# +- JS

[**Что может JavaScript в браузере?**](https://learn.javascript.ru/intro#chto-mozhet-javascript-v-brauzere)

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»).

## [Чего НЕ может JavaScript в браузере?](https://learn.javascript.ru/intro" \l "chego-ne-mozhet-javascript-v-brauzere)

Возможности JavaScript в браузере ограничены ради безопасности пользователя. Цель заключается в предотвращении доступа недобросовестной веб-страницы к личной информации или нанесения ущерба данным пользователя.

Примеры таких ограничений включают в себя:

* JavaScript на веб-странице не может читать/записывать произвольные файлы на жёстком диске, копировать их или запускать программы. Он не имеет прямого доступа к системным функциям ОС.

Современные браузеры позволяют ему работать с файлами, но с ограниченным доступом, и предоставляют его, только если пользователь выполняет определённые действия, такие как «перетаскивание» файла в окно браузера или его выбор с помощью тега <input>.

Существуют способы взаимодействия с камерой/микрофоном и другими устройствами, но они требуют явного разрешения пользователя. Таким образом, страница с поддержкой JavaScript не может незаметно включить веб-камеру, наблюдать за происходящим и отправлять информацию в [ФСБ](https://ru.wikipedia.org/wiki/%D0%A4%D0%B5%D0%B4%D0%B5%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D0%BB%D1%83%D0%B6%D0%B1%D0%B0_%D0%B1%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D1%81%D1%82%D0%B8_%D0%A0%D0%BE%D1%81%D1%81%D0%B8%D0%B9%D1%81%D0%BA%D0%BE%D0%B9_%D0%A4%D0%B5%D0%B4%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8).

* Различные окна/вкладки не знают друг о друге. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае JavaScript с одной страницы не имеет доступа к другой, если они пришли с разных сайтов (с другого домена, протокола или порта).

Это называется «Политика одинакового источника» (Same Origin Policy). Чтобы обойти это ограничение, обе страницы должны согласиться с этим и содержать JavaScript-код, который специальным образом обменивается данными.

Это ограничение необходимо, опять же, для безопасности пользователя. Страница https://anysite.com, которую открыл пользователь, не должна иметь доступ к другой вкладке браузера с URL https://gmail.com и воровать информацию оттуда.

* JavaScript может легко взаимодействовать с сервером, с которого пришла текущая страница. Но его способность получать данные с других сайтов/доменов ограничена. Хотя это возможно в принципе, для чего требуется явное согласие (выраженное в заголовках HTTP) с удалённой стороной. Опять же, это ограничение безопасности.

## Скрипты в HTML

* 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>.
* Нельзя использовать и чтение скрипта из файла, и скрипт в теге <script src="path/to/script.j"> SOME SCRIPT</script>.

# Строгий режим — "use strict"

На протяжении долгого времени JavaScript развивался без проблем с обратной совместимостью. Новые функции добавлялись в язык, в то время как старая функциональность не менялась.

Преимуществом данного подхода было то, что существующий код продолжал работать. А недостатком – что любая ошибка или несовершенное решение, принятое создателями JavaScript, застревали в языке навсегда.

Так было до 2009 года, когда появился ECMAScript 5 (ES5). Он добавил новые возможности в язык и изменил некоторые из существующих. Чтобы устаревший код работал, как и раньше, по умолчанию подобные изменения не применяются. Поэтому нам нужно явно их активировать с помощью специальной директивы: "use strict".

## [«use strict»](https://learn.javascript.ru/strict-mode" \l "use-strict)

Директива выглядит как строка: "use strict" или 'use strict'. Когда она находится в начале скрипта, весь сценарий работает в «современном» режиме.

# Переменные

JavaScript-приложению обычно нужно работать с информацией. Например:

1. Интернет-магазин – информация может включать продаваемые товары и корзину покупок.
2. Чат – информация может включать пользователей, сообщения и многое другое.

Переменные используются для хранения этой информации.

## [Переменная](https://learn.javascript.ru/variables" \l "peremennaya)

[Переменная](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) – это «именованное хранилище» для данных. Мы можем использовать переменные для хранения товаров, посетителей и других данных.

Для создания переменной в JavaScript используйте ключевое слово let.

Приведённая ниже инструкция создаёт (другими словами: объявляет или определяет) переменную с именем «message»:

let message;

**Регистр имеет значение**

Переменные с именами apple и AppLE – это две разные переменные.

**[Константы](https://learn.javascript.ru/variables" \l "konstanty)**

Чтобы объявить константную, то есть, неизменяемую переменную, используйте const вместо let:

const myBirthday = '18.04.1982';

Переменные, объявленные с помощью const, называются «константами». Их нельзя изменить. Попытка сделать это приведёт к ошибке:

Имена констант : 1) известны до начала выполнения – ЗАГЛАВНЫЕ

2) константа заполняется во время выполнения – обычными.

## [Строка](https://learn.javascript.ru/types" \l "stroka)

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

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript.

Обратные же кавычки имеют расширенную функциональность. Они позволяют нам встраивать выражения в строку, заключая их в ${…}. Например:

# Конструкторы, создание объектов через "new"

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество однотипных объектов, таких как пользователи, элементы меню и т.д.

Это можно сделать при помощи функции-конструктора и оператора "new".

## [Функция-конструктор](https://learn.javascript.ru/constructor-new" \l "funktsiya-konstruktor)

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Например:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

alert(user.isAdmin); // false

Когда функция вызывается как new User(...), происходит следующее:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

let name = "Иван";

// Вставим переменную

alert( `Привет, ${name}!` ); // Привет, Иван!

// Вставим выражение

alert( `результат: ${1 + 2}` ); // результат: 3

Выражение внутри ${…} вычисляется, и его результат становится частью строки. Мы можем положить туда всё, что угодно: переменную name, или выражение 1 + 2, или что-то более сложное.

# Типы данных

Значение в JavaScript всегда относится к данным определённого типа. Например, это может быть строка или число.

Есть восемь основных типов данных в JavaScript. В этой главе мы рассмотрим их в общем, а в следующих главах поговорим подробнее о каждом.

Переменная в JavaScript может содержать любые данные. В один момент там может быть строка, а в другой – число:

// Не будет ошибкой

let message = "hello";

message = 123456;

Языки программирования, в которых такое возможно, называются «динамически типизированными». Это значит, что типы данных есть, но переменные не привязаны ни к одному из них.

## [Число](https://learn.javascript.ru/types" \l "chislo)

Кроме обычных чисел, существуют так называемые «специальные числовые значения», которые относятся к этому типу данных: Infinity, -Infinity и NaN.

Если где-то в математическом выражении есть NaN, то результатом вычислений с его участием будет NaN.

**Математические операции – безопасны**

Математические операции в JavaScript «безопасны». Мы можем делать что угодно: делить на ноль, обращаться с нечисловыми строками как с числами и т.д.

Скрипт никогда не остановится с фатальной ошибкой (не «умрёт»). В худшем случае мы получим NaN как результат выполнения.

Чтобы создать значение типа BigInt, необходимо добавить n в конец числового литерала:

## [Значение «undefined»](https://learn.javascript.ru/types" \l "znachenie-undefined)

Специальное значение undefined также стоит особняком. Оно формирует тип из самого себя так же, как и null.

Оно означает, что «значение не было присвоено».

Если переменная объявлена, но ей не присвоено никакого значения, то её значением будет undefined:

let age;

alert(age); // выведет "undefined"

## [Оператор typeof](https://learn.javascript.ru/types" \l "type-typeof)

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.

# Взаимодействие: alert, prompt, confirm

Так как мы будем использовать браузер как демо-среду, нам нужно познакомиться с несколькими функциями его интерфейса, а именно: alert, prompt и confirm.

## [alert](https://learn.javascript.ru/alert-prompt-confirm" \l "alert)

С этой функцией мы уже знакомы. Она показывает сообщение и ждёт, пока пользователь нажмёт кнопку «ОК».

Например:

alert("Hello");

Это небольшое окно с сообщением называется модальным окном. Понятие модальное означает, что пользователь не может взаимодействовать с интерфейсом остальной части страницы, нажимать на другие кнопки и т.д. до тех пор, пока взаимодействует с окном. В данном случае – пока не будет нажата кнопка «OK».

## [prompt](https://learn.javascript.ru/alert-prompt-confirm" \l "prompt)

Функция prompt принимает два аргумента:

result = prompt(title, [default]);

Этот код отобразит модальное окно с текстом, полем для ввода текста и кнопками OK/Отмена.

**title**

Текст для отображения в окне.

**default**

Необязательный второй параметр, который устанавливает начальное значение в поле для текста в окне.

**Квадратные скобки в синтаксисе [...]**

Квадратные скобки вокруг default в описанном выше синтаксисе означают, что параметр факультативный, необязательный.

## confirm

показывает сообщение и ждёт, пока пользователь нажмёт OK или Отмена. Возвращает true, если нажата OK, и false, если нажата кнопка «Отмена» или Esc с клавиатуры.

Все эти методы являются модальными: останавливают выполнение скриптов и не позволяют пользователю взаимодействовать с остальной частью страницы до тех пор, пока окно не будет закрыто.

На все указанные методы распространяются два ограничения:

1. Расположение окон определяется браузером. Обычно окна находятся в центре.
2. Визуальное отображение окон зависит от браузера, и мы не можем изменить их вид.

Такова цена простоты. Есть другие способы показать более приятные глазу окна с богатой функциональностью для взаимодействия с пользователем, но если «навороты» не имеют значения, то данные методы работают отлично.

# Преобразование типов

Чаще всего операторы и функции автоматически приводят переданные им значения к нужному типу.

Например, alert автоматически преобразует любое значение к строке. Математические операторы преобразуют значения к числам.

Есть также случаи, когда нам нужно явно преобразовать значение в ожидаемый тип.

## [Строковое преобразование](https://learn.javascript.ru/type-conversions" \l "strokovoe-preobrazovanie)

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки.

Например, alert(value) преобразует значение к строке.

Также мы можем использовать функцию String(value), чтобы преобразовать значение к строке:

## [Численное преобразование](https://learn.javascript.ru/type-conversions" \l "chislennoe-preobrazovanie)

Численное преобразование происходит в математических функциях и выражениях.

Например, когда операция деления / применяется не к числу:

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

Мы можем использовать функцию Number(value), чтобы явно преобразовать value к числу:

Явное преобразование часто применяется, когда мы ожидаем получить число из строкового контекста, например из текстовых полей форм.

Если строка не может быть явно приведена к числу, то результатом преобразования будет NaN.

Правила численного преобразования:

| **Значение** | **Преобразуется в…** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| string | Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то получаем 0, иначе из непустой строки «считывается» число. При ошибке результат NaN. |

## [Логическое преобразование](https://learn.javascript.ru/type-conversions" \l "logicheskoe-preobrazovanie)

Логическое преобразование самое простое.

Происходит в логических операциях (позже мы познакомимся с условными проверками и подобными конструкциями), но также может быть выполнено явно с помощью функции Boolean(value).

Правило преобразования:

* Значения, которые интуитивно «пустые», вроде 0, пустой строки, null, undefined и NaN, становятся false.
* Все остальные значения становятся true.

## [Приоритет операторов](https://learn.javascript.ru/operators" \l "prioritet-operatorov)

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется приоритетом, или, другими словами, существует определённый порядок выполнения операторов.

Из школы мы знаем, что умножение в выражении 1 + 2 \* 2 выполнится раньше сложения. Это как раз и есть «приоритет». Говорят, что умножение имеет более высокий приоритет, чем сложение.

Скобки важнее, чем приоритет, так что, если мы не удовлетворены порядком по умолчанию, мы можем использовать их, чтобы изменить приоритет. Например, написать (1 + 2) \* 2.

В JavaScript много операторов. Каждый оператор имеет соответствующий номер приоритета. Тот, у кого это число больше, – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.

Отрывок из [таблицы приоритетов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Operator_Precedence) (нет необходимости всё запоминать, обратите внимание, что приоритет унарных операторов выше, чем соответствующих бинарных):

| **Приоритет** | **Название** | **Обозначение** |
| --- | --- | --- |
| … | … | … |
| 17 | унарный плюс | + |
| 17 | унарный минус | - |
| 16 | возведение в степень | \*\* |
| 15 | умножение | \* |
| 15 | деление | / |
| 13 | сложение | + |
| 13 | вычитание | - |
| … | … | … |
| 3 | присваивание | = |
| … | … | … |

Так как «унарный плюс» имеет приоритет 17, который выше, чем 13 у «сложения» (бинарный плюс), то в выражении "+apples + +oranges" сначала выполнятся унарные плюсы, а затем сложение.

**Оператор строгого равенства === проверяет равенство без приведения типов.**

Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования.

# Логические операторы

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

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

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

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

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

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

Оператор НЕ представлен восклицательным знаком !.

# Функции

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

## [Объявление функции](https://learn.javascript.ru/function-basics" \l "obyavlenie-funktsii)

Для создания функций мы можем использовать объявление функции.

Пример объявления функции:

function showMessage() {

alert( 'Всем привет!' );

}

Вначале идёт ключевое слово function, после него имя функции, затем список параметров в круглых скобках через запятую (в вышеприведённом примере он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

## [Внешние переменные](https://learn.javascript.ru/function-basics" \l "vneshnie-peremennye)

У функции есть доступ к внешним переменным, например:

let userName = 'Вася';

function showMessage() {

let message = 'Привет, ' + userName;

alert(message);

}

showMessage(); // Привет, Вася

Функция обладает полным доступом к внешним переменным и может изменять их значение.

Например:

let userName = 'Вася';

function showMessage() {

userName = "Петя"; // (1) изменяем значение внешней переменной

let message = 'Привет, ' + userName;

alert(message);

}

alert( userName ); // Вася перед вызовом функции

showMessage();

alert( userName ); // Петя, значение внешней переменной было изменено функцией

Внешняя переменная используется, только если внутри функции нет такой локальной.

Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю.

## [Параметры по умолчанию](https://learn.javascript.ru/function-basics" \l "parametry-po-umolchaniyu)

Если параметр не указан, то его значением становится undefined.

Например, вышеупомянутая функция showMessage(from, text) может быть вызвана с одним аргументом:

showMessage("Аня");

Это не приведёт к ошибке. Такой вызов выведет "Аня: undefined". В вызове не указан параметр text, поэтому предполагается, что text === undefined.

Если мы хотим задать параметру text значение по умолчанию, мы должны указать его после =:

function showMessage(from, text = "текст не добавлен") {

alert( from + ": " + text );

}

showMessage("Аня"); // Аня: текст не добавлен

## Функции-стрелки, основы

Существует ещё более простой и краткий синтаксис для создания функций, который часто лучше, чем синтаксис Function Expression.

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

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

## [Многострочные стрелочные функции](https://learn.javascript.ru/arrow-functions-basics" \l "mnogostrochnye-strelochnye-funktsii)

В примерах выше аргументы использовались слева от =>, а справа вычислялось выражение с их значениями.

Порой нам нужно что-то посложнее, например, выполнить несколько инструкций. Это также возможно, нужно лишь заключить инструкции в фигурные скобки. И использовать return внутри них, как в обычной функции.

Например:

let sum = (a, b) => { // фигурная скобка, открывающая тело многострочной функции

let result = a + b;

return result; // при фигурных скобках для возврата значения нужно явно вызвать return

};

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

## [Итого](https://learn.javascript.ru/arrow-functions-basics" \l "itogo)

Функции-стрелки очень удобны для однострочных действий. Они бывают двух типов:

1. Без фигурных скобок: (...args) => expression – правая сторона выражение: функция выполняет его и возвращает результат.
2. С фигурными скобками: (...args) => { body } – скобки позволяют нам писать многострочные инструкции внутри функции, но при этом необходимо указывать директиву return, чтобы вернуть какое-либо значение.

# Объекты

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

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

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

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

## [Литералы и свойства](https://learn.javascript.ru/object" \l "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

};

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.

Для обращения к свойствам используется запись «через точку»:

// получаем свойства объекта:

alert( user.name ); // John

alert( user.age ); // 30

**Объект, объявленный как константа, может быть изменён**

Объект, объявленный через const, *может* быть изменён.

Например:

const user = {

name: "John"

};

user.name = "Pete"; // (\*)

alert(user.name); // Pete

Может показаться, что строка (\*) должна вызвать ошибку, но нет, здесь всё в порядке. Дело в том, что объявление const защищает от изменений только саму переменную user, а не её содержимое.

Определение const выдаст ошибку только если мы присвоим переменной другое значение: user=....

Есть ещё один способ сделать константами свойства объекта, который мы рассмотрим в главе [Флаги и дескрипторы свойств](https://learn.javascript.ru/property-descriptors).

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

Для свойств, имена которых состоят из нескольких слов, доступ к значению «через точку» не работает:

// это вызовет синтаксическую ошибку

user.likes birds = true

JavaScript видит, что мы обращаемся к свойству user.likes, а затем идёт непонятное слово birds. В итоге синтаксическая ошибка.

Точка требует, чтобы ключ был именован по правилам именования переменных. То есть не имел пробелов, не начинался с цифры и не содержал специальные символы, кроме $ и \_.

Для таких случаев существует альтернативный способ доступа к свойствам через квадратные скобки. Такой способ сработает с любым именем свойства:

let user = {};

// присваивание значения свойству

user["likes birds"] = true;

// получение значения свойства

alert(user["likes birds"]); // true

// удаление свойства

delete user["likes birds"];

Сейчас всё в порядке. Обратите внимание, что строка в квадратных скобках заключена в кавычки (подойдёт любой тип кавычек).

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

let key = "likes birds";

// то же самое, что и user["likes birds"] = true;

user[key] = true;

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

Мы можем использовать квадратные скобки в литеральной нотации для создания вычисляемого свойства.

Пример:

let fruit = prompt("Какой фрукт купить?", "apple");

let bag = {

[fruit]: 5, // имя свойства будет взято из переменной fruit

};

alert( bag.apple ); // 5, если fruit="apple"

Смысл вычисляемого свойства прост: запись [fruit] означает, что имя свойства необходимо взять из переменной fruit.

Вместо name:name мы можем написать просто name:

function makeUser(name, age) {

return {

name, // то же самое, что и name: name

age // то же самое, что и age: age

// ...

};

}

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

Мы можем использовать как обычные свойства, так и короткие в одном и том же объекте:

let user = {

name, // тоже самое, что и name:name

age: 30

};

## [Ограничения на имена свойств](https://learn.javascript.ru/object" \l "ogranicheniya-na-imena-svoystv)

Как мы уже знаем, имя переменной не может совпадать с зарезервированными словами, такими как «for», «let», «return» и т.д.

Но для свойств объекта такого ограничения нет:

// эти имена свойств допустимы

let obj = {

for: 1,

let: 2,

return: 3

};

alert( obj.for + obj.let + obj.return ); // 6

Иными словами, нет никаких ограничений к именам свойств. Они могут быть в виде строк или символов (специальный тип для идентификаторов, который будет рассмотрен позже).

Все другие типы данных будут автоматически преобразованы к строке.

Например, если использовать число 0 в качестве ключа, то оно превратится в строку "0":

let obj = {

0: "Тест" // то же самое что и "0": "Тест"

};

// обе функции alert выведут одно и то же свойство (число 0 преобразуется в строку "0")

alert( obj["0"] ); // Тест

alert( obj[0] ); // Тест (то же свойство)

## [Проверка существования свойства, оператор «in»](https://learn.javascript.ru/object" \l "proverka-suschestvovaniya-svoystva-operator-in)

В отличие от многих других языков, особенность JavaScript-объектов в том, что можно получить доступ к любому свойству. Даже если свойства не существует – ошибки не будет!

При обращении к свойству, которого нет, возвращается undefined. Это позволяет просто проверить существование свойства:

let user = {};

alert( user.noSuchProperty === undefined ); // true означает "свойства нет"

Также существует специальный оператор "in" для проверки существования свойства в объекте.

Синтаксис оператора:

"key" in object

## [Цикл «for…in»](https://learn.javascript.ru/object" \l "tsikl-for-in)

Для перебора всех свойств объекта используется цикл for..in. Этот цикл отличается от изученного ранее цикла for(;;).

Синтаксис:

for (key in object) {

// тело цикла выполняется для каждого свойства объекта

}

## [Итого](https://learn.javascript.ru/object#itogo)

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.

# [Сравнение по ссылке](https://learn.javascript.ru/object-copy" \l "sravnenie-po-ssylke)

Операторы равенства == и строгого равенства === для объектов работают одинаково.

**Два объекта равны только в том случае, если это один и тот же объект.**

В примере ниже две переменные ссылаются на один и тот же объект, поэтому они равны друг другу:

let a = {};

let b = a; // копирование по ссылке

alert( a == b ); // true, т.к. обе переменные ссылаются на один и тот же объект

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

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

let a = {};

let b = {}; // два независимых объекта

alert( a == b ); // false

Для сравнений типа obj1 > obj2 или для сравнения с примитивом obj == 5 объекты преобразуются в примитивы. Мы скоро изучим, как работают такие преобразования объектов, но, по правде говоря, сравнения такого рода необходимы очень редко и обычно являются результатом ошибки программиста.

## Копирование объектов и ссылки

Одним из фундаментальных отличий объектов от примитивных типов данных является то, что они хранятся и копируются «по ссылке».

Примитивные типы: строки, числа, логические значения – присваиваются и копируются «по значению».

## [Клонирование и объединение объектов, Object.assign](https://learn.javascript.ru/object-copy" \l "klonirovanie-i-obedinenie-obektov-object-assign)

Таким образом, при копировании переменной с объектом создаётся ещё одна ссылка на тот же самый объект.

Но что, если нам всё же нужно дублировать объект? Создать независимую копию, клон?

Это выполнимо, но немного сложно, так как в JavaScript нет встроенного метода для этого. На самом деле, такая нужда возникает редко. В большинстве случаев нам достаточно копирования по ссылке.

Но если мы действительно этого хотим, то нам нужно создавать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их.

Например так:

let user = {

name: "Иван",

age: 30

};

let clone = {}; // новый пустой объект

// скопируем все свойства user в него

for (let key in user) {

clone[key] = user[key];

}

// теперь в переменной clone находится абсолютно независимый клон объекта

clone.name = "Пётр"; // изменим в нём данные

alert( user.name ); // в оригинальном объекте значение свойства `name` осталось прежним – Иван.

Кроме того, для этих целей мы можем использовать метод [Object.assign](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign).

Синтаксис:

Object.assign(dest, [src1, src2, src3...])

* Первый аргумент dest — целевой объект.
* Остальные аргументы src1, ..., srcN (может быть столько, сколько нужно)) являются исходными объектами
* Метод копирует свойства всех исходных объектов src1, ..., srcN в целевой объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект.
* Возвращает объект dest.

## [Вложенное клонирование](https://learn.javascript.ru/object-copy" \l "vlozhennoe-klonirovanie)

До сих пор мы предполагали, что все свойства объекта user хранят примитивные значения. Но свойства могут быть ссылками на другие объекты. Что с ними делать?

Например, есть объект:

let user = {

name: "Иван",

sizes: {

height: 182,

width: 50

}

};

alert( user.sizes.height ); // 182

Теперь при клонировании недостаточно просто скопировать clone.sizes = user.sizes, поскольку user.sizes – это объект, он будет скопирован по ссылке. А значит объекты clone и user в своих свойствах sizes будут ссылаться на один и тот же объект:

let user = {

name: "Иван",

sizes: {

height: 182,

width: 50

}

};

let clone = Object.assign({}, user);

alert( user.sizes === clone.sizes ); // true, один и тот же объект

// user и clone обращаются к одному sizes

user.sizes.width++; // меняем свойство в одном объекте

alert(clone.sizes.width); // 51, видим результат в другом объекте

Чтобы исправить это, мы должны в цикле клонирования делать проверку, не является ли значение user[key] объектом, и если это так – скопировать и его структуру тоже. Это называется «глубокое клонирование».

Мы можем реализовать глубокое клонирование, используя рекурсию. Или, чтобы не изобретать велосипед, использовать готовую реализацию — метод [\_.cloneDeep(obj)](https://lodash.com/docs#cloneDeep) из JavaScript-библиотеки [lodash](https://lodash.com/).

## [Итого](https://learn.javascript.ru/object-copy#itogo)

Объекты присваиваются и копируются по ссылке. Другими словами, переменная хранит не «значение объекта», а «ссылку» (адрес в памяти) на это значение. Поэтому копирование такой переменной или передача её в качестве аргумента функции приводит к копированию этой ссылки, а не самого объекта.

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

Для «простого клонирования» объекта можно использовать Object.assign. Необходимо помнить, что Object.assign не делает глубокое клонирование объекта. Если внутри копируемого объекта есть свойство, значение которого не является примитивом, оно будет передано по ссылке. Для создания «настоящей копии» (полного клона объекта) можно воспользоваться методом из сторонней JavaScript-библиотеки [\_.cloneDeep(obj)](https://lodash.com/docs#cloneDeep).

# Методы объекта, "this"

Объекты обычно создаются, чтобы представлять сущности реального мира, будь то пользователи, заказы и так далее:

// Объект пользователя

let user = {

name: "Джон",

age: 30

};

И так же, как и в реальном мире, пользователь может совершать действия: выбирать что-то из корзины покупок, авторизовываться, выходить из системы, оплачивать и т.п.

Такие действия в JavaScript представлены свойствами-функциями объекта.

## [Примеры методов](https://learn.javascript.ru/object-methods" \l "primery-metodov)

Для начала давайте научим нашего пользователя user здороваться:

let user = {

name: "Джон",

age: 30

};

user.sayHi = function() {

alert("Привет!");

};

user.sayHi(); // Привет!

Здесь мы просто использовали Function Expression (функциональное выражение), чтобы создать функцию для приветствия, и присвоили её свойству user.sayHi нашего объекта.

Затем мы вызвали её. Теперь пользователь может говорить!

Функцию, которая является свойством объекта, называют методом этого объекта.

Итак, мы получили метод sayHi объекта user.

Конечно, мы могли бы заранее объявить функцию и использовать её в качестве метода, примерно так:

let user = {

// ...

};

// сначала объявляем

function sayHi() {

alert("Привет!");

};

// затем добавляем в качестве метода

user.sayHi = sayHi;

user.sayHi(); // Привет!

**Объектно-ориентированное программирование**

Когда мы пишем наш код, используя объекты для представления сущностей реального мира, – это называется [объектно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%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) или сокращённо: «ООП».

ООП является большой предметной областью и интересной наукой само по себе. Как выбрать правильные сущности? Как организовать взаимодействие между ними? Это – создание архитектуры, и есть хорошие книги по этой теме, такие как «Приёмы объектно-ориентированного проектирования. Паттерны проектирования» авторов Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес или «Объектно-ориентированный анализ и проектирование с примерами приложений» Гради Буча, а также ещё множество других книг.

## [Сокращённая запись метода](https://learn.javascript.ru/object-methods" \l "sokraschyonnaya-zapis-metoda)

Существует более короткий синтаксис для методов в литерале объекта:

// эти объекты делают одно и то же (одинаковые методы)

user = {

sayHi: function() {

alert("Привет");

}

};

// сокращённая запись выглядит лучше, не так ли?

user = {

sayHi() { // то же самое, что и "sayHi: function()"

alert("Привет");

}

};

Как было показано, мы можем пропустить ключевое слово "function" и просто написать sayHi().

Нужно отметить, что эти две записи не полностью эквивалентны. Есть тонкие различия, связанные с наследованием объектов (что будет рассмотрено позже), но на данном этапе изучения это неважно. В большинстве случаев сокращённый синтаксис предпочтителен.

## [Ключевое слово «this» в методах](https://learn.javascript.ru/object-methods" \l "klyuchevoe-slovo-this-v-metodah)

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода).

Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user.

**Для доступа к информации внутри объекта метод может использовать ключевое слово this.**

Значение this – это объект «перед точкой», который использовался для вызова метода.

Например:

let user = {

name: "Джон",

age: 30,

sayHi() {

// this - это "текущий объект"

alert(this.name);

}

};

user.sayHi(); // Джон

Здесь во время выполнения кода user.sayHi() значением this будет являться user (ссылка на объект user).

Технически также возможно получить доступ к объекту без ключевого слова this, ссылаясь на него через внешнюю переменную (в которой хранится ссылка на этот объект):

let user = {

name: "Джон",

age: 30,

sayHi() {

alert(user.name); // используем переменную "user" вместо ключевого слова "this"

}

};

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

Это показано ниже:

let user = {

name: "Джон",

age: 30,

sayHi() {

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

}

};

let admin = user;

user = null; // обнулим переменную для наглядности, теперь она не хранит ссылку на объект.

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

Если мы используем this.name вместо user.name внутри alert, тогда этот код будет работать.

# Конструкторы, создание объектов через "new"

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество однотипных объектов, таких как пользователи, элементы меню и т.д.

Это можно сделать при помощи функции-конструктора и оператора "new".

## [Функция-конструктор](https://learn.javascript.ru/constructor-new#funktsiya-konstruktor)

Функции-конструкторы являются обычными функциями. Но есть два соглашения:

1. Имя функции-конструктора должно начинаться с большой буквы.
2. Функция-конструктор должна вызываться при помощи оператора "new".

Например:

function User(name) {

this.name = name;

this.isAdmin = false;

}

let user = new User("Вася");

alert(user.name); // Вася

alert(user.isAdmin); // false

Когда функция вызывается как new User(...), происходит следующее:

1. Создаётся новый пустой объект, и он присваивается this.
2. Выполняется код функции. Обычно он модифицирует this, добавляет туда новые свойства.
3. Возвращается значение this.

Другими словами, вызов new User(...) делает примерно вот что:

function User(name) {

// this = {}; (неявно)

// добавляет свойства к this

this.name = name;

this.isAdmin = false;

// return this; (неявно)

}

То есть, результат вызова new User("Вася") – это тот же объект, что и:

let user = {

name: "Вася",

isAdmin: false

};

Теперь, когда нам необходимо будет создать других пользователей, мы можем использовать new User("Маша"), new User("Даша") и т.д. Данная конструкция гораздо удобнее и читабельнее, чем каждый раз создавать литерал объекта. Это и является основной целью конструкторов – удобное повторное создание однотипных объектов.

Ещё раз заметим: технически любая функция может быть использована как конструктор. То есть, каждая функция может быть вызвана при помощи оператора new, и выполнится алгоритм, указанный выше в примере. Заглавная буква в названии функции является всеобщим соглашением по именованию, она как бы подсказывает разработчику, что данная функция является функцией-конструктором, и её нужно вызывать через new.

## [Возврат значения из конструктора return](https://learn.javascript.ru/constructor-new" \l "vozvrat-znacheniya-iz-konstruktora-return)

Обычно конструкторы ничего не возвращают явно. Их задача – записать все необходимое в this, который в итоге станет результатом.

Но если return всё же есть, то применяется простое правило:

* При вызове return с объектом, будет возвращён объект, а не this.
* При вызове return с примитивным значением, примитивное значение будет отброшено.

Другими словами, return с объектом возвращает объект, в любом другом случае конструктор вернёт this.

В примере ниже return возвращает объект вместо this:

function BigUser() {

this.name = "Вася";

return { name: "Godzilla" }; // <-- возвращает этот объект

}

alert( new BigUser().name ); // Godzilla, получили этот объект

А вот пример с пустым return (или мы могли бы поставить примитив после return, неважно)

function SmallUser() {

this.name = "Вася";

return; // <-- возвращает this

}

alert( new SmallUser().name ); // Вася

Обычно у конструкторов отсутствует return. В данном блоке мы упомянули особое поведение с возвращаемыми объектами, чтобы не оставлять пробелов в изучении языка.

**Отсутствие скобок**

Кстати, мы можем не ставить скобки после new, если вызов конструктора идёт без аргументов.

let user = new User; // <-- без скобок

// то же, что и

let user = new User();

Пропуск скобок считается плохой практикой, но синтаксис языка такое позволяет.

## [Создание методов в конструкторе](https://learn.javascript.ru/constructor-new" \l "sozdanie-metodov-v-konstruktore)

Использование конструкторов для создания объектов даёт большую гибкость. Можно передавать конструктору параметры, определяющие, как создавать объект, и что в него записывать.

В this мы можем добавлять не только свойства, но и методы.

Например, в примере ниже, new User(name) создаёт объект с данным именем name и методом sayHi:

function User(name) {

this.name = name;

this.sayHi = function() {

alert( "Меня зовут: " + this.name );

};

}

let vasya = new User("Вася");

vasya.sayHi(); // Меня зовут: Вася

/\*

vasya = {

name: "Вася",

sayHi: function() { ... }

}

\*/

Для создания сложных объектов есть и более «продвинутый» синтаксис – [классы](https://learn.javascript.ru/classes), которые мы разберём позже.

## [Итого](https://learn.javascript.ru/constructor-new#itogo)

* Функции-конструкторы или просто конструкторы являются обычными функциями, именовать которые следует с заглавной буквы.
* Конструкторы следует вызывать при помощи оператора new. Такой вызов создаёт пустой this в начале выполнения и возвращает заполненный в конце.

Мы можем использовать конструкторы для создания множества похожих объектов.

JavaScript предоставляет функции-конструкторы для множества встроенных объектов языка: например, Date, Set и других, которые нам ещё предстоит изучить.

# Опциональная цепочка '?.'

**Новая возможность**

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифил.

Опциональная цепочка ?. — это безопасный способ доступа к свойствам вложенных объектов, даже если какое-либо из промежуточных свойств не существует.

## [Проблема «несуществующего свойства»](https://learn.javascript.ru/optional-chaining" \l "problema-nesuschestvuyuschego-svoystva)

Если вы только начали читать учебник и изучать JavaScript, то, возможно, эта проблема вам пока незнакома, но она достаточно распространена.

Например, рассмотрим объекты для пользователей user. У большинства пользователей есть адрес user.address с улицей user.address.street, но некоторые адрес не указали.

В этом случае при попытке получить свойство user.address.street будет ошибка:

let user = {}; // пользователь без свойства address

alert(user.address.street); // ошибка!

Это нормальный результат, так работает JavaScript, но во многих реальных ситуациях удобнее было бы получать не ошибку, а просто undefined («нет улицы»).

Или ещё пример. В веб-разработке нам бывает нужно получить данные об HTML-элементе, который иногда может отсутствовать на странице:

// Произойдёт ошибка, если querySelector(...) равен null.

let html = document.querySelector('.my-element').innerHTML;

До появления ?. в языке для решения подобных проблем использовался оператор &&.

Например:

let user = {}; // пользователь без адреса

alert( user && user.address && user.address.street ); // undefined (без ошибки)

Использование логического И со всей цепочкой свойств гарантирует, что все они существуют (а если нет – вычисление прекращается), но это довольно длинная и громоздкая конструкция.

## [Опциональная цепочка](https://learn.javascript.ru/optional-chaining" \l "optsionalnaya-tsepochka)

Опциональная цепочка ?. останавливает вычисление и возвращает undefined, если часть перед ?. имеет значение undefined или null.

**Для краткости в этой статье мы будем говорить о значении, что оно «существует», если оно отличается от null или undefined.**

Вот безопасный способ обратиться к свойству user.address.street:

let user = {}; // пользователь без адреса

alert( user?.address?.street ); // undefined (без ошибки)

Чтение адреса с помощью конструкции user?.address выполняется без ошибок, даже если объекта user не существует:

let user = null;

alert( user?.address ); // undefined

alert( user?.address.street ); // undefined

Обратите внимание, что синтаксис ?. делает необязательным только свойство перед ним, а не какое-либо последующее.

В приведённом выше примере конструкция user?. допускает, что переменная user может содержать null/undefined.

С другой стороны, если объект user существует, то в нём должно быть свойство user.address, иначе выполнение user?.address.street вызовет ошибку из-за второй точки.

**Не злоупотребляйте опциональной цепочкой**

Используйте ?. только тогда, когда допускаете ситуацию, что значение перед ним не существует.

Например, если по нашей логике объект user точно существует, но его свойство address является необязательным, то предпочтительнее использовать следующую конструкцию: user.address?.street.

Тогда если переменная user по ошибке окажется пустой, мы увидим программную ошибку и исправим это.

**Переменная перед ?. должна быть объявлена**

Если переменной user вообще не существует, то выражение user?.anything выдаст ошибку:

// ReferenceError: user is not defined

user?.address;

Объявление переменной (например let/const/var user) обязательно должно быть. Опциональная цепочка работает только с существующими переменными.

## Кроме этого, ?. можно совместно использовать с delete:

delete user?.name; // Удалить user.name, если пользователь существует

**Можно использовать ?. для безопасного чтения и удаления, но не для записи**

Опциональная цепочка ?. не имеет смысла в левой части присваивания.

Например:

let user;

user?.name = "John"; // Ошибка, это не сработает

// это по сути то же самое что undefined = "John"

Она недостаточно «умна» для этого.

## [Итого](https://learn.javascript.ru/optional-chaining#itogo)

Синтаксис опциональной цепочки ?. имеет три формы:

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

Как мы видим, все они просты и понятны в использовании. ?. проверяет левую часть выражения на равенство null/undefined, и продолжает дальнейшее вычисление, только если это не так.

Цепочка ?. позволяет без возникновения ошибок обратиться к вложенным свойствам.

Тем не менее, нужно разумно использовать ?. — только там, где это уместно, если допустимо, что левая часть не существует. Чтобы таким образом не скрывать возможные ошибки программирования.

# Тип данных Symbol

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

До сих пор мы видели только строки. Теперь давайте разберём символы, увидим, что хорошего они нам дают.

## [Символы](https://learn.javascript.ru/symbol" \l "simvoly)

«Символ» представляет собой уникальный идентификатор.

Создаются новые символы с помощью функции Symbol():

// Создаём новый символ - id

let id = Symbol();

При создании символу можно дать описание (также называемое имя), в основном использующееся для отладки кода:

// Создаём символ id с описанием (именем) "id"

let id = Symbol("id");

Символы гарантированно уникальны. Даже если мы создадим множество символов с одинаковым описанием, это всё равно будут разные символы. Описание – это просто метка, которая ни на что не влияет.

Например, вот два символа с одинаковым описанием – но они не равны:

let id1 = Symbol("id");

let id2 = Symbol("id");

alert(id1 == id2); // false

Если вы знаете Ruby или какой-то другой язык программирования, в котором есть своего рода «символы» – пожалуйста, будьте внимательны. Символы в JavaScript имеют свои особенности, и не стоит думать о них, как о символах в Ruby или в других языках.

# Преобразование объектов в примитивы

Что произойдёт, если сложить два объекта obj1 + obj2, вычесть один из другого obj1 - obj2 или вывести их на экран, воспользовавшись alert(obj)?

В этом случае объекты сначала автоматически преобразуются в примитивы, а затем выполняется операция.

В главе [Преобразование типов](https://learn.javascript.ru/type-conversions) мы видели правила для численных, строковых и логических преобразований. Но обделили вниманием объекты. Теперь, поскольку мы уже знаем о методах объектов и символах, можно исправить это упущение.

1. Все объекты в логическом контексте являются true. Существуют лишь их численные и строковые преобразования.
2. Численные преобразования происходят, когда мы вычитаем объекты или выполняем математические операции. Например, объекты Date (мы рассмотрим их в статье [Дата и время](https://learn.javascript.ru/date)) могут вычитаться, и результатом date1 - date2 будет временной отрезок между двумя датами.
3. Что касается строковых преобразований – они обычно происходят, когда мы выводим объект alert(obj), а также в других случаях, когда объект используется как строка.

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

Мы можем тонко настраивать строковые и численные преобразования, используя специальные методы объекта.

Существуют три варианта преобразований («три хинта»), описанные в [спецификации](https://tc39.github.io/ecma262/#sec-toprimitive):

**"string"**

Для преобразования объекта к строке, когда операция ожидает получить строку, например alert:

// вывод

alert(obj);

// используем объект в качестве имени свойства

anotherObj[obj] = 123;

**"number"**

Для преобразования объекта к числу, в случае математических операций:

// явное преобразование

let num = Number(obj);

// математическое (исключая бинарный оператор "+")

let n = +obj; // унарный плюс

let delta = date1 - date2;

// сравнения больше/меньше

let greater = user1 > user2;

**"default"**

Происходит редко, когда оператор «не уверен», какой тип ожидать.

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

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.

## [Итого](https://learn.javascript.ru/object-toprimitive#itogo)

Преобразование объектов в примитивы вызывается автоматически многими встроенными функциями и операторами, которые ожидают примитив в качестве аргумента.

Существует всего 3 типа преобразований (хинтов):

* "string" (для alert и других операций, которым нужна строка)
* "number" (для математических операций)
* "default" (для некоторых операций)

В спецификации явно указано, какой хинт должен использовать каждый оператор. И существует совсем немного операторов, которые не знают, что ожидать, и используют хинт со значением "default". Обычно для встроенных объектов хинт "default" обрабатывается так же, как "number". Таким образом, последние два очень часто объединяют вместе.

Алгоритм преобразований к примитивам следующий:

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

На практике довольно часто достаточно реализовать только obj.toString() как «универсальный» метод для всех типов преобразований, возвращающий «читаемое» представление объекта, достаточное для логирования или отладки.