# [ОСНОВЫ JS](https://learn.javascript.ru/javascript-specials)

## Привет, мир!

* Для добавления кода JavaScript на страницу используется тег <script>
* Атрибуты type и language необязательны.
* Скрипт во внешнем файле можно вставить с помощью <script src="path/to/script.js"></script>.

## Структура кода

* Инструкции – это синтаксические конструкции и команды, которые выполняют действия.
* В большинстве случаев точку с запятой можно не ставить, если есть переход на новую строку.
* (JavaScript не вставляет точку с запятой перед квадратными скобками [...])

## Переменные

Мы можем объявить переменные для хранения данных с помощью ключевых слов var, let или const.

* let – это современный способ объявления.
* var – это устаревший способ объявления.
* const – похоже на let, но значение переменной не может изменяться.

Переменные должны быть названы таким образом, чтобы мы могли легко понять, что у них внутри.

## Типы данных

В JavaScript есть 8 основных типов данных.

Семь из них называют «примитивными» типами данных:

* number для любых чисел: целочисленных или чисел с плавающей точкой; целочисленные значения ограничены диапазоном ±(253-1).
* bigint для целых чисел произвольной длины.
* string для строк. Строка может содержать ноль или больше символов, нет отдельного символьного типа.
* boolean для true/false.
* null для неизвестных значений – отдельный тип, имеющий одно значение null.
* undefined для неприсвоенных значений – отдельный тип, имеющий одно значение undefined.
* symbol для уникальных идентификаторов.

И один не является «примитивным» и стоит особняком:

* object для более сложных структур данных.

Оператор typeof позволяет нам увидеть, какой тип данных сохранён в переменной.

* Имеет две формы: typeof x или typeof(x).
* Возвращает строку с именем типа. Например, "string".
* Для null возвращается "object" – это ошибка в языке, на самом деле это не объект.

В следующих главах мы сконцентрируемся на примитивных значениях, а когда познакомимся с ними, перейдём к объектам.

## Преобразование типов

Существует 3 наиболее широко используемых преобразования: строковое, численное и логическое.

* Строковое – Происходит, когда нам нужно что-то вывести. Может быть вызвано с помощью String(value). Для примитивных значений работает очевидным образом.
* Численное – Происходит в математических операциях. Может быть вызвано с помощью Number(value).

Преобразование подчиняется правилам:

|  |  |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| string | Пробельные символы (пробелы, знаки табуляции \t, знаки новой строки \n и т. п.) по краям обрезаются. Далее, если остаётся пустая строка, то получаем 0, иначе из непустой строки «считывается» число. При ошибке результат NaN. |

* Логическое – Происходит в логических операциях. Может быть вызвано с помощью Boolean(value).

Подчиняется правилам:

|  |  |
| --- | --- |
| 0, null, undefined, NaN, "" | false |
| любое другое значение | true |

Большую часть из этих правил легко понять и запомнить. Особые случаи, в которых часто допускаются ошибки:

* undefined при численном преобразовании становится NaN, не 0.
* "0" и строки из одних пробелов типа " " при логическом преобразовании всегда true.

## Операторы сравнения

* Операторы сравнения возвращают значения логического типа.
* Строки сравниваются посимвольно в лексикографическом порядке.
* Значения разных типов при сравнении приводятся к числу. Исключением является сравнение с помощью операторов строгого равенства/неравенства.
* Значения null и undefined равны == друг другу и не равны любому другому значению.
* Будьте осторожны при использовании операторов сравнений вроде > и < с переменными, которые могут принимать значения null/undefined. Хорошей идеей будет сделать отдельную проверку на null/undefined.

## Логические операторы

Оператор **||** выполняет следующие действия:

* Вычисляет операнды слева направо.
* Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.
* Если все операнды являются ложными (false), возвращает последний из них.
* Значение возвращается в исходном виде, без преобразования.

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

*Оператор* ***||=*** *принимает два операнда и выполняет следующие действия:*

* *Вычисляет операнды слева направо.*
* *Конвертирует a в логическое значение.*
* *Если a false, присваивает a значение b.*

Оператор **&&** выполняет следующие действия:

* Вычисляет операнды слева направо.
* Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.
* Если все операнды были истинными, возвращается последний.

Другими словами, И && возвращает ***первое ложное значение***. Или последнее, если ничего не найдено.

*Оператор* ***&&=*** *принимает два операнда и выполняет следующие действия:*

* *Вычисляет операнды слева направо.*
* *Конвертирует a в логическое значение.*
* *Если a true, присваивает a значение b.*

Оператор **!** принимает один аргумент и выполняет следующие действия:

* Сначала приводит аргумент к логическому типу true/false.
* Затем возвращает противоположное значение.

Оператор нулевого слияния **??**. (*??=*)

Результат выражения a ?? b будет следующим:

* если a определено, то a,
* если a не определено, то b.

Иначе говоря, оператор ?? возвращает первый аргумент, если он не null/undefined, иначе второй.

## Циклы while и for

Мы рассмотрели 3 вида циклов:

* while – Проверяет условие перед каждой итерацией.
* do..while – Проверяет условие после каждой итерации.
* for (;;) – Проверяет условие перед каждой итерацией, есть возможность задать дополнительные настройки.

Чтобы организовать бесконечный цикл, используют конструкцию while (true). При этом он, как и любой другой цикл, может быть прерван директивой break.

Если на данной итерации цикла делать больше ничего не надо, но полностью прекращать цикл не следует – используют директиву continue.

Обе этих директивы поддерживают *метки*, которые ставятся перед циклом. Метки – единственный способ для break/continue выйти за пределы текущего цикла, повлиять на выполнение внешнего.

Заметим, что метки не позволяют прыгнуть в произвольное место кода, в JavaScript нет такой возможности.

## Функции

Объявление функции имеет вид:

*function имя(параметры, через, запятую) {*

*/\* тело, код функции \*/*

*}*

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

Для того, чтобы сделать код более чистым и понятным, рекомендуется использовать локальные переменные и параметры функций, не пользоваться внешними переменными.

Функция, которая получает параметры, работает с ними и затем возвращает результат, гораздо понятнее функции, вызываемой без параметров, но изменяющей внешние переменные, что чревато побочными эффектами.

Именование функций:

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

## Function Expression

* Функции – это значения. Они могут быть присвоены, скопированы или объявлены в любом месте кода.
* Если функция объявлена как отдельная инструкция в основном потоке кода, то это “Function Declaration”.
* Если функция была создана как часть выражения, то это “Function Expression”.
* Function Declaration обрабатываются перед выполнением блока кода. Они видны во всём блоке.
* Функции, объявленные при помощи Function Expression, создаются только когда поток выполнения достигает их.

В большинстве случаев, когда нам нужно объявить функцию, Function Declaration предпочтительнее, т.к функция будет видна до своего объявления в коде. Это даёт нам больше гибкости в организации кода, и, как правило, делает его более читабельным.

Исходя из этого, мы должны использовать Function Expression только тогда, когда Function Declaration не подходит для нашей задачи.

## Стрелочные функции, основы

Стрелочные функции очень удобны для простых действий, особенно для однострочных.

Они бывают двух типов:

1. Без фигурных скобок: (...args) => expression – правая сторона выражения: функция вычисляет его и возвращает результат. Скобки можно не ставить, если аргумент только один: n => n \* 2.
2. С фигурными скобками: (...args) => { body } – скобки позволяют нам писать несколько инструкций внутри функции, но при этом необходимо явно вызывать return, чтобы вернуть значение.

# Объекты

## Объекты

Объекты – это ассоциативные массивы с рядом дополнительных возможностей.

Они хранят свойства (пары ключ-значение), где:

* Ключи свойств должны быть строками или символами (обычно строками).
* Значения могут быть любого типа.

Чтобы получить доступ к свойству, мы можем использовать:

* Запись через точку: obj.property.
* Квадратные скобки obj["property"]. Квадратные скобки позволяют взять ключ из переменной, например, obj[varWithKey].

Дополнительные операторы:

* Удаление свойства: delete obj.prop.
* Проверка существования свойства: "key" in obj.
* Перебор свойств объекта: цикл for for (let key in obj).

То, что мы изучали в этой главе, называется «простым объектом» («plain object») или просто Object.

В JavaScript есть много других типов объектов:

* Array для хранения упорядоченных коллекций данных,
* Date для хранения информации о дате и времени,
* Error для хранения информации об ошибке.
* … и так далее.

У них есть свои особенности, которые мы изучим позже. Иногда люди говорят что-то вроде «тип данных Array» или «тип данных Date», но формально они не являются отдельными типами, а относятся к типу данных Object. Они лишь расширяют его различными способами.

## Копирование объектов и ссылки

Объекты присваиваются и копируются по ссылке. Другими словами, переменная хранит не «значение объекта», а «ссылку» (адрес в памяти) на это значение. Таким образом, копирование такой переменной или передача её в качестве аргумента функции копирует эту ссылку, а не сам объект.

Все операции с использованием скопированных ссылок (например, добавление/удаление свойств) выполняются с одним и тем же объектом.

Чтобы создать «реальную копию» (клон), мы можем использовать Object.assign для так называемой «поверхностной копии» (вложенные объекты копируются по ссылке) или функцию «глубокого клонирования», такую как [\_.cloneDeep(obj)](https://lodash.com/docs#cloneDeep).

## Методы объекта, "this"

* Функции, которые находятся в свойствах объекта, называются «методами».
* Методы позволяют объектам «действовать»: object.doSomething().
* Методы могут ссылаться на объект через this.

Значение this определяется во время исполнения кода.

* При объявлении любой функции в ней можно использовать this, но этот this не имеет значения до тех пор, пока функция не будет вызвана.
* Функция может быть скопирована между объектами (из одного объекта в другой).
* Когда функция вызывается синтаксисом «метода» – object.method(), значением this во время вызова является object.

Также ещё раз заметим, что стрелочные функции являются особенными – у них нет this. Когда внутри стрелочной функции обращаются к this, то его значение берётся извне.

## Конструктор, оператор "new"

* Функции-конструкторы или просто конструкторы, являются обычными функциями, но существует общепринятое соглашение именовать их с заглавной буквы.
* Функции-конструкторы следует вызывать только с помощью new. Такой вызов подразумевает создание пустого this в начале и возврат заполненного в конце.

Мы можем использовать конструкторы для создания множества похожих объектов.

JavaScript предоставляет функции-конструкторы для множества встроенных объектов языка: таких как Date, Set, и других

## Опциональная цепочка '?.'

Синтаксис опциональной цепочки ?. имеет три формы:

1. obj?.prop – возвращает obj.prop если obj существует, в противном случае undefined.
2. obj?.[prop] – возвращает obj[prop] если obj существует, в противном случае undefined.
3. obj.method?.() – вызывает obj.method(), если obj.method существует, в противном случае возвращает undefined.

Как мы видим, все они просты и понятны в использовании. ?. проверяет левую часть на null/undefined и позволяет продолжить вычисление, если это не так.

Цепочка ?. позволяет безопасно получать доступ к вложенным свойствам.

Тем не менее, мы должны использовать ?. осторожно, только там, где по логике кода допустимо, что левая часть не существует. Чтобы он не скрывал от нас ошибки программирования, если они возникнут.

## Тип данных Symbol

Символ (symbol) – примитивный тип данных, использующийся для создания уникальных идентификаторов.

Символы создаются вызовом функции Symbol(), в которую можно передать описание (имя) символа.

Даже если символы имеют одно и то же имя, это – разные символы. Если мы хотим, чтобы одноимённые символы были равны, то следует использовать глобальный реестр: вызов Symbol.for(key) возвращает (или создаёт) глобальный символ с key в качестве имени. Многократные вызовы команды Symbol.for с одним и тем же аргументом возвращают один и тот же символ.

Символы имеют два основных варианта использования:

1. «Скрытые» свойства объектов.

Если мы хотим добавить свойство в объект, который «принадлежит» другому скрипту или библиотеке, мы можем создать символ и использовать его в качестве ключа. Символьное свойство не появится в for..in, так что оно не будет нечаянно обработано вместе с другими. Также оно не будет модифицировано прямым обращением, так как другой скрипт не знает о нашем символе. Таким образом, свойство будет защищено от случайной перезаписи или использования.

Так что, используя символьные свойства, мы можем спрятать что-то нужное нам, но что другие видеть не должны.

1. Существует множество системных символов, используемых внутри JavaScript, доступных как Symbol.\*. Мы можем использовать их, чтобы изменять встроенное поведение ряда объектов. Например, в дальнейших главах мы будем использовать Symbol.iterator для [итераторов](https://learn.javascript.ru/iterable), Symbol.toPrimitive для настройки [преобразования объектов в примитивы](https://learn.javascript.ru/object-toprimitive) и так далее.

Технически символы скрыты не на 100%. Существует встроенный метод [Object.getOwnPropertySymbols(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertySymbols) – с его помощью можно получить все свойства объекта с ключами-символами. Также существует метод [Reflect.ownKeys(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Reflect/ownKeys), который возвращает *все* ключи объекта, включая символьные. Так что они не совсем спрятаны. Но большинство библиотек, встроенных методов и синтаксических конструкций не используют эти методы.

## Преобразование объектов в примитивы

Преобразование объекта в примитив вызывается автоматически многими встроенными функциями и операторами, которые ожидают примитив в качестве значения.

Существует всего 3 типа (хинта) для этого:

* "string" (для alert и других операций, которым нужна строка)
* "number" (для математических операций)
* "default" (для некоторых других операторов, обычно объекты реализуют его как "number")

Спецификация явно описывает для каждого оператора, какой ему следует использовать хинт.

Алгоритм преобразования таков:

1. Сначала вызывается метод obj[Symbol.toPrimitive](hint), если он существует,
2. В случае, если хинт равен "string"
   * происходит попытка вызвать obj.toString() и obj.valueOf(), смотря что есть.
3. В случае, если хинт равен "number" или "default"
   * происходит попытка вызвать obj.valueOf() и obj.toString(), смотря что есть.

Все эти методы должны возвращать примитив (если определены).

На практике часто бывает достаточно реализовать только obj.toString() в качестве универсального метода для преобразований к строке, который должен возвращать удобочитаемое представление объекта для целей логирования или отладки.

# Типы данных

## Методы примитивов

* Все примитивы, кроме null и undefined, предоставляют множество полезных методов. Мы познакомимся с ними поближе в следующих главах.
* Формально эти методы работают с помощью временных объектов, но движки JavaScript внутренне очень хорошо оптимизируют этот процесс, так что их вызов не требует много ресурсов.

## Числа

Чтобы писать числа с большим количеством нулей:

* Используйте краткую форму записи чисел – "e", с указанным количеством нулей. Например: 123e6 это 123 с 6-ю нулями 123000000.
* Отрицательное число после "e" приводит к делению числа на 1 с указанным количеством нулей. Например: 123e-6 это 0.000123 (123 миллионных).

Для других систем счисления:

* Можно записывать числа сразу в шестнадцатеричной (0x), восьмеричной (0o) и бинарной (0b) системах счисления
* parseInt(str, base) преобразует строку в целое число в соответствии с указанной системой счисления: 2 ≤ base ≤ 36.
* num.toString(base) представляет число в строковом виде в указанной системе счисления base.

Для проверки на NaN и Infinity:

* isNaN(value) преобразует аргумент в число и проверяет, является ли оно NaN
* Number.isNaN(value) проверяет, является ли аргумент числом, и если да, то проверяет, является ли оно NaN
* isFinite(value) преобразует аргумент в число и проверяет, что оно не является NaN/Infinity/-Infinity
* Number.isFinite(value) проверяет, является ли аргумент числом, и если да, то проверяет, что оно не является NaN/Infinity/-Infinity

Для преобразования значений типа 12pt и 100px в число:

* Используйте parseInt/parseFloat для «мягкого» преобразования строки в число, данные функции по порядку считывают число из строки до тех пор пока не возникнет ошибка.

Для дробей:

* Используйте округления Math.floor, Math.ceil, Math.trunc, Math.round или num.toFixed(precision).
* Помните, что при работе с дробями происходит потеря точности.

Ещё больше математических функций:

* Документация по объекту [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math). Библиотека маленькая, но содержит всё самое важное.

## Строки

* Есть три типа кавычек. Строки, использующие обратные кавычки, могут занимать более одной строки в коде и включать выражения ${…}.
* Строки в JavaScript кодируются в UTF-16.
* Есть специальные символы, такие как разрыв строки \n.
* Для получения символа используйте [] или метод at.
* Для получения подстроки используйте slice или substring.
* Для того, чтобы перевести строку в нижний или верхний регистр, используйте toLowerCase/toUpperCase.
* Для поиска подстроки используйте indexOf или includes/startsWith/endsWith, когда надо только проверить, есть ли вхождение.
* Чтобы сравнить строки с учётом правил языка, используйте localeCompare.

Строки также имеют ещё кое-какие полезные методы:

* str.trim() — убирает пробелы в начале и конце строки.
* str.repeat(n) — повторяет строку n раз.
* …и другие, которые вы можете найти в [справочнике](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String).

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



## Массивы

Массив – это особый тип объекта, предназначенный для работы с упорядоченным набором элементов.

Объявление:

// квадратные скобки (обычно)

let arr = [item1, item2...];

// new Array (очень редко)

let arr = new Array(item1, item2...);

Вызов new Array(number) создаёт массив с заданной длиной, но без элементов.

* Свойство length отражает длину массива или, если точнее, его последний цифровой индекс плюс один. Длина корректируется автоматически методами массива.
* Если мы уменьшаем length вручную, массив укорачивается.

Получение элементов:

* Мы можем получить элемент по его индексу, например arr[0].
* Также мы можем использовать метод at(i) для получения элементов с отрицательным индексом, для отрицательных значений i, он отступает от конца массива. В остальном он работает так же, как arr[i], если i >= 0.

Мы можем использовать массив как двустороннюю очередь, используя следующие операции:

* push(...items)добавляет items в конец массива.
* pop() удаляет элемент в конце массива и возвращает его.
* shift() удаляет элемент в начале массива и возвращает его.
* unshift(...items) добавляет items в начало массива.

Чтобы пройтись по элементам массива:

* for (let i=0; i<arr.length; i++) – работает быстрее всего, совместим со старыми браузерами.
* for (let item of arr) – современный синтаксис только для значений элементов (к индексам нет доступа).
* for (let i in arr) – никогда не используйте для массивов!

## Методы массивов

* Для добавления/удаления элементов:
  + push(...items) – добавляет элементы в конец,
  + pop() – извлекает элемент с конца,
  + shift() – извлекает элемент с начала,
  + unshift(...items) – добавляет элементы в начало.
  + splice(pos, deleteCount, ...items) – начиная с индекса pos удаляет deleteCount элементов и вставляет items.
  + slice(start, end) – создаёт новый массив, копируя в него элементы с индекса start до end (не включая end).
  + concat(...items) – возвращает новый массив: копирует все члены текущего массива и добавляет к нему items. Если какой-то из items является массивом, тогда берутся его элементы.
* Для поиска среди элементов:
  + indexOf/lastIndexOf(item, pos) – ищет item, начиная с позиции pos, и возвращает его индекс или -1, если ничего не найдено.
  + includes(value) – возвращает true, если в массиве имеется элемент value, в противном случае false.
  + find/filter(func) – фильтрует элементы через функцию и отдаёт первое/все значения, при прохождении которых через функцию возвращается true.
  + findIndex похож на find, но возвращает индекс вместо значения.
* Для перебора элементов:
  + forEach(func) – вызывает func для каждого элемента. Ничего не возвращает.
* Для преобразования массива:
  + map(func) – создаёт новый массив из результатов вызова func для каждого элемента.
  + sort(func) – сортирует массив «на месте», а потом возвращает его.
  + reverse() – «на месте» меняет порядок следования элементов на противоположный и возвращает изменённый массив.
  + split/join – преобразует строку в массив и обратно.
  + reduce/reduceRight(func, initial) – вычисляет одно значение на основе всего массива, вызывая func для каждого элемента и передавая промежуточный результат между вызовами.
* Дополнительно:
  + Array.isArray(arr) проверяет, является ли arr массивом.

Пожалуйста, обратите внимание, что методы push, pop, shift, unshift, sort, reverse и splice изменяют исходный массив.

Эти методы – самые используемые, их достаточно в 99% случаев. Но существуют и другие:

* [arr.some(fn)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/some)/[arr.every(fn)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/every) проверяет массив.

Функция fn вызывается для каждого элемента массива аналогично map. Если какие-либо/все результаты вызовов являются true, то метод возвращает true, иначе false.

Эти методы ведут себя примерно так же, как операторы || и &&: если fn возвращает истинное значение, arr.some() немедленно возвращает true и останавливает перебор остальных элементов; если fn возвращает ложное значение, arr.every() немедленно возвращает false и также прекращает перебор остальных элементов.

Мы можем использовать every для сравнения массивов:

function arraysEqual(arr1, arr2) {

return arr1.length === arr2.length && arr1.every((value, index) => value === arr2[index]);

}

alert( arraysEqual([1, 2], [1, 2])); // true

* [arr.fill(value, start, end)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/fill) – заполняет массив повторяющимися value, начиная с индекса start до end.
* [arr.copyWithin(target, start, end)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/copyWithin) – копирует свои элементы, начиная с позиции start и заканчивая end, в *себя*, на позицию target (перезаписывая существующие).
* [arr.flat(depth)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/flat)/[arr.flatMap(fn)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/flatMap) создаёт новый плоский массив из многомерного массива.

Полный список есть в [справочнике MDN](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array).