
| 5 | rabbit.run(5) // Пробежал 5 метров |

|  |  |
| --- | --- |
| 6 | rabbit.run(7) // Пробежал 7 метров |

Здесь не идет речь о классах, создании экземпляров и тому подобном. Просто - в любой объект в любое время можно добавить новый метод или удалить существующий.

Javascript - очень динамический язык, не правда ли?

### [Доступ к объекту из метода](https://javascript.ru/tutorial/object/intro#dostup-k-obektu-iz-metoda)

Обычно хочется, чтобы метод не просто вызывался из объекта, но имел доступ к самому объекту, мог менять находящиеся в нем данные.

Для этого используется ключевое слово this:

|  |
| --- |
| for(var key in obj) { |

|  |
| --- |
| … obj[key] … |

|  |
| --- |
| } |

В отличие от многих языков, this никак не привязано к объекту, а обозначает просто объект, вызвавший функцию.

Например,

|  |  |
| --- | --- |
| 01 | function this1() { |

|  |  |
| --- | --- |
| 02 | var vasya = { name:'Вася' } |

|  |  |
| --- | --- |
| 03 | var petya = { name:'Петя' } |

|  |  |
| --- | --- |
| 04 |  |

|  |  |
| --- | --- |
| 05 | sayName = function() { |

|  |  |
| --- | --- |
| 06 | alert("Я - "+ (this.name ? this.name : 'безымянный') ) |

|  |  |
| --- | --- |
| 07 | } |

|  |  |
| --- | --- |
| 08 |  |

|  |  |
| --- | --- |
| 09 | vasya.sayName = sayName |

|  |  |
| --- | --- |
| 10 |  |

|  |  |
| --- | --- |
| 11 | // один и тот же метод в двух объектах |

|  |  |
| --- | --- |
| 12 | petya.sayName = vasya.sayName |

|  |  |
| --- | --- |
| 13 |  |

|  |  |
| --- | --- |
| 14 | // тут - this будет petya |

|  |  |
| --- | --- |
| 15 | petya.sayName() // Я - Петя |

|  |  |
| --- | --- |
| 16 |  |

|  |  |
| --- | --- |
| 17 | // тут - this будет vasya |

|  |  |
| --- | --- |
| 18 | vasya.sayName() // Я - Вася |

|  |  |
| --- | --- |
| 19 |  |

|  |  |
| --- | --- |
| 20 | // а тут - вызывается метод глобального объекта window, у которого нет имени |

|  |  |
| --- | --- |
| 21 | sayName() // Я - безымянный |

|  |  |
| --- | --- |
| 22 | } |

### [Перебор свойств объекта](https://javascript.ru/tutorial/object/intro#perebor-svoystv-obekta)

Для перебора всех свойств объекта используется специальный вид конструкции for, [for..in](https://javascript.ru/for..in):

|  |  |
| --- | --- |
| 1 | for(var key in object) { |

|  |  |
| --- | --- |
| 2 | // key - название свойства |

|  |  |
| --- | --- |
| 3 | // object[key] - значение свойства |

|  |  |
| --- | --- |
| 4 | ... |

|  |  |
| --- | --- |
| 5 | } |

Например,

|  |  |
| --- | --- |
| 1 | var o = {a:5, b:true} |

|  |  |
| --- | --- |
| 2 | for (var key in o) { |

|  |  |
| --- | --- |
| 3 | alert(key+':'+o[key]) |

|  |  |
| --- | --- |
| 4 | } |

### 27. Классы JavaScript - Калинина

### Определение классов

На самом деле классы — это "специальные [функции](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions)", поэтому точно также, как вы определяете функции ([function expressions](https://developer.mozilla.org/ru-RU/docs/Web/JavaScript/Reference/Operators/function) и [function declarations](https://developer.mozilla.org/ru-RU/docs/Web/JavaScript/Reference/Statements/function)), вы можете определять и классы с помощью: [class declarations](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/class) и [class expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/class).

ОбъявлениеПервый способ определения класса — class declaration (*объявление класса*). Для этого необходимо воспользоваться ключевым словом class и указать имя класса (в примере — «Rectangle»).

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

#### Подъём (hoisting)

Разница между *объявлением функции* (*function declaration*) и *объявлением класса* (*class declaration*) в том, что *объявление функции* совершает подъём ([hoisted](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/%D0%9F%D0%BE%D0%B4%D0%BD%D1%8F%D1%82%D0%B8%D0%B5)), в то время как *объявление класса* — нет. Поэтому вначале необходимо объявить ваш класс и только затем работать с ним, а код же вроде следующего сгенерирует исключение типа [ReferenceError](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError):

var p = new Rectangle(); // ReferenceError

class Rectangle {}

Выражение класса

Второй способ определения класса — class expression (*выражение класса*). Можно создавать именованные и безымянные выражения. В первом случае имя выражения класса находится в локальной области видимости класса и может быть получено через свойства самого класса, а не его экземпляра.

// безымянный

var Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

console.log(Rectagle.name);

// отобразится: "Rectangle"

// именованный

var Rectangle = class Rectangle2 {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

console.log(Rectangle.name);

// отобразится: "Rectangle2"

Обратите внимание: выражения класса подвержены тем же проблемам с подъёмом (hoisting), что и объявления класса.

**Тело класса и задание методов**

Тело класса — это часть кода, заключенная в фигурные скобки {}. Здесь вы можете объявлять члены класса, такие как методы и конструктор.

Строгий режимТела *объявлений классов* и *выражений классов* выполняются в строгом режиме ([strict mode](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)).

ConstructorМетод [constructor](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Classes/constructor) — специальный метод, необходимый для создания и инициализации объектов, созданных, с помощью класса. В классе может быть только один метод с именем constructor. Исключение типа [SyntaxError](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError) будет выброшено, если класс содержит более одного вхождения метода constructor.

Ключевое слово super можно использовать в методе constructor для вызова конструктора родительского класса.

Методы прототипаСм. также [определение методов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Functions/%D0%9E%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B8%D0%BD%D0%B8%D0%B5_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%BE%D0%B2).

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

get area() {

return this.calcArea();

}

calcArea() {

return this.height \* this.width;

}

}

const square = new Rectangle(10, 10);

console.log(square.area); // 100

Статические методы

Ключевое слово [static](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static), определяет статический метод для класса. Статические методы вызываются без [инстанцирования](https://developer.mozilla.org/ru/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript#The_Object_.28Class_Instance.29) их класса, и не могут быть вызваны у экземпляров (*instance*) класса. Статические методы, часто используются для создания служебных функций для приложения.

class Point {

constructor(x, y) {

this.x = x;

this.y = y;

}

static distance(a, b) {

const dx = a.x - b.x;

const dy = a.y - b.y;

return Math.hypot(dx, dy);

}

}

const p1 = new Point(5, 5);

const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2));

Упаковка в прототипных и статических методах

Когда статический или прототипный метод вызывается без привязки к "this" объекта (или когда "this" является типом boolean, string, number, undefined, null), тогда "this" будет иметь значение "undefined" внутри вызываемой функции. Автоупаковка не будет произведена. Поведение будет таким же как если бы мы писали код в нестрогом режиме.

class Animal {

speak() {

return this;

}

static eat() {

return this;

}

}

let obj = new Animal();

obj.speak(); // Animal {}

let speak = obj.speak;

speak(); // undefined

Animal.eat() // class Animal

let eat = Animal.eat;

eat(); // undefined

Если мы напишем этот же код используя классы основанные на функциях, тогда произойдет автоупаковка основанная на значении "this", в течение которого функция была вызвана.

function Animal() { }

Animal.prototype.speak = function(){

return this;

}

Animal.eat = function() {

return this;

}

let obj = new Animal();

let speak = obj.speak;

speak(); // глобальный объект

let eat = Animal.eat;

eat(); // глобальный объект

**Наследование классов с помощью extends**

Ключевое слово [extends](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/extends) используется в *объявлениях классов* и *выражениях классов* для создания класса, дочернего относительно другого класса.

class Animal {

constructor(name) {

this.name = name;

}

speak() {

console.log(this.name + ' издает звук.');

}

}

class Dog extends Animal {

speak() {

console.log(this.name + ' лает.');

}

}

var d = new Dog('Митци');

d.speak();

Аналогичным образом можно расширять традиционные, основанные на функциях "классы":

function Animal (name) { this.name = name; } Animal.prototype.speak = function () { console.log(this.name + ' издает звук.'); } class Dog extends Animal { speak() { console.log(this.name + ' лает.'); } } var d = new Dog('Митци'); d.speak();

Обратите внимание, что классы не могут расширять обычные (non-constructible) объекты. Если вам необходимо создать наследование от обычного объекта, в качестве замены можно использовать [Object.setPrototypeOf()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf):

var Animal = { speak() { console.log(this.name + ' издает звук.'); } }; class Dog { constructor(name) { this.name = name; } speak() { console.log(this.name + ' лает.'); } } Object.setPrototypeOf(Dog.prototype, Animal); var d = new Dog('Митци'); d.speak();

### 28. Стандартные объекты JavaScript

## Стандартные объекты делятся на следующие группы:

-встроенные объекты;

-обобщенные объекты;

-объекты браузера;

-объекты, соответствующие тегам HTML.

Некоторые объекты, такие как Date и Math, встроены в интерпретатор JavaScript, а другие связаны с соответствующими тегами HTML.

## При определении объектов нужно учитывать следующее:

## **1.** В теги HTML, определяющие объект, необходимо включить атрибут name (Имя), задающий имя, по которому можно ссылаться на данный объект. Например, тег

<input type="text" name="nameField" size=20>

это уже не просто поле ввода текста, а объект, являющийся полем ввода.

## В языке JavaScript HTML-теги вместе с их атрибутами рассматриваются как объекты.

## **2.** Объекту в языке JavaScript присущи свойства, которые являются характеристиками объекта.

В браузере атрибуты тега HTML по умолчанию представляются как свойства языка JavaScript.

### **Стандартные объекты**

* [Array](https://javascript.ru/array)Массив пронумерованных элементов, также может служить стеком или очередью
* [Boolean](https://javascript.ru/Boolean)Объект для булевых значений
* [Date](https://javascript.ru/Date)Функции для работы с датой и временем
* [Error](https://javascript.ru/Error)объект для представления ошибок
* [EvalError](https://javascript.ru/EvalError)Ошибка при выполнении функции [eval](https://javascript.ru/eval)
* [Function](https://javascript.ru/Function)Каждая функция в яваскрипт является объектом класса Function.
* [Math](https://javascript.ru/Math)Встроенный объект, предоставляющий константы и методы для математических вычислений.
* [Number](https://javascript.ru/Number)Объект для работы с числами
* [Object](https://javascript.ru/Object)Базовый объект javascript
* [RangeError](https://javascript.ru/RangeError)Ошибка, когда число не лежит в нужном диапазоне
* [ReferenceError](https://javascript.ru/ReferenceError)Ошибку при ссылке на несуществующую переменную
* [RegExp](https://javascript.ru/RegExp)Позволяет работать с регулярными выражениями.
* [String](https://javascript.ru/String)Базовый объект для строк. Позволяет управлять текстовыми строками, форматировать их и выполнять поиск подстрок.
* [SyntaxError](https://javascript.ru/SyntaxError)Ошибка при интерпретации синтаксически неверного кода
* [TypeError](https://javascript.ru/TypeError)Ошибка в типе значения
* [URIError](https://javascript.ru/URIError)Ошибка при некорректном URI

### 29. Представление чисел

В современном JavaScript существует два типа чисел:

1. Обычные числа в JavaScript хранятся в 64-битном формате [IEEE-754](http://en.wikipedia.org/wiki/IEEE_754-1985), который также называют «числа с плавающей точкой двойной точности» (double precision floating point numbers). Это числа, которые мы будем использовать чаще всего. Мы поговорим о них в этой главе.
2. BigInt числа дают возможность работать с целыми числами произвольной длины. Они нужны достаточно редко и используются в случаях, когда необходимо работать со значениями более чем 253 или менее чем -253. Так как BigInt числа нужны достаточно редко, мы рассмотрим их в отдельной главе [BigInt](https://learn.javascript.ru/bigint).

### Безопасные числа

Скорее всего, вы хотите, чтобы значение числа всегда оставалось тем же самым, что вы ему задали. Но если вы используете в коде, к примеру, число 900719925474099164 оно превратится в 900719925474099200. Дело в том, что оно выходит за пределы так называемого безопасного диапазона.

Узнать эти пределы можно, обратившись к свойствам объекта Number [MIN\_SAFE\_INTEGER](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MIN_SAFE_INTEGER) и [MAX\_SAFE\_INTEGER](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/MAX_SAFE_INTEGER). А с помощью функции [Number.isSafeInteger](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/isSafeInteger) можно определить, является ли число безопасным.

Используйте метод isInteger объекта Number. Если число не имеет десятичной части, он возвращает true. Обратите внимание, что числа с пустой десятичной частью автоматически конвертируются в целые.

### Изменение количества знаков после запятой

Используйте метод Number.toFixed, который принимает количество знаков десятичной части, а возвращает строковое представление числа в нужном формате.

Также можно воспользоваться методом Number.toPrecision, которому нужно передать количество значащих цифр числа.

### Преобразование в экспоненциальную форму

Экспоненциальная форма известна также известна как научная нотация. Для преобразования существует специальный метод toExponential. Его единственный необязательный параметр – количество цифр после запятой.

### Глобальные функции для работы с числами и методы Number

Вы, возможно, знаете, что существуют глобальные функции вроде parseInt, но только что мы использовали Number.parseInt().

Дело в том, что последний стандарт JavaScript пытается модуляризировать глобальные сущности и обновляет некоторые существующие модули. Например, метод isNaN отличается от нового метода Number.isNaN.

### 30. Представление дат, класс Date

Объекты Date могут быть созданы только путём вызова функции Date в качестве конструктора:

обычный вызов функции (то есть, без использования оператора [new](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new)) вернёт строку вместо объекта

Date; в отличие от других объектных типов JavaScript, объекты Date не имеют литерального

синтаксиса.

Если функция Date вызывается в качестве конструктора с более чем одним аргументом, то указанные аргументы интерпретируются как локальное время. Если аргументы указывают время в UTC, используйте new Date([Date.UTC(...)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/UTC)) с теми же аргументами.

**value**

Целое значение, представляющее количество миллисекунд, прошедших с 1 января 1970 00:00:00 по UTC (эпохи Unix).

**dateString**

Строковое значение, представляющее дату. Строка должна быть в одном из форматов, распознаваемых методом [Date.parse()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/parse) (совместимые с IETF RFC 2822 временные метки [[на английском](http://tools.ietf.org/html/rfc2822#page-14), [на русском](http://rfc2.ru/5322.rfc/print#p3.3)], а также версия ISO8601 [[на английском](http://www.ecma-international.org/ecma-262/5.1/#sec-15.9.1.15), [на русском](http://es5.javascript.ru/x15.9.html#x15.9.1.15)]).

**year**

Целое значение, представляющее год. Значения с 0 по 99 отображаются на года с 1900 по 1999. Смотрите [пример ниже](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date#Example:_Two_digit_years_map_to_1900_-_1999).

**month**

Целое значение, представляющее ме

Необязательный параметр. Целое значение, представляющее день месяца.

**hour**

Необязательный параметр. Целое значение, представляющее часы дня.

**minute**

Необязательный параметр. Целое значение, представляющее минуты времени.

**second**

Необязательный параметр. Целое значение, представляющее секунды времени.

**millisecond**

Необязательный параметр. Целое значение, представляющее миллисекунды времени.

## **Описание**

* Если никаких аргументов передано не было, конструктор создаёт объект Date для текущих даты и времени, согласно системным настройкам.
* Если передано как минимум два аргумента, отсутствующие аргументы устанавливаются в стартовые значения - день месяца 1 и время полуночи.
* Дата в JavaScript измеряется в миллисекундах, прошедших с полуночи 1 января 1970 года по UTC. День содержит 86 400 000 миллисекунд. Диапазон дат объекта Date варьируется от -100 000 000 до 100 000 000 дней относительно 1 января 1970 года по UTC.
* Объект Date обеспечивает универсальное поведение на всех платформах. Значение времени может передаваться между системами для представления одинакового момента во времени и, если оно используется для создания локального объекта даты, будет отражать местный эквивалент времени.
* Объект Date поддерживает несколько методов для работы с UTC (всемирным координированным временем), наряду с методами работы с местным временем. UTC, также известное как среднее время по Гринвичу (GMT), ссылается на время, установленное Всемирным стандартом времени. Местное время — это время на компьютере, на котором выполняется JavaScript.
* Вызов объекта Date в качестве функции (то есть, без использования оператора [new](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/new)) вернёт строку, представляющую текущие дату и время.

### 31. Строковый тип, методы м свойства класса String.

Объект **String** (строковый объект) используется для хранения и обработки текстовой информации. Он незаменим, когда требуется, например, найти позицию вхождения одной строки в другую, вырезать из строки подстроку, разбить строку на отдельные символы и создать из них массив и т. д.

## **Методы String**

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| [charAt()](http://wm-school.ru/js/string_charat.php) | Возвращает символ строки с указанным индексом (позицией). |
| [charCodeAt()](http://wm-school.ru/js/string_charcodeat.php) | Возвращает числовое значение Unicode символа, индекс которого был передан методу в качестве аргумента. |
| [concat()](http://wm-school.ru/js/string_concat.php) | Возвращает строку, содержащую результат объединения двух и более предоставленных строк. |
| [fromCharCode()](http://wm-school.ru/js/string_fromcharcode.php) | Возвращает строку, созданную с помощью указанной последовательности значений символов Unicode. |
| [indexOf()](http://wm-school.ru/js/string_indexof.php) | Возвращает позицию первого символа первого вхождения указанной подстроки в строке. |
| [lastIndexOf()](http://wm-school.ru/js/string_lastindexof.php) | Возвращает позицию последнего найденного вхождения подстроки или -1, если подстрока не найдена. |
| [localeCompare()](http://wm-school.ru/js/string_localecompare.php) | Возвращает значение, указывающее, эквивалентны ли две строки в текущем языковом стандарте. |
| [match()](http://wm-school.ru/js/string_match.php) | Ищет строку, используя предоставленный шаблон регулярного выражения, и возвращает результат в виде массива. Если совпадений не найдено, метод возвращает значение null. |
| [replace()](http://wm-school.ru/js/string_replace.php) | Ищет строку для указанного значения или регулярного выражения и возвращает новую строку, где указанные значения будут заменены. Метод не изменяет строку, для которой он вызывается. |
| [search()](http://wm-school.ru/js/string_search.php) | Возвращает позицию первого соответствия указанной подстроки или регулярного выражения в строке. |
| [slice()](http://wm-school.ru/js/string_slice.php) | Позволяет извлечь подстроку из строки. Первый аргумент указывает индекс с которого нужно начать извлечение. Второй необязательный аргумент указывает позицию, на которой должно остановиться извлечение. Если второй аргумент не указан, то извлечено будет все с той позиции, которую указывает первый аргумент, и до конца строки. |
| [split()](http://wm-school.ru/js/string_split.php) | Разбивает строку на подстроки, возвращая массив подстрок. В качестве аргумента можно передать символ разделитель (например запятую), используемый для разбора строки на подстроки. |
| [substr()](http://wm-school.ru/js/string_substr.php) | Позволяет извлечь подстроку из строки. Первый аргумент указывает индекс с которого нужно начать извлечение. Второй аргумент указывает количество символов, которое нужно извлечь. |
| [substring()](http://wm-school.ru/js/string_substring.php) | Извлекает символы из строки между двух указанных индексов, если указан только один аргумент, то извлекаются символы от первого индекса и до конца строки. |
| [toLocaleLowerCase()](http://wm-school.ru/js/string_tolocalelowercase.php) | Преобразует символы строки в нижний регистр с учетом текущего языкового стандарта. |
| [toLocaleUpperCase()](http://wm-school.ru/js/string_tolocaleuppercase.php) | Преобразует символы строки в верхний регистр с учетом текущего языкового стандарта. |
| [toLowerCase()](http://wm-school.ru/js/string_tolowercase.php) | Конвертирует все символы строки в нижний регистр и возвращает измененную строку. |
| [toString()](http://wm-school.ru/js/string_tostring.php) | Возвращает строковое представление объекта. |
| [toUpperCase()](http://wm-school.ru/js/string_touppercase.php) | Конвертирует все символы строки в верхний регистр и возвращает измененную строку. |
| [trim()](http://wm-school.ru/js/string_trim.php) | Удаляет пробелы в начале и конце строки и возвращает измененную строку. |
| [valueOf()](http://wm-school.ru/js/string_valueof.php) | Возвращает примитивное значение объекта. |

### **32.** **Объект Math, его свойства и методы. (Виноградова)**

Объект **Math** является встроенным объектом, хранящим в своих свойствах и методах различные математические константы и функции. Объект **Math** не является функциональным объектом.

В отличие от других глобальных объектов, объект Math не является конструктором. Все свойства и методы объекта Math являются статическими. Вы ссылаетесь на константу π через Math.PI и вызываете функцию синуса через Math.sin(x), где x является аргументом метода. Константы в JavaScript определены с полной точностью действительных чисел.

***Свойства Math:***

[***Math.E***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/E)

Число Эйлера или Непера, основание натуральных логарифмов, приблизительно равное 2,718.

[***Math.LN2***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/LN2)

Натуральный логарифм из 2, приблизительно равен 0,693.

[***Math.LN10***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/LN10)

Натуральный логарифм из 10, приблизительно равен 2,303.

[***Math.LOG2E***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/LOG2E)

Двоичный логарифм из E, приблизительно равен 1,443.

[***Math.LOG10E***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/LOG10E)

Десятичный логарифм из E, приблизительно равен 0,434.

[***Math.P***I](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/PI)

Отношение длины окружности круга к его диаметру, приблизительно равно 3,14159.

[***Math.SQRT1\_2***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/SQRT1_2)

Квадратный корень из 1/2; или, что тоже самое, 1, делённая на квадратный корень из 2, приблизительно равен 0,707.

[***Math.SQRT2***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/SQRT2)

Квадратный корень из 2, приблизительно равен 1,414.

***Методы Math:***

[***Math.abs(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/abs) - Возвращает абсолютное значение числа.

[***Math.acos(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/acos) - Возвращает арккосинус числа.

[***Math.asin(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/asin) - Возвращает арксинус числа.

[***Math.atan(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/atan) - Возвращает арктангенс числа.

[***Math.ceil(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/ceil) - Возвращает значение числа, округлённое к большему целому.

[***Math.cos(x***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/cos)[***)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/ceil) - Возвращает косинус числа.

[***Math.exp(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/exp) - Возвращает E^x, где x — аргумент, а E — число Эйлера (2,718…), основание натурального логарифма.

[***Math.floor(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/floor) - Возвращает значение числа, округлённое к меньшему целому.

[***Math.log(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/log) - Возвращает натуральный логарифм числа (loge, также известен как ln).

[***Math.max([x[, y[, …]]])***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/max) - Возвращает наибольшее число из своих аргументов.

[***Math.min([x[, y[, …]]])***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/min) - Возвращает наименьшее число из своих аргументов.

[***Math.pow(x, y)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/pow) - Возвращает основание в степени экспоненты.

[***Math.random()***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/random) - Возвращает псевдослучайное число в диапазоне от 0 до 1.

[***Math.round(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/round) - Возвращает значение числа, округлённое до ближайшего целого.

[***Math.sin(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/sin) - Возвращает синус числа.

[***Math.sqrt(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/sqrt) - Возвращает положительный квадратный корень числа.

[***Math.tan(x)***](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/tan) - Возвращает тангенс числа.

### 

### 33. Объектная нотация JSON

JSON — текстовый формат обмена данными, основанный на JavaScript. Общий формат для представления значений и объектов. Его описание задокументировано в стандарте RFC 4627. Первоначально он был создан для JavaScript, но многие другие языки также имеют библиотеки, которые могут работать с ним. Таким образом, JSON легко использовать для обмена данными, когда клиент использует JavaScript, а сервер написан на Ruby/PHP/Java или любом другом языке.

JavaScript предоставляет методы:

JSON.stringify для преобразования объектов в JSON. (Метод берёт объект и преобразует его в строку

let student = {

name: 'John',

age: 30};

let json = JSON.stringify(student);

alert(typeof json); // мы получили строку!

alert(json);

/\* выведет объект в формате JSON:

{"name": "John",

"age": 30,} \*/

JSON.parse для преобразования JSON обратно в объект. (Обычно JSON используется для обмена данными с сервером. При получении с сервера данные всегда передаются в виде строки. Если обработать эти данные при помощи функции JSON.parse(), то они станут объектом JavaScript.)

var obj = JSON.parse('{ "name":"John", "age":30, "city":"New York"}');

### 34. Значения Infinity, NaN, undefined, null. Оператор typeof. - Диана Ахметова

Infinity является свойством *глобального объекта*, то есть, это переменная в глобальной области видимости.

Значение Infinity (положительная бесконечность) больше любого другого числа, включая саму положительную бесконечность. Это значение ведёт себя как математическая бесконечность; например, любое положительное число, умноженное на Infinity даёт Infinity, а любое число, делённое на Infinity даёт 0.

Глобальное свойство **NaN** является значением, представляющим не-число (Not-A-Number).

В программах NaN используется довольно редко. Это возвращаемое значение в ситуациях, когда математические ([Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math)) функции не срабатывают должным образом (например, при вызове Math.sqrt(-1)) или когда функция, пытающаяся считать число из строки, терпит неудачу по причине того, что в строке не число (parseInt('blabla')).

undefined является свойством *глобального объекта*, то есть, это переменная в глобальной области видимости. Начальным значением undefined является примитивное значение undefined.

Переменная, не имеющая присвоенного значения, обладает типом undefined. Также undefined возвращают метод или инструкция, если переменная, участвующая в вычислениях, не имеет присвоенного значения. Функция возвращает undefined, если она не [возвращает](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/return) какого-либо значения.

Значение null представляет отсутствие какого-либо объектного значения. В JavaScript, null является [примитивом](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/Primitive), и в контексте логических операций, рассматривается как [ложное (falsy)](https://developer.mozilla.org/ru/docs/%D0%A1%D0%BB%D0%BE%D0%B2%D0%B0%D1%80%D1%8C/Falsy).

null является определенным значением отсутствия объекта, тогда как [undefined](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/undefined) обозначает неопределенность.

Оператор typeof возвращает тип аргумента. Он позволяет выяснить, какой тип находится в x, возвращая его в виде строки.

У него есть два синтаксиса: со скобками и без:

1. Синтаксис оператора: typeof x.
2. Синтаксис функции: typeof(x).

**Результатом typeof является строка, содержащая тип, например:**

typeof undefined // "undefined"   
typeof 0 // "number"  
typeof true // "boolean"  
typeof "foo" // "string" и т.д.

### 35. Регулярные выражения- Бутова Катя

Регулярные выражения - это шаблоны используемые для сопоставления последовательностей символов в строках. В JavaScript, регулярные выражения также являются объектами. Эти шаблоны используются в методах exec и test объекта RegExp, а также match, replace, search, и split объекта String.

Создание регулярного выражения

Регулярное выражение можно создать двумя способами:

Используя литерал регулярного выражения, например:

var re = /ab+c/;

Литералы регулярных выражений вызывают предварительную компиляцию регулярного выражения при анализе скрипта. Если ваше регулярное выражение постоянно, то пользуйтесь им чтобы увеличить производительность.

Вызывая функцию конструктор объекта RegExp, например:

var re = new RegExp("ab+c");

Использование конструктора влечет за собой компиляцию регулярного выражения во время исполнения скрипта. Используйте данный способ, если знаете, что выражение будет изменяться или не знаете шаблон заранее. Например вы получаете его из стороннего источника, при пользовательском вводе.

Написание шаблона регулярного выражения

Шаблон регулярного выражения состоит из обычных символов, например /abc/, или комбинаций обычных и специальных символов, например /ab\*c/ или /Chapter (\d+)\.\d\*/. Последний пример включает в себя скобки, которые используются как "запоминающий механизм". Соответствие этой части шаблона запоминается для дальнейшего использования как описано в Использование совпадений подстрок заключённых в скобки.

Использование простых шаблонов

Простые шаблоны используются для нахождения прямого соответствия в тексте. Например, шаблон /abc/ соответствует комбинации символов в строке только когда символы 'abc' встречаются вместе и в том же порядке. Такое сопоставление произойдет в строке "Hi, do you know your abc's?" и "The latest airplane designs evolved from slabcraft." В обоих случаях сопоставление произойдет с подстрокой 'abc'. Сопоставление не произойдет в строке "Grab crab", потому что она не содержит подстроку 'abc'.

Использование специальных символов

В случае когда поиск соответствия требует чего-то большего, чем прямое сопоставление, например нахождение последовательности символов 'b' или нахождение пробела, шаблон включает в себя специальные символы. Например, шаблон /ab\*c/ соответствует любой комбинации символов, в которой за 'a' следует ноль или более символов 'b' (\* означает ноль или более вхождений предыдущего символа), за которыми сразу же следует символ 'c'. В строке "cbbabbbbcdebc," этому шаблону сопоставляется подстрока 'abbbbc'.

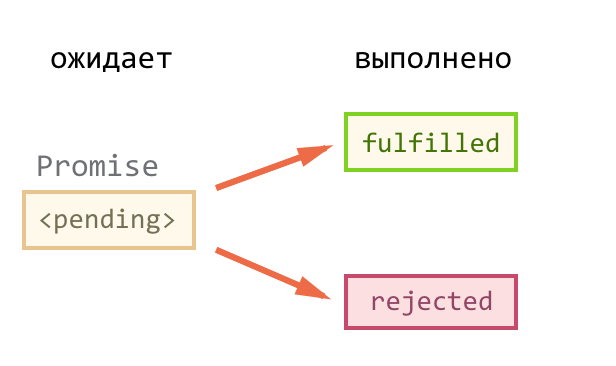
### 36. Объект Promise+ - Даша Сопоцько

Интерфейс **Promise** представляет собой обертку для значения, неизвестного на момент создания промиса. Он позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными: вместо конечного результата асинхронного метода возвращается *обещание* (промис)получить результат в некоторый момент в будущем.

Promise может находиться в трёх состояниях:

* *ожидание (pending)*: начальное состояние, не исполнен и не отклонен.
* *исполнено (fulfilled)*: операция завершена успешно.
* *отклонено (rejected)*: операция завершена с ошибкой.

При создании промис находится в *ожидании (pending)*, а затем может стать *исполненным* (*fulfilled)*, вернув полученный результат (значение), или *отклоненным* (*rejected),* вернув причину отказа. В любом из этих случаев вызывается обработчик, прикрепленный к промису методом then.



**Свойства**

**Promise.length**

Значение свойства всегда равно 1 (количество аргументов конструктора).

[**Promise.prototype**](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/prototype)

Представляет прототип для конструктора Promise.

**Методы**

[Promise.all(iterable)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/all)

Ожидает исполнения всех промисов или отклонения любого из них.

Возвращает промис, который исполнится после исполнения всех промисов в iterable. В случае, если любой из промисов будет отклонен, Promise.all будет также отклонен.

[Promise.allSettled(iterable)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled)

Ожидает завершения всех полученных промисов (как исполнения так и отклонения).

Возвращает промис, который исполняется когда все полученные промисы завершены (исполнены или отклонены), содержащий массив результатов исполнения полученных промисов.

[Promise.race(iterable)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/race)

Ожидает исполнения или отклонения любого из полученных промисов.

Возвращает промис, который будет исполнен или отклонен с результатом исполнения первого исполненного или отклонённого промиса из .iterable.

[Promise.reject(reason)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject)

Возвращает промис, отклонённый из-за reason.

[Promise.resolve(value)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve)

Возвращает промис, исполненный с результатом value.

**Создание promise**

Объект Promise создается при помощи ключевого слова new и своего конструктора. Конструктор Promise принимает в качестве аргумента функцию, называемую "исполнитель" (*executor function*). Эта функция должна принимать две функции-коллбэка в качестве параметров. Первый из них (resolve) вызывается, когда асинхронная операция завершилась успешно и вернула результат своего исполнения в виде значения. Второй коллбэк (reject) вызывается, когда операция не удалась, и возвращает значение, указывающее на причину неудачи, чаще всего объект ошибки.