**Тема 1. Основы JavaScript**

1. Введение в JavaScript.
2. Отладка скриптов.
3. Синтаксис JavaScript.
4. Переменные.
5. Типы данных.
6. Преобразование типов.
7. Операторы JavaScript.
8. Операторы сравнения.
9. Модальные окна.
10. Условные операторы.
11. Логические операторы.
12. Циклы while, for.
13. Конструкция switch.
14. Функции. Объявление функции.
15. Функциональные выражения и функции-стрелки.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. **Введение в JavaScript**

**История**

Язык JavaScript (JS) был представлен в 1995 году как способ добавлять программы на веб-страницы в браузере Netscape Navigator. Изначально он назывался LiveScript. Но в то время широко использовался язык Java, и с целью популярности LiveScript переименовали в JavaScript.

Планировалось, что JavaScript будет упрощенной версией языка Java. Однако, JavaScript очень развился и сейчас это совершенно независимый язык, и к языку Java не имеет никакого отношения. JavaScript имеет своею спецификацию, которая называется [ECMAScript](https://ru.wikipedia.org/wiki/ECMAScript).

После того, как язык стал широко использоваться в других браузерах, был составлен документ, описывающий работу языка. Он называется стандарт ECMAScript по имени организации ECMA (European Computer Manufacturers Association – Европейская ассоциация производителей компьютеров). ECMAScript является стандартом, а JavaScript – это самая популярная реализация этого стандарта.

**Версии ECMAScript**

ES – принятое сокращение для ECMAScript. Каждое издание ECMAScript получает аббревиатуру ES с последующим его номером.

* ES1 был выпущен в июне 1997 года,
* ES2 – в июне 1998 года,
* ES3 – в декабре 1999 года,
* ES4 – не была принята,
* ES5 был выпущен в декабре 2009 года,
* ES6/ES2015 вышел в июне 2015 года. С выходом этого стандарта комитет принял решение перейти к ежегодным обновлениям. Поэтому издание было переименовано в ES2015, чтобы отражать год релиза. Последующие версии также называются в соответствии с годом их выпуска.
* ES2016 (ES7) вышла в июне 2016 года,
* ES2017 (ES8) – в июне 2017 года,
* ES2018 – в июне 2018 года,
* ES2019 – в июне 2019 года.

Термин ES.Next ссылается на новую версию ECMAScript. Стоит отметить, что каждая новая версия приносит новые функции для языка.

**Выполнение JS**

Программы на языке JavaScript называются скриптами. В браузере они подключаются напрямую к HTML и выполняются сразу после загрузки страницы.

JavaScript может выполняться не только в браузере, а где угодно, нужна лишь специальная программа – [*интерпретатор*](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BF%D1%80%D0%B5%D1%82%D0%B0%D1%82%D0%BE%D1%80). Процесс выполнения скрипта называют *интерпретацией*.

*Транслятор* (англ. translator - переводчик) представляет собой программу, на основе которой компьютер преобразует вводимые в него программы на машинный язык, поскольку он может выполнять программы, записанные только на языке его процессора, и алгоритмы, заданные на другом языке, должны быть перед их выполнением переведены на машинный язык. Трансляторы реализуются в виде компиляторов или интерпретаторов. С точки зрения выполнения работы компилятор и интерпретатор существенно различаются.

*Компилятор* (англ. compiler - составитель, собиратель) читает всю программу целиком, делает ее перевод и создает законченный вариант программы на машинном языке, который затем и выполняется. Результат работы компилятора — бинарный исполняемый файл.

*Интерпретатор* (англ. interpreter - истолкователь, устный переводчик) переводит и выполняет программу строка за строкой. После того, как программа откомпилирована, ни исходный текст программы, ни компилятор более не нужны для исполнения программы. В то же время программа, обрабатываемая интерпретатором, должна заново переводиться на машинный язык при каждом очередном запуске программы. То есть исходный файл является непосредственно исполняемым.

Откомпилированные программы работают быстрее, но интерпретируемые проще исправлять и изменять.

Современные интерпретаторы перед выполнением преобразуют JavaScript в машинный код или близко к нему (байт-код), оптимизируют и затем выполняют. И даже во время выполнения скрипта оптимизируют его. Поэтому JavaScript работает очень быстро.

Во все основные браузеры встроен интерпретатор («движок») JavaScript, именно поэтому они могут выполнять скрипты на странице. Но JavaScript можно использовать не только в браузере. Это полноценный язык, программы на котором можно запускать на сервере, и даже в других устройствах, если в них установлен соответствующий «движок». Разные движки имеют разные «кодовые имена». Например, [V8](https://ru.wikipedia.org/wiki/V8_(%D0%B4%D0%B2%D0%B8%D0%B6%D0%BE%D0%BA_JavaScript)) – для Chrome и Opera, [SpiderMonkey](https://ru.wikipedia.org/wiki/SpiderMonkey) – для Firefox, Trident и Chakra для разных версий IE, ChakraCore для Microsoft Edge, Nitro и SquirrelFish для Safari и т.д.

**Возможности JavaScript в браузере**

Современный JavaScript – это «безопасный» язык программирования. Он не предоставляет низкоуровневый доступ к памяти или процессору, потому что он 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:

* Добавлять новый HTML 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»).

Возможности JavaScript в браузере ограничены ради безопасности пользователя. Цель заключается в предотвращении доступа вредоносной 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 операционной системы.

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 куда-либо.

* Различные окна/вкладки не могут взаимодействовать между собой. Иногда одно окно, используя JavaScript, открывает другое окно. Но даже в этом случае, JavaScript с одной страницы не имеет доступа к другой, если они из разных сайтов (с другого домена, протокола или порта).

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

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

Подобные ограничения не действуют, если JavaScript используется вне браузера, например — на сервере. Современные браузеры предоставляют плагины/расширения с помощью которых можно запрашивать дополнительные разрешения.

[**Языки «над» JavaScript**](https://learn.javascript.ru/intro#yazyki-nad-javascript)

Существует много языков, которые *транспилируются* (конвертируются) в JavaScript прежде, чем запустятся в браузере. Современные инструменты делают транспиляцию очень быстрой и прозрачной, фактически позволяя разработчикам писать код на другом языке, автоматически преобразуя его в JavaScript.

Примеры таких языков:

* [*CoffeeScript*](http://coffeescript.org/) добавляет «синтаксический сахар» для JavaScript. Он вводит более короткий синтаксис, который позволяет писать чистый и лаконичный код.
* [*TypeScript*](http://www.typescriptlang.org/) концентрируется на добавлении «строгой типизации» для упрощения разработки и поддержки больших и сложных систем. Разработан Microsoft.
* [*Flow*](http://flow.org/) тоже добавляет типизацию, но иначе. Разработан Facebook.
* [*Dart*](https://www.dartlang.org/)  имеет собственный движок работающий вне браузера (например, в мобильных приложениях). Первоначально был предложен Google как замена JavaScript, но на данный момент, необходима его транспиляция для запуска, так же как для языков выше.

**Справочники и спецификации**

*Спецификация* ECMA-262 содержит самую глубокую, детальную и формализованную информацию о JavaScript. Она определяет сам язык.

Самые последние возможности, которые включены или будут включены в следующую версию ES, можно найти на <https://tc39.es/ecma262/> и <https://github.com/tc39/proposals>.

MDN (Mozilla) JavaScript Reference – это *справочник* с примерами и другой информацией. Хороший источник для получения подробных сведений о функциях языка, методах встроенных объектов и так далее.

Располагается по адресу <https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference>.

MSDN – справочник от Microsoft, содержащий много информации, в том числе по JavaScript. Если вам нужно найти что-то специфическое по разработке для браузера Internet Explorer, то лучше искать тут: <http://msdn.microsoft.com/>.

JavaScript – это развивающийся язык, в который постоянно добавляется что-то новое. Посмотреть, какие возможности поддерживаются в разных браузерах и других движках, можно в следующих источниках:

* [http://caniuse.com](http://caniuse.com/) – таблицы совместимости с информацией о поддержке по каждой возможности языка.
* <https://kangax.github.io/compat-table> – таблица с возможностями языка и движками, которые их поддерживают и не поддерживают.

**Редакторы кода**

Термином [IDE](https://ru.wikipedia.org/wiki/Integrated_development_environment) (Integrated Development Environment, «интегрированная среда разработки») называют мощные редакторы с множеством функций, которые работают в рамках целого проекта. Это не просто редактор, а нечто большее.

IDE загружает проект (который может состоять из множества файлов), позволяет переключаться между файлами, предлагает автодополнение по коду всего проекта (а не только открытого файла), также она интегрирована с системой контроля версий (например, такой как [git](https://git-scm.com/)), средой для тестирования и другими инструментами на уровне всего проекта.

Кроссплатформенные IDE:

* [Visual Studio Code](https://code.visualstudio.com/) (бесплатная).
* [WebStorm](http://www.jetbrains.com/webstorm/) (платная).

Обычные редакторы менее мощные, чем IDE, но они отличаются скоростью, удобным интерфейсом и простотой. Их используют для того, чтобы быстро открыть и отредактировать нужный файл с кодом. Такие редакторы могут иметь множество плагинов, включая автодополнение и анализаторы синтаксиса на уровне директории. Наиболее полпулярные рекдакторы кода:

* [Atom](https://atom.io/) (кроссплатформенный, бесплатный).
* [Sublime Text](http://www.sublimetext.com/) (кроссплатформенный, частично бесплатный).
* [Notepad++](https://notepad-plus-plus.org/) (Windows, бесплатный).
* [Vim](http://www.vim.org/) и [Emacs](https://www.gnu.org/software/emacs/).

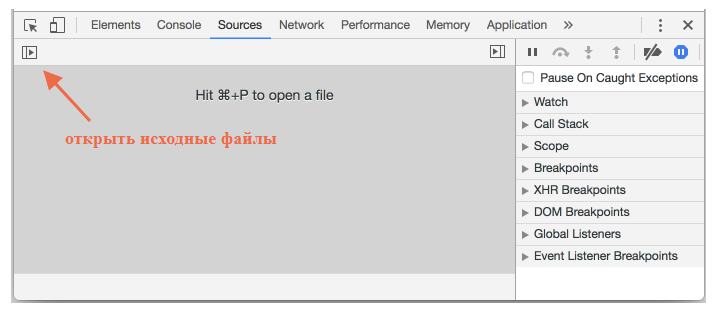
1. **Отладка скриптов**

Все современные браузеры и многие среды разработки поддерживают средства отладки кода — специальный графический интерфейс для быстрого поиска и устранения ошибок. Он также позволяет по шагам отследить, что именно происходит в коде.

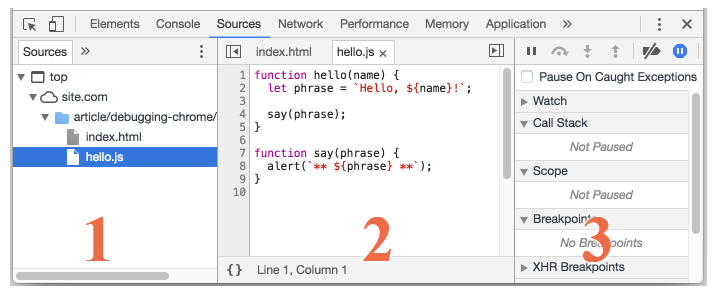
Рассмотрим процесс отладки в браузере Chrome, так как у него достаточно возможностей, в большинстве других браузеров процесс будет схожим.

[**Панель «Исходный код» («Sources»)**](https://learn.javascript.ru/debugging-chrome#panel-ishodnyy-kod-sources)

Чтобы включить инструменты разработчика, необходимо нажать F12 (Mac: Cmd+Opt+I). Щёлкните по панели sources («исходный код»). При первом запуске получаем следующее:



Кнопка-переключатель  откроет вкладку со списком файлов. Кликните на неё и файл с кодом JS. Исходный код появится в соответствующей вкладке:



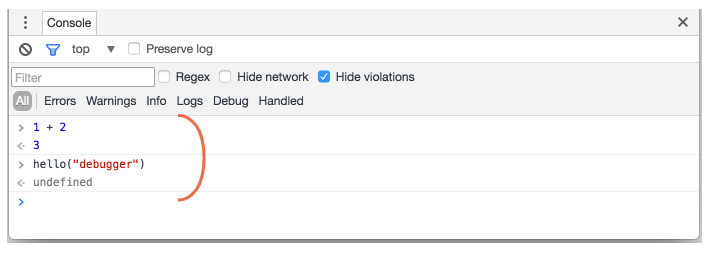
Интерфейс состоит из трёх зон:

1. В зоне Resources (Ресурсы) показаны файлы HTML, JavaScript, CSS, включая изображения, используемые на странице. Здесь также могут быть файлы различных расширений Chrome.
2. Зона Source показывает исходный код.
3. Зона Information and control (Сведения и контроль) отведена для отладки.

Чтобы скрыть список ресурсов и освободить экранное место для исходного кода, необходимо щёлкнуть по тому же переключателю  .

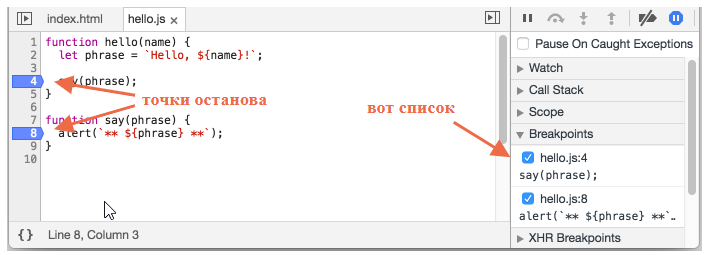
[**Консоль**](https://learn.javascript.ru/debugging-chrome#konsol)

При нажатии на клавишу Esc в нижней части экрана появляется консоль, где можно вводить команды и выполнять их клавишей Enter. Результат выполнения инструкций сразу же отображается в консоли. Например, результатом 1+2 будет 3, а инструкция hello("debugger") ничего не возвращает, так что получаем undefined:



**[Точки останова (breakpoints)](https://learn.javascript.ru/debugging-chrome" \l "tochki-ostanova-breakpoints)**

Точки останова позволяют разобраться как работает код. Чтобы поставить точку останова необходимо щёлкнуть по номеру строчки. Номер строки будет окрашен в синий цвет. Вот что в итоге должно получиться:



Точка останова – это участок кода, где отладчик автоматически приостановит исполнение JavaScript. Пока исполнение поставлено «на паузу», можно просмотреть текущие значения переменных, выполнить команды в консоли и т.д., т.е. выполнить отладку кода.

Список точек останова располагается в правой части графического интерфейса. Когда точек выставлено много, да ещё и в разных файлах, этот список поможет эффективно ими управлять:

* Быстро переместиться к любой точке останова в коде – нужно щёлкнуть по точке в правой части экрана.
* Временно деактивировать точку – в общем списке нужно снять галочку напротив ненужной в данный момент точки.
* Удалить точку – щёлкнуть по ней правой кнопкой мыши и выбрать Remove (Удалить).
* и так далее.

Можно задать и так называемую условную точку останова. Для этого необходимо щёлкнуть правой кнопкой мыши по номеру строки в коде. Если задать выражение, то именно при его истинности выполнение кода будет приостановлено.

Этот метод используется, когда выполнение кода нужно остановить при присвоении определённого выражения какой-либо переменной или при определённых параметрах функции.

[**Команда Debugger**](https://learn.javascript.ru/debugging-chrome#komanda-debugger)

Выполнение кода можно также приостановить с помощью команды debugger прямо изнутри самого кода:

function hello(name) {

let phrase = `Привет, ${name}!`;

debugger; // <-- здесь выполнение прерывается

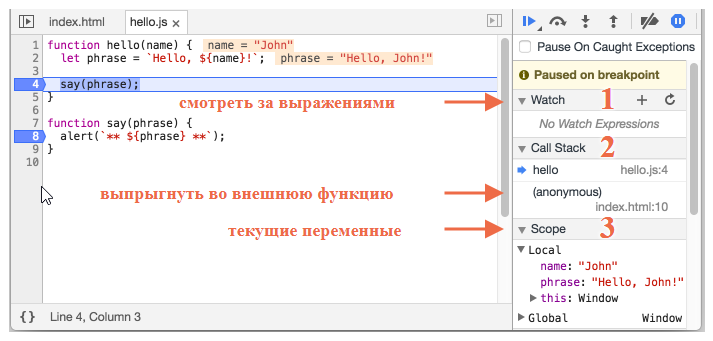
say(phrase);

}

Способ удобен тем, что можно продолжить работать в редакторе кода без необходимости переключения в браузер для выставления точки останова.

[**Пошаговое выполнение скрипта**](https://learn.javascript.ru/debugging-chrome#poshagovoe-vypolnenie-skripta)

В примере функция hello() вызывается во время загрузки страницы, поэтому для начала отладки (после того, как выставлены точки останова) необходимо её перезагрузить (F5 (Windows, Linux) или Cmd+R (Mac)). Выполнение прервётся на четвёртой строчке:



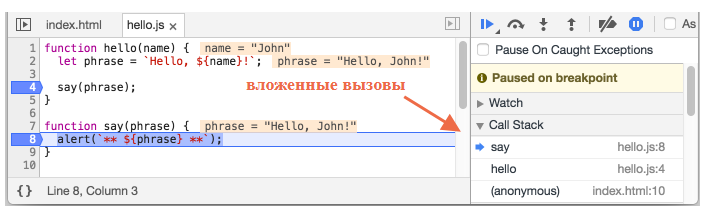
Содержимое панели справа:

1. *Watch* показывает текущие значения выражений. Чтобы ввести выражение необходимо нажать на +. В процессе выполнения отладчик автоматически пересчитывает и выводит его значение.
2. *Call Stack* показывает последовательность вызовов функций. Так как в примере отладчик работает с функцией hello(), вызванной скриптом из файла index.html, и там нет функции, то вызов «анонимный». При нажатии на элемент списка (например, на «anonymous») отладчик переходит к соответствующему коду, и его можно проанализировать.
3. *Scope* показывает текущие переменные. В Local отображаются локальные переменные функций, а их значения подсвечены в исходном коде. В Global перечисляются глобальные переменные (т.е. объявленные за пределами функций).

Для отладки кода по шагам в правой части панели есть несколько кнопок:

 –  продолжить выполнение. Быстрая клавиша – F8.

Возобновляет выполнение кода. Если больше нет точек останова, отладчик прекращает работу и позволяет приложению работать дальше:



Выполнение кода возобновилось, дошло до другой точки останова внутри say(), и отладчик снова приостановил выполнение. В списке Call stack (справа) появился ещё один вызов. Выполнение кода перешло внутрь функции say().

 –  сделать шаг (выполнить следующую команду), не заходя в функцию (полностью игнорируя её содержимое). Быстрая клавиша –F10.

 –  сделать шаг. Быстрая клавиша – F11.

В отличие от предыдущей команды, здесь выполнение «заходит» во вложенные функции и шаг за шагом проходим по скрипту.

 –  продолжить выполнение до завершения текущей функции. Быстрая клавиша – Shift+F11.

Выполнение кода остановится на самой последней строчке текущей функции. Используется, когда разработчик случайно нажал и зашел в функцию и хочет из неё выйти.

–  активировать/деактивировать все точки останова.

Эта кнопка не влияет на выполнение кода, она лишь позволяет массово включить/отключить точки останова.

 –  разрешить/запретить остановку выполнения в случае возникновения ошибки.

Если опция включена и инструменты разработчика открыты, любая ошибка в скрипте приостанавливает выполнение кода, что позволяет его проанализировать. Поэтому если скрипт завершается с ошибкой, необходимо открыть отладчик, включить эту опцию, перезагрузить страницу и локализовать проблему.

Если щёлкнуть правой кнопкой мыши по строчке кода, в контекстном меню можно выбрать опцию *«Continue to here»* («продолжить до этого места»). Этот метод используется, когда нужно продвинуться на несколько шагов вперёд до нужной строки, но не хочется выставлять точки останова.

[**Логирование**](https://learn.javascript.ru/debugging-chrome#logirovanie)

Если нужно что-то вывести в консоль из кода, применяется функция console.log. К примеру, выведем в консоль значения от нуля до четырёх:

for (let i = 0; i < 5; i++) {

console.log("значение", i);

}

Консоль можно открыть через инструменты разработчика – выбрав вкладку «Консоль» или нажать Esc, находясь в другой вкладке – консоль откроется в нижней части интерфейса. Если правильно выстроить логирование в приложении, то можно и без отладчика разобраться, что происходит в коде.

**3. Синтаксис JavaScript**

Код должен быть максимально читаемым и понятным. Для этого нужен хороший стиль написания кода.

**К**[**оманды**](https://learn.javascript.ru/structure#komandy)

Каждая команда пишется на отдельной строке – так код лучше читается:

alert('Привет');

alert('Мир');

**То**[**чка с запятой**](https://learn.javascript.ru/structure#semicolon)

В JavaScript рекомендуется точку с запятой ставить в конце команды. Это не требование стандарта, но в некоторых ситуациях может привести к ошибкам.

Рассмотрим некоторые примеры. Точку с запятой во многих случаях можно не ставить, если есть переход на новую строку:

alert('Привет')

alert('Мир')

В этом случае JavaScript интерпретирует переход на новую строчку как разделитель команд и автоматически вставляет «виртуальную» точку с запятой между ними. Однако, так происходит не всегда. Например, этот код:

alert(3 +

1

+ 2);

выведет 6. То есть, точка с запятой не ставится, так как это незавершённое выражение, конца которого JavaScript ждёт с первой строки.

Но в некоторых важных ситуациях JavaScript не вставит точку с запятой там, где она нужна. Например, такой код работать не будет:

alert("Сейчас будет ошибка")

[1, 2].forEach(alert)

Выведется первый alert, а дальше – ошибка. Потому что перед квадратной скобкой JavaScript точку с запятой не ставит, а она здесь нужна.

**К**[**омментарии**](https://learn.javascript.ru/structure#kommentarii)

Со временем программа становится большой и сложной. Появляется необходимость добавить комментарии, которые объясняют, что происходит и почему.

Один из показателей хорошего разработчика – качество комментариев, которые позволяют эффективно поддерживать код, возвращаться к нему после любой паузы и легко вносить изменения.

Должен быть минимум комментариев, которые отвечают на вопрос "что происходит в коде?". Хороший код и так понятен. Если вам кажется, что нужно добавить комментарий для улучшения понимания, это значит, что ваш код недостаточно прост и стоит его переписать.

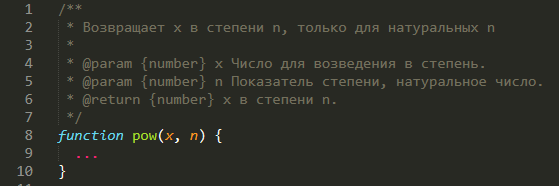
Комментарий должен содержать:

* Архитектурный комментарий – «как оно, вообще, устроено».

Какие компоненты есть, какие технологии использованы, поток взаимодействия. О чём и зачем этот скрипт. Эти комментарии особенно нужны, если вы не один, а проект большой.

* Справочный комментарий перед функцией – о том, что именно она делает, какие параметры принимает и что возвращает.

Для таких комментариев существует синтаксис [JSDoc](http://en.wikipedia.org/wiki/JSDoc).



Такие комментарии позволяют сразу понять, что принимает и что делает функция, не вникая в код. Они автоматически обрабатываются многими редакторами, например [Aptana](http://aptana.com/) и редакторами от [JetBrains](http://www.jetbrains.com/), которые учитывают их при автодополнении, а также выводят их в автоподсказках при наборе кода. Есть инструменты, например [JSDoc 3](https://github.com/jsdoc3/jsdoc), которые умеют генерировать по таким комментариям документацию в формате HTML.

Более важными могут быть комментарии, которые объясняют не *что*, а *почему* в коде происходит именно это. Например, есть несколько способов решения задачи. Вы выбрали именно это решение. Вы пробовали решить задачу по-другому, но не получилось – напишите об этом. Особенно это важно в тех случаях, когда используется не первый приходящий в голову способ, а какой-то другой. Вы открываете код, который был написан какое-то время назад, и видите, что он «неоптимален», захотите его переписать. Только этот вариант вы уже обдумали раньше. И отказались, а почему – забыли. Комментарии, которые объясняют выбор решения, очень важны. Они помогают понять происходящее и предпринять правильные шаги при развитии кода.

Комментарии могут находиться в любом месте программы и никак не влияют на её выполнение. Интерпретатор JavaScript попросту игнорирует их.

*Однострочные комментарии* начинаются с двойного слэша //. Текст считается комментарием до конца строки:

// Команда ниже говорит "Привет"

alert( 'Привет' );

alert( 'Мир' ); // Второе сообщение выводим отдельно

*Многострочные комментарии* начинаются слешем-звездочкой «/\*» и заканчиваются звездочкой-слэшем «\*/», вот так:

/\* Пример с двумя сообщениями.

Это - многострочный комментарий.

\*/

alert( 'Привет' );

alert( 'Мир' );

Всё содержимое комментария игнорируется. Если поместить код внутрь   
/\* … \*/ или после // – он не выполнится.

/\* Закомментировали код

alert( 'Привет' );

\*/

alert( 'Мир' );

В большинстве редакторов комментарий можно поставить горячей клавишей, обычно это Ctrl+/ для однострочных и Ctrl+Shift+/ – для многострочных комментариев (нужно выделить блок и нажать сочетание клавиш). Вложенные комментарии не поддерживаются.

В этом коде будет ошибка:

/\*

/\* вложенный комментарий ?!? \*/

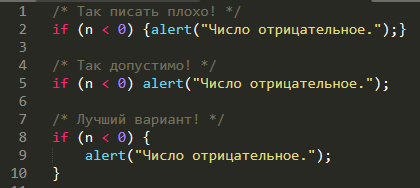
\*/

alert('Мир');

Не бойтесь комментариев. Чем больше кода в проекте – тем они важнее. Что же касается увеличения размера кода – это не страшно, т.к. существуют инструменты сжатия JavaScript, которые при публикации кода легко их удалят.

[**Фигурные скобки**](https://learn.javascript.ru/coding-style#figurnye-skobki)

Фигурные скобки располагаются на той же строке, это так называемый «египетский» стиль. Перед скобкой ставится пробел.



**[Длина строки](https://learn.javascript.ru/coding-style" \l "dlina-stroki)**

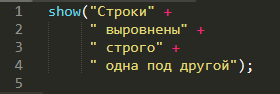
Максимальную длину строки согласовывают в команде. Как правило, это либо 80, либо 120 символов, в зависимости от того, какие мониторы у разработчиков. Более длинные строки необходимо разбивать для улучшения читаемости.

**[Отступы](https://learn.javascript.ru/coding-style" \l "otstupy)**

Отступы нужны двух типов:

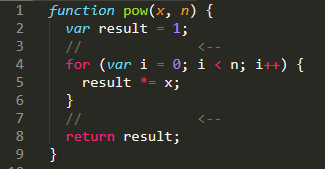
*Горизонтальный отступ, при вложенности – два (или четыре) пробела.*

Как правило, используются именно пробелы, т.к. они позволяют установить более точное выравнивание, чем символ «Tab», например, выровнять аргументы относительно открывающей скобки:



*Вертикальный отступ, для лучшей разбивки кода – перевод строки.*

Используется, чтобы разделить логические блоки внутри одной функции. В примере разделены инициализация переменных, главный цикл и возвращение результата:



Вставляйте дополнительный перевод строки туда, где это сделает код более читаемым. Не должно быть более 9 строк кода подряд без вертикального отступа.

[**Именование**](https://learn.javascript.ru/coding-style#imenovanie)

Общее правило:

Имя переменной – существительное.

Имя функции – глагол или начинается с глагола.

Для имён используется английский язык (не транслит) и верблюжья нотация.

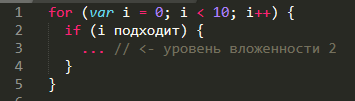
Верблюжья нотация (CamelCase) – стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово внутри фразы пишется с прописной буквы.

**Ур**[**овни вложенности**](https://learn.javascript.ru/coding-style#urovni-vlozhennosti)

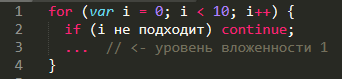
Уровней вложенности должно быть немного.

Например, [проверки в циклах можно делать через «continue»](https://learn.javascript.ru/while-for#continue), чтобы не было дополнительного уровня if(..) { ... }.

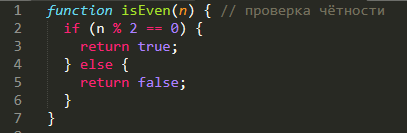
Вместо:



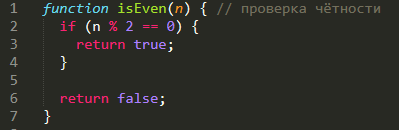
Используйте:



Аналогичная ситуация – с if/else и return. Следующие две конструкции идентичны. Первая:



Вторая:

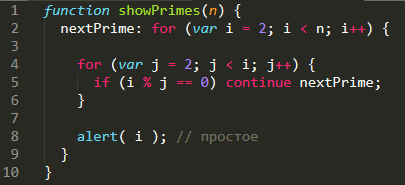


Если в блоке if есть return, то else за ним не нужен.

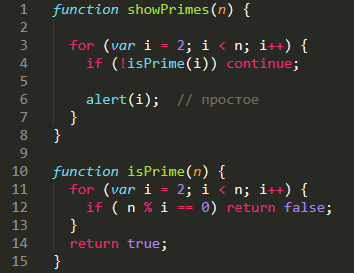
Рекомендуется проверять сначала простые условия, вернуть результат, а затем обрабатывать более сложные, без дополнительного уровня вложенности.

**Функции**

Функции должны быть небольшими. Если функция большая – желательно разбить её на несколько. Сравните, например, две функции showPrimes(n) для вывода простых чисел до n. В первом варианте используется метка:

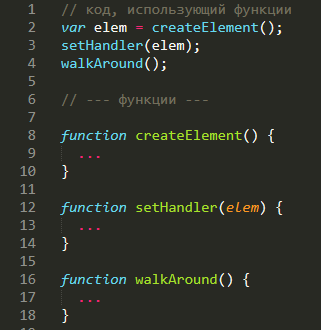


Во втором варианте используется дополнительная функция isPrime(n) для проверки на простоту:



Второй вариант проще и понятнее. Вместо участка кода видно описание действия, которое там совершается (проверка isPrime).

Рекомендуется располагать функции под кодом, который их использует, так как при чтении кода разработчик в первую очередь хочет знать, что код делает, а уже затем какие функции реализованы*.*



**[Руководства по стилю](https://learn.javascript.ru/coding-style" \l "rukovodstva-po-stilyu)**

Когда написанием проекта занимается целая команда, то должен существовать один стандарт кода, описывающий где и когда ставить пробелы, запятые, переносы строк и т.п. Существует много готовых руководст, которые можно использовать. Большинство из них на английском языке:

* [Google JavaScript Style Guide](https://google.github.io/styleguide/jsguide.html);
* [jQuery JavaScript Style Guide](http://contribute.jquery.org/style-guide/js/);
* [Airbnb JavaScript Style Guide](https://github.com/airbnb/javascript);
* [Idiomatic.JS](https://github.com/rwaldron/idiomatic.js);
* [Dojo Style Guide](https://dojotoolkit.org/reference-guide/1.10/developer/styleguide.html).

**[Автоматизированные средства проверки](https://learn.javascript.ru/coding-style" \l "avtomatizirovannye-sredstva-proverki) стиля**

Существуют средства, проверяющие стиль кода. Самые известные – это:

* [JSLint](http://www.jslint.com/) – проверяет код на соответствие [стилю JSLint](http://www.jslint.com/lint.html), в онлайн-интерфейсе можно ввести код и различные настройки проверки, чтобы сделать её более мягкой.
* [JSHint](http://www.jshint.com/) – это JSLint с большим количеством настроек.

В частности, JSLint и JSHint интегрированы с большинством редакторов, они гибко настраиваются под нужный стиль и совершенно незаметно улучшают разработку, подсказывая, где и что поправить.

**Полифилы**

JavaScript – динамично развивающийся язык программирования. Регулярно появляются новые возможности. Разработчики JavaScript-движков сами принимают решение, какие возможности реализовать в первую очередь и часто реализуется только часть стандарта. Поэтому современные возможности JavaScript, некоторые движки могут не поддерживать.

Новые возможности языка могут включать встроенные функции и синтаксические конструкции. Существуют [транспилер](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BF%D0%B0%D0%B9%D0%BB%D0%B5%D1%80)ы (например, [Babel](https://babeljs.io/)), которые переписывают синтаксические конструкции в старые. Новые встроенные функции, нужно реализовать с помощью полифилов.

*Полифил* – это код, реализующий какую-либо функциональность, которая не поддерживается в некоторых версиях веб-браузеров.

Два популярных полифила:

* [core.js](https://github.com/zloirock/core-js) поддерживает много функций, можно подключать только нужные.
* [polyfill.io](http://polyfill.io/) – сервис, который автоматически создаёт скрипт с полифилом в зависимости от необходимых функций и браузера пользователя.

Таким образом, чтобы современные функции поддерживались в старых движках, необходимо установить транспилер и добавить полифил.

[**«use strict»**](https://learn.javascript.ru/strict-mode#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 старый код работал, большинство таких модификаций по умолчанию отключены. Необходимо явно включить их с помощью специальной директивы: "use strict". Когда она находится в верхней части скрипта, весь сценарий работает в «современном» режиме.

Например:

"use strict";

// этот код работает в современном режиме

...

"use strict" можно расположить в начале большинства видов функций, вместо всего скрипта. Это позволяет включить строгий режим только в этой функции. Но обычно, он используется для всего файла.

Нет какой-либо директивы или другого способа, который возвращает движок к старому поведению. Как только включается строгий режим, отменить это невозможно.

В консоле браузера use strict по умолчанию выключен.

1. **Переменные.**

Для создания переменной в JavaScript, используйте ключевое слово let:

let message;

message = 'Hello'; // сохранить строку

Можно объявить несколько переменных в одной строке, но это не рекомендуется, так как ухудшает читаемость кода:

let user = 'John', age = 25, message = 'Hello';

Многострочный вариант немного длиннее, но легче для чтения:

let user = 'John';

let age = 25;

let message = 'Hello';

Можно объявлять переменные на нескольких строках и даже с запятой в начале строки:

let user = 'John',

age = 25,

message = 'Hello';

let user = 'John'

, age = 25

, message = 'Hello';

В старых скриптах можно найти ключевое слово: var вместо let:

var message = 'Hello';

Ключевое слово var – почти то же самое, что и let. Оно объявляет переменную, но немного по-другому, «устаревшим» способом.

[**Имена переменных**](https://learn.javascript.ru/variables#variable-naming)

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

Именование переменных – это один из самых важных и сложных навыков в программировании. Быстрый взгляд на имена переменных может показать, какой код был написан новичком, а какой опытным разработчиком.

В реальном проекте большая часть времени тратится на изменение и расширение существующей кодовой базы, а не на написание чего-то совершенно нового с нуля. Когда мы возвращаемся к коду после какого-то промежутка времени, гораздо легче найти информацию, которая хорошо размечена. Или, другими словами, когда переменные имеют хорошие имена.

Несколько советов:

* Используйте легко читаемые имена, такие как userName или shoppingCart.
* Избегайте использования аббревиатур или коротких имён, таких как a, b, c, за исключением тех случаев, когда вы точно знаете, что так нужно.
* Делайте имена максимально описательными и лаконичными. Примеры плохих имён: data и value. Такие имена ничего не говорят. Их можно использовать только в том случае, если из контекста кода очевидно, какие данные хранит переменная.
* Договоритесь с вашей командой о используемых терминах. Если посетитель сайта называется «user» тогда мы должны назвать связанные с ним переменные currentUser или newUser вместо того, чтобы называть их currentVisitorили newManInTown.

В JavaScript есть два ограничения, касающиеся имён переменных:

1. Имя переменной должно содержать только буквы, цифры или символы $ и \_.
2. Первый символ не должен быть цифрой.

Если имя содержит несколько слов, обычно используется [верблюжья нотация](https://ru.wikipedia.org/wiki/CamelCase), например,  myVeryLongName.

Примеры допустимых имён:

let userName;

let test123;

let $ = 1; // объявили переменную с именем "$"

let \_ = 2; // а теперь переменную с именем "\_"

Примеры неправильных имён переменных:

let 1a; // не может начинаться с цифры

let my-name; // дефис '-' не разрешён в имени

Регистр имеет значение. Переменные с именами apple and AppLE – это две разные переменные.

Нелатинские буквы разрешены, но не рекомендуются. Можно использовать любой язык, включая кириллицу или даже иероглифы, например:

let имя = '...';

let 我 = '...';

Технически здесь нет ошибки, такие имена разрешены, но есть международная традиция использовать английский язык в именах переменных.

Существует [список зарезервированных слов](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Lexical_grammar#%D0%9A%D0%BB%D1%8E%D1%87%D0%B5%D0%B2%D1%8B%D0%B5_%D1%81%D0%BB%D0%BE%D0%B2%D0%B0), которые нельзя использовать в качестве имён переменных, потому что они используются самим языком. Например: let, class, return и function зарезервированы. Приведённый ниже код даёт синтаксическую ошибку:

let let = 5; // нельзя назвать переменную "let", ошибка!

let return = 5; // также нельзя назвать переменную "return", ошибка!

Обычно нужно определить переменную перед её использованием. Но старые стандарты позволяли создавать переменную простым присвоением значения без использования let. Это все ещё работает, если не включить use strict, для поддержания совместимости со старыми скриптами.

// заметка: "use strict" в этом примере не используется

num = 5; // если переменная "num" не существовала, она создаётся

alert(num); // 5

Это плохая практика и приведёт к ошибке в строгом режиме:

"use strict";

num = 5; // error: num is not defined

[**Константы**](https://learn.javascript.ru/variables#konstanty)

Чтобы объявить константную переменную необходимо использовать const вместо let:

const myBirthday = '18.04.1982';

Изменение константы приведет к ошибке:

const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // ошибка, константу нельзя перезаписать!

Широко распространена практика использования констант в качестве псевдонимов для трудно запоминаемых значений, которые известны до начала исполнения скрипта. Названия таких констант пишутся с использованием заглавных букв и подчёркивания, например, для различных цветов в шестнадцатиричном формате:

const COLOR\_RED = "#F00";

const COLOR\_GREEN = "#0F0";

const COLOR\_BLUE = "#00F";

const COLOR\_ORANGE = "#FF7F00";

let color = COLOR\_ORANGE;

alert(color); // #FF7F00

Преимущества такого подхода:

* COLOR\_ORANGE гораздо легче запомнить, чем "#FF7F00".
* Гораздо легче допустить ошибку при вводе "#FF7F00", чем при вводе COLOR\_ORANGE.
* При чтении кода, COLOR\_ORANGE намного понятнее, чем #FF7F00.

Когда мы должны использовать для констант заглавные буквы, а когда называть их нормально? Давайте разберёмся и с этим.

1. **Типы данных.**

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. Есть семь основных типов данных в JavaScript, пять из них называются «примитивными», потому что их значениями могут быть только простые значения (будь то строка или число, или что-то ещё).

[**Число**](https://learn.javascript.ru/types#chislo)

let n = 123;

n = 12.345;

Числовой тип данных (number) представляет как целочисленные значения, так и числа с плавающей точкой. Существует множество операций для чисел, например, умножение \*, деление /, сложение +, вычитание - и так далее.

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

*Infinity* представляет собой математическую [бесконечность](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D1%81%D0%BA%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C#%D0%92_%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B5) ∞. Это особое значение, которое больше любого числа. Мы можем получить его в результате деления на ноль или задать его явно:

alert( 1 / 0 ); // Infinity

alert( Infinity ); // Infinity

*NaN* означает вычислительную ошибку. Это результат неправильной или неопределённой математической операции, например:

alert( "не число" / 2 ); // NaN, такое деление является ошибкой

Любая операция с NaN возвращает NaN:

alert( "не число" / 2 + 5 ); // NaN

Математические операции в JavaScript позволяют делать что угодно: делить на ноль, обращаться со строками как с числами и т.д. Скрипт никогда не остановится с фатальной ошибкой. В худшем случае просто в результате выполнения будет возвращен NaN .

Тип *BigInt* –  представляет собой встроенный объект, который предоставляет способ представлять целые числа больше pow(2, 53) - 1, наибольшего числа, которое JavaScript может надежно представить с Number примитивом.

Чтобы обозначить, что число относится к типу BigInt нужно добавить n в конце числа. n означает, что это BigInt.

let oldMax = Number.MAX\_SAFE\_INTEGER; // 9007199254740991

++oldMax; // 9007199254740992

++oldMax; // 9007199254740992 <- такое же значение

let newMax = 9007199254740992n;

++newMax; // 9007199254740993n

++newMax; // 9007199254740994n

const a = 9007199254740991n; // 9007199254740991n

const b = BigInt(9007199254740991n); // 9007199254740991n

a === b; // true

typeof 10; // "number";

typeof 10n; // "bigint";

[**Строка**](https://learn.javascript.ru/types#stroka)

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

let str = "Привет";

let str2 = 'Одинарные кавычки тоже подойдут';

let phrase = `Обратные кавычки позволяют встраивать переменные ${str}`;

В JavaScript существует три типа кавычек.

* Двойные кавычки: "Привет".
* Одинарные кавычки: 'Привет'.
* Обратные кавычки: `Привет`.

Двойные или одинарные кавычки являются «простыми», между ними нет разницы в JavaScript. Обратные кавычки же имеют расширенный функционал. Они позволяют встраивать выражения в строку, заключая их в ${…}. Например:

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. Можно записать туда всё, что угодно: переменную name или выражение 1 + 2, или что-то более сложное.

В некоторых языках для одного символа существует специальный «символьный» тип. Например, в C и Java это char. В JavaScript подобного типа нет, есть только тип string. Строка может содержать один символ или множество.

[**Булевый (логический) тип**](https://learn.javascript.ru/types#bulevyy-logicheskiy-tip)

Булевый тип (boolean) может принимать только два значения: true (истина) и false (ложь). Такой тип, как правило, используется для хранения значений да/нет: true значит «да, правильно», а false значит «нет, не правильно». Например:

let nameFieldChecked = true; // да, поле отмечено

let ageFieldChecked = false; // нет, поле не отмечено

Булевы значения также могут быть результатом сравнений:

let isGreater = 4 > 1;

alert( isGreater ); // true (результатом сравнения будет "да")

[**Значение «null»**](https://learn.javascript.ru/types#znachenie-null)

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

let age = null;

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое представляет собой «ничего», «пусто» или «значение неизвестно». В приведённом выше коде указано, что переменная age неизвестна или не имеет значения по какой-то причине.

[**Значение «undefined»**](https://learn.javascript.ru/types#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 x;

alert(x); // выведет "undefined"

Технически можно присвоить значение undefined любой переменной:

let x = 123;

x = undefined;

alert(x); // "undefined"

Но так делать не рекомендуется. Обычно null используется для присвоения переменной «пустого» или «неизвестного» значения, а undefined для проверок, была ли переменная назначена.

[**Объекты и символы**](https://learn.javascript.ru/types#obekty-i-simvoly)

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

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

Рассмотрим объекты и символы позднее после того, как изучим подробнее примитивы.

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

Оператор typeof возвращает тип аргумента. Это полезно, когда необходимо обрабатывать значения различных типов по-разному или просто сделать проверку.

У него есть два синтаксиса, результат одинаковый:

* Синтаксис оператора: typeof x.
* Синтаксис функции: typeof(x).

Вызов typeof x возвращает строку с именем типа:

typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

typeof null // "object" (1)

typeof alert // "function" (2)

Результатом вызова typeof null в строке (1) является "object". Это неверно. Это официально признанная ошибка в typeof, сохранённая для совместимости. null не является объектом, это специальное значение с отдельным типом.

Вызов typeof alert в строке (2) возвращает "function", потому что alert является функцией. В JavaScript нет специального типа «функция». Функции относятся к объектному типу. Но typeof обрабатывает их особым образом, возвращая "function". Формально это неверно, но очень удобно на практике.

1. **Преобразование типов**

Чаще всего, операторы и функции автоматически приводят переданные им значения к нужному типу. Например, alert автоматически преобразует любое значение к строке, а математические операторы преобразуют значения к числам. Есть также случаи, когда нужно явно преобразовать значение в ожидаемый тип.

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

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. Например, alert(value) преобразует значение к строке.

Также можно использовать функцию String(value) чтобы преобразовать значение к строке:

let value = true;

alert(typeof value); // boolean

value = String(value); // теперь value это строка "true"

alert(typeof value); // string

Преобразование происходит очевидным способом. False становится "false", null становится "null" и т.п.

Почти все математические операторы выполняют численное преобразование. Исключение составляет +. Если одно из слагаемых является строкой, тогда и все остальные приводятся к строкам и они конкатенируются (присоединяются) друг к другу:

alert( 1 + '2' ); // '12' (строка справа)

alert( '1' + 2 ); // '12' (строка слева)

Так происходит, только если хотя бы один из аргументов является строкой. Во всех остальных случаях, значения складываются как числа.

[**Численное преобразование**](https://learn.javascript.ru/type-conversions#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

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

let str = "123";

alert(typeof str); // string

let num = Number(str); // становится числом 123

alert(typeof num); // number

Явноe преобразование часто применяется, когда необходимо получить число из строковых источников, вроде форм текстового ввода. Если строка не может быть явно приведена к числу, то результатом преобразования будет NaN. Например:

let age = Number("Любая строка вместо числа");

alert(age); // NaN, преобразование не удалось

Правила численного преобразования:

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

Примеры:

alert( Number(" 123 ") ); // 123

alert( Number("123z") ); // NaN (ошибка чтения числа в "z")

alert( Number(true) ); // 1

alert( Number(false) ); // 0

null и undefined ведут себя по-разному: null становится нулём, undefined приводится к NaN.

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

Логическое преобразование происходит в логических операторах но так же может быть выполнено явно с помощью функции Boolean(value).

Правила преобразования:

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

Например:

alert( Boolean(1) ); // true

alert( Boolean(0) ); // false

alert( Boolean("Привет!") ); // true

alert( Boolean("") ); // false

Строка с нулём "0" преобразуется в true. Некоторые языки воспринимают строку "0" как false. Но в JavaScript, если строка не пустая, то она всегда true.

alert( Boolean("0") ); // true

alert( Boolean(" ") ); // пробел это тоже true

1. **Операторы.**

*Операнд* – то, к чему применяется оператор. Например, в умножении 5 \* 2 есть два операнда: левый операнд равен 5, а правый операнд равен 2. Иногда их называют аргументами.

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

let x = 1;

x = -x;

alert( x ); // -1, применили унарный минус

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

let x = 1, y = 3;

alert( y - x ); // 2, бинарный минус

Формально мы говорим о двух разных операторах: унарное отрицание (один операнд: меняет знак) и бинарное вычитание (два операнда: вычитает).

[**Сложение строк, бинарный +**](https://learn.javascript.ru/operators#slozhenie-strok-binarnyy)

Обычно при помощи оператора плюс '+' складывают числа. Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну. Если хотя бы один операнд является строкой, то второй будет также преобразован к строке.

let s = "моя" + "строка";

alert(s); // моястрока

Например:

alert( '1' + 2 ); // "12"

alert( 2 + '1' ); // "21"

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

alert(2 + 2 + '1' ); // будет "41", а не "221"

Сложение и преобразование строк – это особенность бинарного плюса +. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа. Например, вычитание и деление:

alert( 2 - '1' ); // 1

alert( '6' / '2' ); // 3

[**Преобразование к числу, унарный плюс +**](https://learn.javascript.ru/operators#preobrazovanie-k-chislu-unarnyy-plyus)

Плюс + существует в двух формах: бинарной, которая рассматривалась выше, и унарной. Унарный плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число. Это то же самое, что и Number(...), только короче. Например:

// Не влияет на числа

let x = 1;

alert( +x ); // 1

let y = -2;

alert( +y ); // -2

// Преобразует нечисла в число

alert( +true ); // 1

alert( +"" ); // 0

Необходимость преобразовывать строки в числа возникает очень часто. Например, обычно значения полей HTML-формы – это строки. Если их надо сложить, то сначала необходимо преобразовать к числам. Бинарный плюс сложит их как строки:

let apples = "2";

let oranges = "3";

alert( apples + oranges ); // "23"

Поэтому использутся унарный плюс, чтобы преобразовать к числу:

let apples = "2";

let oranges = "3";

// оба операнда предварительно преобразованы в числа

alert( +apples + +oranges ); // 5

// более длинный вариант

// alert( Number(apples) + Number(oranges) ); // 5

Сначала выполнятся унарные плюсы, приведут строки к числам, а затем – бинарный '+' их сложит. Этот порядок определяет приеоритет оператора.

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

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

Известно, что умножение в выражении 2 \* 2 + 1 выполнится раньше сложения. Это потому, что умножение имеет более высокий приоритет, чем сложение.

Скобки важнее, чем приоритет, так что, если необходимо изменить порядок по умолчанию, можно использовать их, чтобы изменить приоритет. Например, написать (1 + 2) \* 2.

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

Ниже представлена часть [таблицы приоритетов](https://developer.mozilla.org/ru/JavaScript/Reference/operators/operator_precedence). Обратите внимание, что у унарных операторов приоритет выше, чем у соответствующих бинарных:

| **Приоритет** | **Название** | **Обозначение** |
| --- | --- | --- |
| … | … | … |
| 16 | унарный плюс | + |
| 16 | унарный минус | - |
| 14 | умножение | \* |
| 14 | деление | / |
| 13 | сложение | + |
| 13 | вычитание | - |
| … | … | … |
| 3 | присваивание | = |
| … | … | … |

Так как «унарный плюс» имеет приоритет 16, который выше, чем 13 у «сложения» (бинарный плюс), то в выражении "+apples + +oranges" сначала выполнятся унарные плюсы, а затем сложение.

**[Присваивание](https://learn.javascript.ru/operators" \l "prisvaivanie)**

В таблице приоритетов также есть 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 присвоится 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. Писать в таком стиле не рекомендуется, так как это делает ваш код менее понятным и читабельным.

**[Остаток от деления %](https://learn.javascript.ru/operators" \l "ostatok-ot-deleniya)**

Оператор взятия остатка в выражении a % b возвращает остаток от деления a на b. Например:

alert( 5 % 2 ); // 1, остаток от деления 5 на 2

alert( 8 % 3 ); // 2, остаток от деления 8 на 3

alert( 6 % 3 ); // 0, остаток от деления 6 на 3

[**Возведение в степень \*\***](https://learn.javascript.ru/operators#vozvedenie-v-stepen)

Оператор возведения в степень \*\* недавно добавили в язык. Для натурального числа b результат a \*\* b равен a, умноженному на само себя b раз. Например:

alert( 2 \*\* 2 ); // 4 (2 \* 2)

alert( 2 \*\* 3 ); // 8 (2 \* 2 \* 2)

alert( 2 \*\* 4 ); // 16 (2 \* 2 \* 2 \* 2)

Оператор работает и для нецелых чисел. Например:

alert( 4 \*\* (1/2) ); // 2

alert( 8 \*\* (1/3) ); // 2

[**Инкремент/декремент**](https://learn.javascript.ru/operators#inkrement-dekrement)

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу. Для этого существуют даже специальные операторы:

*Инкремент* ++ увеличивает на 1:

let counter = 2;

counter++; // работает как counter = counter + 1, просто запись короче

alert( counter ); // 3

*Декремент* -- уменьшает на 1:

let counter = 2;

counter--; // работает как counter = counter - 1, просто запись короче

alert( counter ); // 1

Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа 5++, вернёт ошибку.

Операторы ++ и -- могут быть расположены не только после, но и до переменной. Когда оператор идёт после переменной – это *постфиксная форма*: counter++. *Префиксная форма* – это когда оператор идёт перед переменной: ++counter. Обе эти формы записи делают одно и то же: увеличивают counter на 1. Разница в том, что префиксная форма возвращает новое значение, а постфиксная форма возвращает старое значение (до увеличения/уменьшения числа). Чтобы увидеть разницу, вот небольшой пример:

let counter = 1;

let a = ++counter; // (\*)

alert(a); // 2

В строке (\*) префиксная форма увеличения counter, она возвращает новое значение 2.

Пример использования постфиксной формы:

let counter = 1;

let a = counter++; // (\*)

alert(a); // 1

В строке (\*) постфиксная форма counter++ также увеличивает counter, но возвращает старое значение, которое было до увеличения. Так что alert покажет 1.

Операторы ++/-- могут также использоваться внутри выражений. Их приоритет выше, чем у арифметических операций. Например:

let counter = 1;

alert( 2 \* ++counter ); // 4

Сравните с:

let counter = 1;

alert( 2 \* counter++ ); // 2

Такая запись делает код менее читабельным. При беглом чтении кода можно с лёгкостью пропустить counter++, и будет неочевидно, что переменая увеличивается. Лучше использовать стиль «одна строка – одно действие»:

let counter = 1;

alert( 2 \* counter );

counter++;

[**Побитовые операторы**](https://learn.javascript.ru/operators#pobitovye-operatory)

Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), на уровне их внутреннего двоичного представления. Эти операторы не являются чем-то специфичным для JavaScript, они поддерживаются в большинстве языков программирования. Поддерживаются следующие побитовые операторы:

* AND (и) ( & )
* OR (или) ( | )
* XOR (побитовое исключающее или) ( ^ )
* NOT (не) ( ~ )
* LEFT SHIFT (левый сдвиг) ( << )
* RIGHT SHIFT (правый сдвиг) ( >> )
* ZERO-FILL RIGHT SHIFT (правый сдвиг с заполнением нулями) ( >>> )

Они используются редко. Чтобы понять их, нужно углубиться в низкоуровневое представление чисел.

[**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

[**Оператор запятая**](https://learn.javascript.ru/operators#operator-zapyataya)

Оператор «запятая», редко используется и является одним из самых необычных. Иногда он используется для написания более короткого кода, поэтому нужно знать его, чтобы понимать, что при этом происходит.

Оператор запятая предоставляет возможность вычислять несколько выражений, разделяя их запятой. Каждое выражение выполняется, но возвращается результат только последнего. Например:

let a = (1 + 2, 3 + 4);

alert( a ); // 7

Первое выражение 1 + 2 выполняется, а результат отбрасывается. Затем идёт 3 + 4, выражение выполняется и возвращается результат.

Оператор запятая имеет очень низкий приоритет, приоритет которого ниже =, поэтому скобки важны в приведённом примере выше. Без них в a = 1 + 2, 3 + 4 сначала выполнится +, суммируя числа в a = 3, 7, затем оператор присваивания = присвоит a = 3, а то что идёт дальше, будет игнорировано. Всё так же, как в (a = 1 + 2), 3+4.

Этот оператор иногда используют в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:

// три операции в одной строке

for (a = 1, b = 3, c = a \* b; a < 10; a++) {

...

}

Такое написание кода используется во многих JavaScript-фреймворках и о нем стоит знать. Но обычно это не улучшает читабельность кода, поэтому этот опреатор не рекомендуется использовать.

1. **Операторы сравнения.**

Операторы сравнения известные из математики:

* Больше/меньше: a > b, a < b.
* Больше/меньше или равно: a >= b, a <= b.
* Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства =. Один знак равенства a = b означал бы присваивание.
* Не равно. В математике обозначается символом ≠. В JavaScript записывается как знак равенства с предшествующим ему восклицательным знаком: a != b.

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

* true – означает «да», «верно», «истина».
* false – означает «нет», «неверно», «ложь».

Например:

alert( 2 > 1 ); // true (верно)

alert( 2 == 1 ); // false (неверно)

alert( 2 != 1 ); // true (верно)

Результат сравнения можно присвоить переменной, как и любое значение:

let result = 5 > 4; // результат сравнения присваивается переменной result

alert( result ); // true

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

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

alert( 'Я' > 'А' ); // true

alert( 'Кот' > 'Код' ); // true

alert( 'Сонный' > 'Сон' ); // true

Алгоритм сравнения двух строк довольно прост:

1. Сначала сравниваются первые символы строк.
2. Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй.
3. Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
4. Сравнение продолжается, пока не закончится одна из строк.
5. Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

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

1. К равна К.
2. о равна о.
3. т больше чем д. На этом сравнение заканчивается. Первая строка больше.

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

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

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

alert( '2' > 1 ); // true, строка '2' становится числом 2

alert( '01' == 1 ); // true, строка '01' становится числом 1

Логическое значение true становится 1, а false – 0. Например:

alert( true == 1 ); // true

alert( false == 0 ); // true

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

let a = 0;

alert( Boolean(a) ); // false

let b = "0";

alert( Boolean(b) ); // true

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

С точки зрения JavaScript, результат ожидаем. Равенство преобразует значения, используя числовое преобразование, поэтому "0" становится 0. В то время как явное преобразование с помощью Boolean использует другой набор правил.

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

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

alert( 0 == false ); // true

Та же проблема с пустой строкой:

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

Это происходит из-за того, что операнды разных типов преобразуются оператором == к числу. В итоге, и пустая строка, и false становятся нулём.

Оператор строгого равенства === проверяет равенство без приведения типов. Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования:

alert( 0 === false ); // false, так как сравниваются разные типы

Оператор строгого равенства делает код более очевидным и оставляет меньше мест для ошибок.

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

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

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

* При строгом равенстве === эти значения различны, так как различны их типы.

alert( null === undefined ); // false

* При нестрогом равенстве == эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

alert( null == undefined ); // true

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

Сравнение null с нулём:

alert( null > 0 ); // (1) false

alert( null == 0 ); // (2) false

alert( null >= 0 ); // (3) true

Результат последнего сравнения говорит о том, что "null больше или равно нулю", тогда результат одного из сравнений выше должен быть true, но они оба ложны. Причина в том, что нестрогое равенство и сравнения >, <, >=, <= работают по-разному. Сравнения преобразуют null в число, рассматривая его как 0. Поэтому выражение (3) null >= 0 истинно, а null > 0 ложно.

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

Значение undefined несравнимо с другими значениями:

alert( undefined > 0 ); // false (1)

alert( undefined < 0 ); // false (2)

alert( undefined == 0 ); // false (3)

Сравнение undefined с нулём всегда ложно, по следующим причинам:

* Сравнения (1) и (2) возвращают false, потому что undefined преобразуется в NaN, а NaN – это специальное числовое значение, которое возвращает false при любых сравнениях.
* Нестрогое равенство (3) возвращает false, потому что undefined равно только null и ни чему больше.

Относитесь к любому сравнению с undefined/null, кроме строгого равенства ===, с осторожностью. Не используйте сравнения >=, >, <, <= с переменными, которые могут принимать значения null/undefined, если вы не уверены в том, что делаете. Если переменная может принимать эти значения, то добавьте для них отдельные проверки.

1. **Модальные окна**

Рассмотрим базовые UI операции: alert, prompt и confirm, которые позволяют работать с данными, полученными от пользователя.

[**alert**](https://learn.javascript.ru/uibasic#alert)

Синтаксис:

alert(сообщение)

alert выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «ОК».

alert( "Привет" );

Окно сообщения, которое выводится, является *модальным окном*. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберётся с окном. В данном случае – пока не нажмёт на «OK».

**[prompt](https://learn.javascript.ru/uibasic" \l "prompt)**

Функция prompt принимает два аргумента:

result = prompt(title, default);

Она выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL. Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием Esc на клавиатуре. Вызов prompt возвращает то, что ввёл посетитель – строку или специальное значение null, если ввод отменён. Как и в случае с alert, окно prompt модальное.

var years = prompt('Сколько вам лет?', 100);

alert('Вам ' + years + ' лет!')

Всегда указывайте default. Второй параметр может отсутствовать. Однако при этом браузер IE вставит в диалог значение по умолчанию "undefined". Поэтому рекомендуется *всегда* указывать второй аргумент, хотябы пустую строку.

[**confirm**](https://learn.javascript.ru/uibasic#confirm)

Синтаксис:

result = confirm(question);

confirm выводит окно с вопросом question с двумя кнопками: OK и CANCEL. Результатом будет true при нажатии OK и false – при CANCEL (Esc). Например:

var isAdmin = confirm("Вы - администратор?");

alert( isAdmin );

1. **Условные операторы.**

Чтобы выполнить различные действия в зависимости от условий, нам нужно использовать оператор if и условный оператор ?, который также называют «оператор вопросительный знак».

**[Оператор «if»](https://learn.javascript.ru/ifelse" \l "operator-if)**

Оператор if(...) вычисляет условие в скобках и, если результат true, то выполняет блок кода. Например:

let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');

if (year == 2015) alert( 'Вы правы!' );

В примере выше, условие – это простая проверка на равенство (year == 2015), но оно может быть и гораздо более сложным. Если надо выполнить более одной инструкции, то нужно заключить блок кода в фигурные скобки:

if (year == 2015) {

alert( "Правильно!" );

alert( "Вы такой умный!" );

}

Рекомендуется использовать фигурные скобки {} всегда, когда используется оператор if, даже если выполняется только одна команда. Это улучшает читабельность кода.

[**Преобразование к логическому типу**](https://learn.javascript.ru/ifelse#preobrazovanie-k-logicheskomu-tipu)

Оператор if (…) вычисляет выражение в скобках и преобразует результат к логическому типу. Вспомним правила преобразования типов:

* Число 0, пустая строка "", null, undefined и NaN становятся false. Из-за этого их называют «ложными» значениями.
* Остальные значения становятся true, поэтому их называют «правдивыми».

Таким образом, код при таком условии никогда не выполнится:

if (0) {

...

}

А при таком – выполнится всегда:

if (1) {

...

}

Также можно передать заранее вычисленное в переменной логическое значение в if, например, так:

let cond = (year == 2015); // преобразуется к true или false

if (cond) {

...

}

[**Блок «else»**](https://learn.javascript.ru/ifelse#blok-else)

Оператор if может содержать необязательный блок «else» («иначе»). Выполняется, когда условие ложно. Например:

let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');

if (year == 2015) {

alert( 'Да вы знаток!' );

} else {

alert( 'А вот и неправильно!' ); // любое значение, кроме 2015

}

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

Иногда, нужно проверить несколько вариантов условия. Для этого используется блок else if. Например:

let year = prompt('В каком году появилась спецификация ECMAScript-2015?', '');

if (year < 2015) {

alert( 'Это слишком рано...' );

} else if (year > 2015) {

alert( 'Это поздновато' );

} else {

alert( 'Верно!' );

}

В приведённом выше коде, JavaScript сначала проверит year < 2015. Если это неверно, он переходит к следующему условию year > 2015. Если оно тоже ложно, тогда сработает последний alert. Блоков else if может быть и больше. Присутствие блока else не является обязательным.

**[Условный оператор „?“](https://learn.javascript.ru/ifelse" \l "uslovnyy-operator)**

Иногда нужно назначить переменную в зависимости от условия. Например:

let accessAllowed;

let age = prompt('Сколько вам лет?', '');

if (age > 18) {

accessAllowed = true;

} else {

accessAllowed = false;

}

alert(accessAllowed);

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

let result = условие ? значение1 : значение2;

Сначала вычисляется условие: если оно истинно, тогда возвращается значение1, в противном случае – значение 2. Например:

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

Технически, мы можем опустить круглые скобки вокруг age > 18. Оператор вопросительного знака имеет низкий приоритет, поэтому он выполняется после сравнения >. Этот пример будет делать то же самое, что и предыдущий:

// оператор сравнения "age > 18" выполняется первым в любом случае

// (нет необходимости заключать его в скобки)

let accessAllowed = age > 18 ? true : false;

Но скобки делают код более читабельным, поэтому рекомендуется их использовать.

В примере выше, можно избежать использования оператора вопросительного знака ?, т.к. сравнение само по себе уже возвращает true/false:

// то же самое

let accessAllowed = age > 18;

[**Несколько операторов „?“**](https://learn.javascript.ru/ifelse#neskolko-operatorov)

Последовательность операторов вопросительного знака ? позволяет вернуть значение, которое зависит от более чем одного условия. Например:

let age = prompt('Возраст?', 18);

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

(age < 18) ? 'Привет!' :

(age < 100) ? 'Здравствуйте!' :

'Какой необычный возраст!';

alert( message );

Поначалу может быть сложно понять, что происходит. Но при ближайшем рассмотрении видно, что это обычная последовательная проверка:

1. Первый знак вопроса проверяет age < 3.
2. Если верно – возвращает 'Здравствуй, малыш!'. В противном случае, проверяет выражение после двоеточия „":"“, вычисляет age < 18.
3. Если это верно – возвращает 'Привет!'. В противном случае, проверяет выражение после следующего двоеточия „":"“, вычисляет age < 100.
4. Если это верно – возвращает 'Здравствуйте!'. В противном случае, возвращает выражение после последнего двоеточия – 'Какой необычный возраст!'.

Те же проверки при использовании if..else:

if (age < 3) {

message = 'Здравствуй, малыш!';

} else if (age < 18) {

message = 'Привет!';

} else if (age < 100) {

message = 'Здравствуйте!';

} else {

message = 'Какой необычный возраст!';

}

Иногда оператор вопросительный знак ? используется в качестве замены if:

let company = prompt('Какая компания создала JavaScript?', '');

(company == 'Netscape') ?

alert('Верно!') : alert('Неправильно.');

В зависимости от условия company == 'Netscape', будет выполнена либо первая, либо вторая часть после ?. Здесь не присваивается результат переменной. Вместо этого выполняется различный код в зависимости от условия. Не рекомендуется использовать оператор вопросительного знака таким образом, так как она менее читабельна.

Тот же код, использующий if:

let company = prompt('Какая компания создала JavaScript?', '');

if (company == 'Netscape') {

alert('Верно!');

} else {

alert('Неправильно.');

}

1. **Логические операторы.**

В JavaScript есть три логических оператора: || (ИЛИ), && (И) и ! (НЕ). Данные операторы могут применяться к значениям любых типов. Полученные результаты также могут иметь различный тип.

[**|| (ИЛИ)**](https://learn.javascript.ru/logical-operators#ili)

Оператор «ИЛИ» выглядит как двойной символ вертикальной черты:

result = a || b;

Традиционно в программировании ИЛИ предназначено только для манипулирования булевыми значениями: в случае, если какой-либо из аргументов true, он вернёт true, в противной ситуации возвращается false. В JavaScript этот оператор работает несколько иным образом.

Существует всего четыре возможные логические комбинации:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

Как видно, результат операций всегда равен true, за исключением случая, когда оба аргумента false. Если значение не логического типа, то оно к нему приводится в целях вычислений. Например, число 1 будет воспринято как true, а 0 – как false:

if (1 || 0) {

alert( 'truthy!' );

}

Обычно оператор || используется в if для проверки истинности любого из заданных условий. К примеру:

let hour = 9;

if (hour < 10 || hour > 18) {

alert( 'Офис закрыт.' );

}

Можно передать и больше условий:

let hour = 12;

let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {

alert( 'Офис закрыт.' ); // это выходной

}

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

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

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

Значение возвращается в исходном виде, без преобразования.

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

alert( 1 || 0 ); // 1

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

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

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

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

Такой принцип действия позволяет применять оператор нетрадиционным способом для решения некоторых задач. Например, чтобы получить первое истинное значение из списка переменных или выражений. Допустим имеется ряд переменных, которые могут содержать данные или быть null/undefined. С помощью || можно найти первую переменную с данными:

let currentUser = null;

let defaultUser = "John";

let name = currentUser || defaultUser || "unnamed";

alert( name ); // выбирается "John" – первое истинное значение

Еще один пример использования ИЛИ для сокращённого вычисления. Операндами могут быть как отдельные значения, так и произвольные выражения. ИЛИ вычисляет их слева направо. Вычисление останавливается при достижении первого истинного значения. Этот процесс называется «сокращённым вычислением», поскольку второй операнд вычисляется только в том случае, если первого не достаточно для вычисления всего выражения. Это хорошо заметно, когда выражение, указанное в качестве второго аргумента, имеет побочный эффект, например, изменение переменной. В приведённом ниже примере x не изменяется:

let x;

true || (x = 1);

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

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

let x;

false || (x = 1);

alert(x); // 1

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

**[&& (И)](https://learn.javascript.ru/logical-operators" \l "i)**

Оператор И пишется как два амперсанда &&:

result = a && b;

В традиционном программировании И возвращает true, если оба аргумента истинны, а иначе – false:

alert( true && true ); // true

alert( false && true ); // false

alert( true && false ); // false

alert( false && false ); // false

Пример с if:

let hour = 12;

let minute = 30;

if (hour == 12 && minute == 30) {

alert( 'The time is 12:30' );

}

Как и в случае с ИЛИ, любое значение допускается в качестве операнда И:

if (1 && 0) { // вычисляется как true && false

alert( "won't work, because the result is falsy" );

}

[И «&&» находит первое ложное значение](https://learn.javascript.ru/logical-operators#i-nahodit-pervoe-lozhnoeznachenie). При нескольких подряд операторах И result = value1 && value2 && value3; оператор && выполняет следующие действия:

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

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

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

// Если первый операнд истинный,

// И возвращает второй:

alert( 1 && 0 ); // 0

alert( 1 && 5 ); // 5

// Если первый операнд ложный,

// И возвращает его. Второй операнд игнорируется

alert( null && 5 ); // null

alert( 0 && "no matter what" ); // 0

Можно передать несколько значений подряд. В таком случае возвратится первое «ложное» значение, на котором остановились вычисления.

alert( 1 && 2 && null && 3 ); // null

Когда все значения верны, возвращается последнее

alert( 1 && 2 && 3 ); // 3

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

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

let x = 1;

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

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

let x = 1;

if (x > 0) {

alert( 'Greater than zero!' );

}

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

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

Оператор НЕ представлен восклицательным знаком !. Синтаксис:

result = !value;

Оператор принимает один аргумент и выполняет следующие действия:

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

Например:

alert( !true ); // false

alert( !0 ); // true

Двойное НЕ используют для преобразования значений к логическому типу:

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

alert( !!null ); // false

Первое НЕ преобразует значение в логическому типу и возвращает обратное, а второе НЕ снова инвертирует его. В результате получится простое преобразование значения в логическое. С помощью встроенной функции Boolean можно сделать то же самое:

alert( Boolean("non-empty string") ); // true

alert( Boolean(null) ); // false

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

**?? (нуллевое слияние)**

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

В отличие от логического ИЛИ (||), левая часть оператора вычисляется и возвращается даже если его результат после приведения к логическому типу оказывается ложным, но не является null или undefined. Другими словами, если вы используете || чтобы установить значение по умолчанию, вы можете столкнуться с неожиданным поведением если считаете некоторые ложные значения пригодными для использования (например, "" или 0). Примеры:

const foo = null ?? 'default string';

console.log(foo); // "default string"

const baz = 0 ?? 42;

console.log(baz); // 0

1. **Циклы while, for.**

При написании скриптов зачастую встаёт задача сделать однотипное действие много раз. Например, вывести товары из списка один за другим. Или просто перебрать все числа от 1 до 10 и для каждого выполнить одинаковый код. Для многократного повторения одного участка кода предусмотрены *циклы*.

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

Цикл while имеет следующий синтаксис:

while (condition) {

// код – тело цикла

}

Код из тела цикла выполняется, пока условие condition истинно. Например, цикл ниже выводит i, пока i < 3:

let i = 0;

while (i < 3) { // выводит 0, затем 1, затем 2

alert( i );

i++;

}

Одно выполнение тела цикла называется *итерация*. Цикл в примере выше совершает три итерации. Если бы строка i++ отсутствовала, то цикл бы повторялся (в теории) вечно. Но в действительности браузер предоставит пользователю возможность остановить «подвисший» скрипт, а JavaScript на стороне сервера придётся «убить» процесс.

Любое выражение или переменная может быть условием цикла, а не только сравнение: условие while вычисляется и преобразуется в логическое значение. Например, while (i) – более краткий вариант while (i != 0):

let i = 3;

while (i) { // когда i будет равно 0, условие станет ложным, и цикл остановится

alert( i );

i--;

}

Если тело цикла состоит лишь из одной инструкции, мы можем опустить фигурные скобки {…}:

let i = 3;

while (i) alert(i--);

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

Проверку условия можно разместить под телом цикла, используя специальный синтаксис do..while:

do {

// тело цикла

} while (condition);

Цикл сначала выполнит тело, а затем проверит условие condition, и пока его значение равно true, он будет выполняться снова и снова. Например:

let i = 0;

do {

alert( i );

i++;

} while (i < 3);

Цикл do..while стоит использовать, если необходимо, чтобы тело цикла выполнилось хотя бы один раз, даже если условие окажется ложным. На практике чаще используется форма с предусловием: while(…) {…}.

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

Более сложный, но при этом самый распространённый цикл – цикл for. Синтаксис:

for (начало; условие; шаг) {

// ... тело цикла ...

}

Рассмотрим пример. Цикл ниже выполняет alert(i) для i от 0 до (но не включая) 3:

for (let i = 0; i < 3; i++) { // выведет 0, затем 1, затем 2

alert(i);

}

То есть, *начало* выполняется один раз, а затем каждая итерация заключается в проверке *условия*, после которой выполняется *тело* и *шаг*.

В примере переменная счётчика *i* была объявлена прямо в цикле. Это так называемое встроенное объявление переменной. Такие переменные видны только внутри цикла.

for (let i = 0; i < 3; i++) {

alert(i); // 0, 1, 2

}

alert(i); // ошибка, нет такой переменной

Вместо объявления новой переменной можно использовать уже существующую:

let i = 0;

for (i = 0; i < 3; i++) {

alert(i); // 0, 1, 2

}

alert(i); // 3, переменная доступна

Любая часть for может быть пропущена. Например, можно пропустить начало если ничего не нужно делать перед стартом цикла:

let i = 0;

for (; i < 3; i++) {

alert( i ); // 0, 1, 2

}

Можно убрать шаг, это сделает цикл аналогичным while (i < 3):

let i = 0;

for (; i < 3;) {

alert( i++ );

}

Можно убрать всё, получив бесконечный цикл:

for (;;) {

// будет выполняться вечно

}

При этом сами точки с запятой ; обязательно должны присутствовать, иначе будет ошибка синтаксиса.

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

Обычно цикл завершается при вычислении *условия* в false. Но можно выйти из цикла в любой момент с помощью специальной директивы break. Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

let sum = 0;

while (true) {

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

if (!value) break; // (\*)

sum += value;

}

alert( 'Сумма: ' + sum );

Директива break в строке (\*) полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на alert. Использование директивы break в бесконечном цикле удобно в том случае, если условие, по которому нужно прерваться, находится не в начале или конце цикла, а посередине.

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

При выполнении директивы continue цикл не прерывается, а переходит к следующей итерации (если условие все ещё равно true). Её используют, если известно, что в текущей итерации цикла нет необходимости. Например, цикл ниже использует continue, чтобы выводить только нечётные значения:

for (let i = 0; i < 10; i++) {

// если true, пропустить оставшуюся часть тела цикла

if (i % 2 == 0) continue;

alert(i); // 1, затем 3, 5, 7, 9

}

Для чётных значений *i*, директива continue прекращает выполнение тела цикла и передаёт управление на следующую итерацию for (со следующим числом). Таким образом alert вызывается только для нечётных значений.

Директива continue позволяет избегать вложенности. Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

for (let i = 0; i < 10; i++) {

if (i % 2) {

alert( i );

}

}

Но здесь появился дополнительный уровень вложенности. Если код внутри if более длинный, то это ухудшает читаемость, в отличие от варианта с continue.

Эти синтаксические конструкции break/continue не являются выражениями и не могут быть использованы с тернарным оператором ?. Это приведет к синтаксической ошибке. Например:

(i > 5) ? alert(i) : continue; // continue здесь приведёт к ошибке

1. **Конструкция switch.**

Конструкция switch заменяет собой сразу несколько if. Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Конструкция switch имеет один или более блок *case* и необязательный блок *default*. Синтаксис:

switch(x) {

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

...

[break]

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

...

[break]

default:

...

[break]

}

* Переменная *x* проверяется на строгое равенство первому значению *value1*, затем второму *value2* и так далее.
* Если соответствие установлено – switch начинает выполняться от соответствующей директивы case и далее, до ближайшего break (или до конца switch).
* Если ни один case не совпал – выполняется (если есть) вариант default.

Пример использования switch:

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Маловато' );

break;

case 4:

alert( 'В точку!' );

break;

case 5:

alert( 'Перебор' );

break;

default:

alert( "Нет таких значений" );

}

Здесь оператор switch последовательно сравнит *a* со всеми вариантами из case. Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert( 'В точку!' ) и далее, до ближайшего break, который прервёт выполнение. Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются. Пример без break:

let a = 2 + 2;

switch (a) {

case 3:

alert( 'Маловато' );

case 4:

alert( 'В точку!' );

case 5:

alert( 'Перебор' );

default:

alert( "Нет таких значений" );

}

В примере выше последовательно выполнятся три alert:

alert( 'В точку!' );

alert( 'Перебор' );

alert( "Нет таких значений" );

И switch и case допускают любое выражение в качестве аргумента. Например:

let a = "1";

let b = 0;

switch (+a) {

case b + 1:

alert("Выполнится, т.к. значением +a будет 1, что в точности равно b+1");

break;

default:

alert("Это не выполнится");

}

В этом примере результатом выражения +a будет 1, что совпадает с выражением b + 1 в case, и, следовательно, код в этом блоке будет выполнен.

Несколько вариантов case, использующих один код, можно группировать. Для примера, выполним один и тот же код для case 3 и case 5, сгруппировав их:

let a = 2 + 2;

switch (a) {

case 4:

alert('Правильно!');

break;

case 3: // (\*) группируем оба case

case 5:

alert('Неправильно!');

alert("Может вам посетить урок математики?");

break;

default:

alert('Результат выглядит странновато. Честно.');

}

Теперь оба варианта 3 и 5 выводят одно сообщение. Возможность группировать case – это побочный эффект того, как switch/case работает без break. Здесь выполнение case 3 начинается со строки (\*) и продолжается в case 5, потому что отсутствует break.

Стоит отметить, что проверка на равенство всегда строгая. Значения должны быть одного типа, чтобы выполнялось равенство. Для примера, рассмотрим следующий код:

let arg = prompt("Введите число?");

switch (arg) {

case '0':

case '1':

alert( 'Один или ноль' );

break;

case '2':

alert( 'Два' );

break;

case 3:

alert( 'Никогда не выполнится!' );

break;

default:

alert( 'Неизвестное значение' );

}

1. Для '0' и '1' выполнится первый alert.
2. Для '2' – второй alert.
3. Но для 3, результат выполнения prompt будет строка "3", которая не соответствует строгому равенству === с числом 3. Таким образом, код в case 3 не выполнится. Выполнится вариант default.
4. **Функции.**

Зачастую нам надо повторять одно и то же действие во многих частях программы. Чтобы не повторять один и тот же код во многих местах, придуманы функции. Примеры встроенных функций: alert(message), prompt(message, default) и confirm(question). Можно создавать и свои.

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

Для создания функций можно использовать объявление функции. Такой синтаксис называется *Function Declaration*. 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 примере выше он пустой) и, наконец, код функции, также называемый «телом функции», внутри фигурных скобок.

function имя(параметры) {

...тело...

}

Функция может быть вызвана по её имени: showMessage(). Например:

function showMessage() {

alert( 'Всем привет!' );

}

showMessage();

showMessage();

Вызов showMessage() выполняет код функции. В результате появится сообщение дважды. Этот пример явно демонстрирует одно из главных предназначений функций: избавление от дублирования кода. Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

Переменные объявленные внутри функции, видны только внутри этой функции и являются *локальными*. Например:

function showMessage() {

let message = "Привет, я JavaScript!"; // локальная переменная

alert( message );

}

showMessage(); // Привет, я JavaScript!

alert( message ); // <-- будет ошибка, т.к. переменная видна только внутри функции

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

showMessage();

alert( userName ); // Петя

Внешняя переменная используется только если внутри функции нет такой локальной. Если одноимённая переменная объявляется внутри функции, тогда она перекрывает внешнюю. Например, в коде ниже функция использует локальную переменную userName. Внешняя будет проигнорирована:

let userName = 'Вася';

function showMessage() {

let userName = "Петя"; // объявляем локальную переменную

let message = 'Привет, ' + userName; // Петя

alert(message);

}

// функция создаст и будет использовать свою собственную локальную переменную userName

showMessage();

alert( userName ); // Вася, не изменилась, функция не трогала внешнюю переменную

Переменные, объявленные снаружи всех функций, такие как внешняя переменная userName в коде выше – называются *глобальными*. Глобальные переменные видимы для любой функции (если только их не перекрывают одноимённые локальные переменные).

Желательно сводить использование глобальных переменных к минимуму. В современном коде их нет или они используются редко.

[**Параметры**](https://learn.javascript.ru/function-basics#parametry)

Можно передать внутрь функции любую информацию, используя параметры (также называемые аргументы функции). В примере ниже функции передаются два параметра: from и text.

function showMessage(from, text) { // аргументы: from, text

alert(from + ': ' + text);

}

showMessage('Аня', 'Привет!'); // Аня: Привет! (\*)

showMessage('Аня', "Как дела?"); // Аня: Как дела? (\*\*)

Когда функция вызывается в строках (\*) и (\*\*), переданные значения копируются в локальные переменные from и text. Затем они используются в теле функции.

В примере ниже, есть переменная from, и она передаётся функции. Функция изменяет значение from, но это изменение не видно снаружи, так она получает только копию значения:

function showMessage(from, text) {

from = '\*' + from + '\*';

alert( from + ': ' + text );

}

let from = "Аня";

showMessage(from, "Привет"); // \*Аня\*: Привет

alert( from ); // Аня

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

Если параметр не указан, то его значением становится undefined. Например, вышеупомянутая функция showMessage(from, text) может быть вызвана с одним аргументом:

showMessage("Аня");

Это не приведёт к ошибке. Такой вызов выведет "Аня: undefined". В вызове не указан параметр text, поэтому предполагается, что text === undefined. Если необходимо задать параметру text значение по умолчанию, то надо указать его после =:

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

alert( from + ": " + text );

}

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

Теперь, если параметр text не указан, его значением будет "текст не добавлен". В данном случае "текст не добавлен" это строка, но на её месте могло бы быть и более сложное выражение, которое бы вычислялось и присваивалось при отсутствии параметра. Например:

function showMessage(from, text = anotherFunction()) {

// anotherFunction() выполнится только если не передан text

// результатом будет значение text

}

**Вычисление параметров по умолчанию**

В JavaScript параметры по умолчанию вычисляются каждый раз, когда функция вызывается без соответствующего параметра. В примере выше anotherFunction() будет вызываться каждый раз, когда showMessage()вызывается без параметра text.

Ранние версии JavaScript не поддерживали параметры по умолчанию. Поэтому существуют альтернативные способы, которые могут встречаться в старых скриптах. Например, явная проверка на undefined или с помощью оператора ||:

function showMessage(from, text) {

if (text === undefined) {

text = 'текст не добавлен';

}

alert( from + ": " + text );

}

function showMessage(from, text) {

// Если значение text ложно, тогда присвоить параметру text значение по умолчанию

text = text || 'текст не добавлен';

...

}

[**Возврат значения**](https://learn.javascript.ru/function-basics#vozvrat-znacheniya)

Функция может возвратить результат, который будет передан в вызвавший её код. Простейшим примером может служить функция сложения двух чисел:

function sum(a, b) {

return a + b;

}

let result = sum(1, 2);

alert( result ); // 3

Директива return может находиться в любом месте тела функции. Как только выполнение доходит до этого места, функция останавливается, и значение возвращается в вызвавший её код (присваивается переменной result выше). Вызовов return может быть несколько, например:

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('А родители разрешили?');

}

}

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

if ( checkAge(age) ) {

alert( 'Доступ получен' );

} else {

alert( 'Доступ закрыт' );

}

Возможно использовать return и без значения. Это приведёт к немедленному выходу из функции. Например:

function showMovie(age) {

if ( !checkAge(age) ) {

return;

}

alert( "Вам показывается кино" ); // (\*)

// ...

}

В коде выше, если checkAge(age) вернёт false, showMovie не выполнит alert. Результат функции с пустым return или без него – undefined. Если функция не возвращает значения, это то же самое, что она возвращает undefined:

function doNothing() { /\* пусто \*/ }

alert( doNothing() === undefined ); // true

Пустой return аналогичен return undefined:

function doNothing() {

return;

}

alert( doNothing() === undefined ); // true

Для длинного выражения в return не стоит добавлять перевод строки между return и его значением, например так:

return

(some + long + expression + or + whatever \* f(a) + f(b))

Код не выполнится, потому что интерпретатор JavaScript подставит точку с запятой после return. Для него это будет выглядеть так:

return;

(some + long + expression + or + whatever \* f(a) + f(b))

Таким образом, это фактически стало пустым return. Если необходимо, чтобы возвращаемое выражение занимало несколько строк, то нужно начать его на той же строке, что и return. Или, хотя бы, поставить там открывающую скобку:

return (

some + long + expression

+ or +

whatever \* f(a) + f(b)

)

**[Выбор имени функции](https://learn.javascript.ru/function-basics" \l "function-naming)**

Функция – это действие. Поэтому имя функции обычно является глаголом. Оно должно быть простым, точным и описывать действие функции. Чтобы программист, который будет читать код, получил верное представление о том, что делает функция. Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение. Обычно в командах разработчиков действуют соглашения, касающиеся значений этих префиксов. Например, функции, начинающиеся с "show" обычно что-то показывают. Примеры префиксов:

* "get…" – возвращают значение,
* "calc…" – что-то вычисляют,
* "create…" – что-то создают,
* "check…" – что-то проверяют и возвращают логическое значение, и т.д.

Примеры имён функций с префиксами:

showMessage(..) // показывает сообщение

getAge(..) // возвращает возраст

calcSum(..) // вычисляет сумму и возвращает результат

createForm(..) // создаёт форму и обычно возвращает её

checkPermission(..) // проверяет доступ, возвращая true/false

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одним действием. Два независимых действия обычно подразумевают две функции, даже если предполагается, что они будут вызываться вместе. Например, функция getAge должна только возвращать возвраст, а не выводить alert с возрастом; createForm – должна только создавать форму и возвращать её, а не изменять документ, добавляя форму в него; checkPermission – должна только выполнять проверку и возвращать её результат, а не отображать сообщение с текстом доступ разрешён/запрещён и т.д.

Имена функций, которые используются очень часто, иногда делают сверхкороткими. Например, во фреймворке [jQuery](http://jquery.com/) есть функция с именем $. В библиотеке [Lodash](http://lodash.com/) основная функция представлена именем \_. Это исключения. В основном имена функций должны быть в меру краткими и описательными.

Функции должны быть короткими и делать только что-то одно. Если это что-то большое, имеет смысл разбить функцию на несколько меньших. Небольшие функции не только облегчает тестирование и отладку, но и являются хорошим комментарием. Например, сравним ниже две функции showPrimes(n). Каждая из них выводит [простое число](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%81%D1%82%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE) до *n*.

Первый вариант использует метку nextPrime:

function showPrimes(n) {

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

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

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

}

alert( i ); // простое

}

}

Второй вариант использует дополнительную функцию isPrime(n) для проверки на простое:

function showPrimes(n) {

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

if (!isPrime(i)) continue;

alert(i); // простое

}

}

function isPrime(n) {

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

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

}

return true;

}

Второй вариант легче для понимания. Не надо разбираться с кодом, сразу видно название действия (isPrime). Разработчики называют такой код самодокументируемым.

Таким образом, рекомендуется создавать функции даже если не планируется повторно использовать их. Такие функции структурируют код и делают его более понятным.

1. **Функциональные выражения и функции-стрелки**

Существует ещё один синтаксис создания функций, который называется Function Expression (Функциональное Выражение):

let sayHi = function() {

alert( "Привет" );

};

В коде выше функция создаётся и явно присваивается переменной, как любое другое значение. Не зивисимо от того, как определена функция (Function Expression или Function Declaration), это просто значение, хранимое в переменной sayHi. Можно даже вывести это значение с помощью alert:

function sayHi() {

alert( "Привет" );

}

alert( sayHi ); // выведет код функции

Обратите внимание, что последняя строка не вызывает функцию sayHi, так как после её имени нет круглых скобок. В JavaScript функции – это значения, поэтому и обращаться с ними, надо как со значениями. Код выше выведет строковое представление функции, которое является её исходным кодом.

С функцией можно делать то же самое, что и с любым другим значением. Например, скопировать функцию в другую переменную:

function sayHi() { // (1)

alert( "Привет" );

}

let func = sayHi; // (2)

func(); // Привет // (3)

sayHi(); // Привет

Рассмотрим пример подробнее:

* 1. Объявление Function Declaration (1) создало функцию и присвоило её значение переменной с именем sayHi.
  2. В строке (2) ее значение скопировано в переменную func. Обратите внимание: нет круглых скобок после sayHi. Если бы они были, то выражение func = sayHi() записало бы результат вызова sayHi() в переменную func, а не саму функцию sayHi.
  3. Теперь функция может быть вызвана с помощью обеих переменных sayHi() и func().

Можно использовать и Function Expression для того, чтобы создать sayHi в первой строке. Результат будем таким же:

let sayHi = function() {

alert( "Привет" );

};

let func = sayHi;

// ...

Заметьте, что в Function Expression ставится точка с запятой ; в конце, а в Function Declaration нет:

function sayHi() {

// ...

}

let sayHi = function() {

// ...

};

Это потому, что Function Expression использует внутри себя инструкции присваивания let sayHi = ...; как значение. Это не блок кода, а выражение с присваиванием. Таким образом, точка с запятой не относится непосредственно к Function Expression, она лишь завершает инструкцию.

[**Функции-«колбэки»**](https://learn.javascript.ru/function-expressions-arrows#funktsii-kolbeki)

Рассмотрим функцию ask(question, yes, no) с тремя параметрами: question – текст вопроса, yes – функция, которая будет вызываться, если ответ будет «Yes», no – функция, которая будет вызываться, если ответ будет «No». В браузерах такие функции обычно отображают красивые диалоговые окна. Функция задает вопрос question и, в зависимости от того, как ответит пользователь, вызвать yes() или no():

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

function showOk() {

alert( "Вы согласны." );

}

function showCancel() {

alert( "Вы отменили выполнение." );

}

ask("Вы согласны?", showOk, showCancel);

Аргументы функции ask ещё называют функциями-колбэками или просто колбэками (от англ. «call back» – обратный вызов). Т.е. функция передаётся в качестве агрумента и вызывается обратно тогда, когда это необходимо. В нашем случае, showOk становится колбэком для ответа «yes», а showCancel – для ответа «no».

Можно переписать пример короче, используя Function Expression:

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

ask(

"Вы согласны?",

function() { alert("Вы согласились."); },

function() { alert("Вы отменили выполнение."); }

);

Здесь функции объявляются прямо внутри вызова ask(...). У них нет имён, поэтому они называются анонимными. Такие функции недоступны снаружи ask (потому что они не присвоены переменным).

Одним из отличий Function Expression от Function Declaration, кроме синтаксиса, является то, что Function Expression создаётся, когда выполнение доходит до него, и затем уже может использоваться. После того, как поток выполнения достигнет правой части выражения присваивания let sum = function… – с этого момента, функция считается созданной и может быть использована (присвоена переменной, вызвана и т.д. ).

Function Declaration можно использовать во всем скрипте (или блоке кода, если функция объявлена в блоке). Другими словами, когда движок JavaScript готовится выполнять скрипт или блок кода, прежде всего он ищет в нём Function Declaration и создаёт все такие функции. Обычно этот процесс называют «стадией инициализации». И только после того, как все объявления Function Declaration будут обработаны, продолжится выполнение. В результате, функции, созданные, как Function Declaration могут быть вызваны раньше своих определений. Например, так будет работать:

sayHi("Вася"); // Привет, Вася

function sayHi(name) {

alert( `Привет, ${name}` );

}

Функция sayHi была создана, когда движок JavaScript подготавливал скрипт к выполнению, и такая функция видна повсюду в этом скрипте. Если бы это было Function Expression, то такой код привел бы к ошибке:

sayHi("Вася");

let sayHi = function(name) {

alert( `Привет, ${name}` );

};

Ещё одна важная особенность Function Declaration заключается в их блочной области видимости. В строгом режиме, когда Function Declaration находится в блоке {...}, функция доступна везде внутри блока, но не снаружи него. Для примера представим, что нужно создать функцию welcome() в зависимости от значения переменной age, которое будет получено во время выполнения кода. Такой код, использующий Function Declaration, работать не будет:

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

// в зависимости от условия объявляем функцию

if (age < 18) {

function welcome() {

alert("Привет!");

}

} else {

function welcome() {

alert("Здравствуйте!");

}

}

welcome(); // Error: welcome is not defined

Это произошло, так как объявление Function Declaration видимо только внутри блока кода, в котором располагается.

Верным подходом будет воспользоваться функцией, объявленной при помощи Function Expression, и присвоить значение welcome переменной, объявленной снаружи if, что обеспечит нужную видимость. Такой код работает как требуется:

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

let welcome;

if (age < 18) {

welcome = function() {

alert("Привет!");

};

} else {

welcome = function() {

alert("Здравствуйте!");

};

}

welcome(); // теперь всё в порядке

Можно упростить этот код ещё больше, используя условный оператор ?:

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

let welcome = (age < 18) ?

function() { alert("Привет!"); } :

function() { alert("Здравствуйте!"); };

welcome();

**[Функции-стрелки](https://learn.javascript.ru/function-expressions-arrows" \l "arrow-functions)**

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

Такой код создаёт функцию func с аргументами arg1..argN и вычисляет expression с правой стороны с их использованием, возвращая результат. Это то же самое, что и:

let func = function(arg1, arg2, ...argN) {

return expression;

};

Рассмотрим пример:

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

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

Если у передается только один аргумент, то круглые скобки вокруг параметров можно опустить, сделав запись ещё короче:

let double = n => n \* 2;

alert( double(3) ); // 6

Если нет аргументов, используются пустые круглые скобки (их указывать обязательно):

let sayHi = () => alert("Hello!");

sayHi();

В примерах выше аргументы использовались слева от =>, а справа вычислялось выражение с их значениями. Но если требуется вычислить несколько выражений или инструкций, то необходимо заключить такие выражения в фигурные скобки с использованием директивы return внутри них, как в обычной функции. Например:

let sum = (a, b) => {

let result = a + b;

return result;

};

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

Важной особенностью стрелочных функций является то, что у них нет переменной arguments.

**Тема 2. Структуры данных**

1. Методы примитивов.
2. Числа.
3. Строки.
4. Массивы.
5. Методы массивов.
6. Объекты.
7. Тип данных Symbol.
8. Set, Map, WeakSet и WeakMap.
9. Деструктурирующее присваивание
10. Дата и время.
11. Формат JSON.
12. Intl: интернационализация в JavaScript.
13. Регулярные выражения.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. **Методы примитивов.**

JavaScript позволяет работать с примитивными типами данных – строками, числами, и т.д., так, как если бы они были объектами. У них есть и методы. Рассмотрим ключевые различия между примитивами и объектами.

Примитив – это значение «примитивного» типа. Есть 6 примитивных типов: string, number, boolean, symbol, null и undefined.

Объект может хранить множество значений как свойства. Объявляется при помощи фигурных скобок {}, например: {name: "Рома", age: 30}. В JavaScript есть и другие виды объектов: например, функции тоже являются объектами.

Одна из лучших особенностей объектов – это то, что можно хранить функцию как одно из свойств объекта:

let roma = {

name: "Рома",

sayHi: function() {

alert("Привет, дружище!");

}

};

roma.sayHi(); // Привет, дружище!

Здесь создан объект roma с методом sayHi.

Существует множество встроенных объектов. Например, те, которые работают с датами, ошибками, HTML-элементами и т.д. Они имеют различные свойства и методы. Поэтому объекты «тяжелее» примитивов, они нуждаются в дополнительных ресурсах для поддержания внутренней структуры.

Язык JavaScript позволяет осуществлять доступ к методам и свойствам строк, чисел, булевых значений и символов. Чтобы это работало, при таком доступе создаётся специальный «объект-обёртка», который предоставляет нужный функционал, а после удаляется. Каждый примитив имеет свой собственный «объект-обёртку», которые называются: String, Number, Boolean и Symbol. Таким образом, они имеют разный набор методов. К примеру, существует метод [str.toUpperCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase), который возвращает cтроку в верхнем регистре:

let str = "Привет";

alert( str.toUpperCase() );

Рассмотрим, как это работет:

1. Строка str – примитив. В момент обращения к его свойству, создаётся специальный объект, который содержит значение строки и метод toUpperCase().
2. Этот метод вызывается и возвращает новую строку (выводится в alert).
3. Специальный объект удаляется, оставляя только примитив str.

Таким образом, примитивы могут предоставлять методы, и в то же время оставаться «лёгкими». Движок JavaScript сильно оптимизирует этот процесс. Он даже может пропустить создание специального объекта.

Число имеет собственный набор методов. Например, [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) округляет число до *n* знаков после запятой.

let n = 1.23456;

alert( n.toFixed(2) ); // 1.23

В JavaScript также можно явное создавать «объекты-обёртки» с помощью конструкторов String/Number/Boolean (new Number(1) или new Boolean(false)), но это очень не рекомендуется, так как иногда приводит к необжиданным результатам:

alert( typeof 0 ); // "число"

alert( typeof new Number(0) ); // "object"

Объекты в if всегда дают true, поэтому в примере ниже будет показан alert:

let zero = new Number(0);

if (zero) {

// zero возвращает "true", так как является объектом

alert( "zero имеет «истинное» значение?!?" );

}

С другой стороны, использование функций String/Number/Boolean (без оператора new) полезно, так как они превращают значение в соответствующий примитивный тип: в строку, в число, в булевой тип. Например:

let num = Number("123"); // превращает строку в число

Особенные примитивы null and undefined являются исключениями – у них нет соответствующих «объектов-обёрток» и они не имеют никаких методов.

1. **Числа.**

Все 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).

[**Способы записи числа**](https://learn.javascript.ru/number#sposoby-zapisi-chisla)

Допустим надо записать число 1 миллиард:

let billion = 1000000000;

Но в реальной жизни обычно множество нулей опускается, а укороченная запись может выглядеть как "1млрд" или "7.3млрд".  Такой принцип работает для всех больших чисел. В JavaScript можно использовать букву "e", чтобы укоротить запись числа. Она добавляется к числу и заменяет указанное количество нулей:

let billion = 1e9; // 1 миллиард

alert( 7.3e9 ); // 7.3 миллиардов

Другими словами, "e" производит операцию умножения числа на 1 с указанным количеством нулей.

1e3 = 1 \* 1000

1.23e6 = 1.23 \* 1000000

Допустим необходимо записать что-нибудь очень маленькое: 1 микросекунду (одна миллионная секунды):

let ms = 0.000001;

Записать микросекунду в укороченном виде также можно с помощью "e".

let ms = 1e-6; // шесть нулей, слева от 1

Т.е., отрицательное число после "e" подразумевает деление на 1 с указанным количеством нулей:

// -3 делится на 1 с 3 нулями

1e-3 = 1 / 1000 (=0.001)

// -6 делится на 1 с 6 нулями

1.23e-6 = 1.23 / 1000000 (=0.00000123)

[**Шестнадцатеричные, двоичные и восьмеричные числа**](https://learn.javascript.ru/number#shestnadtsaterichnye-dvoichnye-i-vosmerichnye-chisla)

[Шестнадцатеричные](https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D1%81%D1%82%D0%BD%D0%B0%D0%B4%D1%86%D0%B0%D1%82%D0%B5%D1%80%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%81%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F) числа широко используются в JavaScript для представления цветов, кодировки символов и многое другое. Естественно, есть короткий стиль записи: 0x, после которого указывается число. Например:

alert( 0xff ); // 255

alert( 0xFF ); // 255 (регистр не имеет значения)

Не так часто используются двоичные и восьмеричные числа, но они также поддерживаются 0b для двоичных и 0o для восьмеричных:

let a = 0b11111111; // бинарная форма записи числа 255

let b = 0o377; // восьмеричная форма записи числа 255

alert( a == b ); // true

Есть только 3 системы счисления с такой поддержкой. Для других систем счисления рекомендуется использовать функцию parseInt (рассмотрим позже).

[**toString(base)**](https://learn.javascript.ru/number#tostring-base)

Метод num.toString(base) возвращает строковое представление числа *num* в системе счисления base. Например:

let num = 255;

alert( num.toString(16) ); // ff

alert( num.toString(2) ); // 11111111

Значение *base* может варьироваться от 2 до 36 (по умолчанию 10):

* base = 16 – для шестнадцатеричного представления цвета, кодировки символов и т.д., цифры могут быть 0..9 или A..F.
* base = 2 – обычно используется для отладки побитовых операций, цифры 0 или 1.
* base = 36 – максимальное основание, цифры могут быть 0..9 или A..Z. То есть, используется весь латинский алфавит для представления числа.

Две точки в 123456..toString(36) используется, если надо вызвать метод toString непосредственно на числе. Если поставить одну точку: 123456.toString(36), тогда это будет ошибкой, поскольку синтаксис JavaScript предполагает, что после первой точки начинается десятичная часть. А если поставить две точки, то JavaScript понимает, что десятичная часть отсутствует, и начинается метод. Также можно записать как (123456).toString(36).

[**Округление**](https://learn.javascript.ru/number#okruglenie)

Одна из часто используемых операций при работе с числами – это округление. В JavaScript есть несколько встроенных функций для работы с округлением:

* Math.floor – округление в меньшую сторону: 3.1 становится 3, а -1.1 – -2.
* Math.ceil – округление в большую сторону: 3.1 становится 4, а -1.1 – -1.
* Math.round – округление до ближайшего целого: 3.1 становится 3, 3.6 – 4, а -1.1 – -1.
* Math.trunc (не поддерживается в Internet Explorer) – производит удаление дробной части без округления: 3.1 становится 3, а -1.1 – -1.

Таблица с различиями между функциями округления:

|  | **Math.floor** | **Math.ceil** | **Math.round** | **Math.trunc** |
| --- | --- | --- | --- | --- |
| 3.1 | 3 | 4 | 3 | 3 |
| 3.6 | 3 | 4 | 4 | 3 |
| -1.1 | -2 | -1 | -1 | -1 |
| -1.6 | -2 | -1 | -2 | -1 |

Эти функции охватывают все возможные способы обработки десятичной части. Если надо округлить число до *n*-ого количества цифр в дробной части, то это можно сделать одним из следующих способов:

1. Умножить и разделить.

Например, чтобы округлить число до второго знака после запятой, можно умножить число на 100, вызвать функцию округления и разделить обратно.

let num = 1.23456;

alert( Math.floor(num \* 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23

1. Метод [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) округляет число до *n* знаков после запятой и возвращает строковое представление результата.

Если десятичная часть короче, чем необходима, будут добавлены нули в конец строки:

let num = 12.34;

alert( num.toFixed(5) ); // "12.34000", добавлены нули, чтобы получить 5 знаков после запятой

Можно преобразовать полученное значение в число, используя унарный оператор + или Number(), пример с унарным оператором: +num.toFixed(5).

[**Неточные вычисления**](https://learn.javascript.ru/number#netochnye-vychisleniya)

Внутри JavaScript число представлено в виде 64-битного формата [IEEE-754](https://ru.wikipedia.org/wiki/IEEE_754-1985). Для хранения числа используется 64 бита: 52 из них используется для хранения цифр, 11 из них для хранения положения десятичной точки (если число целое, то хранится 0), и один бит отведён на хранения знака. Если число слишком большое, оно переполнит 64-битное хранилище, JavaScript вернёт бесконечность:

alert( 1e500 ); // Infinity

Наиболее часто встречающаяся ошибка при работе с числами в JavaScript – это потеря точности. Например, сумма 0.1 и 0.2 не равна 0.3:

alert( 0.1 + 0.2 == 0.3 ); // false

alert( 0.1 + 0.2 ); // 0.30000000000000004

Так происходит потому, что число хранится в памяти в бинарной форме, как последовательность бит – единиц и нулей. Но дроби, такие как 0.1, 0.2, которые выглядят довольно просто в десятичной системе счисления, на самом деле являются бесконечной дробью в двоичной форме.

Число 0.1 – это единица, делённая на десять – 1/10, одна десятая. В десятичной системе счисления такие числа легко представимы, по сравнению с одной третьей: 1/3, которая становится бесконечной дробью 0.33333(3). Деление на 10 хорошо работает в десятичной системе, но деление на 3 – нет. По той же причине и в двоичной системе счисления, деление на 2 обязательно сработает, а 1/10 становится бесконечной дробью.

В JavaScript нет возможности для хранения точных значений 0.1 или 0.2, используя двоичную систему, точно также, как нет возможности хранить одну третью в десятичной системе счисления. Числовой формат IEEE-754 решает эту проблему путём округления до ближайшего возможного числа. Правила округления обычно не позволяют увидеть эту нехначительную потерю точности, но она существует:

alert( 0.1.toFixed(20) ); // 0.10000000000000000555

Когда складываются 2 числа, их неточности тоже суммируются, поэтому 0.1 + 0.2 – это не совсем 0.3.

Наиболее надёжный способ обойти эту проблему – это округлить результат используя метод [toFixed(n)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed):

let sum = 0.1 + 0.2;

alert( sum.toFixed(2) ); // 0.30

Метод toFixed всегда возвращает строку. Это гарантирует, что результат будет с заданным количеством цифр в десятичной части. В других случаях можно использовать унарный оператор +, чтобы преобразовать строку в число:

let sum = 0.1 + 0.2;

alert( +sum.toFixed(2) ); // 0.3

Также можно умножить число на 100 (или на большее), чтобы привести его к целому, выполнить математические действия, а после разделить обратно. Суммируя целые числа, мы уменьшаем погрешность, но она все равно появляется при финальном делении:

alert( (0.1 \* 10 + 0.2 \* 10) / 10 ); // 0.3

alert( (0.28 \* 100 + 0.14 \* 100) / 100); // 0.4200000000000001

Таким образом, метод умножения/деления уменьшает погрешность, но полностью её не решает.

Еще одно следствие внутреннего представления чисел – наличие двух нулей: 0 и -0. Это возможно потому, что знак представлен отдельным битом, так что, любое число может быть положительным и отрицательным, включая нуль. В большинстве случаев это поведение незаметно, так как операторы в JavaScript воспринимают их одинаковыми.

[**Проверка: isFinite и isNaN**](https://learn.javascript.ru/number#proverka-isfinite-i-isnan)

Специальные числовые значения Infinity ( -Infinity) и NaN  принадлежат типу number, но они не являются «обычными» числами, поэтому есть функции для их проверки:

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

alert( isNaN(NaN) ); // true

alert( isNaN("str") ); // true

Нельзя просто сравнить === NaN, так как значение NaN уникально тем, что оно не является равным ни чему другому, даже самому себе:

alert( NaN === NaN ); // false

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

alert( isFinite("15") ); // true

alert( isFinite("str") ); // false

alert( isFinite(Infinity) ); // false

Иногда isFinite используется для проверки, содержится ли в строке число:

let num = +prompt("Enter a number", '');

alert( isFinite(num) );

Не стоит забывать, что пустая строка интерпретируется как 0 во всех числовых функциях, включая isFinite.

**Сравнение Object.is**

Существует специальный метод [Object.is](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/is), который сравнивает значения примерно как ===, но более надёжен в двух особых ситуациях:

1. Сравнивает NaN: Object.is(NaN, NaN) === true.
2. Значения 0 и -0 разные: Object.is(0, -0) === false, это редко используется, но технически эти значения разные.

Во всех других случаях Object.is(a, b) идентичен a === b.

Этот способ сравнения часто используется в спецификации JavaScript. Когда алгоритму необходимо сравнить 2 значения на предмет точного совпадения, он использует Object.is.

[**parseInt и parseFloat**](https://learn.javascript.ru/number#parseint-i-parsefloat)

Для явного преобразования к числу можно использовать + или Number(). Если строка не является в точности числом, то результат будет NaN:

alert( +"100px" ); // NaN

Единственное исключение – это пробелы в начале строки и в конце, они игнорируются. На практике часто встречаются значения, у которых есть единица измерения, например, "100px" или "12pt" в CSS, во множестве стран символ валюты записывается после номинала "19€".

Чтобы получить числовое значение из таких строк есть методы parseInt и parseFloat. Они считывают число из строки. Если в процессе чтения возникает ошибка, они возвращают полученное до ошибки число. Функция parseInt возвращает целое число, а parseFloat возвращает число с плавающей точкой:

alert( parseInt('100px') ); // 100

alert( parseFloat('12.5em') ); // 12.5

alert( parseInt('12.3') ); // 12

alert( parseFloat('12.3.4') ); // 12.3

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

alert( parseInt('a123') ); // NaN

Функция parseInt(str, radix) имеет необязательный второй параметр. Он определяет систему счисления, таким образом parseInt может также читать строки с шестнадцатеричными числами, двоичными числами и т.д.:

alert( parseInt('0xff', 16) ); // 255

alert( parseInt('ff', 16) ); // 255

alert( parseInt('2n9c', 36) ); // 123456

[**Другие математические функции**](https://learn.javascript.ru/number#drugie-matematicheskie-funktsii)

В JavaScript встроен объект [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math), который содержит различные математические функции и константы. Несколько примеров:

* Math.random() – возвращает псевдослучайное число в диапазоне [0, 1)

alert( Math.random() ); // 0.1234567894322

alert( Math.random() ); // 0.5435252343232

alert( Math.random() ); // ...

* Math.max(a, b, c...) / Math.min(a, b, c...) – возвращает наибольшее/наименьшее число из перечисленных аргументов.

alert( Math.max(3, 5, -10, 0, 1) ); // 5

alert( Math.min(1, 2) ); // 1

* Math.pow(n, power) – возвращает число n, возведённое в степень power

alert( Math.pow(2, 10) ); // 2 в степени 10 = 1024

В объекте Math есть множество других функций и констант, включая тригонометрические функции, с которыми подробнее можно ознакомиться в документации по объекту [Math](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math).

1. **Строки.**

В JavaScript любые текстовые данные являются строками. Не существует отдельного типа «символ», который есть в ряде других языков. Внутренний формат для строк – всегда [UTF-16](https://ru.wikipedia.org/wiki/UTF-16), вне зависимости от кодировки страницы.

[**Кавычки**](https://learn.javascript.ru/string#kavychki)

В JavaScript есть разные типы кавычек. Строку можно создать с помощью одинарных, двойных либо обратных кавычек:

let single = 'single-quoted';

let double = "double-quoted";

let backticks = `backticks`;

Одинарные и двойные кавычки работают, по сути, одинаково, а если использовать обратные кавычки, то в такую строку можно вставлять произвольные выражения, обернув их в ${…}:

function sum(a, b) {

return a + b;

}

alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.

Ещё одно преимущество обратных кавычек – они могут занимать более одной строки, вот так:

let guestList = `Guests:

\* John

\* Pete

\* Mary

`;

alert(guestList);

Если использовать точно так же одинарные или двойные кавычки, то будет ошибка:

let guestList = "Guests: // Error: Unexpected token ILLEGAL

\* John";

Обратные кавычки также позволяют задавать «шаблонную функцию» перед первой обратной кавычкой. Используемый синтаксис: func`string`. Автоматически вызываемая функция func получает строку и встроенные в неё выражения и может их обработать. Подробнее об этом можно прочитать в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Template_literals#Tagged_template_literals). Если перед строкой есть выражение, то шаблонная строка называется «теговым шаблоном». Это позволяет использовать свою шаблонизацию для строк, но на практике теговые шаблоны применяются редко.

[**Спецсимволы**](https://learn.javascript.ru/string#spetssimvoly)

Многострочные строки также можно создавать с помощью одинарных и двойных кавычек, используя так называемый «символ перевода строки», который записывается как \n:

let guestList = "Guests:\n \* John\n \* Pete\n \* Mary";

alert(guestList); // список гостей, состоящий из нескольких строк

В частности, эти две строки эквивалентны, просто записаны по-разному:

let str1 = "Hello\nWorld";

let str2 = `Hello

World`;

alert(str1 == str2); // true

Есть и другие, реже используемые спецсимволы:

| **Символ** | **Описание** |
| --- | --- |
| \n | Перевод строки |
| \r | Возврат каретки: самостоятельно не используется. В текстовых файлах Windows для перевода строки используется комбинация символов \r\n. |
| \', \" | Кавычки |
| \\ | Обратный слеш |
| \t | Знак табуляции |
| \b, \f, \v | Backspace, Form Feed и Vertical Tab – оставлены для обратной совместимости, сейчас не используются. |
| \xXX | Символ с шестнадцатеричным юникодным кодом XX, например, '\x7A' – то же самое, что 'z'. |
| \uXXXX | Символ в кодировке UTF-16 с шестнадцатеричным кодом XXXX, например, \u00A9 – юникодное представление знака копирайта, ©. Код должен состоять ровно из 4 шестнадцатеричных цифр. |
| \u{X…XXXXXX} (от 1 до 6 шестнадцатеричных цифр) | Символ в кодировке UTF-32 с шестнадцатеричным кодом от U+0000 до U+10FFFF. Некоторые редкие символы кодируются двумя 16-битными словами и занимают 4 байта. Так можно вставлять символы с длинным кодом. |

Примеры с Юникодом:

// ©

alert( "\u00A9" );

// 佫, редкий китайский иероглиф

alert( "\u{20331}" );

// 😍

alert( "\u{1F60D}" );

Все спецсимволы начинаются с обратного слеша, \ – так называемого «символа экранирования». Он также используется, если необходимо вставить в строку кавычку. Например:

alert( 'I\'m the Walrus!' ); // I'm the Walrus!

Здесь перед входящей в строку кавычкой необходимо добавить обратный слеш  \, иначе она бы обозначала окончание строки. Требование экранировать относится только к таким же кавычкам, как те, в которые заключена строка. Можно использовать для этой строки двойные или обратные кавычки:

alert( `I'm the Walrus!` ); // I'm the Walrus!

Заметим, что обратный слеш \ служит лишь для корректного прочтения строки интерпретатором, но он не записывается в строку после её прочтения. Когда строка сохраняется в оперативную память, в неё не добавляется символ \. Это можно увидеть в выводах alert в примерах выше. Но если надо добавить в строку сам обратный слеш \, то это можно сделать, добавив перед ним ещё один обратный слеш:

alert( `The backslash: \\` ); // The backslash: \

[**Длина строки**](https://learn.javascript.ru/string#dlina-stroki)

Свойство length содержит длину строки:

alert( `My\n`.length ); // 3

Обратите внимание, \n – это один спецсимвол, поэтому длина строки 3.

Так как str.length – это числовое свойство, а не функция, добавлять скобки не нужно.

[**Доступ к символам**](https://learn.javascript.ru/string#dostup-k-simvolam)

Получить символ, который занимает позицию pos, можно с помощью квадратных скобок: [pos]. Также можно использовать метод charAt: [str.charAt(pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/charAt). Первый символ занимает нулевую позицию:

let str = `Hello`;

// первый символ

alert( str[0] ); // H

alert( str.charAt(0) ); // H

// последний символ

alert( str[str.length - 1] ); // o

Квадратные скобки – современный способ получить символ, в то время как charAt существует в основном по историческим причинам. Разница только в том, что если символ с такой позицией отсутствует, тогда [] вернёт undefined, а charAt – пустую строку:

let str = `Hello`;

alert( str[1000] ); // undefined

alert( str.charAt(1000) ); // ''

Также можно перебрать строку посимвольно, используя for..of:

for (let char of "Hello") {

alert(char); // H,e,l,l,o

}

Содержимое строки в JavaScript нельзя изменить. Нельзя взять символ посередине и заменить его. Как только строка создана – она такая навсегда:

let str = 'Hi';

str[0] = 'h'; // ошибка

alert( str[0] ); // не работает

Можно создать новую строку и записать её в ту же самую переменную вместо старой. Например:

let str = 'Hi';

str = 'h' + str[1]; // заменяем строку

alert( str ); // hi

[**Изменение регистра**](https://learn.javascript.ru/string#izmenenie-registra)

Методы [toLowerCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toLowerCase) и [toUpperCase()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/toUpperCase) меняют регистр символов:

alert( 'Interface'.toUpperCase() ); // INTERFACE

alert( 'Interface'.toLowerCase() ); // interface

Если необходимо перевести в нижний регистр какой-то конкретный символ:

alert( 'Interface'[0].toLowerCase() ); // 'i'

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

Существует несколько способов поиска подстроки.

Первый метод – [str.indexOf(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/indexOf). Он ищет подстроку substr в строке str, начиная с позиции pos, и возвращает позицию, на которой располагается совпадение, либо -1 при отсутствии совпадений. Например:

let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, т.к. подстрока 'Widget' найдена в начале

alert( str.indexOf('widget') ); // -1, поиск чувствителен к регистру

alert( str.indexOf("id") ); // 1

Необязательный второй аргумент позволяет начать поиск с определённой позиции. Например, первое вхождение "id" – на позиции 1. Для того, чтобы найти следующее, необходимо начать поиск с позиции 2:

let str = 'Widget with id';

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

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

let str = 'Ослик Иа-Иа посмотрел на виадук';

let target = 'Иа'; // цель поиска

let pos = 0;

while (true) {

let foundPos = str.indexOf(target, pos);

if (foundPos == -1) break;

alert( `Найдено тут: ${foundPos}` );

pos = foundPos + 1; // продолжаем со следующей позиции

}

Также есть похожий метод [str.lastIndexOf(substr, position)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/lastIndexOf), который ищет с конца строки к её началу. Он используется тогда, когда нужно получить самое последнее вхождение: перед концом строки или начинающееся до (включительно) определённой позиции.

**Методы** [**includes, startsWith, endsWith**](https://learn.javascript.ru/string#includes-startswith-endswith)

Более современный метод [str.includes(substr, pos)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/includes) возвращает true, если в строке str есть подстрока substr, либо false, если нет. Стоит его использовать, если необходимо проверить, есть ли совпадение, но позиция не нужна:

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

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

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

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

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

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

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

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

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

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

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

Например:

let str = "stringify";

// 'strin', символы от 0 до 5 (не включая 5)

alert( str.slice(0, 5) );

// 's', от 0 до 1, не включая 1

alert( str.slice(0, 1) );

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

let str = "stringify";

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

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

let str = "stringify";

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

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

* str.substring(start [, end]) – возвращает часть строки между start и end.

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

let str = "stringify";

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

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

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

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

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

* str.substr(start [, length]) – возвращает часть строки от start длины length.

В противоположность предыдущим методам, этот позволяет указать длину вместо конечной позиции:

let str = "stringify";

alert( str.substr(2, 4) ); // ring

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

let str = "stringify";

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

Сравнительная таблица рассмотренных выше методов:

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

Метод slice более гибок, он поддерживает отрицательные аргументы, и его короче писать, поэтому его используют наиболее часто.

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

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

1. Строчные буквы больше заглавных:

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

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

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

Это может привести к своеобразным результатам при сортировке названий стран: ожидается, что Zealand будет после Österreich в списке. Чтобы разобраться, что происходит на самом деле, необходимо ознакомимся с внутренним представлением строк в JavaScript.

Строки кодируются в [UTF-16](https://ru.wikipedia.org/wiki/UTF-16). Таким образом, у любого символа есть соответствующий код. Есть специальные методы, позволяющие получить символ по его коду и наоборот:

* str.codePointAt(pos) – возвращает код для символа, находящегося на позиции pos. Одна и та же буква в нижнем и верхнем регистре будет иметь разные коды:

alert( "z".codePointAt(0) ); // 122

alert( "Z".codePointAt(0) ); // 90

* String.fromCodePoint(code) – создаёт символ по его коду code

alert( String.fromCodePoint(90) ); // Z

Также можно добавлять юникодные символы по их кодам, используя \u с шестнадцатеричным кодом символа:

// 90 – 5a в шестнадцатеричной системе счисления

alert( '\u005a' ); // Z

Можно сформировать строку, содержащую символы с кодами от 65 до 220 – это латиница и ещё некоторые распространённые символы:

let str = '';

for (let i = 65; i <= 220; i++) {

str += String.fromCodePoint(i);

}

alert( str );

// ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~

// ¡¢£¤¥¦§¨©ª«¬­®¯°±²³´µ¶·¸¹º»¼½¾¿ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜ

Как видино, сначала идут заглавные буквы, затем несколько спецсимволов, затем строчные и Ö ближе к концу вывода. Это объясняет, почему a > Z. Символы сравниваются по их кодам. Больший код – больший символ. Код a (97) больше кода Z (90):

* Все строчные буквы идут после заглавных, так как их коды больше.
* Некоторые буквы, такие как Ö, находятся вне основного алфавита. У этой буквы код больше, чем у любой буквы от a до z.

[**Правильное сравнение**](https://learn.javascript.ru/string#pravilnoe-sravnenie)

«Правильный» алгоритм сравнения строк сложнее, чем может показаться, так как разные языки используют разные алфавиты. Поэтому браузеру нужно знать, какой язык использовать для сравнения. Все современные браузеры (для IE10 нужна дополнительная библиотека [Intl.JS](https://github.com/andyearnshaw/Intl.js/)) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), обеспечивающий правильное сравние строк на разных языках с учётом их правил. Для этого есть соответствующий метод.

Вызов [str.localeCompare(str2)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare) возвращает число, которое показывает, какая строка больше в соответствии с правилами языка:

* Отрицательное число, если str меньше str2.
* Положительное число, если str больше str2.
* 0, если строки равны.

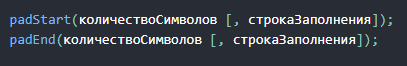
Например:

alert( 'Österreich'.localeCompare('Zealand') ); // -1

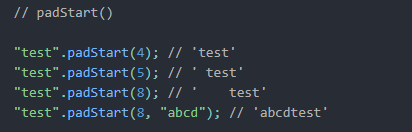
У этого метода есть два дополнительных аргумента, которые указаны в [документации](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/localeCompare). Первый позволяет указать язык (по умолчанию берётся из окружения) – от него зависит порядок букв. Второй – определить дополнительные правила, такие как чувствительность к регистру, а также следует ли учитывать различия между "a" и "á".

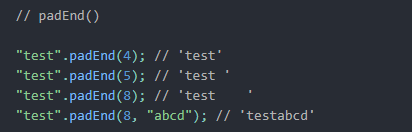
**Заполнение строки**

Целью заполнения строки является добавление символов в строку, чтобы она достигла определенной длины. ES2017 представляет два метода для строк padStart() и padEnd() - которые позволяют добавлять либо пустую строку, либо любую другую строку к началу или, концу исходной строки. Это оказывается удобным, если нужно выровнять текст, например, при выводе в консоль.



Пример использования:





1. **Массивы.**

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

[**Объявление**](https://learn.javascript.ru/array#obyavlenie)

Существует два варианта синтаксиса для создания пустого массива:

let arr = new Array();

let arr = [];

Практически всегда используется второй вариант синтаксиса. В скобках можно указать начальные значения элементов:

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

Элементы массива нумеруются, начиная с нуля. Можно получить элемент, указав его номер в квадратных скобках:

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

alert( fruits[0] ); // Яблоко

alert( fruits[1] ); // Апельсин

alert( fruits[2] ); // Слива

Можно заменить элемент:

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

Или добавить новый элемент к существующему массиву:

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

Общее число элементов массива содержится в его свойстве length:

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

alert( fruits.length ); // 3

Вывести массив целиком можно при помощи alert:

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

alert( fruits ); // Яблоко, Апельсин, Слива

В массиве могут храниться элементы любого типа:

let arr = [ 'Яблоко', { name: 'Джон' }, true, function() { alert('привет'); } ];

alert( arr[1].name ); // Джон

arr[3](); // привет

**«Висячая» запятая**

Массив может оканчиваться запятой:

let fruits = [

"Яблоко",

"Апельсин",

"Слива",

];

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

[**Методы pop/push, shift/unshift**](https://learn.javascript.ru/array#metody-pop-push-shift-unshift)

[Очередь](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_(%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)) – один из самых распространённых вариантов применения массива. В области компьютерных наук так называется упорядоченная коллекция элементов, поддерживающая два вида операций:

* push добавляет элемент в конец.
* shift удаляет элемент в начале, сдвигая очередь, так что второй элемент становится первым.

Массивы поддерживают обе операции. На практике необходимость в этом возникает очень часто. Например, очередь сообщений, которые надо показать на экране. Существует и другой вариант применения для массивов – структура данных, называемая [стек](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%BA)). Она поддерживает два вида операций:

* push добавляет элемент в конец.
* pop удаляет последний элемент.

Таким образом, новые элементы всегда добавляются или удаляются из «конца». Примером стека обычно служит колода карт: новые карты кладутся наверх и берутся тоже сверху. Массивы в JavaScript могут работать и как очередь, и как стек. Можно добавлять/удалять элементы как в начало, так и в конец массива.

В компьютерных науках структура данных, делающая это возможным, называется [двусторонняя очередь](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D1%83%D1%85%D1%81%D1%82%D0%BE%D1%80%D0%BE%D0%BD%D0%BD%D1%8F%D1%8F_%D0%BE%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C).

Методы, работающие с концом массива:

* pop – удаляет последний элемент из массива и возвращает его:

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

alert( fruits.pop() ); // удаляем "Груша" и выводим его

alert( fruits ); // Яблоко, Апельсин

* push – добавляет элемент в конец массива:

let fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");

alert( fruits ); // Яблоко, Апельсин, Груша

Вызов fruits.push(...) равнозначен fruits[fruits.length] = ....

Методы, работающие с началом массива:

* shift – удаляет из массива первый элемент и возвращает его:

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

alert( fruits.shift() ); // удаляем Яблоко и выводим его

alert( fruits ); // Апельсин, Груша

* unshift – добавляет элемент в начало массива:

let fruits = ["Апельсин", "Груша"];

fruits.unshift('Яблоко');

alert( fruits ); // Яблоко, Апельсин, Груша

Методы push и unshift могут добавлять сразу несколько элементов:

let fruits = ["Яблоко"];

fruits.push("Апельсин", "Груша");

fruits.unshift("Ананас", "Лимон");

alert( fruits ); // ["Ананас", "Лимон", "Яблоко", "Апельсин", "Груша"]

[**Внутреннее устройство массива**](https://learn.javascript.ru/array#vnutrennee-ustroystvo-massiva)

Массив – это особый подвид объектов. Квадратные скобки, используемые для того, чтобы получить доступ к свойству arr[0] – это обычный синтаксис доступа по ключу, как obj[key], где в роли obj выступает arr, а в качестве ключа – числовой индекс. Массивы расширяют объекты, так как предусматривают специальные методы для работы с упорядоченными коллекциями данных, а также свойство length. Но в основе все равно лежит объект.

Следует помнить, что в JavaScript существует всего 7 основных типов данных. Массив является объектом и, следовательно, ведёт себя как объект. Например, копируется по ссылке:

let fruits = ["Банан"]

let arr = fruits; // копируется по ссылке

alert( arr === fruits ); // true

arr.push("Груша"); // массив заполняется по ссылке

alert( fruits ); // Банан, Груша

Но то, что действительно делает массивы особенными – это их внутреннее представление. Движок JavaScript старается хранить элементы массива в непрерывной области памяти, один за другим. Существуют и другие способы оптимизации, благодаря которым массивы работают очень быстро. Но все они утратят эффективность, если перестать работать с массивом как с «упорядоченной коллекцией данных», и начать использовать его как обычный объект. Например, технически, можно сделать следующее:

let fruits = []; // создаём массив

fruits[99999] = 5; // создаём свойство с индексом, больше длины массива

fruits.age = 25; // создаём свойство с произвольным именем

Это возможно, потому что в основе массива лежит объект. Можно присвоить ему любые свойства. Но движок поймёт, что осуществляется работа с массивом, как с обычным объектом. Способы оптимизации, используемые для массивов, в этом случае не подходят и поэтому они будут отключены и никакой выгоды не принесут.

Варианты неправильного применения массива:

* Добавление нечислового свойства, например, arr.test = 5.
* Создание пустого пространства, например: добавление arr[0], затем arr[1000] (между ними ничего нет).
* Заполнение массива в обратном порядке, например: arr[1000], arr[999] и т.д.

Массив следует считать особой структурой, позволяющей работать с упорядоченными данными. Для этого массивы предоставляют специальные методы. Массивы тщательно настроены в движках JavaScript для работы с однотипными упорядоченными данными, поэтому, следует использовать их в таких случаях. Если нужны произвольные ключи, то лучше подойдёт обычный объект {}.

[**Эффективность**](https://learn.javascript.ru/array#effektivnost)

Методы push/pop выполняются быстро, а методы shift/unshift – медленно, т.е. у работать с концом массива быстрее, чем с его началом. Давайте посмотрим, что происходит во время выполнения:

fruits.shift(); // удаляем 1-ый элемент с начала

Просто взять и удалить элемент с номером 0 недостаточно. Нужно также заново пронумеровать остальные элементы. Операция shift должна выполнить 3 действия:

1. Удалить элемент с индексом 0.
2. Сдвинуть все элементы влево, заново пронумеровать их, заменив 1 на 0, 2 на 1 и т.д.
3. Обновить свойство length .

Чем больше элементов содержит массив, тем больше времени потребуется для того, чтобы их переместить, больше операций с памятью.

То же самое происходит с unshift: чтобы добавить элемент в начало массива, нужно сначала сдвинуть существующие элементы вправо, увеличивая их индексы.

В процессе работы методам push/pop не нужно ничего перемещать. Чтобы удалить элемент в конце массива, метод pop очищает индекс и уменьшает значение length. Действия при операции pop:

fruits.pop(); // удаляем 1 элемент с конца

Метод pop не требует перемещения, потому что остальные элементы остаются на тех же индексах. Именно поэтому он выполняется очень быстро.

Аналогично работает метод push.

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

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

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

for (let i = 0; i < arr.length; i++) {

alert( arr[i] );

}

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

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

// проходит по значениям

for (let fruit of fruits) {

alert( fruit );

}

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

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

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

for (let key in arr) {

alert( arr[key] ); // Яблоко, Апельсин, Груша

}

Но так делать не рекомендуется. Существуют скрытые недостатки этого способа:

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

В браузере и других программных средах также существуют так называемые «псевдомассивы» – объекты, которые выглядят, как массив. То есть, у них есть свойство length и индексы, но также они могут иметь дополнительные нечисловые свойства и методы, которые обычно не нужны. Тем не менее, цикл for..in выведет и их. Поэтому, если приходится иметь дело с объектами, похожими на массив, такие «лишние» свойства могут стать проблемой.

1. Цикл for..in оптимизирован под произвольные объекты, не массивы, и поэтому в 10-100 раз медленнее. Увеличение скорости выполнения может иметь значение только при возникновении узких мест.

Поэтому не следует использовать цикл for..in для массивов.

[**Свойство length**](https://learn.javascript.ru/array#nemnogo-o-length)

Свойство length автоматически обновляется при изменении массива. Если быть точными, это не количество элементов массива, а наибольший цифровой индекс плюс один. Например, единственный элемент, имеющий большой индекс, даёт большую длину:

let fruits = [];

fruits[123] = "Яблоко";

alert( fruits.length ); // 124

Свойство length можно перезаписать. Если вручную увеличить его, то ничего особенного не произойдет. Но если уменьшить – массив станет короче. Этот процесс необратим:

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

arr.length = 2; // укорачиваем до 2 элементов

alert( arr ); // [1, 2]

arr.length = 5; // возвращаем length как было

alert( arr[3] ); // undefined: значения не восстановились

Таким образом, самый простой способ очистить массив – это arr.length = 0;.

[**new Array()**](https://learn.javascript.ru/array#new-array)

Существует ещё один вариант синтаксиса для создания массива:

let arr = new Array("Яблоко", "Груша", "и тд");

Он редко применяется, так как квадратные скобки [] короче. Кроме того, у него есть одна особенность: если new Array вызывается с одним аргументом, который представляет собой число, он создаёт массив без элементов, но с заданной длиной:

let arr = new Array(2); // создаем массив [2]

alert( arr[0] ); // undefined! нет элементов.

alert( arr.length ); // length 2

Как видно из кода, представленного выше, в new Array(number) все элементы равны undefined. Чтобы избежать появления таких неожиданных ситуаций, обычно используются квадратные скобки, если конечно нет какой-то причины для использования именно Array.

[**Многомерные массивы**](https://learn.javascript.ru/array#mnogomernye-massivy)

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

let matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

alert( matrix[1][1] ); // 5, центральный элемент

**Метод** [**toString**](https://learn.javascript.ru/array#tostring)

Массивы по-своему реализуют метод toString, который возвращает список элементов, разделённых запятыми. Например:

let arr = [1, 2, 3];

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

alert( String(arr) === '1,2,3' ); // true

alert( [] + 1 ); // "1"

alert( [1] + 1 ); // "11"

alert( [1,2] + 1 ); // "1,21"

Массивы не имеют ни Symbol.toPrimitive, ни функционирующего valueOf, они реализуют только преобразование toString, таким образом здесь [] становится пустой строкой, [1] становится "1", а [1,2] становится "1,2". Когда бинарный оператор плюс "+" добавляет что-либо к строке, он тоже преобразует это в строку, таким образом:

alert( "" + 1 ); // "1"

alert( "1" + 1 ); // "11"

alert( "1,2" + 1 ); // "1,21"

1. **Методы массивов.**

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

Методы, которые добавляют и удаляют элементы из начала или конца, были рассмотрены ранее:

* arr.push(...items) – добавляет элементы в конец,
* arr.pop() – извлекает элемент из конца,
* arr.shift() – извлекает элемент из начала,
* arr.unshift(...items) – добавляет элементы в начало.

Есть и другие.

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

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

Его синтаксис:

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

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

Рассмотрим пример удаления элементов:

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

arr.splice(1, 1); // начиная с позиции 1, удалить 1 элемент

alert( arr ); // осталось ["Я", "JavaScript"]

Начиная с позиции 1, метод удалил 1 элемент.

Удалим 3 элемента и заменим их двумя другими:

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

arr.splice(0, 3, "Давай", "танцевать");

alert( arr ) // теперь ["Давай", "танцевать", "прямо", "сейчас"]

Здесь видно, что splice возвращает массив из удалённых элементов:

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

// удалить 2 первых элемента

let removed = arr.splice(0, 2);

alert( removed ); // "Я", "изучаю" – массив из удалённых элементов

Метод splice также может вставлять элементы без удаления, для этого достаточно установить deleteCount в 0:

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

// с позиции 2 удалить 0 элементов и вставить "сложный", "язык"

arr.splice(2, 0, "сложный", "язык");

alert( arr ); // "Я", "изучаю", "сложный", "язык", "JavaScript"

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

let arr = [1, 2, 5];

// начиная с индекса -1 (перед последним элементом) удалить 0 элементов,

// затем вставить числа 3 и 4

arr.splice(-1, 0, 3, 4);

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

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

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

arr.slice(start, end)

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

let arr = ["t", "e", "s", "t"];

alert( arr.slice(1, 3) ); // e,s

alert( arr.slice(-2) ); // s,t

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

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

arr.concat(arg1, arg2...)

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

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

let arr = [1, 2];

alert( arr.concat([3, 4])); // 1,2,3,4

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

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

Обычно он просто копирует элементы из массивов. Другие объекты, даже если они выглядят как массивы, добавляются как есть:

let arr = [1, 2];

let arrayLike = {

0: "что-то",

length: 1

};

alert( arr.concat(arrayLike) ); // 1,2,[object Object]

//[1, 2, arrayLike]

Но если объект имеет специальное свойство Symbol.isConcatSpreadable, то он обрабатывается concat как массив: вместо него добавляются его элементы:

let arr = [1, 2];

let arrayLike = {

0: "что-то",

1: "ещё",

[Symbol.isConcatSpreadable]: true,

length: 2

};

alert( arr.concat(arrayLike) ); // 1,2,что-то,ещё

[**Перебор массива: метод forEach**](https://learn.javascript.ru/array-methods#perebor-foreach)

Метод [arr.forEach](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) позволяет запускать функцию для каждого элемента массива. Его синтаксис:

arr.forEach(function(item, index, array) {

// вычисления с item

});

Например, этот код выведет на экран каждый элемент массива:

["Bilbo", "Gandalf", "Nazgul"].forEach(alert);

А этот выведет значение элемента и его позицию в массиве:

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

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

});

Результат функции (если она вообще что-то возвращает) отбрасывается и игнорируется.

**Методы** [**поиска**](https://learn.javascript.ru/array-methods#poisk-v-massive) **в массиве**

Методы [arr.indexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf), [arr.lastIndexOf](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/lastIndexOf) и [arr.includes](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/includes) имеют одинаковый синтаксис и делают по сути то же самое, что и их строковые аналоги, но работают с элементами вместо символов:

* arr.indexOf(item, from) ищет item, начиная с индекса from, и возвращает индекс, на котором был найден искомый элемент, в противном случае -1.
* arr.lastIndexOf(item, from) – то же самое, но ищет справа налево.
* arr.includes(item, from) – ищет item, начиная с индекса from, и возвращает true, если поиск успешен.

Например:

let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1

alert( arr.indexOf(false) ); // 2

alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true

Обратите внимание, что методы используют строгое сравнение ===. Таким образом, если осуществляется поиск false, то он находит именно false, а не ноль.

Если надо проверить наличие элемента и нет необходимости знать его точный индекс, тогда предпочтительным является arr.includes.

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

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

// если true - возвращается текущий элемент и перебор прерывается

// если все итерации оказались ложными возвращается undefined

});

Функция вызывается по очереди для каждого элемента массива:

* item – очередной элемент.
* index – его индекс.
* array – сам массив.

Если функция вернёт true, поиск прерывается и возвращается item. Если ничего не найдено, возвращается undefined. Например, имеется массив пользователей, каждый из которых имеет поля id и name. Найдем того, кто с id == 1:

let users = [

{id: 1, name: "Вася"},

{id: 2, name: "Петя"},

{id: 3, name: "Маша"}

];

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

alert(user.name); // Вася

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

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

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

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

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

});

Например:

let users = [

{id: 1, name: "Вася"},

{id: 2, name: "Петя"},

{id: 3, name: "Маша"}

];

// возвращает массив, состоящий из двух первых пользователей

let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2

[**Преобразование массива**](https://learn.javascript.ru/array-methods#preobrazovanie-massiva)

Метод [arr.map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/map) является одним из наиболее полезных и часто используемых. Он вызывает функцию для каждого элемента массива и возвращает массив результатов выполнения этой функции. Синтаксис:

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

// возвращается новое значение вместо элемента

});

Например, в коде ниже осуществляется преобразование каждого элемента в его длину:

let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);

alert(lengths); // 5,7,6

Вызов [arr.sort()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/sort) сортирует массив на месте, меняя в нём порядок элементов. Он возвращает отсортированный массив, но обычно возвращаемое значение игнорируется, так как изменяется сам arr. Например:

let arr = [ 1, 2, 15 ];

arr.sort();

alert( arr ); // 1, 15, 2

Порядок стал 1, 15, 2 так как элементы преобразуются в строки при сравнении и по умолчанию сортируются как строки. Для сортировки строк применяется лексикографический порядок где "2" > "15". Чтобы задать свой порядок сортировки, нужно предоставить функцию в качестве аргумента arr.sort(). Функция может возвращать следующие значения:

function compare(a, b) {

if (a > b) return 1; // если первое значение больше второго

if (a == b) return 0; // если равны

if (a < b) return -1; // если первое значение меньше второго

}

Например, для сортировки чисел:

function compareNumeric(a, b) {

if (a > b) return 1;

if (a == b) return 0;

if (a < b) return -1;

}

let arr = [ 1, 2, 15 ];

arr.sort(compareNumeric);

alert(arr); // 1, 2, 15

Функция сравнения может возвращать любое положительное число, если первый сравниваемый элемент больше второго и отрицательное – если меньше. Это позволяет писать более короткие функции:

let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });

alert(arr); // 1, 2, 15

Можно использовать стрелочные функции, чтобы сортировка выглядела более компактной:

arr.sort( (a, b) => a - b );

Будет работать точно так же, как и более длинная версия выше.

Метод [arr.reverse](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse) меняет порядок элементов в arr на обратный. Например:

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

arr.reverse();

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

Он также возвращает массив arr с изменённым порядком элементов.

Метод [str.split(delim)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/String/split) разбивает строку на массив по заданному разделителю delim. В примере ниже таким разделителем является строка из запятой и пробела.

let names = 'Вася, Петя, Маша';

let arr = names.split(', ');

for (let name of arr) {

alert( `Сообщение получат: ${name}.` );

}

У метода split есть необязательный второй числовой аргумент – ограничение на количество элементов в массиве. Если их больше, чем указано, то остаток массива будет отброшен. На практике это редко используется:

let arr = 'Вася, Петя, Маша, Саша'.split(', ', 2);

alert(arr); // Вася, Петя

Вызов split(s) с пустым аргументом s разделяет строку на массив букв:

let str = "тест";

alert( str.split('') ); // т,е,с,т

Вызов [arr.join(glue)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/join) выполняет действие противоположное split. Он создаёт строку из элементов arr, вставляя glue между ними. Например:

let arr = ['Вася', 'Петя', 'Маша'];

let str = arr.join(';'); // объединить массив в строку через ;

alert( str ); // Вася;Петя;Маша

Если надо перебрать массив – можно использовать forEach, for или for..of. Если надо перебрать массив и возвратить данные для каждого элемента – стоит используем map. Методы [arr.reduce](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce) и [arr.reduceRight](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) похожи на методы выше, но они немного сложнее. Они используются для вычисления какого-нибудь единого значения на основе всего массива. Синтаксис:

let value = arr.reduce(function(previousValue, item, index, array) {

// ...

}, [initial]);

Функция применяется по очереди ко всем элементам массива и «переносит» свой результат на следующий вызов. Аргументы:

* previousValue – результат предыдущего вызова этой функции, равен initial при первом вызове (если передан initial),
* item – очередной элемент массива,
* index – индекс элемента,
* array – массив.

При вызове функции результат её вызова на предыдущем элементе массива передаётся как первый аргумент. Звучит сложновато, но всё становится проще, если думать о первом аргументе как «аккумулирующем» результат предыдущих вызовов функции. По окончании он становится результатом reduce. Этот метод проще всего понять на примере. Получим сумму всех элементов массива:

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

let result = arr.reduce((sum, current) => sum + current, 0);

alert(result); // 15

Здесь использован наиболее распространённый вариант reduce, который использует только 2 аргумента. Рассмотрим, как он работает:

1. При первом запуске, sum равен initial (последний аргумент reduce), то есть 0, а current – первый элемент массива, равнй 1. Таким образом, результат функции равен 1.
2. При втором запуске sum = 1, и к нему добавляется второй элемент массива (2).
3. На 3-м запуске sum = 3, к которому добавляется следующий элемент и так далее.

Ниже представлена таблица, где каждая строка – вызов функции на очередном элементе массива:

|  | **sum** | **current** | **result** |
| --- | --- | --- | --- |
| первый вызов | 0 | 1 | 1 |
| второй вызов | 1 | 2 | 3 |
| третий вызов | 3 | 3 | 6 |
| четвёртый вызов | 6 | 4 | 10 |
| пятый вызов | 10 | 5 | 15 |

Здесь отчётливо видно, как результат предыдущего вызова передаётся в первый аргумент следующего. Также можно опустить начальное значение:

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

let result = arr.reduce((sum, current) => sum + current);

alert( result ); // 15

Результат такой же потому, что при отсутствии initial в качестве первого значения берётся первый элемент массива, а перебор стартует со второго. Таблица вычислений будет такая же за вычетом первой строки. Но такое использование требует крайней осторожности. Если массив пуст, то вызов reduce без начального значения выдаст ошибку, поэтому рекомендуется всегда его указывать:

let arr = [];

// Error: Reduce of empty array with no initial value

arr.reduce((sum, current) => sum + current);

Метод [arr.reduceRight](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/reduceRight) работает аналогично, но проходит по массиву справа налево.

Массивы не образуют отдельный тип языка. Они основаны на объектах. Поэтому typeof не может отличить простой объект от массива:

alert(typeof {}); // object

alert(typeof []); // object

Метод [Array.isArray(value)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/isArray) возвращает true, если value массив, и false, если нет:

alert(Array.isArray({})); // false

alert(Array.isArray([])); // true

Почти все методы массива, которые вызывают функции – такие как find, filter, map, за исключением метода sort, принимают необязательный параметр thisArg. Этот параметр очень редко используется. Полный синтаксис этих методов:

arr.find(func, thisArg);

arr.filter(func, thisArg);

arr.map(func, thisArg);

Значение параметра thisArg становится this для func. Например, если необходимо использовать метод объекта как фильтр, то thisArg с этим поможет:

let user = {

age: 18,

younger(otherUser) {

return otherUser.age < this.age;

}

};

let users = [

{age: 12},

{age: 16},

{age: 32}

];

// найти число пользователей моложе, чем заданный

let youngerUsers = users.filter(user.younger, user);

alert(youngerUsers.length); // 2

В вызове выше используется user.younger как фильтр, а user – в качестве контекста для него. Если бы не предоставлялся контекст, users.filter(user.younger) вызвал бы user.younger как ни к чему не привязанную функцию с this=undefined. Это привело бы кошибке.

1. **Объекты.**

Как известно, в JavaScript существует семь типов данных. Шесть из них называются примитивными, так как содержат только одно значение. Объекты же используются для хранения коллекций различных значений и более сложных сущностей. В JavaScript объекты используются очень часто, это одна из основ языка. Поэтому стоит их досконально изучить.

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

Пустой объект можно создать, используя один из двух вариантов синтаксиса:

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

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

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 {...} сразу можно поместить в объект несколько свойств в виде пар «ключ: значение»:

let user = { // объект

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

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

};

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

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. Добавим свойство с логическим значением:

user.isAdmin = true;

Для удаления свойства можно использовать оператор delete:

delete user.age;

Имя свойства может состоять из нескольких слов, но тогда оно должно быть заключено в кавычки. Последнее свойство объекта может заканчиваться висячей запятой:

let user = {

name: "John",

age: 30,

"likes birds": true,

};

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

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

// приведет к синтаксической ошибкк

user.likes birds = true

Так происходит, потому что 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"];

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

let key = "likes birds";

user[key] = true;

Здесь переменная key может быть вычислена во время выполнения кода или зависеть от пользовательского ввода. После этого можно использовать её для доступа к свойству. Пример:

let user = {

name: "John",

age: 30

};

let key = prompt("Что вы хотите узнать о пользователе?", "name");

alert( user[key] ); // John (если ввели "name")

Запись «через точку» такого не позволяет:

let user = {

name: "John",

age: 30

};

let key = "name";

user.key // undefined

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

Можно 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. И если посетитель введёт слово "apple", то в объекте bag теперь будет храниться свойство {apple: 5}.

Можно использовать и более сложные выражения в квадратных скобках:

let fruit = 'apple';

let bag = {

[fruit + 'Computers']: 5 // bag.appleComputers = 5

};

Таким образом, когда имена свойств известны и просты, используется запись через точку. Если же нужно что-то более сложное, то следует использовать квадратные скобки. Зарезервированные слова разрешено использовать как имена свойств:

let obj = {

for: 1,

let: 2,

return: 3

};

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

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

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

function makeUser(name, age) {

return {

name: name,

age: age

// ...другие свойства

};

}

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

alert(user.name); // John

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

function makeUser(name, age) {

return {

name,

age

height: 170// ...

};

}

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

Особенность 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 – сравнением его с undefined:

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

Пример:

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

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

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

Обратите внимание, что слева от оператора in должно быть имя свойства. Обычно это строка в кавычках. Если кавычки опускаются, это значит, что используется переменная, в которой находится имя свойства. Например:

let user = { age: 30 };

let key = "age";

alert( key in user ); // true

Обычно строгого сравнения "=== undefined" достаточно для проверки наличия свойства. Но есть особый случай, когда оно не подходит, и нужно использовать "in". Это когда свойство существует, но содержит значение undefined:

let obj = {

test: undefined

};

alert( obj.test ); // выведет undefined

alert( "test" in obj ); // true, свойство существует

В примере выше свойство obj.test технически существует в объекте. Оператор in сработал правильно. Подобные ситуации случаются очень редко, так как undefined обычно явно не присваивается. Для «неизвестных» или «пустых» свойств используется значение null.

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

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

for (key in object) {

// тело цикла выполняется для каждого свойства объекта

}

Например, выведем все свойства объекта user:

let user = {

name: "John",

age: 30,

isAdmin: true

};

for (let key in user) {

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

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

}

Обратите внимание, что все конструкции «for» позволяют объявлять переменную внутри цикла, как, например, let key здесь. Кроме того, можно использовать другое имя переменной. Например, часто используется вариант "for (let prop in obj)".

[**Упорядочение свойств объекта**](https://learn.javascript.ru/object#uporyadochenie-svoystv-obekta)

Свойства объекта упорядочены особым образом: свойства с целочисленными ключами сортируются по возрастанию, остальные располагаются в порядке создания. В качестве примера рассмотрим объект с телефонными кодами:

let codes = {

"49": "Германия",

"41": "Швейцария",

"44": "Великобритания",

// ..,

"1": "США"

};

for (let code in codes) {

alert(code); // 1, 41, 44, 49

}

Телефонные коды идут в порядке возрастания, потому что они являются целыми числами: 1, 41, 44, 49.

Термин «целочисленное свойство» означает строку, которая может быть преобразована в целое число и обратно без изменений. То есть, "49" – это целочисленное имя свойства, потому что если его преобразовать в целое число, а затем обратно в строку, то оно не изменится. А вот свойства "+49" или "1.2" таковыми не являются. Если ключи не целочисленные, то они перебираются в порядке создания, например:

let user = {

name: "John",

surname: "Smith"

};

user.age = 25;

for (let prop in user) {

alert( prop ); // name, surname, age

}

Таким образом, чтобы телефонные коды выводились, в том порядке в котром записаны, надо сделать коды не целочисленными свойствами. Для этого надо добавить знак "+" перед каждым кодом. Пример:

let codes = {

"+49": "Германия",

"+41": "Швейцария",

"+44": "Великобритания",

// ..,

"+1": "США"

};

for (let code in codes) {

alert( +code ); // 49, 41, 44, 1

}

**[Копирование по ссылке](https://learn.javascript.ru/object" \l "kopirovanie-po-ssylke)**

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». Например:

let message = "Hello!";

let phrase = message;

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

let user = {

name: "John"

};

Сам объект хранится где-то в памяти. А в переменной user лежит «ссылка» на эту область памяти. Когда переменная объекта копируется – копируется ссылка, сам же объект не дублируется:

let user = { name: "John" };

let admin = user; // копируется ссылка

В результате есть две переменные, каждая из которых содержит ссылку на один и тот же объект. Можно использовать любую из переменных для доступа к объекту и изменения его содержимого:

let user = { name: 'John' };

let admin = user;

admin.name = 'Pete'; // изменено по ссылке из переменной "admin"

alert(user.name); // 'Pete', изменения видны по ссылке из переменной "user"

**[Сравнение объектов](https://learn.javascript.ru/object" \l "sravnenie-obektov)**

Операторы равенства == и строгого равенства === для объектов работают одинаково. Два объекта равны только в том случае, если это один и тот же объект. Например, две переменные ссылаются на один и тот же объект, они равны:

let a = {};

let b = a; // копирование по ссылке

alert( a == b ); // true

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

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

let a = {};

let b = {}; // два независимых объекта

alert( a == b ); // false

Для сравнений типа obj1 > obj2 или для сравнения с примитивом obj == 5 объекты преобразуются в примитивы. Такое сравнение используется очень редко и не рекомендуется.

**[Объекты-константы](https://learn.javascript.ru/object" \l "obekty-konstanty)**

Объект, объявленный через const, может быть изменен:

const user = {

name: "John"

};

user.age = 25; // (\*)

alert(user.age); // 25

Объявление const защищает от изменений только само значение user. В примере значение user – это ссылка на объект, и это значение не меняется. В строке (\*) вносятся изменения внутри объекта, а значение user не изменяется. Если же попытаться присвоить user другое значение, то const выдаст ошибку:

const user = {

name: "John"

};

// Ошибка (нельзя переопределять константу user)

user = {

name: "Pete"

};

Cделать константами свойства объекта тоже возможно с помощью флагов и дескрипторов свойств, кторые будут рассматриваться позже.

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

При копировании переменной объекта создаётся ещё одна ссылка на тот же самый объект. Если надо создать независимую копию (клон), то необходимо создать новый объект и повторять структуру дублируемого объекта, перебирая его свойства и копируя их. Например так:

let user = {

name: "John",

age: 30

};

let clone = {};

for (let key in user) {

clone[key] = user[key];

}

clone.name = "Pete";

alert( user.name );

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

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

* Аргументы dest, и src1, ..., srcN (может быть столько, сколько нужно) являются объектами.
* Метод копирует свойства всех объектов src1, ..., srcN в объект dest. То есть, свойства всех перечисленных объектов, начиная со второго, копируются в первый объект. После копирования метод возвращает объект dest.

Например, объединим несколько объектов в один:

let user = { name: "John" };

let permissions1 = { canView: true };

let permissions2 = { canEdit: true };

// user = { name: "John", canView: true, canEdit: true }

Object.assign(user, permissions1, permissions2);

Если принимающий объект (user) уже имеет свойство с таким именем, оно будет перезаписано:

let user = { name: "John" };

// user = { name: "Pete", isAdmin: true }

Object.assign(user, { name: "Pete", isAdmin: true });

Также можно использовать Object.assign для простого клонирования:

let user = {

name: "John",

age: 30

};

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

Все свойства объекта user будут скопированы в пустой объект, и ссылка на этот объект будет в переменной clone. Такое клонирование работает так же, как и через цикл, но короче.

Если свойство не примитивно, а явлется ссылкой на другой объект, то 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: "John",

sizes: {

height: 182,

width: 50

}

};

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

alert( user.sizes === clone.sizes );

user.sizes.width++;

alert(clone.sizes.width); // 51

Чтобы исправить это, необходимо в цикле клонирования делать проверку, не является ли значение user[key] объектом, и, если это так, – копировать и его структуру тоже. Это называется «глубокое клонирование». Существует стандартный алгоритм глубокого клонирования, [Structured cloning algorithm](http://w3c.github.io/html/infrastructure.html#safe-passing-of-structured-data). Он решает описанную выше задачу, а также более сложные задачи.

1. **Тип данных Symbol**

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

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

let id = Symbol();

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

let id = Symbol("id");

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

let id1 = Symbol("id");

let id2 = Symbol("id");

alert(id1 == id2); // false

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

let id = Symbol("id");

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

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

let id = Symbol("id");

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

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

let id = Symbol("id");

alert(id.description); // id

[**«Скрытые» свойства**](https://learn.javascript.ru/symbol#skrytye-svoystva) **объекта**

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

let user = { name: "Егор" };

let id = Symbol("id");

user[id] = "ID Value";

alert( user[id] );

Лучше использовать Symbol("id"), а не строку "id", так как объект user принадлежит стороннему коду, и этот код также работает с ним, то не следует добавлять к нему какие-либо поля. Это небезопасно. Но к символу нечаянно обратиться сложно, сторонний код вряд ли его вообще увидит. Сторонний код может создать для этого свой символ Symbol("id"):

let id = Symbol("id");

user[id] = "Их идентификатор";

Конфликта между двумя идентификатором не будет, так как символы всегда уникальны, даже если их имена совпадают. А вот если использовать строку "id" вместо символа, то тогда будет конфликт:

let user = { name: "Егор" };

user.id = "Первый идентификатор";

// свойство перезаписано

user.id = "Второй идентификатор"

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

let id = Symbol("id");

let user = {

name: "Егор",

[id]: 123

};

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

Свойства, чьи ключи – символы, не перебираются циклом for..in. Например:

let id = Symbol("id");

let user = {

name: "Егор",

age: 30,

[id]: 123

};

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

alert( "Напрямую: " + user[id] );

Это невозможно с целью скрытия символьных свойств. Если другая библиотека или скрипт будут работать с текущим объектом, то при переборе они не получат символьное свойство. Object.keys(user) также игнорирует символы. А вот [Object.assign](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/assign), в отличие от цикла for..in, копирует и строковые, и символьные свойства:

let id = Symbol("id");

let user = {

[id]: 123

};

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

alert( clone[id] ); // 123

Так происходит потому, что, когда осуществляется клонирование или объединение объектов, обычно необходимо скопировать все свойства (включая такие свойства с ключами-символами, как, например, id в примере выше).

[**Глобальные символы**](https://learn.javascript.ru/symbol#globalnye-simvoly)

Иногда необходимо, чтобы символы с одинаковыми именами были одной сущностью. Например, разные части приложения хотят получить доступ к символу "id", подразумевая именно одно и то же свойство. Для этого существует глобальный реестр символов. Можно создавать в нём символы и обращаться к ним позже, и при каждом обращении гарантированно будет возвращаться один и тот же символ.

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

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

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

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

Символы, содержащиеся в реестре, называются глобальными символами. Если нужен символ, доступный везде в коде – используйте глобальные символы.

Для глобальных символов, кроме Symbol.for(key), который ищет символ по имени, существует обратный метод: Symbol.keyFor(sym), который, наоборот, принимает глобальный символ и возвращает его имя. Например:

// получаем символ по имени

let sym = Symbol.for("name");

let sym2 = Symbol.for("id");

// получаем имя по символу

alert( Symbol.keyFor(sym) ); // name

alert( Symbol.keyFor(sym2) ); // id

Внутри метода Symbol.keyFor используется глобальный реестр символов для нахождения имени символа. Так что этот метод не будет работать для неглобальных символов. Если символ неглобальный, метод не сможет его найти и вернёт undefined. Не стоит забывать, что для любых символов доступно свойство description.

**[Системные символы](https://learn.javascript.ru/symbol" \l "sistemnye-simvoly)**

Существует множество «системных» символов, использующихся внутри самого JavaScript, и можно использовать их, чтобы настраивать различные аспекты поведения объектов. Эти символы перечислены в спецификации в таблице [Well-known symbols](https://tc39.github.io/ecma262/#sec-well-known-symbols):

* Symbol.hasInstance
* Symbol.isConcatSpreadable
* Symbol.iterator
* Symbol.toPrimitive и др.

В частности, Symbol.toPrimitive позволяет описать правила для объекта, согласно которым он будет преобразовываться к примитиву.

1. **Set, Map, WeakSet и WeakMap**

Ранее рассматривались следующие сложные структуры данных: объекты – для хранения именованных коллекций, массивы – для хранения упорядоченных коллекций. Существуют и другие более сложные коллекции.

[**Map**](https://learn.javascript.ru/map-set#map)

[Map](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Map) – это коллекция ключ/значение, как и Object. Но основное отличие в том, что Map позволяет использовать ключи любого типа. Методы и свойства:

* new Map() – создаёт коллекцию.
* map.set(key, value) – записывает по ключу key значение value.
* map.get(key) – возвращает значение по ключу или undefined, если ключ key отсутствует.
* map.has(key) – возвращает true, если ключ key присутствует в коллекции, иначе false.
* map.delete(key) – удаляет элемент по ключу key.
* map.clear() – очищает коллекцию от всех элементов.
* map.size – возвращает текущее количество элементов.

Например:

let map = new Map();

map.set("1", "str1"); // строка в качестве ключа

map.set(1, "num1"); // цифра как ключ

map.set(true, "bool1"); // булево значение как ключ

alert(map.get(1)); // "num1"

alert(map.get("1")); // "str1"

alert(map.size); // 3

Как мы видим, в отличие от объектов, ключи не были приведены к строкам. Можно использовать любые типы данных для ключей, даже объекты. Например:

let john = { name: "John" };

let visitsCountMap = new Map();

visitsCountMap.set(john, 123);

alert(visitsCountMap.get(john)); // 123

Использование объектов в качестве ключей – это одна из известных и часто применяемых возможностей объекта Map. При строковых ключах обычный объект Object может подойти, но для ключей-объектов – уже нет.

Чтобы сравнивать ключи, объект Map использует алгоритм [SameValueZero](https://tc39.github.io/ecma262/#sec-samevaluezero). Это почти такое же сравнение, что и ===, с той лишь разницей, что NaN считается равным NaN. Так что NaN также может использоваться в качестве ключа. Этот алгоритм не может быть заменён или модифицирован.

Каждый вызов map.set возвращает объект map, так что можно объединить вызовы в цепочку:

map.set("1", "str1")

.set(1, "num1")

.set(true, "bool1");

Для перебора коллекциии Map есть 3 метода:

* map.keys() – возвращает итерируемый объект по ключам,
* map.values() – возвращает итерируемый объект по значениям,
* map.entries() – возвращает итерируемый объект по парам вида [ключ, значение], этот вариант используется по умолчанию в for..of.

Например:

let recipeMap = new Map([

["огурец", 500],

["помидор", 350],

["лук", 50]

]);

// перебор по ключам (овощи)

for (let vegetable of recipeMap.keys()) {

alert(vegetable); // огурец, помидор, лук

}

// перебор по значениям (числа)

for (let amount of recipeMap.values()) {

alert(amount); // 500, 350, 50

}

// перебор по элементам в формате [ключ, значение]

for (let entry of recipeMap) {

alert(entry); // огурец,500 (и так далее)

}

В отличие от обычных объектов Object, в Map перебор происходит в том же порядке, в каком происходило добавление элементов. Кроме этого, Map имеет встроенный метод forEach, схожий со встроенным методом массивов Array:

// выполняем функцию для каждой пары (ключ, значение)

recipeMap.forEach((value, key, map) => {

alert(`${key}: ${value}`); // огурец: 500 и так далее

});

[**Создание Map из Object**](https://learn.javascript.ru/map-set#object-entries-map-iz-object)

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

let obj = {

name: "John",

age: 30

};

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

Здесь Object.entries возвращает массив пар ключ-значение: [ ["name","John"], ["age", 30] ]. Это именно то, что нужно для создания Map.

**[Создание Object из Map](https://learn.javascript.ru/map-set" \l "object-fromentries-object-iz-map)**

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

let prices = Object.fromEntries([

['banana', 1],

['orange', 2],

['meat', 4]

]);

// prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2

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

let map = new Map();

map.set('banana', 1);

map.set('orange', 2);

map.set('meat', 4);

let obj = Object.fromEntries(map);

// obj = { banana: 1, orange: 2, meat: 4 }

alert(obj.orange); // 2

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

**[Set](https://learn.javascript.ru/map-set" \l "set)**

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

* new Set(iterable) – создаёт Set, и если в качестве аргумента был предоставлен итерируемый объект (обычно это массив), то копирует его значения в новый Set.
* set.add(value) – добавляет значение (если оно уже есть, то ничего не делает), возвращает тот же объект set.
* set.delete(value) – удаляет значение, возвращает true если value было в множестве на момент вызова, иначе false.
* set.has(value) – возвращает true, если значение присутствует в множестве, иначе false.
* set.clear() – удаляет все имеющиеся значения.
* set.size – возвращает количество элементов в множестве.

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

let set = new Set();

let john = { name: "John" };

let pete = { name: "Pete" };

let mary = { name: "Mary" };

// считаем гостей, некоторые приходят несколько раз

set.add(john);

set.add(pete);

set.add(mary);

set.add(john);

set.add(mary);

// set хранит только 3 уникальных значения

alert(set.size); // 3

for (let user of set) {

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

}

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

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

let set = new Set(["апельсин", "яблоко", "банан"]);

for (let value of set) alert(value);

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

alert(value);

});

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

* set.keys() – возвращает перебираемый объект для значений,
* set.values() – то же самое, что и set.keys(), присутствует для обратной совместимости с Map,
* set.entries() – возвращает перебираемый объект для пар вида [значение, значение], присутствует для обратной совместимости с Map.

**WeakMap и WeakSet**

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

let john = { name: "John" };

// объект доступен, переменная john -- это ссылка на него

// перепишем ссылку

john = null;

// объект будет удалён из памяти

Обычно свойства объекта, элементы массива или другой структуры данных считаются достижимыми и сохраняются в памяти до тех пор, пока эта структура данных содержится в памяти. Например, если поместить объект в массив, то до тех пор, пока массив существует, объект также будет существовать в памяти, несмотря на то, что других ссылок на него нет. Например:

let john = { name: "John" };

let array = [ john ];

john = null; // перезаписываем ссылку на объект

// объект john хранится в массиве, поэтому он не будет удалён

Аналогично, если используется объект как ключ в Map, то до тех пор, пока существует Map, также будет существовать и этот объект. Он занимает место в памяти и не может быть удалён сборщиком мусора. Например:

let john = { name: "John" };

let map = new Map();

map.set(john, "...");

john = null; // перезаписываем ссылку на объект

// объект john сохранён внутри объекта `Map`

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

let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok");

weakMap.set("test", "Whoops"); // Ошибка, потому что "test" не объект

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

let john = { name: "John" };

let weakMap = new WeakMap();

weakMap.set(john, "...");

john = null;

// объект john удалён из памяти

Теперь john существует только как ключ в WeakMap и может быть удалён оттуда автоматически. WeakMap не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё. В WeakMap присутствуют только следующие методы:

* weakMap.get(key)
* weakMap.set(key, value)
* weakMap.delete(key)
* weakMap.has(key)

Такие ограничения связаны с особенностью технической реализации. Если объект станет недостижим (как объект john в примере выше), то он будет автоматически удалён сборщиком мусора. Но нет информации, в какой момент произойдет эта очистка. Решение о том, когда делать сборку мусора, принимает движок JavaScript. Он может посчитать необходимым как удалить объект прямо сейчас, так и отложить эту операцию, чтобы удалить большее количество объектов за раз позже. Так что технически количество элементов в коллекции WeakMap неизвестно. Движок может произвести очистку сразу или потом, или сделать это частично. По этой причине методы для доступа ко всем сразу ключам/значениям недоступны.

В основном, WeakMap используется в качестве дополнительного хранилища данных или кеширования, когда результат вызова функции должен где-то запоминаться («кешироваться») для того, чтобы дальнейшие её вызовы на том же объекте могли просто брать уже готовый результат, повторно используя его. Для хранения результатов можно использовать Map:

// cache.js

let cache = new Map();

// вычисляем и запоминаем результат

function process(obj) {

if (!cache.has(obj)) {

let result = /\* какие-то вычисления \*/ obj;

cache.set(obj, result);

}

return cache.get(obj);

}

// main.js

let obj = {/\* какой-то объект \*/};

let result1 = process(obj);

let result2 = process(obj); // ранее вычисленный результат взят из кеша

obj = null;

alert(cache.size); // 1, объект всё ещё в кеше и занимает память

Многократные вызовы process(obj) с тем же самым объектом в качестве аргумента ведут к тому, что результат вычисляется только в первый раз, а затем последующие вызовы берут его из кеша. Недостатком является то, что необходимо вручную очищать cache от ставших ненужными объектов. Но если использовать WeakMap вместо Map, то эта проблема исчезнет: закешированные результаты будут автоматически удалены из памяти сборщиком мусора.

// cache.js

let cache = new WeakMap();

function process(obj) {

if (!cache.has(obj)) {

let result = /\* вычисляем результат для объекта \*/ obj;

cache.set(obj, result);

}

return cache.get(obj);

}

// main.js

let obj = {/\* какой-то объект \*/};

let result1 = process(obj);

let result2 = process(obj);

obj = null;

// Нет возможности получить cache.size, так как это WeakMap,

// но он равен 0 или скоро будет равен 0

// Когда сборщик мусора удаляет obj, связанные с ним данные из кеша тоже удаляются

[**WeakSet**](https://learn.javascript.ru/weakmap-weakset#weakset)

Коллекция WeakSet ведёт себя похоже:

* Она аналогична Set, но можно добавлять в WeakSet только объекты (не примитивные значения).
* Объект присутствует в множестве только до тех пор, пока доступен где-то ещё.
* Как и Set, она поддерживает add, has и delete, но не size, keys() и не является перебираемой.

WeakSet тоже служит в качестве дополнительного хранилища. Но не для произвольных данных, а скорее для значений типа «да/нет». Присутствие во множестве WeakSet может что-то сказать об объекте. Например, можно добавлять пользователей в WeakSet для учёта тех, кто посещал наш сайт:

let visitedSet = new WeakSet();

let john = { name: "John" };

let pete = { name: "Pete" };

let mary = { name: "Mary" };

visitedSet.add(john); // John

visitedSet.add(pete); // Pete

visitedSet.add(john); // John

// visitedSet сейчас содержит двух пользователей

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

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

john = null;

// структура данных visitedSet будет очищена автоматически

Наиболее значительным ограничением WeakMap и WeakSet является то, что их нельзя перебрать или взять всё содержимое. Это может доставлять неудобства, но не мешает WeakMap/WeakSet выполнять их главную задачу – быть дополнительным хранилищем данных для объектов, управляемых из каких-то других мест в коде.

1. **Деструктурирующее присваивание**

В JavaScript есть две чаще всего используемые структуры данных – это Object и Array. Объекты позволяют создавать одну сущность, которая хранит элементы данных по ключам, а массивы – хранить упорядоченные коллекции данных.

Но когда они передаются в функцию, то ей может понадобится не объект/массив целиком, а элементы по отдельности.

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

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

Пример:

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

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

let [firstName, surname] = arr;

alert(firstName); // Steve

alert(surname); // Jobs

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

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

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

Ненужные элементы массива также могут быть отброшены через запятую:

// второй элемент не нужен

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

alert( title ); // Consul

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

Можно использовать любой перебираемый объект, не только массивы:

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

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

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

let user = {};

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

alert(user.name); // Ilya

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

let user = {

name: "John",

age: 30

};

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

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

}

То же самое для map:

let user = new Map();

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

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

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

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

}

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

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

alert(name1); // Julius

alert(name2); // Caesar

alert(rest[0]); // Consul

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

alert(rest.length); // 2

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

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

let [firstName, surname] = [];

alert(firstName); // undefined

alert(surname); // undefined

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

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

alert(name); // Julius (из массива)

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

Значения по умолчанию могут быть гораздо более сложными выражениями или даже функциями. Они выполняются, только если значения отсутствуют. Например, в в примере ниже используется функция prompt для указания двух значений по умолчанию. Но она будет запущена только для отсутствующего значения:

let [name = prompt('name?'), surname = prompt('surname?')] = ["Julius"];

alert(name); // Julius (из массива)

alert(surname); // результат prompt

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

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

let {var1, var2} = {var1:…, var2:…}

У нас есть существующий объект с правой стороны, который необходимо разделить на переменные. Левая сторона содержит «шаблон» для соответствующих свойств. В простом случае это список названий переменных в {...}. Например:

let options = {

title: "Menu",

width: 100,

height: 200

};

let {title, width, height} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

Свойства options.title, options.width и options.height присваиваются соответствующим переменным. Порядок не имеет значения. Вот так тоже работает:

// изменён порядок в let {...}

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

Шаблон с левой стороны может быть более сложным и определять соответствие между свойствами и переменными.

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

let options = {

title: "Menu",

width: 100,

height: 200

};

// { sourceProperty: targetVariable }

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

// width -> w, height -> h, title -> title

alert(title); // Menu

alert(w); // 100

alert(h); // 200

Двоеточие показывает «что : куда идёт». В примере выше свойство width сохраняется в переменную *w*, свойство height сохраняется в *h*, а title присваивается одноимённой переменной.

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

let options = {

title: "Menu"

};

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

alert(title); // Menu

alert(width); // 100

alert(height); // 200

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

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

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

let options = {

title: "Menu",

height: 200,

width: 100

};

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

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

alert(rest.height); // 200

alert(rest.width); // 100

В примерах выше переменные были объявлены в присваивании: let {…} = {…}. Если использовать существующие переменные и не указывать let, то это не будет работать. Проблема в том, что JavaScript обрабатывает {...} в основном потоке кода (не внутри другого выражения) как блок кода. Такие блоки кода могут быть использованы для группировки операторов, например:

{

// блок кода

let message = "Hello";

// ...

alert( message );

}

Так что здесь JavaScript считает, что видит блок кода, отсюда и ошибка. На самом-то деле здесь деструктуризация. Чтобы показать JavaScript, что это не блок кода, можно заключить выражение в скобки (...):

let title, width, height;

({title, width, height} = {title: "Menu", width: 200, height: 100});

alert( title ); // Menu

[**Вложенная деструктуризация**](https://learn.javascript.ru/destructuring-assignment#vlozhennaya-destrukturizatsiya)

Если объект или массив содержит другие вложенные объекты или массивы, то можно использовать более сложные шаблоны с левой стороны, чтобы извлечь более глубокие свойства. В приведённом ниже коде options хранит другой объект в свойстве size и массив в свойстве items. Шаблон в левой части присваивания имеет такую же структуру, чтобы извлечь данные из них:

let options = {

size: {

width: 100,

height: 200

},

items: ["Cake", "Donut"],

extra: true

};

let {

size: {

width,

height

},

items: [item1, item2],

title = "Menu"

} = options;

alert(title); // Menu

alert(width); // 100

alert(height); // 200

alert(item1); // Cake

alert(item2); // Donut

Весь объект options, кроме свойства extra, присваивается в соответствующие переменные. В итоге есть width, height, item1, item2 и title со значением по умолчанию. Заметьте, что переменные для size и items отсутствуют, так как сразу использовали их содержимое.

[**Умные параметры функций**](https://learn.javascript.ru/destructuring-assignment#umnye-parametry-funktsiy)

Есть ситуации, когда функция имеет много параметров, большинство из которых не обязательны. Это особенно верно для пользовательских интерфейсов. Представьте себе функцию, которая создаёт меню. Она может иметь ширину, высоту, заголовок, список элементов и так далее. Вот так – плохой способ писать подобные функции:

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

// ...

}

В реальной жизни проблема заключается в том, как запомнить порядок всех аргументов. Обычно IDE помогают в этом, особенно если код хорошо документирован. Но есть и другая проблема, которая заключается в том, как вызвать функцию, когда большинство параметров передавать не надо, и значения по умолчанию вполне подходят. Разве что вот так:

// undefined там, где подходят значения по умолчанию

showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])

Это выглядит плохо. И становится нечитаемым, когда имеется большое количество параметров. Решить эту проблему помогает деструктуризация. Можно передать параметры как объект, и функция немедленно деструктурирует его в переменные:

let options = {

title: "My menu",

items: ["Item1", "Item2"]

};

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

alert( `${title} ${width} ${height}` ); // My Menu 200 100

alert( items ); // Item1, Item2

}

showMenu(options);

Также можно использовать более сложное деструктурирование с вложенными объектами и двоеточием:

let options = {

title: "My menu",

items: ["Item1", "Item2"]

};

function showMenu({

title = "Untitled",

width: w = 100,

height: h = 200,

items: [item1, item2]

}) {

alert( `${title} ${w} ${h}` ); // My Menu 100 200

alert( item1 ); // Item1

alert( item2 ); // Item2

}

showMenu(options);

Полный синтаксис – такой же, как для деструктурирующего присваивания:

function({

incomingProperty: varName = defaultValue

...

})

Тогда для объекта с параметрами, будет создана переменная varName для свойства с именем incomingProperty, по умолчанию равная defaultValue. Обратите внимание, что такое деструктурирование подразумевает, что в showMenu() будет обязательно передан аргумент. Если нужны все значения по умолчанию, то следует определить пустой объект:

showMenu({}); // все значения по умолчанию

showMenu(); // ошибка

Можно исправить это, сделав {} значением по умолчанию для всего объекта параметров:

function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {

alert( `${title} ${width} ${height}` );

}

showMenu(); // Menu 100 200

В приведённом коде выше весь объект аргументов по умолчанию равен {}, поэтому всегда есть что-то, что можно деструктурировать.

1. **Дата и Время.**

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

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

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

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

let now = new Date();

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

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

// 0 соответствует 01.01.1970 UTC+0

let Jan01\_1970 = new Date(0);

alert( Jan01\_1970 );

// добавим 24 часа и получим 02.01.1970 UTC+0

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

alert( Jan02\_1970 );

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

* new Date(datestring) – если аргумент всего один, и это строка, то из неё считывается дата. В примере ниже время не указано, поэтому оно ставится в полночь по Гринвичу и меняется в соответствии с временной зоной места выполнения кода, так что в результате можно получить Thu Jan 26 2017 11:00:00 GMT+1100 (восточноавстралийское время) или Wed Jan 25 2017 16:00:00 GMT-0800 (тихоокеанское время):

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

alert(date);

* new Date(year, month, date, hours, minutes, seconds, ms) – создать объект Date с заданными компонентами в местной временной зоне. Обязательны только первые два аргумента.
* year должен состоять из четырёх цифр: значение 2013 корректно, 98 – нет.
* month начинается с 0 (январь) по 11 (декабрь).
* Параметр date здесь представляет собой день месяца. Если параметр не задан, то принимается значение 1.
* Если параметры hours/minutes/seconds/ms отсутствуют, их значением становится 0.

Например:

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

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

Минимальная точность – 1 мс (1/1000 секунды):

let date = new Date(2011, 0, 1, 2, 3, 4, 567);

alert( date ); // 1.01.2011, 02:03:04.567

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

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

* [getFullYear()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getFullYear) – получить год (4 цифры)
* [getMonth()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMonth) – получить месяц, от 0 до 11.
* [getDate()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getDate) – получить день месяца, от 1 до 31, что несколько противоречит названию метода.
* [getHours()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getHours), [getMinutes()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMinutes), [getSeconds()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getSeconds), [getMilliseconds()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getMilliseconds) – получить, соответственно, часы, минуты, секунды или миллисекунды.

Многие интерпретаторы JavaScript реализуют нестандартный и устаревший метод getYear(), который порой возвращает год в виде двух цифр. Не используйте его. Если нужно значение года, используйте getFullYear().

* [getDay()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getDay) – вернуть день недели от 0 (воскресенье) до 6 (суббота). Несмотря на то, что в ряде стран за первый день недели принят понедельник, в JavaScript начало недели приходится на воскресенье.

Все вышеперечисленные методы возвращают значения в соответствии с местной временной зоной. Однако существуют и их UTC-варианты, возвращающие день, месяц, год для временной зоны UTC+0: [getUTCFullYear()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCFullYear), [getUTCMonth()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCMonth), [getUTCDay()](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/getUTCDay). Для их использования требуется после "get" подставить "UTC". Если ваша местная временная зона смещена относительно UTC, то следующий код покажет разные часы:

// текущая дата

let date = new Date();

// час в вашей текущей временной зоне

alert( date.getHours() );

// час во временной зоне UTC+0 (лондонское время без перехода на летнее время)

alert( date.getUTCHours() );

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

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

// если вы во временной зоне UTC-1, то выводится 60

// если вы во временной зоне UTC+3, выводится -180

alert( new Date().getTimezoneOffset() );

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

Следующие методы позволяют установить компоненты даты и времени:

* [setFullYear(year, [month], [date])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setFullYear)
* [setMonth(month, [date])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMonth)
* [setDate(date)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setDate)
* [setHours(hour, [min], [sec], [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setHours)
* [setMinutes(min, [sec], [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMinutes)
* [setSeconds(sec, [ms])](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setSeconds)
* [setMilliseconds(ms)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setMilliseconds)
* [setTime(milliseconds)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date/setTime) (устанавливает дату в виде целого количества миллисекунд, прошедших с 01.01.1970 UTC)

У всех этих методов, кроме setTime(), есть UTC-вариант, например: setUTCHours(). Некоторые методы могут устанавливать сразу несколько компонентов даты, например: setHours. Если какая-то компонента не указана, она не меняется. Пример:

let today = new Date();

today.setHours(0);

alert(today); // выводится сегодняшняя дата, значение часа будет 0

today.setHours(0, 0, 0, 0);

alert(today); // выводится сегодняшняя дата, время будет ровно 00:00:00.

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

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

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

alert(date); // 1st Feb 2013

Неправильные компоненты даты автоматически распределяются по остальным. Предположим, требуется увеличить дату «28 февраля 2016» на два дня. В зависимости от того, високосный это год или нет, результатом будет «2 марта» или «1 марта». Объект Date вычислит это сам, надо просто прибавить два дня:

let date = new Date(2016, 1, 28);

date.setDate(date.getDate() + 2);

alert( date ); // 1 Mar 2016

Эту возможность часто используют, чтобы получить дату по прошествии заданного отрезка времени. Например, получим дату «спустя 70 секунд с текущего момента»:

let date = new Date();

date.setSeconds(date.getSeconds() + 70);

alert( date ); // выводит правильную дату

Также можно установить нулевые или даже отрицательные значения. Например:

let date = new Date(2016, 0, 2); // 2 Jan 2016

date.setDate(1); // задать первое число месяца

alert( date );

// первый день месяца - это 1, так что выводится последнее число предыдущего месяца

date.setDate(0);

alert( date ); // 31 Dec 2015

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

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

let date = new Date();

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

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

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

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

let doSomething = i \* i \* i;

}

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

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

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

Если нужно просто померить время, для этого существует особый метод Date.now(), возвращающий текущий таймстамп. Семантически он эквивалентен new Date().getTime(), однако метод не создаёт промежуточный объект Date. Так что этот способ работает быстрее и не нагружает сборщик мусора. Данный метод используется из соображений удобства или когда важно быстродействие, например, при разработке игр на JavaScript или других специализированных приложений. Поэтому, предыдущий пример лучше переписать так:

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

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

let doSomething = i \* i \* i;

}

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

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

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

* YYYY-MM-DD – это дата: год-месяц-день.
* Символ "T" используется в качестве разделителя.
* HH:mm:ss.sss – время: часы, минуты, секунды и миллисекунды.
* Необязательная часть 'Z' обозначает временную зону в формате +-hh:mm. Если указать просто букву Z, то получим UTC+0.

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

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

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

alert(ms); // 1327611110417 (таймстамп)

Можно тут же создать объект new Date из таймстампа:

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

alert(date);

1. **Формат JSON.**

Допустим, есть сложный объект, и необходимо преобразовать его в строку, чтобы отправить по сети или просто вывести для логирования. Естественно, такая строка должна включать в себя все важные свойства. Можно реализовать преобразование следующим образом:

let user = {

name: "John",

age: 30,

toString() {

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

}

};

alert(user); // {name: "John", age: 30}

Но в процессе разработки добавляются новые свойства, старые свойства переименовываются и удаляются. Обновление такого toString каждый раз может стать проблемой. Можно попытаться перебрать свойства в нем, но если объект является сложным и имеет вложенные объекты в свойствах, тогда надо также осуществить их преобразование. Решением данной проеблемы является формат JSON.

**[JSON.stringify](https://learn.javascript.ru/json" \l "json-stringify)**

[JSON](https://ru.wikipedia.org/wiki/JSON) (JavaScript Object Notation) – это общий формат для представления значений и объектов. Его описание задокументировано в стандарте [RFC 4627](http://tools.ietf.org/html/rfc4627). 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/любой другой язык.

JavaScript предоставляет методы:

* JSON.stringify для преобразования объектов в JSON.
* JSON.parse для преобразования JSON обратно в объект.

Например, преобразуем через JSON.stringify данные студента:

let student = {

name: 'John',

age: 30,

isAdmin: false,

courses: ['html', 'css', 'js'],

wife: null

};

let json = JSON.stringify(student);

alert(typeof json); // получили строку

alert(json);

/\* выведет объект в формате JSON:

{

"name": "John",

"age": 30,

"isAdmin": false,

"courses": ["html", "css", "js"],

"wife": null

}

\*/

Метод JSON.stringify(student) берет объект и преобразует его в строку. Полученная строка json называется JSON-форматированным или сериализованным объектом. Можно отправить его по сети или поместить в обычное хранилище данных.

Объект в формате JSON имеет несколько важных отличий от объектного литерала:

* Строки используют двойные кавычки. Никаких одинарных кавычек или обратных кавычек в JSON. Так 'John' становится "John".
* Имена свойств объекта также заключаются в двойные кавычки. Это обязательно. Так age:30 становится "age":30.

JSON.stringify может быть применён и к примитивам.

JSON поддерживает следующие типы данных:

* объекты { ... },
* массивы [ ... ],
* примитивы: строки, числа, логические значения true/false, null.

Например:

alert( JSON.stringify(1) ) // 1

alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]

JSON является независимой от языка спецификацией для данных, поэтому JSON.stringify пропускает некоторые специфические свойства объектов JavaScript. А именно:

* Свойства-функции (методы).
* Символьные свойства.
* Свойства, содержащие undefined.

let user = {

sayHi() { // пропущено

alert("Hello");

},

[Symbol("id")]: 123, // пропущено

something: undefined // пропущено

};

alert( JSON.stringify(user) ); // {} (пустой объект)

Вложенные объекты поддерживаются и конвертируются автоматически. Например:

let meetup = {

title: "Conference",

room: {

number: 23,

participants: ["john", "ann"]

}

};

alert( JSON.stringify(meetup) );

/\* вся структура преобразована в строку:

{

"title":"Conference",

"room":{"number":23,"participants":["john","ann"]},

}

\*/

Важное ограничение: не должно быть циклических ссылок. Например:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: ["john", "ann"]

};

meetup.place = room;

room.occupiedBy = meetup;

JSON.stringify(meetup); // Ошибка: Преобразование цикличной структуры в JSON

Здесь преобразование завершается неудачно из-за циклической ссылки: room.occupiedBy ссылается на meetup, и meetup.place ссылается на room:

Полный синтаксис JSON.stringify:

let json = JSON.stringify(value[, replacer, space])

* value – значение для кодирования,
* replacer – массив свойств для кодирования или функция соответствия function(key, value),
* space – дополнительное пространство (отступы), используемое для форматирования.

В большинстве случаев JSON.stringify используется только с первым аргументом. Но если нужно настроить процесс замены, например, отфильтровать циклические ссылки, то можно использовать второй аргумент JSON.stringify. Если передать ему массив свойств, будут закодированы только эти свойства. Например:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, ['title', 'participants']) );

// {"title":"Conference","participants":[{},{}]}

Список свойств применяется ко всей структуре объекта. Так что свойство внутри participants объекты пустые, потому что name нет в списке. Включим в список все свойства, кроме room.occupiedBy, из-за которого появляется цикличная ссылка:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, ['title', 'participants', 'place', 'name', 'number']) );

/\*

{

"title":"Conference",

"participants":[{"name":"John"},{"name":"Alice"}],

"place":{"number":23}

}

\*/

Теперь все, кроме occupiedBy, сериализовано. Но список свойств довольно длинный. В качестве replacer можно использовать функцию, а не массив. Функция будет вызываться для каждой пары (key, value) и должна возвращать «заменённое» значение, которое будет использоваться вместо исходного. Или undefined, чтобы пропустить значение. В рассматриваемом примере можно вернуть value «как есть» для всего, кроме occupiedBy. Чтобы игнорировать occupiedBy, код ниже возвращает undefined:

let room = {

number: 23

};

let meetup = {

title: "Conference",

participants: [{name: "John"}, {name: "Alice"}],

place: room // meetup ссылается на room

};

room.occupiedBy = meetup; // room ссылается на meetup

alert( JSON.stringify(meetup, function replacer(key, value) {

alert(`${key}: ${value}`);

return (key == 'occupiedBy') ? undefined : value;

}));

/\* пары ключ:значение, которые приходят в replacer:

: [object Object]

title: Conference

participants: [object Object],[object Object]

0: [object Object]

name: John

1: [object Object]

name: Alice

place: [object Object]

number: 23

\*/

Функция replacer получает каждую пару ключ/значение, включая вложенные объекты и элементы массива. И она применяется рекурсивно. Значение this внутри replacer – это объект, который содержит текущее свойство. При первом вызове ему передаётся специальный «объект-обёртка»: {"": meetup}. Другими словами, первая (key, value) пара имеет пустой ключ, а значением является целевой объект. Вот почему первая строка из примера выше будет ":[object Object]".

Идея состоит в том, чтобы дать как можно больше возможностей replacer – у него есть возможность проанализировать и заменить/пропустить даже весь объект целиком, если это необходимо.

Третий аргумент в JSON.stringify(value, replacer, spaces) – это количество пробелов, используемых для удобного форматирования. Ранее все JSON-форматированные объекты не имели отступов и лишних пробелов. Аргумент spacer используется исключительно для вывода в удобочитаемом виде. Ниже spacer = 2 указывает JavaScript отображать вложенные объекты в несколько строк с отступом в 2 пробела внутри объекта:

let user = {

name: "John",

age: 25,

roles: {

isAdmin: false,

isEditor: true

}

};

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

/\* отступ в 2 пробела:

{

"name": "John",

"age": 25,

"roles": {

"isAdmin": false,

"isEditor": true

}

}

\*/

/\* для JSON.stringify(user, null, 4) результат содержал больше отступов:

{

"name": "John",

"age": 25,

"roles": {

"isAdmin": false,

"isEditor": true

}

}

\*/

Параметр spaces применяется для логирования и красивого вывода.

**[JSON.parse](https://learn.javascript.ru/json" \l "json-parse)**

Чтобы декодировать JSON-строку, нужен другой метод с именем [JSON.parse](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse). Синтаксис:

let value = JSON.parse(str, [reviver]);

* str – JSON для преобразования в объект,
* reviver – необязательная функция, которая будет вызываться для каждой пары (ключ, значение) и может преобразовывать значение.

Например:

let numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1

Или для вложенных объектов:

let user = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

alert( user.friends[1] ); // 1

JSON может быть настолько сложным, насколько это необходимо, объекты и массивы могут включать другие объекты и массивы. Но они должны быть в том же JSON-формате. Типичные ошибки в написанном от руки JSON:

let json = `{

name: "John", // Ошибка: имя свойства без кавычек

"surname": 'Smith', // Ошибка: одинарные кавычки в значении (должны быть двойными)

'isAdmin': false // Ошибка: одинарные кавычки в ключе (должны быть двойными)

"birthday": new Date(2000, 2, 3), // Ошибка: не допускается конструктор "new", только значения.

"friends": [0,1,2,3] // Здесь все в порядке

}`;

Кроме того, JSON не поддерживает комментарии. Добавление комментария в JSON делает его недействительным.

Существует ещё один формат [JSON5](http://json5.org/), который поддерживает ключи без кавычек, комментарии и т.д. Но это самостоятельная библиотека, а не спецификация языка.

1. **Intl: интернационализация в JavaScript.**

Общая проблема строк, дат, чисел в JavaScript заключается в том, что они не учитывают языуовые и временные особенностей стран, где находится посетитель. В частности:

* Строки: при сравнении сравниваются коды символов, а это неправильно, к примеру, в русском языке оказывается, что "ё" > "я" и "а" > "Я", хотя всем известно, что я – последняя буква алфавита и это она должна быть больше любой другой.
* Даты: в разных странах принята разная запись дат. Где-то пишут 31.12.2014 (Россия), а где-то 12/31/2014 (США), где-то иначе.
* Числа: в одних странах выводятся цифрами, в других – иероглифами, длинные числа разделяются где-то пробелом, где-то запятой.

Все современные браузеры, кроме IE10- (но есть библиотеки и для него) поддерживают стандарт [ECMA 402](http://www.ecma-international.org/ecma-402/1.0/ECMA-402.pdf), предназначенный решить эти проблемы навсегда.

[Основные объекты](https://learn.javascript.ru/intl" \l "osnovnye-obekty):

* Intl.Collator – умеет правильно сравнивать и сортировать строки.
* Intl.DateTimeFormat – умеет форматировать дату и время в соответствии с нужным языком.
* Intl.NumberFormat – умеет форматировать числа в соответствии с нужным языком.

[**Локаль**](https://learn.javascript.ru/intl#lokal)

Локаль – первый и самый важный аргумент всех методов, связанных с интернационализацией. Локаль описывается строкой из трёх компонентов, которые разделяются дефисом: код языка, код способа записи, код страны.

На практике не всегда указаны три, обычно меньше:

* ru – русский язык, без уточнений.
* en-GB – английский язык, используемый в Англии (GB).
* en-US – английский язык, используемый в США (US).
* zh-Hans-CN – китайский язык (zh), записываемый упрощённой иероглифической письменностью (Hans), используемый в Китае.

Также через суффикс -u-\* можно указать расширения локалей, например "th-TH-u-nu-thai" – тайский язык (th), используемый в Тайланде (TH), с записью чисел тайскими буквами (๐, ๑, ๒, ๓, ๔, ๕, ๖, ๗, ๘, ๙).

Стандарт, который описывает локали – [RFC 5464](http://tools.ietf.org/html/rfc5646), языки описаны в [IANA language registry](http://www.iana.org/assignments/language-subtag-registry/language-subtag-registry). Все методы принимают локаль в виде строки или массива, содержащего несколько локалей в порядке предпочтения. Если локаль не указана или undefined – берётся локаль по умолчанию, установленная в окружении (браузере).

[**Подбор локали localeMatcher**](https://learn.javascript.ru/intl#podbor-lokali-localematcher)

localeMatcher – вспомогательная настройка, которую тоже можно везде указать, она определяет способ подбора локали, если желаемая недоступна. У него два значения:

* "lookup" – означает простейший порядок поиска путём обрезания суффикса, например, zh-Hans-CN → zh-Hans → zh → локаль по умолчанию.
* "best fit" – использует встроенные алгоритмы и предпочтения браузера (или другого окружения) для выбора подходящей локали.

Если локалей несколько, например, ["zh-Hans-CN", "ru-RU"] то localeMatcher пытается подобрать наиболее подходящую локаль для первой из списка (китайская), если не получается – переходит ко второй (русской) и так далее. Если ни одной не нашёл, например, на компьютере не совсем поддерживается ни китайский ни русский, то используется локаль по умолчанию. Как правило, "best fit" является здесь наилучшим выбором.

[**Строки, Intl.Collator**](https://learn.javascript.ru/intl#intl-collator)

Синтаксис:

var collator = new Intl.Collator([locales, [options]])

Параметры:

* Locales – локаль, одна или массив в порядке предпочтения.
* Options – объект с дополнительными настройками:
* localeMatcher – алгоритм выбора подходящей локали.
* usage – цель сравнения: сортировка "sort" или поиск "search", по умолчанию "sort".
* sensitivity – чувствительность: какие различия в символах учитывать, а какие – нет, варианты:
* base – учитывать только разные символы, без диакритических знаков и регистра, например: а ≠ б, е = ё, а = А.
* accent – учитывать символы и диакритические знаки, например: а ≠ б, е ≠ ё, а = А.
* case – учитывать символы и регистр, например: а ≠ б, е = ё, а ≠ А.
* variant – учитывать всё: символ, диакритические знаки, регистр, например: а ≠ б, е ≠ ё, а ≠ А, используется по умолчанию.
* ignorePunctuation – игнорировать знаки пунктуации: true/false, по умолчанию false.
* numeric – использовать ли численное сравнение: true/false, если true, то будет 12 > 2, иначе 12 < 2.
* caseFirst – в сортировке должны идти первыми прописные или строчные буквы, варианты: "upper" (прописные), lower (строчные) или false (стандартное для локали, также является значением по умолчанию). Не поддерживается IE11-.

В подавляющем большинстве случаев подходят стандартные параметры, то есть options указывать не нужно. Использование:

var result = collator.compare(str1, str2);

Результат compare имеет значение 1 (больше), 0 (равно) или -1 (меньше). Например:

var collator = new Intl.Collator();

alert( "ёжик" > "яблоко" ); // true

alert( collator.compare("ёжик", "яблоко") ); // -1

Выше были использованы полностью стандартные настройки. Они различают регистр символа, но это различие можно убрать, если настроить чувствительность sensitivity:

var collator = new Intl.Collator();

alert( collator.compare("ЁжиК", "ёжик") ); // 1

var collator = new Intl.Collator(undefined, {

sensitivity: "accent"

});

alert( collator.compare("ЁжиК", "ёжик") ); // 0

[**Даты, Intl.DateTimeFormat**](https://learn.javascript.ru/intl#intl-datetimeformat)

Синтаксис:

var formatter = new Intl.DateTimeFormat([locales, [options]])

Первый аргумент – такой же, как и в Collator, а в объекте options можно определить, какие именно части даты показывать (часы, месяц, год…) и в каком формате. Полный список свойств options:

| **Свойство** | **Описание** | **Возможные значения** | **По умолчанию** |
| --- | --- | --- | --- |
| localeMatcher | Алгоритм подбора локали | lookup,best fit | best fit |
| formatMatcher | Алгоритм подбора формата | basic, best fit | best fit |
| hour12 | Включать ли время в 12-часовом формате | true -- 12-часовой формат, false -- 24-часовой |  |
| timeZone | Временная зона | Временная зона, например Europe/Moscow | UTC |
| weekday | День недели | narrow, short, long |  |
| era | Эра | narrow, short, long |  |
| year | Год | 2-digit, numeric | undefined или numeric |
| month | Месяц | 2-digit, numeric, narrow, short, long | undefined или numeric |
| day | День | 2-digit, numeric | undefined или numeric |
| hour | Час | 2-digit, numeric |  |
| minute | Минуты | 2-digit, numeric |  |
| second | Секунды | 2-digit, numeric |  |
| timeZoneName | Название таймзоны (нет в IE11) | short, long |  |

Все локали обязаны поддерживать следующие наборы настроек:

* weekday, year, month, day, hour, minute, second
* weekday, year, month, day
* year, month, day
* year, month
* month, day
* hour, minute, second

Если указанный формат не поддерживается, то настройка formatMatcher задаёт алгоритм подбора наиболее близкого формата: basic – по [стандартным правилам](http://www.ecma-international.org/ecma-402/1.0/#BasicFormatMatcher) и best fit – по умолчанию, на усмотрение окружения (браузера). Использование:

var dateString = formatter.format(date);

Например:

var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru");

alert( formatter.format(date) ); // 31.12.2014

var formatter = new Intl.DateTimeFormat("en-US");

alert( formatter.format(date) ); // 12/31/2014

Длинная дата, с настройками:

var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru", {

weekday: "long",

year: "numeric",

month: "long",

day: "numeric"

});

alert( formatter.format(date) ); // среда, 31 декабря 2014 г.

Только время:

var date = new Date(2014, 11, 31, 12, 30, 0);

var formatter = new Intl.DateTimeFormat("ru", {

hour: "numeric",

minute: "numeric",

second: "numeric"

});

alert( formatter.format(date) ); // 12:30:00

[**Числа: Intl.NumberFormat**](https://learn.javascript.ru/intl#chisla-intl-numberformat)

Форматтер Intl.NumberFormat умеет красиво форматировать не только числа, но и валюту, а также проценты. Синтаксис:

var formatter = new Intl.NumberFormat([locales[, options]]);

formatter.format(number); // форматирование

Параметры, как и раньше – локаль и опции. Список опций:

|  |  |  |  |
| --- | --- | --- | --- |
| **Свойство** | **Описание** | **Возможные значения** | **По умолчанию** |
| localeMatcher | Алгоритм подбора локали | lookup, best fit | best fit |
| style | Стиль форматирования | decimal, percent, currency | decimal |
| currency | Алфавитный код валюты | См. [Список кодов валюты](http://www.currency-iso.org/en/home/tables/table-a1.html), например USD |  |
| currencyDisplay | Показывать валюту в виде кода, локализованного символа или локализованного названия | code, symbol, name | symbol |
| useGrouping | Разделять ли цифры на группы | true, false | true |
| minimumIntegerDigits | Минимальное количество цифр целой части | от `1` до `21` | 21 |
| minimumFractionDigits | Минимальное количество десятичных цифр | от 0 до 20 | для чисел и процентов 0, для валюты зависит от кода. |
| maximumFractionDigits | Максимальное количество десятичных цифр | от minimumFractionDigits до 20. | для чисел max(minimumFractionDigits, 3), для процентов 0, для валюты зависит от кода. |
| minimumSignificantDigits | Минимальное количество значимых цифр | от 1 до 21 | 1 |
| maximumSignificantDigits | Максимальное количество значимых цифр | от minimumSignificantDigitsдо 21 | minimumSignificantDigits |

Пример без опций:

var formatter = new Intl.NumberFormat("ru");

alert( formatter.format(1234567890.123) ); // 1 234 567 890,123

С ограничением значимых цифр (важны только первые 3):

var formatter = new Intl.NumberFormat("ru", {

maximumSignificantDigits: 3

});

alert( formatter.format(1234567890.123) ); // 1 230 000 000

С опциями для валюты:

var formatter = new Intl.NumberFormat("ru", {

style: "currency",

currency: "GBP"

});

alert( formatter.format(1234.5) ); // 1 234,5 £

С двумя цифрами после запятой:

var formatter = new Intl.NumberFormat("ru", {

style: "currency",

currency: "GBP",

minimumFractionDigits: 2

});

alert( formatter.format(1234.5) ); // 1 234,50 £

[**Методы в Date, String, Number**](https://learn.javascript.ru/intl#metody-v-date-string-number)

Методы форматирования также поддерживаются в обычных строках, датах, числах:

String.prototype.localeCompare(that [, locales [, options]]) – сравнивает строку с другой, с учётом локали, например:

var str = "ёжик";

alert( str.localeCompare("яблоко", "ru") ); // -1

Date.prototype.toLocaleString([locales [, options]]) – форматирует дату в соответствии с локалью, например:

var date = new Date(2014, 11, 31, 12, 00);

alert( date.toLocaleString("ru", { year: 'numeric', month: 'long' }) ); // Декабрь 2014

Date.prototype.toLocaleDateString([locales [, options]]) – то же, что и выше, но опции по умолчанию включают в себя год, месяц, день.

Date.prototype.toLocaleTimeString([locales [, options]]) – то же, что и выше, но опции по умолчанию включают в себя часы, минуты, секунды.

Number.prototype.toLocaleString([locales [, options]]) – форматирует число, используя опции Intl.NumberFormat.

Все эти методы при запуске создают соответствующий объект Intl.\* и передают ему опции, можно рассматривать их как укороченные варианты вызова.

1. **Регулярные выражения.**

Регулярные выражения – это шаблоны используемые для сопоставления последовательностей символов в строках. В JavaScript, регулярные выражения также являются объектами. Эти шаблоны используются в методах [exec](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp/exec) и [test](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp/test) объекта[RegExp](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp), а также [match](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/match), [replace](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/replace),[search](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/search), и [split](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/split) объекта [String](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String).

**Создание регулярного выражения**

Регулярное выражение можно создать двумя способами:

1. Используя литерал регулярного выражения, например:

let 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.

1. Вызывая функцию конструктор объекта [RegExp](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp), например:

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

Использование конструктора влечет за собой компиляцию регулярного выражения во время исполнения скрипта. Данный способ можно использовать, если известно, что выражение будет изменяться или не известен шаблон заранее. Например, при получении 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".

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'.

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'.

В следующей таблице приводится полный список специальных символов регулярных выражений с их описаниями.

|  | |
| --- | --- |
| **Символ** | **Значение** |
| [\](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-backslash) | Одно из следующего:   * Для символов обычно обрабатываемых буквально, означает что следующий символ является специальным и не должен интерпретироваться буквально. * Например, /b/ сопоставляется символу 'b'. Добавляя слэш перед b, т.е используя /\b/, символ становится специальным символом, означающим границу слова. * Для символов обычно обрабатываемых особым образом означает, что следующий символ не является специальным и должен интерпретироваться буквально. * Например, \* является специальным символом, сопоставляемым 0 или более повторений предыдущего символа; например, /a\*/ означает соответствие 0 или более символов а. Для буквальной интерпретации \*, поставьте перед ней обратный слэш; например, /a\\*/ соответствует 'a\*'. * Также не забудьте заэкранировать сам \ при его использовании в записи new RegExp("pattern") поскольку \ также является экранирующим символом в обычных строках. |
| [^](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-caret) | Соответствует началу ввода. Если установлен флаг многострочности, также производит сопоставление непосредственно после переноса строки.  Например, /^A/ не соответствует 'A' в "an A", но соответствует 'A' в "An E".  Этот символ имеет другое значение при появлении в начале шаблона набора символов. Например, /[^a-z\s]/ соответствует 'I' в "I have 3 sisters". |
| [$](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-dollar) | Соответствует концу ввода. Если установлен битовый флаг многострочности, также сопоставляется содержимому до переноса строки.  Например, /t$/ не соответствует 't' в строке "eater", но соответствует строке "eat". |
| [\*](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-asterisk) | Соответствует предыдущему символу повторенному 0 или более раз. Эквивалентно {0,}.  Например, /bo\*/ соответствует 'boooo' в "A ghost booooed" и 'b' в "A bird warbled", но не в "A goat grunted". |
| [+](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-plus) | Соответствует предыдущему символу повторенному 1 или более раз. Эквивалентно {1,}. Например, /a+/ соответствует 'a' в "candy" и всем символам 'a' в "caaaaaaandy". |
| [?](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-questionmark) | Соответствует предыдущему символу повторенному  0 или 1 раз. Эквивалентно {0,1}.  Например, /e?le?/ соответствует 'el' в "angel" и 'le' в "angle" а также 'l' в "oslo".  Если использован сразу после квалификаторов \*, +, ?, or {}, делает квалификатор "нежадным" (соответствующим минимальному количеству символов), в отличие от режима по умолчанию, являющимся "жадным" (соответствующим максимальному числу символов). Например, используя /\d+/ не глобальное сопоставление "123abc" возвращает "123", если использовать /\d+?/, только "1" будет возвращена.  Также используется в упреждающих утверждениях (assertions), описанных в строках x(?=y) и x(?!y) данной таблицы. |
| [.](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-dot) | (десятичная точка) соответствует любому символу кроме переноса строки.  Например, /.n/ соответствует 'an' и 'on' в "nay, an apple is on the tree", но не 'nay'. |
| [(x)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-capturing-parentheses) | Соответствует 'x' и запоминает это соответствие. Это называется захватывающие скобки.  Например, /(foo)/ соответствует 'foo' в "foo bar." Сопоставленная строка может быть получена из элементов результирующего массива [1], ..., [n]. |
| [(?:x)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-capturing-parentheses) | Соответствует 'x' но не запоминает соответствие. Это называется не-захватывающие скобки. Сопоставленная строка не может быть получена из элементов результирующего массива [1], ..., [n]. |
| [x(?=y)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-lookahead) | Соответствует 'x' только если за 'x' следует 'y'. Это называется упреждение.  Например, /Jack(?=Sprat)/ соответствует 'Jack' только если за ним следует 'Sprat'. /Jack(?=Sprat|Frost)/ соответствует 'Jack' только если за ним следует 'Sprat' или 'Frost'. Тем не менее, ни 'Sprat' ни 'Frost' не являются частью сопоставленного результата. |
| [x(?!y)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-negated-look-ahead) | Соответствует 'x' только если за 'x' не следует 'y'. Это называется отрицательное упреждение. Например, /\d+(?!\.)/ соответствует числу только если за ним не следует десятичная точка. Регулярное выражение /\d+(?!\.)/.exec("3.141") сопоставит '141' но не '3.141'. |
| [x|y](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-or) | Соответствует либо 'x' либо 'y'.  Например, /green|red/ соответствует 'green' в "green apple" и 'red' в "red apple." |
| [{n}](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-quantifier) | n - положительное целое. Соответствует ровно n вхождениям предыдущего символа. Например, /a{2}/ не соответствует 'a' в "candy," но соответствует всем а в "caandy," первым двум а в "caaandy." |
| [{n,m}](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-quantifier-range) | m и n - положительные целые. Соответствует как минимум n и максимум m вхождениям предыдущего символа. При m=n=0 пропускается.  Например, /a{1,3}/ ничему не соответствует в строке "cndy", символу 'a' в "candy," двум а в "caandy," и трем первым а в "caaaaaaandy". Отметим, что при сопоставлении "caaaaaaandy", совпадает "aaa", хотя изначальная строка содержит больше а. |
| [[xyz]](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-character-set) | Набор символов. Соответствует любому символу из перечисленных. Можно указать диапазон символов используя тире. Специальные символы (как точка (.) и звездочка (\*)) не имеют специального значения внутри такого набора.Их не надо экранировать. Экранирование работает так же.  Например, [abcd] эквивалентна [a-d]. Они соответствуют 'b' в "brisket" и 'c' в "city". /[a-z.]+/ и /[\w.]+/ обе соответствуют всему в "test.i.ng". |
| [[^xyz]](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-negated-character-set) | Инвертированный или дополняющий набор символов. Это означает соответствие всему, что не в скобках. Можно указать диапазон символов с помощью тире. Все, что действует в обычном наборе символов, действует и здесь. Например, [^abc] эквивалентно [^a-c]. Они соответствуют изначально 'r' в "brisket" и 'h' в "chop." |
| [[\b]](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-backspace) | Соответствует бэкспейсу (U+0008). (Не путать с \b.) |
| [\b](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-word-boundary) | Соответствует границе слова. Граница слова соответствует позиции где за символом слова не следует другой символ слова или предшествует ему. Отметим, что граница слова не включается в соответствие. Другими словами, длина сопоставленной границы слова равна нулю. (Не путать с [\b].)  Примеры: /\bmoo/ соответствует 'moo' в слове "moon" ; /oo\b/ не соответствует 'oo' в слове "moon", поскольку за 'oo' следует символ 'n' , являющимся символом слова; /oon\b/ соответствует 'oon' в слове "moon", поскольку 'oon' является окончанием строки, и таким образом, за этими символами не следует другой символ слова; /\w\b\w/ никогда не будет ничему соответствовать, поскольку за символом слова никогда не может следовать и граница слова, и символ слова. |
| [\B](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-word-boundary) | Соответствует несловообразующей границе. Несловообразующая граница соответствует позиции, в которой предыдущий и следующий символы являются символами одного типа: либо оба должны быть словообразующими символами, либо несловообразующими. Начало и конец строки считаются несловообразующими символами.  Например, /\B../ соответствует 'oo' в слове "noonday" (, а /y\B./ соответствует 'ye' в "possibly yesterday." |
| [\c*X*](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-control) | Где *X* является символом случайного выбора из последовательности от А до Я. Соответствует управляющему символу в строке.  Например, /\cM/ соответствует control-M (U+000D) в строке. |
| [\d](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-digit) | Соответствует цифровому символу. Эквивалентно выражению [0-9].  Например, /\d/ or /[0-9]/ соотвествует  '2' в "B2 is the suite number." |
| [\D](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-digit) | Соответствует любому нецифровому символу. Эквивалентно выражению [^0-9].  Например, /\D/ or /[^0-9]/ соответствует 'B' в предложении "B2 is the suite number." |
| [\f](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-form-feed) | Соответствует символу прогона страницы (U+000C). Особый символ управления печатью. |
| [\n](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-line-feed) | Соответствует символу перевода строки (U+000A). |
| [\r](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-carriage-return) | Соответствует символу возврата каретки (U+000D). |
| [\s](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-white-space) | Соответствует одиночному символу пустого пространства, включая пробел, табуляция, прогон страницы, перевод строки. Эквивалентен [ \f\n\r\t\v​\u00A0\u1680​\u180e\u2000​\u2001\u2002​\u2003\u2004​\u2005\u2006​\u2007\u2008​\u2009\u200a​\u2028\u2029​\u2028\u2029​\u202f\u205f​\u3000].  Например, /\s\w\*/ совпадает с ' bar' в "foo bar." |
| [\S](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-white-space) | Соответствует одиночному символу непустого пространства. Эквивалентен [^ \f\n\r\t\v​\u00A0\u1680​\u180e\u2000-\u200a​​\u2028\u2029​\u202f\u205f​\u3000].  Например, /\S\w\*/ совпадает с 'foo' в "foo bar." |
| [\t](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-tab) | Соответствует символу горизонтальной табуляции (U+0009). |
| [\v](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-vertical-tab) | Соответствует символу вертикальной табуляции (U+000B). |
| [\w](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-word) | Соответствует любому цифробуквенному символу включая нижнее подчеркивание. Эквивалентен [A-Za-z0-9\_].  Например, /\w/ совпадает с 'a' в "apple," '5' в "$5.28," и '3' в "3D." |
| [\W](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-non-word) | Соответствует любому не цифробуквенному символу. Равносилен [^A-Za-z0-9\_].  Например, /\W/ or /[^A-Za-z0-9\_]/ совпадает с '%' в "50%." |
| [\*n*](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-backreference) | Где *n* это положительное целое, обратная ссылка на последнюю найденную подстроку, соответствующую *n* , заключенную в круглые скобки в регуляном выражении (считая левые скобки).  Например, /apple(,)\sorange\1/ соответствует 'apple, orange,' в "apple, orange, cherry, peach." |
| [\0](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-null) | Соответствует символу NULL (U+0000). Не следует ставить за ним другой цифровой символ, поскольку \0<digits> является восьмеричной экранирующей последовательностью. |
| [\xhh](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-hex-escape) | Соответствует символам кода hh (две шестнадцатиричные цифры) |
| [\uhhhh](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#special-unicode-escape) | Соответствует символам кода hhhh (четыре шестнадцатиричные цифры). |

Экранирование пользовательского ввода, соответствующего буквенной строке внутри регулярного выражения, может быть достигнуто простой заменой:

function escapeRegExp(string){

return string.replace(/[.\*+?^${}()|[\]\\]/g, '\\$&');

}

**Использование скобок**

Скобки вокруг любой части регулярного выражения означают что эта часть совпадаемой подстроки будет запомнена. Например, паттерн /Chapter (\d+)\.\d\*/ включает в себя дополнительные экранирующие и специальные символы и указывает на то, что часть шаблона должна быть запомнена. Он точно соответствует символам слова 'Chapter ', за которыми следует один или более цифровых символов (\d означает любой цифровой символ, а '+' означает 1 или более раз), за которым следует десятичная точка (сама по себе являющаяся специальным символом; предшествующий десятичной точке слэш ' \' означает, что паттерн должен искать литеральный символ '.'), после которой следует любой цифровой символ 0 или более раз ('\d' обозначает цифровой символ, '\*' обозначает 0 или более раз). Кроме того, круглые скобки используются для запоминания первых же совпавших цифровых символов.

Этот шаблон будет найден во фразе "Open Chapter 4.3, paragraph 6" и цифра '4' будет запомнена. Но он не будет найден во фразе "Chapter 3 and 4", поскольку эта строка не имеет точки после цифры '3'.

Для того, чтобы сопоставить подстроку без вызова совпавшей части для запоминания, внутри круглых скобок необходимо предварить паттерн сочетанием символов ' ?:'. Например, шаблон (?:\d+) будет соответствовать одному или более цифровому символу, но не запомнит совпавших символов.

**Работа с регулярными выражениями**

Регулярные выражения используются в методах test и exec объекта RegExp и с методами match, replace, search, и split объекта String. Эти методы подробно объясняются в [Справочнике JavaScript](https://developer.mozilla.org/en-US/docs/JavaScript/Reference).

| Методы пользующиеся регулярными выражениями: | |
| --- | --- |
| **Метод** | **Описание** |
| [exec](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp/exec) | Метод RegExp, который выполняет поиск совпадения в  строке. Он возвращает массив данных. |
| [test](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/RegExp/test) | Метод RegExp, который тестирует совпадение в строке. Возвращет либо истину либо ложь. |
| [match](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/match) | Метод String, который выполняет поиск совпадения в строке. Он возвращет массив данных либо null если совпадения отсутствуют. |
| [match](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/match)All | Метод String, который выполняет поиск совпадений в строке и возвращает итератор, который в свою очередь возвращает все совпадающие группы одну за другой. |
| [search](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/search) | Метод String, который тестирует на совпадение в строке. Он возвращет индекс совпадения, или -1 если совпадений не будет найдено. |
| [replace](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/replace) | Метод String, который выполняет поиск совпадения в строке, и заменяет совпавшую подстроку другой подстрокой переданной как аргумент в этот метод. |
| [split](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/split) | Метод String, который использует регулярное выражение или фиксированую строку чтобы разбить строку на массив подстрок. |

Чтобы просто узнать есть ли в строке что либо соотвествующее шаблону, воспользуйтесь методами test или search; а чтобы получить больше информации пользуйтесь методами exec или match (хотя эти метода работают медленее), которые вернут массив и обновленные свойства объекта ассоцированного регулярного выражения, а также предопределенного объекта RegExp регулярного выражения. Если совпадений нет, метод exec вернет null (который сконвертируется в  false).

В следующем примере, скрипт использует метод exec чтобы найти совпадения в строке:

let myRe = /d(b+)d/g;

let myArray = myRe.exec("cdbbdbsbz");

Если не нужен доступ к свойствам регулярного выражения, то альтернативный способ получить myArray можно так:

let myArray = /d(b+)d/g.exec("cdbbdbsbz");

Если надо сконструировать регулярное выражение из строки, другой способ сделать это приведен ниже:

let myRe = new RegExp("d(b+)d", "g");

let myArray = myRe.exec("cdbbdbsbz");

С помощью этих скриптов, поиск совпадения завершается и возвращает массив и обновленные ствойства, показанные в следующей таблице:

|  | | | |
| --- | --- | --- | --- |
| **Объект** | **Свойство или индекс** | **Описание** | **В этом примере.** |
| myArray |  | Совпавшая строка и все запомненные подстроки. | ["dbbd", "bb"] |
| index | Индекс совпавшей подстроки (индекс начинается с нуля). | 1 |
| input | Исходная строка. | "cdbbdbsbz" |
| [0] | Последние совпавшие символы. | "dbbd" |
| myRe | lastIndex | Индекс с которого начнется след. поиск совпадения. (Это свойство определяется только если регулярное выражение использует параметр g, описанный в [Advanced Searching With Flags](https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Regular_Expressions#Advanced_Searching_With_Flags).) | 5 |
| source | Текст шаблона. Обновляется в момент создания регулярного выражения, а не во время выполнения. | "d(b+)d" |

Как показано во втором варианте этого примера, можно использовать регулярное выражение, созданное при помощи инициализатора объекта, без присваивания его переменной. Таким образом, если используется данная форма записи без присваивания переменной, то в процессе дальнейшего использования нельзя получить доступ к свойствам данного регулярного выражения. Например:

var myRe = /d(b+)d/g;

var myArray = myRe.exec("cdbbdbsbz");

console.log("The value of lastIndex is " + myRe.lastIndex);

Этот скрипт выведет:

The value of lastIndex is 5

Однако, следующий скрипт:

var myArray = /d(b+)d/g.exec("cdbbdbsbz");

console.log("The value of lastIndex is " + /d(b+)d/g.lastIndex);

выведет

The value of lastIndex is 0

Совпадения /d(b+)d/g в двух случаях являются разными объектами регулярного выражения и, следовательно, имеют различные значения для свойства lastIndex. Если необходим доступ к свойствам объекта, созданного при помощи инициализатора, то надо сначала присвоить его переменной.

**Использование скобочных выражений для нахождения подстрок.**

Использование скобок в шаблоне регулярного выражения повлечет "запоминание" совпавшей подстроки. Для примера, /a(b)c/ вызовет совпадение 'abc' и запомнит 'b'. Чтобы получить совпадения скобочного выражения используйте Array elements [1], ..., [n].

Число возможных скобочных подстрок неограничено. Возвращаемый массив содержит все полученные совпадения, удовлетворяющие выражению в скобках. Следующий пример показывает, как использовать скобочные выражения для нахождения подстрок.

Следующий скрипт использует метод [replace()](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/String/replace), чтобы поменять местами слова (символы) в строке. Для замены текста скрипт использует обозначения $1 и $2 для обозначения первого и второго совпадения скобочного выражения.

let re = /(\w+)\s(\w+)/;

let str = "John Smith";

let newstr = str.replace(re, "$2, $1");

console.log(newstr);

Выведет "Smith, John".

**Расширенный поиск с флагами**

Регулярные выражения имеют четыре опциональных флага, которые делают возможным глобальный и регистронезависимый поиск. Флаги могут использоваться самостоятельно или вместе в любом порядке, а также могут являться частью регулярного выражения.

| **Флаги регулярных выражений** | |
| --- | --- |
| **Flag** | **Description** |
| g | Глобальный поиск. |
| i | Регистронезависимый поиск. |
| m | Многострочный поиск. |
| y | Выполняет поиск начиная с символа, который находится на позиции свойства lastindex текущего регулярного выражения. |

Чтобы использовать флаги в шаблоне регулярного выражения используйте следующий синтаксис:

var re = /pattern/flags;

или

var re = new RegExp("pattern", "flags");

Обратите внимание, что флаги являются неотъемлемой частью регулярного выражения. Флаги не могут быть добавлены или удалены позднее. Для примера, re = /\w+\s/g создаёт регулярное выражение, которое ищет один или более символов, после котрых следует пробел и ищет данное совпадение на протяжении всей строки.

var re = /\w+\s/g;

var str = "fee fi fo fum";

var myArray = str.match(re);

console.log(myArray);

Выведет ["fee ", "fi ", "fo "]. В этом примере можно заменить строку:

var re = /\w+\s/g;

на следующую:

var re = new RegExp("\\w+\\s", "g");

и получить тот же результат.

Флаг m используется, чтобы входная строка рассматривалась как многострочная. Если флаг m используется, то ^ и $ вызовет совпадение в начале или конце любой строки в строке ввода вместо начала или конца вводимой строки целиком.

Следующий пример иллюстрирует формирование регулярного выражения и использование string.split() и string.replace(). Он очищает неправильно сформатированную исходную строку, которая содержит имена в неправильном порядке (имя идет первым) разделенные пробелами, табуляцией и одной точкой с запятой. В конце, изменяется порядок следования имен (фамилия станет первой) и сортируется список.

var names = "Harry Trump ;Fred Barney; Helen Rigby ; Bill Abel ; Chris Hand ";

var output = ["---------- Original String\n", names + "\n"];

var pattern = /\s\*;\s\*/;

var nameList = names.split(pattern);

pattern = /(\w+)\s+(\w+)/;

var bySurnameList = [];

output.push("---------- After Split by Regular Expression");

var i, len;

for (i = 0, len = nameList.length; i < len; i++){

output.push(nameList[i]);

bySurnameList[i] = nameList[i].replace(pattern, "$2, $1");

}

output.push("---------- Names Reversed");

for (i = 0, len = bySurnameList.length; i < len; i++){

output.push(bySurnameList[i]);

}

bySurnameList.sort();

output.push("---------- Sorted");

for (i = 0, len = bySurnameList.length; i < len; i++){

output.push(bySurnameList[i]);

}

output.push("---------- End");

console.log(output.join("\n"));

В следующем примере, ожидается что пользователь введет телефонный номер и требуется проверить правильность символов, набранных пользователем. Когда пользователь нажмет кнопку "Check", скрипт проверит правильность введеного номера. Если номер правильный (совпадает с символами, определенными в регулярном выражении), то скрипт покажет сообщение благодарности для пользователя и подтвердит номер. Если нет, то скрипт проинформирует пользователя, что телефонный номер неправильный.

Внутри незахватывающих скобок (?:, регуляное выражение ищет три цифры \d{3} ИЛИ |  открывающую скобку \(, затем три цифры \d{3}, затем закрывающую скобку \), (закрывающая незахватывающая скобка )), затем тире, слэш, или десятичная точка, и когда это выражение найдено, запоминает символ ([-\/\.]),  следующие за ним и запомненные три цифры \d{3}, следующее соответствие тире, слэша или десятичной точки \1, и следующие четыре цифры \d{4}.

Регулярное выражение ищет сначала 0 или одну открывающую скобку \(?, затем три цифры \d{3}, затем 0 или одну закрывающую скобку \)?, потом одно тире, слеш или точка и когда найдет это, запомнит символ([-\/\.]), след. три цифры \d{3}, followed by the remembered match of a dash, forward slash, or decimal point \1, followed by four digits \d{4}.

Событие "Изменить" активируется, когда пользователь подтвердит ввод значения регулярного выражения, нажав клавишу "Enter".

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

<meta http-equiv="Content-Script-Type" content="text/javascript">

<script type="text/javascript">

var re = /\(?\d{3}\)?([-\/\.])\d{3}\1\d{4}/;

function testInfo(phoneInput){

var OK = re.exec(phoneInput.value);

if (!OK)

window.alert(RegExp.input + " isn't a phone number with area code!");

else

window.alert("Thanks, your phone number is " + OK[0]);

}

</script>

</head>

<body>

<p>Enter your phone number (with area code) and then click "Check".

<br>The expected format is like ###-###-####.</p>

<form action="#">

<input id="phone"><button onclick="testInfo(document.getElementById('phone'));">Check</button>

</form>

</body>

</html>

autoPreviousNext("JSGChapters");

Тема 3. Функции и замыкания

1. Глобальный объект.
2. Замыкания, внутренняя работа функции.
3. Конструкция new Function.
4. Объект функции.
5. Остаточные параметры и оператор расширения.
6. Каррирование.
7. Генераторы.
8. Колбэки.
9. Промис (Promise).
10. Промисификация.
11. Async/await.
12. Управление памятью в JavaScript.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. **Глобальный объект**

Глобальный объект предоставляет переменные и функции, доступные в любом месте программы. По умолчанию это те, что встроены в язык или среду исполнения. В браузере он называется window, в Node.js – global, в другой среде исполнения может называться иначе. Недавно globalThis был добавлен в язык как стандартизированное имя для глобального объекта, которое должно поддерживаться в любом окружении. В некоторых браузерах (например, Edge) globalThis ещё не поддерживается, но легко реализуется с помощью полифила.

Прежде, решение было таким:

const getGlobal = function () {

if (typeof self !== undefined) {

return self;

}

if (typeof window !== undefined) {

return window;

}

if (typeof global !== undefined) {

return global;

}

throw new Error("unable to locate global object");

};

const globals = getGlobal();

// Сейчас есть `globalThis`

globalThis === window; // true

Рассмотрим подробнее глобальный объект window, так как наша среда – браузер. Ко всем свойствам глобального объекта можно обращаться напрямую:

alert("Привет");

// это то же самое, что и

window.alert("Привет");

В браузере глобальные функции и переменные, объявленные с помощью var (не let/const), становятся свойствами глобального объекта:

var gVar = 5;

alert(window.gVar); // 5 (становится свойством глобального объекта)

Такое поведение поддерживается для совместимости. В современных проектах, использующих [JavaScript-модули](https://learn.javascript.ru/modules), такого не происходит.

Если объявить переменную при помощи let, то такого не произойдет:

let gLet = 5;

alert(window.gLet); // undefined (не становится свойством глобального объекта)

Если свойство настолько важное, что надо сделать его доступным для всей программы, то запишите его в глобальный объект напрямую:

window.currentUser = {

name: "John"

};

alert(currentUser.name); // John

alert(window.currentUser.name); // John

При этом обычно не рекомендуется использовать глобальные переменные. Следует применять их как можно реже. Дизайн кода, при котором функция получает входные параметры и выдаёт определённый результат, чище, надёжнее и удобнее для тестирования, чем когда используются внешние, а тем более глобальные переменные.

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

if (!window.Promise) {

alert("Ваш браузер очень старый!");

}

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

if (!window.Promise) {

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

}

1. **Замыкания, внутренняя работа функции.**

JavaScript – язык с сильным функционально-ориентированным уклоном. Функция может быть динамически создана, скопирована в другую переменную или передана как аргумент другой функции и позже вызвана из совершенно другого места. Функция может получить доступ к переменным из внешнего окружения, эта возможность используется очень часто.

Рассмотрим две ситуации:

1. Функция sayHi использует внешнюю переменную name. Какое значение будет использовать функция при выполнении?

let name = "John";

function sayHi() {

alert("Hi, " + name);

}

name = "Pete";

sayHi(); // что будет показано: "John" или "Pete"?

Такие ситуации распространены и в браузерной и в серверной разработке. Выполнение функции может быть запланировано позже, чем она была создана, например, после какого-нибудь пользовательского действия или сетевого запроса.

1. Функция makeWorker создаёт другую функцию и возвращает её. Новая функция может быть вызвана откуда-то ещё. Получит ли она доступ к внешним переменным из места своего создания или места выполнения или из обоих?

function makeWorker() {

let name = "Pete";

return function() {

alert(name);

};

}

let name = "John";

let work = makeWorker();

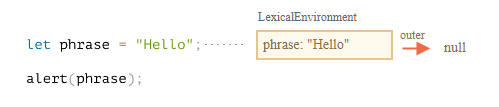
work(); // что будет показано? "Pete" (из места создания) или "John" (из места выполнения)

[**Лексическое окружение**](https://learn.javascript.ru/closure#leksicheskoe-okruzhenie)

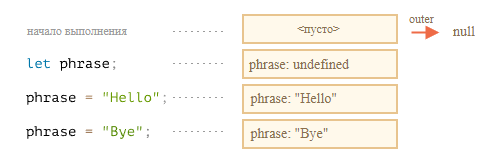
Чтобы понять, что происходит, давайте для начала, обсудим, что такое «переменная» на самом деле. В JavaScript у каждой выполняемой функции, блока кода и скрипта есть связанный с ними внутренний (скрытый) объект, называемый лексическим окружением LexicalEnvironment. Объект лексического окружения состоит из двух частей:

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

Переменная – это просто свойство специального внутреннего объекта: Environment Record. «Получить или изменить переменную», означает, «получить или изменить свойство этого объекта». Например, в этом простом коде только одно лексическое окружение:



Это, так называемое, глобальное лексическое окружение, связанное со всем скриптом. На картинке выше прямоугольник означает Environment Record (хранилище переменных), а стрелка означает ссылку на внешнее окружение. У глобального лексического окружения нет внешнего окружения, так что она указывает на null. А вот как оно изменяется при объявлении и присваивании переменной:



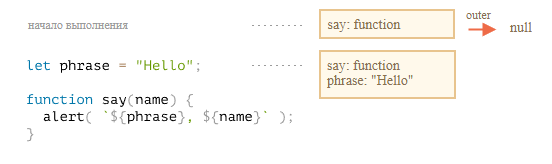
Прямоугольники с правой стороны демонстрируют, как глобальное лексическое окружение изменяется в процессе выполнения кода:

1. В начале скрипта лексическое окружение пустое.
2. Появляется определение переменной let phrase. У неё нет присвоенного значения, поэтому присваивается undefined.
3. Переменной phrase присваивается значение.
4. Переменная phrase меняет значение.

[**Function Declaration**](https://learn.javascript.ru/closure#function-declaration)

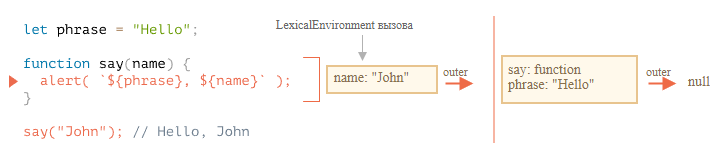
До сих поры рассматривались только переменные. Теперь рассмотрим Function Declaration. В отличие от переменных, объявленных с помощью let, они полностью инициализируются не тогда, когда выполнение доходит до них, а раньше, когда создаётся лексическое окружение. Для верхнеуровневых функций это означает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

Следующий код демонстрирует, что уже с самого начала в лексическом окружении что-то есть. Там есть say, потому что это Function Declaration. И позже там появится phrase, объявленное через let:



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

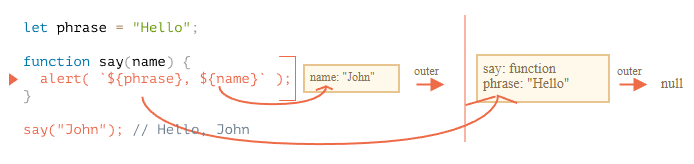
В течение вызова say() использует внешнюю переменную phrase. При запуске функции для неё автоматически создаётся новое лексическое окружение, для хранения локальных переменных и параметров вызова. Например, для say("John") это выглядит так (выполнение находится на строке, отмеченной стрелкой):



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

Рассмотрим, как происходит поиск в рассматриваемом примере:

* когда alert внутри say хочет получить доступ к name, он немедленно находит переменную в лексическом окружении функции;
* когда он хочет получить доступ к phrase, которой нет локально, он следует дальше по ссылке к внешнему лексическому окружению и находит переменную там.



Это ответ на первый вопрос из начала темы. Функция получает текущее значение внешних переменных, то есть, их последнее значение. Старые значения переменных нигде не сохраняются. Когда функция хочет получить доступ к переменной, она берёт её текущее значение из своего или внешнего лексического окружения.

Так что, ответ на первый вопрос – Pete:

let name = "John";

function sayHi() {

alert("Hi, " + name);

}

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

sayHi(); // Pete

Порядок выполнения кода, приведённого выше:

1. В глобальном лексическом окружении есть name: "John".
2. На строке (\*) глобальная переменная изменяется, теперь name: "Pete".
3. Выполняется функция sayHi() и берёт переменную name извне. Теперь из глобального лексического окружения, где переменная уже равна "Pete".

Обратите внимание, что новое лексическое окружение функции создаётся каждый раз, когда функция выполняется. Если функция вызывается несколько раз, то для каждого вызова будет своё лексическое окружение, со своими, специфичными для этого вызова, локальными переменными и параметрами.

Лексическое окружение – это специальный внутренний объект. Нельзя получить его в коде и изменять напрямую. Сам движок JavaScript может оптимизировать его, уничтожать неиспользуемые переменные для освобождения памяти и выполнять другие внутренние уловки, но видимое поведение объекта должно оставаться таким, как было описано.

[**Вложенные функции**](https://learn.javascript.ru/closure#vlozhennye-funktsii)

Функция называется вложенной, когда она создаётся внутри другой функции. Можно использовать это для упорядочивания кода, например:

function sayHiBye(firstName, lastName) {

function getFullName() {

return firstName + " " + lastName;

}

alert( "Hello, " + getFullName() );

alert( "Bye, " + getFullName() );

}

Здесь вложенная функция getFullName() создана для удобства. Она может получить доступ к внешним переменным и, значит, вывести полное имя. В JavaScript вложенные функции используются очень часто.

Вложенная функция может быть возвращена: либо в качестве свойства нового объекта, либо сама по себе, и затем может быть использована в любом месте. Не важно где, она всё так же будет иметь доступ к тем же внешним переменным. Например, здесь, вложенная функция присваивается новому объекту в [конструкторе](https://learn.javascript.ru/constructor-new):

function User(name) {

this.sayHi = function() {

alert(name);

};

}

let user = new User("John");

user.sayHi();

Создаём и возвращаем функцию «счётчик»:

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter = makeCounter();

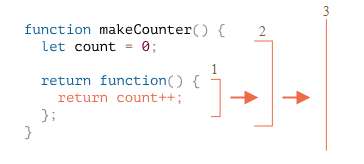
alert( counter() ); // 0

alert( counter() ); // 1

alert( counter() ); // 2

Рассмотрим далее makeCounter. Он создаёт функцию «counter», которая возвращает следующее число при каждом вызове. Несмотря на простоту, немного модифицированные варианты этого кода применяются на практике, например, в [генераторе псевдослучайных чисел](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_%D0%BF%D1%81%D0%B5%D0%B2%D0%B4%D0%BE%D1%81%D0%BB%D1%83%D1%87%D0%B0%D0%B9%D0%BD%D1%8B%D1%85_%D1%87%D0%B8%D1%81%D0%B5%D0%BB) и во многих других случаях.

Когда внутренняя функция начинает выполняться, начинается поиск переменной count++ изнутри-наружу. Для примера выше порядок будет такой:



1. Локальные переменные вложенной функции.
2. Переменные внешней функции.
3. И так далее, пока не будут достигнуты глобальные переменные.

В этом примере count будет найден на шаге 2. Когда внешняя переменная модифицируется, она изменится там, где была найдена. Значит, count++ найдёт внешнюю переменную и увеличит её значение в лексическом окружении, которому она принадлежит. Как если бы было let count = 1.

Теперь рассмотрим два вопроса:

1. Можно ли каким-нибудь образом сбросить счётчик count из кода, который не принадлежит makeCounter? Например, после вызова alert в коде выше.
2. Если вызвать makeCounter несколько раз – возвращается много функций counter. Они независимы или разделяют одну и ту же переменную count?

Ответы на вопросы:

1. Такой возможности нет: count – локальная переменная функции, нельзя получить к ней доступ извне.
2. Для каждого вызова makeCounter() создаётся новое лексическое окружение функции, со своим собственным count. Так что, получившиеся функции counter – независимы.

Вот демо:

function makeCounter() {

let count = 0;

return function() {

return count++;

};

}

let counter1 = makeCounter();

let counter2 = makeCounter();

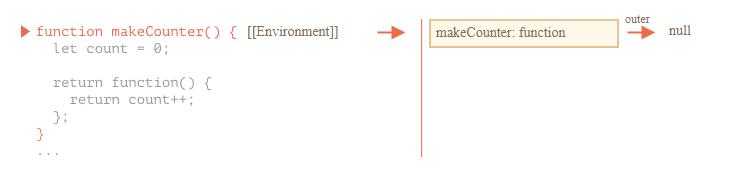
alert( counter1() ); // 0

alert( counter1() ); // 1

alert( counter2() ); // 0

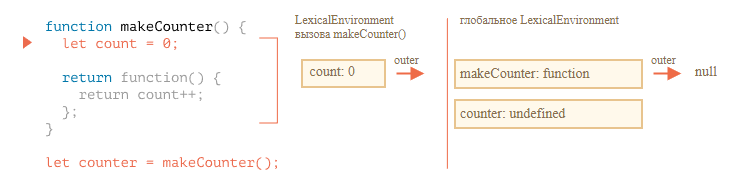
Рассмотрим, что происходит в примере с makeCounter шаг за шагом. Обратите внимание на дополнительное свойство [[Environment]].

1. Когда скрипт только начинает выполняться, есть только глобальное лексическое окружение:



В этот начальный момент есть только функция makeCounter, потому что это Function Declaration. Она ещё не выполняется. Все функции при создании получают скрытое свойство [[Environment]], которое ссылается на лексическое окружение места, где они были созданы. В данном случае, makeCounter создан в глобальном лексическом окружении, так что [[Environment]] содержит ссылку на него. Другими словами, функция навсегда запоминает ссылку на лексическое окружение, где она была создана. И [[Environment]] – скрытое свойство функции, которое содержит эту ссылку.

1. Код продолжает выполняться, объявляется новая глобальная переменная counter, которой присваивается результат вызова makeCounter. На рисунке изображено состояние, когда интерпретатор находится на первой строке внутри makeCounter():



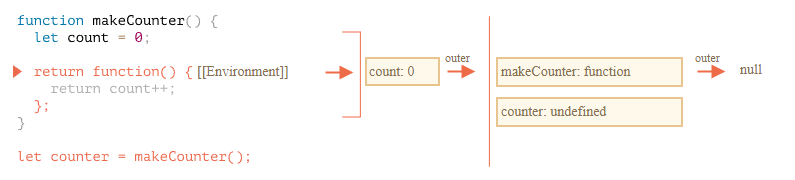
В момент вызова makeCounter() создаётся лексическое окружение, для хранения его переменных и аргументов. Как и все лексические окружения, оно содержит две вещи:

* Environment Record с локальными переменными. В рассматриваемом примере count – единственная локальная переменная (появляющаяся, когда выполняется строчка с let count).
* Ссылка на внешнее окружение, которая устанавливается в значение [[Environment]] функции. В данном случае, [[Environment]] функции makeCounter ссылается на глобальное лексическое окружение.

Итак, теперь есть два лексических окружения: первое – глобальное, второе – для текущего вызова makeCounter, с внешней ссылкой на глобальный объект.

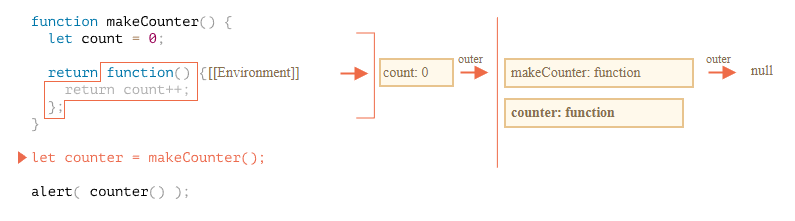
1. В процессе выполнения makeCounter() создаётся небольшая вложенная функция.

Не имеет значения, какой способ объявления функции используется: Function Declaration или Function Expression. Все функции получают свойство [[Environment]], которое ссылается на лексическое окружение, в которым они были созданы. Тоже происходит и с новой функцией. Для нее значением [[Environment]] будет текущее лексическое окружение makeCounter() (где она была создана):



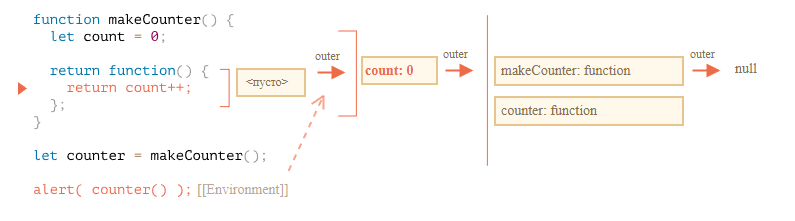
Обратите внимание, что на этом шаге внутренняя функция была создана, но ещё не вызвана. Код внутри function() { return count++ } не выполняется.

1. Выполнение продолжается, вызов makeCounter() завершается, и результат (небольшая вложенная функция) присваивается глобальной переменной counter:



В этой функции есть только одна строчка: return count++, которая будет выполнена при вызове функции.

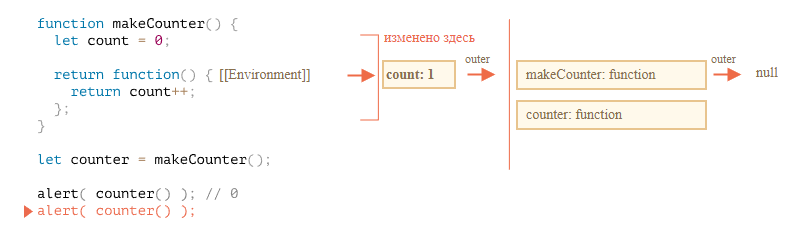
1. При вызове counter() для этого вызова создаётся новое лексическое окружение. Оно пустое, так как в самом counter локальных переменных нет. Но [[Environment]] counter используется, как ссылка на внешнее лексическое окружение outer, которое даёт доступ к переменным предшествующего вызова makeCounter, где counter был создан.



Теперь, когда вызов ищет переменную count, он сначала ищет в собственном лексическом окружении (пустое), а затем в лексическом окружении предшествующего вызова makeCounter(), где и находит её.

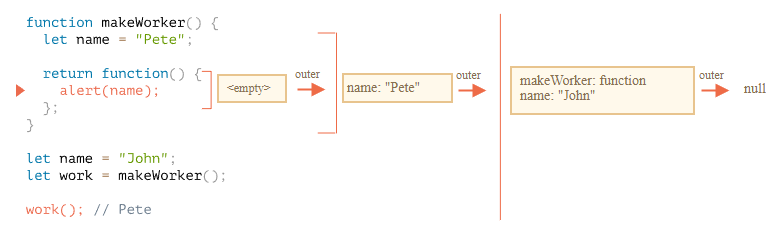
Обратите внимание, как здесь работает управление памятью. Хотя makeCounter() закончил выполнение некоторое время назад, его лексическое окружение остаётся в памяти, потому что есть вложенная функция с [[Environment]], который ссылается на него. В большинстве случаев, объект лексического окружения существует до того момента, пока есть функция, которая может его использовать. И только тогда, когда таких не остаётся, окружение уничтожается.

1. Вызов counter() не только возвращает значение count, но также увеличивает его. Модификация происходит «на месте». Значение count изменяется конкретно в том окружении, где оно было найдено.



1. Следующие вызовы counter() сделают то же самое.

Теперь ответ на второй вопрос из начала темы очевиден: функция work() в коде ниже получает name из того места, где была создана, через ссылку на внешнее лексическое окружение:



Так что, результатом будет "Pete". Но, если бы в makeWorker() не было let name, тогда бы поиск продолжился дальше и была бы взята глобальная переменная. В таком случае, результатом было бы "John".

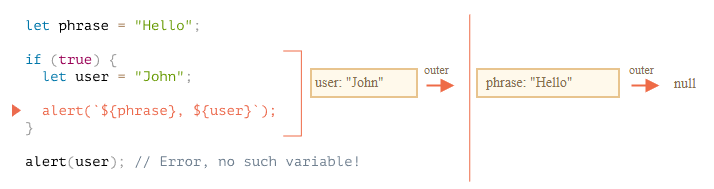
**Замыкания**

[Замыкание](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%BC%D1%8B%D0%BA%D0%B0%D0%BD%D0%B8%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)) – это функция, которая запоминает свои внешние переменные и может получить к ним доступ. В некоторых языках это невозможно, или функция должна быть написана специальным образом, чтобы получилось замыкание. Но, как было описано выше, в JavaScript, все функции изначально являются замыканиями (есть только одно исключение, про которое будет рассказано в [Синтаксис "new Function"](https://learn.javascript.ru/new-function)). То есть, они автоматически запоминают, где были созданы, с помощью скрытого свойства [[Environment]] и все они могут получить доступ к внешним переменным.

[**Блоки кода и циклы**](https://learn.javascript.ru/closure#bloki-koda-i-tsikly-iife)

Предыдущие примеры сосредоточены на функциях. Но лексическое окружение существует для любых блоков кода {...}. Лексическое окружение создаётся при выполнении блока кода и содержит локальные переменные для этого блока.

В следующем примере переменная user существует только в блоке if:



Когда выполнение попадает в блок if, для этого блока создаётся новое лексическое окружение. У него есть ссылка на внешнее окружение, так что phrase может быть найдена. Но все переменные и Function Expression, объявленные внутри if, остаются в его лексическом окружении и не видны снаружи. Например, после завершения if следующий alert не увидит user, что вызовет ошибку.

Для цикла у каждой итерации своё отдельное лексическое окружение. Если переменная объявлена в for(let ...), то она также в нём:

for (let i = 0; i < 10; i++) {

// {i: value}

}

alert(i); // Ошибка, нет такой переменной

Обявление *let i* визуально находится снаружи {...}, но конструкция for – особенная в этом смысле, у каждой итерации цикла своё собственное лексическое окружение с текущим *i* в нём. И так же, как и в if, ниже цикла *i* невидима.

Также можно использовать «простые» блоки кода {...}, чтобы изолировать переменные в «локальной области видимости». Например, в браузере все скрипты (кроме type="module") разделяют одну общую глобальную область. Так что, если создать глобальную переменную в одном скрипте, она станет доступна и в других. Но это становится источником конфликтов, если два скрипта используют одно и тоже имя переменной и перезаписывают друга друга. Это может произойти, если название переменной – широко распространённое слово, а авторы скрипта не знают друг о друге. Если надо этого избежать, то можно использовать блок кода для изоляции всего скрипта или какой-то его части:

{

let message = "Hello";

alert(message); // Hello

}

alert(message); // Ошибка: переменная message не определена

Из-за того, что у блока есть собственное лексическое окружение, код снаружи него (или в другом скрипте) не видит переменные этого блока.

Обычно лексическое окружение очищается и удаляется после того, как функция выполнилась. Например:

function f() {

let value1 = 123;

let value2 = 456;

}

f();

Здесь два значения, которые технически являются свойствами лексического окружения. Но после того, как f() завершится, это лексическое окружение станет недоступно, поэтому оно удалится из памяти. Но, если есть вложенная функция, которая всё ещё доступна после выполнения f, то у неё есть свойство [[Environment]], которое ссылается на внешнее лексическое окружение, тем самым оставляя его достижимым:

function f() {

let value = 123;

function g() { alert(value); }

return g;

}

let g = f();

Но на практике движки JavaScript пытаются это оптимизировать. Они анализирует использование переменных и, если легко по коду понять, что внешняя переменная не используется – она удаляется. Одним из важных побочных эффектов в V8 (Chrome, Opera) является то, что такая переменная становится недоступной при отладке. Это может привести к тому, что можно увидеть не ту внешнюю переменную при совпадающих названиях:

let value = "Сюрприз!";

function f() {

let value = "ближайшее значение";

function g() {

debugger; // в консоли: напишите alert(value);

}

return g;

}

let g = f();

g();

1. **Конструкция new Function.**

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

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

Функция создается с заданными аргументами arg1...argN и телом functionBody. Это проще понять на конкретном примере. Здесь объявлена функция с двумя аргументами:

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

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

А вот функция без аргументов, в этом случае достаточно указать только тело:

let sayHi = new Function('alert("Hello")');

sayHi(); // Hello

Главное отличие от других способов объявления функции, которые были рассмотрены ранее, заключается в том, что функция создаётся полностью «на лету» из строки, переданной во время выполнения.

Все предыдущие объявления требовали писать объявление функции в скрипте. Но new Function позволяет превратить любую строку в функцию. Например, можно получить новую функцию с сервера и затем выполнить ее:

let str = ... код, полученный с сервера динамически ...

let func = new Function(str);

func();

Это используется в очень специфических случаях, например, когда получаем код с сервера для динамической компиляции функции из шаблона, в сложных веб-приложениях.

Когда функция создаётся с использованием new Function, в её [[Environment]] записывается ссылка не на текущее лексическое окружение, а на глобальное. Поэтому такая функция не имеет доступа к внешним переменным, только к глобальным.

function getFunc() {

let value = "test";

let func = new Function('alert(value)');

return func;

}

getFunc()(); // ошибка: value не определено

Сравним это с обычным объявлением:

function getFunc() {

let value = "test";

let func = function() { alert(value); };

return func;

}

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

Эта особенность new Function очень полезна на практике. Представьте, что нужно создать функцию из строки. Код этой функции неизвестен во время написания скрипта (вот поэтому не используем обычные функции), а будет определён только в процессе выполнения. Можно получить код с сервера или другого ресурса.

Новая функция должна взаимодействовать с основным скриптом. Если бы она имела доступ к внешним переменным это привело бы к проблеме. Проблема в том, что перед отправкой JavaScript-кода на реальные работающие проекты код сжимается с помощью *минификатора* – специальной программы, которая уменьшает размер кода, удаляя комментарии, лишние пробелы, и, что самое главное, локальным переменным даются укороченные имена. Например, если в функции объявляется переменная let userName, то минификатор изменяет её на let a (или другую букву, если она не занята), и изменяет её везде. Обычно так делать безопасно, потому что переменная является локальной и никто снаружи не имеет к ней достп. И внутри функции минификатор заменяет каждое её упоминание. Минификаторы анализируют структуру кода, и поэтому ничего не ломают.

Так что если бы new Function и имела доступ к внешним переменным, она не смогла бы найти переименованную userName. Кроме того, такой код был бы архитектурно хуже и более подвержен ошибкам. Чтобы передать что-то в функцию, созданную как new Function, можно использовать ее аргументы.

1. **Объект функции.**

Как известно, в JavaScript функция – это значение. Каждое значение в JavaScript имеет свой тип. В JavaScript, функции – это объекты.

Можно представить функцию как «объект, который может делать какое-то действие». Функции можно не только вызывать, но использовать их как обычные объекты: добавлять/удалять свойства, передавать их по ссылке и т.д.

[**Свойство name**](https://learn.javascript.ru/function-object#svoystvo-name)

Объект функции содержит несколько полезных свойств. Например, имя функции доступно как свойство name:

function sayHi() {

alert("Hi");

}

alert(sayHi.name); // sayHi

Корректное имя присваивается даже в случае, если функция создается без имени и тут же присваивается:

let sayHi = function() {

alert("Hi");

};

alert(sayHi.name); // sayHi

Это даже работает в случае присваивания значения по умолчанию:

function f(sayHi = function() {}) {

alert(sayHi.name); // sayHi

}

f();

В спецификации это называется «контекстное имя»: если функция не имеет name, то JavaScript пытается понять его из контекста.

Также имена имеют и методы объекта:

let user = {

sayHi() {

// ...

},

sayBye: function() {

// ...

}

}

alert(user.sayHi.name); // sayHi

alert(user.sayBye.name); // sayBye

В этом нет никакой магии. Бывает, что корректное имя определить невозможно. В таких случаях свойство name имеет пустое значение. Например:

let arr = [function() {}];

alert( arr[0].name ); // <пустая строка>

На практике такое бывает редко, обычно функции имеют name.

[**Свойство length**](https://learn.javascript.ru/function-object#svoystvo-length)

Ещё одно встроенное свойство length содержит количество параметров функции в её объявлении. Например:

function f1(a) {}

function f2(a, b) {}

function many(a, b, ...more) {}

alert(f1.length); // 1

alert(f2.length); // 2

alert(many.length); // 2

Как видно, троеточие, обозначающее «остальные параметры», здесь не учитываются.

Свойство length иногда используется для [интроспекций](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D1%80%D0%BE%D1%81%D0%BF%D0%B5%D0%BA%D1%86%D0%B8%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)) в функциях, которые работают с другими функциями. Например, в коде ниже функция ask принимает в качестве параметров вопрос question и произвольное количество функций-обработчиков ответа handler. Когда пользователь отвечает на вопрос, функция вызывает обработчики. Можно передать два типа обработчиков:

* Функцию без аргументов, которая будет вызываться только в случае положительного ответа.
* Функцию с аргументами, которая будет вызываться в обоих случаях и возвращать ответ.

Чтобы вызвать обработчик handler правильно, надо проверить свойство handler.length. Идея состоит в том, чтобы иметь простой синтаксис обработчика без аргументов для положительных ответов (наиболее распространённый случай), но также и возможность передавать универсальные обработчики:

function ask(question, ...handlers) {

let isYes = confirm(question);

for(let handler of handlers) {

if (handler.length == 0) {

if (isYes) handler();

} else {

handler(isYes);

}

}

}

ask("Вопрос?", () => alert('Вы ответили да'), result => alert(result));

[**Пользовательские свойства**](https://learn.javascript.ru/function-object#polzovatelskie-svoystva)

Также можно добавить свои собственные свойства. Например, свойство counter для отслеживания общего количества вызовов:

function sayHi() {

alert("Hi");

sayHi.counter++;

}

sayHi.counter = 0;

sayHi(); // Hi

sayHi(); // Hi

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

**Свойство не есть переменная**

Свойство функции, назначенное как sayHi.counter = 0, не объявляет локальную переменную counter внутри неё. Другими словами, свойство counter и переменная let counter – это две независимые вещи. Можно использовать функцию как объект, хранить в ней свойства, но они никак не влияют на её выполнение. Переменные – это не свойства функции и наоборот.

Иногда свойства функции могут использоваться вместо замыканий. Например, можно переписать функцию-счетчик из вопроса про [замыкание](https://learn.javascript.ru/closure), используя её свойство:

function makeCounter() {

function counter() {

return counter.count++;

};

counter.count = 0;

return counter;

}

let counter = makeCounter();

alert( counter() ); // 0

alert( counter() ); // 1

Свойство count теперь хранится прямо в функции, а не в её внешнем лексическом окружении. Основное отличие такого подхода от замыкания в том, что если значение count живет во внешней переменной, то она не доступна для внешнего кода. Изменить её могут только вложенные функции. А если оно присвоено как свойство функции, то можно его получить:

function makeCounter() {

function counter() {

return counter.count++;

};

counter.count = 0;

return counter;

}

let counter = makeCounter();

counter.count = 10;

alert( counter() ); // 10

Поэтому выбор реализации зависит от целей разработчика.

[**Named Function Expression**](https://learn.javascript.ru/function-object#named-function-expression)

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

let sayHi = function func(who) {

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

};

Заметьте, что функция всё ещё задана как Function Expression. Добавление "func" после function не превращает объявление в Function Declaration, потому что оно все еще является частью выражения присваивания. Добавление такого имени ничего не ломает. Функция все еще доступна как sayHi():

let sayHi = function func(who) {

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

};

sayHi("John"); // Hello, John

Есть две важные особенности имени func, ради которого оно даётся:

1. Оно позволяет функции ссылаться на себя же.
2. Оно не доступно за пределами функции.

Например, ниже функция sayHi вызывает себя с "Guest", если не передан параметр who:

let sayHi = function func(who) {

if (who) {

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

} else {

func("Guest");

}

};

sayHi(); // Hello, Guest

func(); // Ошибка

Не следует использовать имя sayHi для вложенного вызова, так как значение sayHi может быть изменено. Функция может быть присвоена другой переменной, и тогда код начнет выдавать ошибки:

let sayHi = function(who) {

if (who) {

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

} else {

sayHi("Guest"); // Ошибка

}

};

let welcome = sayHi;

sayHi = null;

welcome(); // Ошибка

Так происходит, потому что функция берет sayHi из внешнего лексического окружения. Так как локальная переменная sayHi отсутствует, используется внешняя. И на момент вызова эта внешняя sayHi равна null. Необязательное имя, которое можно вставить в Function Expression, как раз и призвано решать такого рода проблемы. Все работает, потому что имя "func" локальное и находится внутри функции. Теперь оно взято не снаружи (и недоступно оттуда). Спецификация гарантирует, что оно всегда будет ссылаться на текущую функцию. Внешний код все еще содержит переменные sayHi и welcome, но теперь func – это «внутреннее имя функции», таким образом она может вызвать себя изнутри.

Задать «внутреннее» имя можно только для Function Expression, и не нельзя для Function Declaration. Если нужно надёжное «внутреннее» имя, стоит переписать Function Declaration на Named Function Expression.

1. **Остаточные параметры и оператор расширения.**

Многие встроенные функции JavaScript поддерживают произвольное количество аргументов. Например: Math.max(arg1, arg2, ..., argN) – вычисляет максимальное число из переданных; Object.assign(dest, src1, ..., srcN) – копирует свойства из исходных объектов src1..N в целевой объект dest и др.

[**Остаточные параметры (...)**](https://learn.javascript.ru/rest-parameters-spread-operator#ostatochnye-parametry)

Вызывать функцию можно с любым количеством аргументов, независимо от того, как она была определена. Например:

function sum(a, b) {

return a + b;

}

alert( sum(1, 2, 3, 4, 5) );

Лишние аргументы не вызовут ошибку, но приняты будут только первые два.

Остаточные параметры могут быть обозначены через три точки «...». Суть его в том, что оставшиеся параметры помещаются в массив. Например, соберём все аргументы в массив args:

function sumAll(...args) {

let sum = 0;

for (let arg of args) sum += arg;

return sum;

}

alert( sumAll(1) ); // 1

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

alert( sumAll(1, 2, 3) ); // 6

Можно положить первые несколько параметров в переменные и собрать в массив остальные. В примере ниже первые два аргумента функции станут именем и фамилией, а третий и последующие превратятся в массив titles:

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

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

// titles = ["Консул", "Император"]

alert( titles[0] ); // Консул

alert( titles[1] ); // Император

alert( titles.length ); // 2

}

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

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

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

}

Все аргументы функции находятся в псевдомассиве arguments под своими порядковыми номерами. Например:

function showName() {

alert( arguments.length );

alert( arguments[0] );

alert( arguments[1] );

}

// Вывод: 2, Юлий, Цезарь

showName("Юлий", "Цезарь");

// Вывод: 1, Илья, undefined

showName("Илья");

Раньше в языке не было остаточных параметров, и получить все аргументы функции можно было только с помощью arguments. Этот способ всё ещё работает, его можно найти в старом коде. Но у него есть один недостаток. Хотя arguments похож на массив, и он тоже перебираемый, это всё же не массив. Он не поддерживает методы массивов, поэтому нельзя, например, вызвать arguments.map(...). К тому же, arguments всегда содержит все аргументы функции – нельзя получить их часть. А остаточные параметры позволяют это сделать.

Соответственно, для более удобной работы с аргументами лучше использовать остаточные параметры.

У стрелочных функций нет своего объекта arguments. Если обратиться к arguments из стрелочной функции, то получим аргументы внешней обычной функции. Пример:

function f() {

let showArg = () => alert(arguments[0]);

showArg(2);

}

f(1); // 1

**[Оператор расширения](https://learn.javascript.ru/rest-parameters-spread-operator" \l "spread-operator)**

Иногда нужно массив преобразовать в список параметров. Например, есть встроенная функция [Math.max](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Math/max). Она возвращает наибольшее число из списка:

alert( Math.max(3, 5, 1) ); // 5

Если вызвать эту функцию для массива чисел [3, 5, 1], то она его не обработает, так как ожидает список параметров:

let arr = [3, 5, 1];

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

Чтобы преобразовать массив в список необходимо использовать оператор расширения. Он похож на остаточные параметры – тоже использует ..., но делает совершенно противоположное. Когда ...arr используется при вызове функции, он «расширяет» перебираемый объект arr в список аргументов. Для Math.max:

let arr = [3, 5, 1];

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

Этим же способом можно передать несколько итерируемых объектов и комбинировать оператор расширения с обычными значениями:

let arr1 = [1, -2, 3, 4];

let arr2 = [8, 3, -8, 1];

alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25

Оператор расширения можно использовать и для слияния массивов:

let arr = [3, 5, 1];

let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15

Оператора расширения работает с любым перебираемым объектом. Например, оператор расширения подойдёт для того, чтобы превратить строку в массив символов:

let str = "Привет";

alert( [...str] ); // П,р,и,в,е,т

Оператор расширения использует итераторы, чтобы собирать элементы. Так же, как это делает for..of. Цикл for..of перебирает строку как последовательность символов, поэтому из ...str получается "П", "р", "и", "в", "е", "т". Получившиеся символы собираются в массив при помощи стандартного объявления массива: [...str].

Для этой задачи можно использовать и Array.from. Он тоже преобразует перебираемый объект (такой как строка) в массив:

let str = "Привет";

alert( Array.from(str) ); // П,р,и,в,е,т

Результат аналогичен [...str]. Но между Array.from(obj) и [...obj] есть разница: Array.from работает как с псевдомассивами, так и с итерируемыми объектами; оператор расширения работает только с итерируемыми объектами. Таким образом, если нужно сделать из чего угодно массив, Array.from – более универсальный метод.

1. **Каррирование**

[*Каррирование*](https://ru.wikipedia.org/wiki/%D0%9A%D0%B0%D1%80%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) или карринг (currying) в функциональном программирование – это преобразование функции с множеством аргументов в набор вложенных функций с одним аргументом. При вызове каррированной функции с передачей ей одного аргумента, она возвращает новую функцию, которая ожидает поступления следующего аргумента. Новые функции, ожидающие следующего аргумента, возвращаются при каждом вызове каррированной функции – до тех пор, пока функция не получит все необходимые ей аргументы. Ранее полученные аргументы, благодаря механизму замыканий, ждут того момента, когда функция получит всё, что ей нужно для выполнения вычислений. После получения последнего аргумента функция выполняет вычисления и возвращает результат.

Говоря о [каррировании](https://medium.com/@kbrainwave/currying-in-javascript-ce6da2d324fe), можно сказать, что это процесс превращения функции с несколькими аргументами в функцию с меньшей арностью.

[*Арность*](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%BD%D0%BE%D1%81%D1%82%D1%8C) – это количество аргументов функции. Например – вот объявление пары функций:

function fn(a, b) {

*//...*

}

function \_fn(a, b, c) {

*//...*

}

Функция fn принимает два аргумента (это бинарная или 2-арная функция), функция \_fn принимает три аргумента (тернарная, 3-арная функция).

В результате каррирования, функция с несколькими аргументами преобразуется в набор функций, каждая из которых принимает один аргумент. Рассмотрим пример:

function multiply(a, b, c) {

   return a \* b \* c;

}

Функция multiply принимает три аргумента и возвращает их произведение:  
  
multiply(1,2,3); *// 6*

Преобразовать её к набору функций, каждая из которых принимает один аргумент. Создадим каррированный вариант этой функции и посмотрим на то, как получить тот же результат в ходе вызова нескольких функций:

function multiply(a) {

   return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

}

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

В примере вызов единственной функции с тремя аргументами multiply(1,2,3) преобразован к вызову трёх функций – multiply(1)(2)(3). При использовании новой конструкции каждая функция, кроме последней, возвращающей результат вычислений, принимает аргумент и возвращает другую функцию, также способную принять аргумент и возвратить другую функцию. Распишем конструкцию вида multiply(1)(2)(3), чтобы она была более понятной:

const mul1 = multiply(1)*;*

const mul2 = mul1(2)*;*

const result = mul2(3)*;*

log(result)*; // 6*

Подробнее разберём, что здесь происходит. Сначала передается аргумент 1 функции multiply:

const mul1 = multiply(1);

При работе этой функции срабатывает такая конструкция:

return (b) => {

       return (c) => {

           return a \* b \* c

       }

   }

Теперь в mul1 имеется ссылка на функцию, принимающую аргумент b. Вызовем функцию mul1, передав ей 2:

const mul2 = mul1(2);

В результате этого вызова выполнится следующий код:

return (c) => {

           return a \* b \* c

       }

Константа mul2 будет содержать ссылку на функцию, которая могла бы оказаться в ней, например, в результате выполнения следующей операции:

mul2 = (c) => {

           return a \* b \* c

       }

Если теперь вызвать функцию mul2, передав ей 3, то функция выполнит необходимые вычисления, воспользовавшись аргументами a и b:

const result = mul2(3);

Результатом этих вычислений будет 6:

log(result)*; // 6*

Функция mul2, обладающая самым большим уровнем вложенности, имеет доступ к областям видимости, к замыканиям, формируемым функциями multiply и mul1. Именно поэтому в функции mul2можно производить вычисления с переменными, объявленными в функциях, выполнение которых уже завершено, которые уже возвратили некие значения и обработаны сборщиком мусора.

**Каррирование и частичное применение функций**

Сейчас, возникает ощущение, что количество вложенных функций, при представлении функции в виде набора вложенных функций, зависит от количества аргументов функции. И, если речь идёт о каррировании, то это так. Особый вариант функции для вычисления объёма, можно сделать таким:

function volume(l) {

   return (w, h) => {

       return l \* w \* h

   }

}

Здесь применены идеи, очень похожие на рассмотренные выше. Пользоваться этой функцией можно так:

const hV = volume(70)*;*

hV(203,142)*;*

hV(220,122)*;*

hV(120,123)*;*

А можно и так:

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

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

volume(70)(940,340)*;*

Фактически, здесь командой volume(70), создана специализированная функция для вычисления объёма тел, одно из измерений которых (а именно – длина, l), зафиксировано. Функция volume ожидает 3 аргумента и содержит 2 вложенных функции, в отличие от предыдущей версии подобной функции, каррированный вариант которой содержал 3 вложенных функции.

Та функция, которая получилась после вызова volume(70) реализует концепцию частичного применения функции (partial function application). Каррирование и частичное применение функций очень похожи друг на друга, но концепции это разные.

При частичном применении функцию преобразуют в другую функцию, обладающую меньшим числом аргументов (меньшей арностью). Некоторые аргументы такой функции оказываются зафиксированными (для них задаются значения по умолчанию).

Каррирование и частичное применение функций может оказаться полезным в различных ситуациях. Например – при разработке небольших модулей, подходящих для повторного использования.

Частичное применение функций позволяет облегчить использование универсальных модулей. Например, есть интернет-магазин, в коде которого имеется функция, которая используется для вычисления суммы к оплате с учётом скидки.

function discount(price, discount) {

   return price \* discount

}

Есть определённая категория клиентов, которые получают скидку в 10%. Например, если такой клиент покупает что-то на $500, то он получает скидку размером $50:

const price = discount(500,0.10); // $50

// $500 - $50 = $450

Несложно заметить, что при таком подходе, постоянно придётся вызывать эту функцию с двумя аргументами:

const price = discount(1500,0.10); // $150

// $1,500 - $150 = $1,350

const price = discount(2000,0.10); // $200

// $2,000 - $200 = $1,800

const price = discount(50,0.10); // $5

// $50 - $5 = $45

const price = discount(5000,0.10); // $500

// $5,000 - $500 = $4,500

const price = discount(300,0.10); // $30

// $300 - $30 = $270

Исходную функцию можно привести к такому виду, который позволял бы получать новые функции с заранее заданным уровнем скидки, при вызове которых им достаточно передавать сумму покупки. Функция discount() в примере имеет два аргумента. Преобразуем ее следующим образом:

function discount(discount) {

   return (price) => {

       return price \* discount;

   }

}

const tenPercentDiscount = discount(0.1);

Функция tenPercentDiscount() представляет собой результат частичного применения функции discount(). При вызове tenPercentDiscount() этой функции достаточно передать цену, а скидка в 10%, то есть – аргумент discount, уже будет задана:

tenPercentDiscount(500); // $50

// $500 - $50 = $450

Если в магазине имеются покупатели, которым решено дать скидку размером в 20%, то получить соответствующую функцию для работы с ними можно так:

const twentyPercentDiscount = discount(0.2);

Теперь функцию twentyPercentDiscount() можно вызывать для расчёта стоимости товаров с учётом скидки в 20%:

twentyPercentDiscount(500); // 100

// $500 - $100 = $400

twentyPercentDiscount(5000); // 1000

// $5,000 - $1,000 = $4,000

twentyPercentDiscount(1000000); // 200000

// $1,000,000 - $200,000 = $600,000

1. **Генераторы**

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

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

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

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

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

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

alert(generator); // [object Generator]

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

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

* value: значение из yield.
* done: true, если выполнение функции завершено, иначе false.

Создадим генератор и получим первое из возвращаемых им значений:

function\* generateSequence() {

yield 1;

yield 2;

return 3;

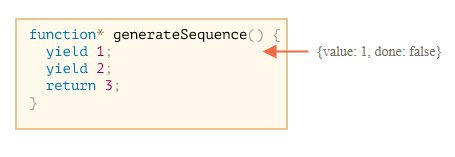
}

let generator = generateSequence();

let one = generator.next();

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

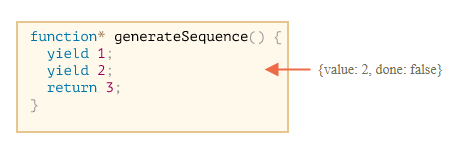
На данный момент получено только первое значение, выполнение функции остановлено на второй строке:



Повторный вызов generator.next() возобновит выполнение кода и вернёт результат следующего yield:

let two = generator.next();

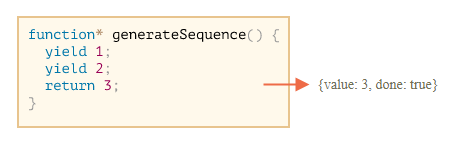
alert(JSON.stringify(two)); // {value: 2, done: false}



И, наконец, последний вызов завершит выполнение функции и вернёт результат return:

let three = generator.next();

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



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

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

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

function\* generateSequence() {

yield 1;

yield 2;

return 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, затем 2

}

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

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

let generator = generateSequence();

for(let value of generator) {

alert(value); // 1, затем 2, затем 3

}

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

function\* generateSequence() {

yield 1;

yield 2;

yield 3;

}

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

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

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

В примерах выше генерируются конечные последовательности, но также можно сделать генератор, который будет возвращать значения бесконечно. Например, бесконечная последовательность псевдослучайных чисел. Тут потребуется break (или return) в цикле for..of по такому генератору, иначе цикл будет продолжаться бесконечно и скрипт «зависнет».

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

Композиция генераторов – это особенная возможность генераторов, которая позволяет прозрачно «встраивать» генераторы друг в друга. Например, есть функция для генерации последовательности чисел:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

Надо использовать её при генерации более сложной последовательности: сначала цифры 0..9 (с кодами символов 48…57), за которыми следуют буквы алфавита a..z (коды символов 65…90), за которыми следуют буквы в верхнем регистре A..Z (коды символов 97…122). Можно использовать такую последовательность для генерации паролей, выбирать символы из неё (может быть, ещё добавить символы пунктуации), но сначала её нужно сгенерировать.

В обычной функции, чтобы объединить результаты из нескольких других функций, надо вызвать их, сохранить промежуточные результаты, а затем в конце их объединить. Для генераторов есть особый синтаксис yield\*, который позволяет «вкладывать» генераторы один в другой (осуществлять их композицию). Пример генератора с композицией:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

function\* generatePasswordCodes() {

// 0..9

yield\* generateSequence(48, 57);

// A..Z

yield\* generateSequence(65, 90);

// a..z

yield\* generateSequence(97, 122);

}

let str = '';

for(let code of generatePasswordCodes()) {

str += String.fromCharCode(code);

}

alert(str); // 0..9A..Za..z

Директива yield\* делегирует выполнение другому генератору. Этот термин означает, что yield\* gen перебирает генератор gen и прозрачно направляет его вывод наружу. Как если бы значения были сгенерированы внешним генератором.

Результат – такой же, как если встроить код из вложенных генераторов:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) yield i;

}

function\* generateAlphaNum() {

// yield\* generateSequence(48, 57);

for (let i = 48; i <= 57; i++) yield i;

// yield\* generateSequence(65, 90);

for (let i = 65; i <= 90; i++) yield i;

// yield\* generateSequence(97, 122);

for (let i = 97; i <= 122; i++) yield i;

}

let str = '';

for(let code of generateAlphaNum()) {

str += String.fromCharCode(code);

}

alert(str); // 0..9a..zA..Z

Композиция генераторов – естественный способ вставлять вывод одного генератора в поток другого. Она не использует дополнительную память для хранения промежуточных результатов.

Директива yield не только возвращает результат наружу, но и может передавать значение извне в генератор. Чтобы это сделать, нужно вызвать generator.next(arg) с аргументом. Этот аргумент становится результатом yield. Продемонстрируем это на примере:

function\* gen() {

// Передаём вопрос во внешний код и ожидаем ответа

let result = yield "2 + 2 = ?"; // (\*)

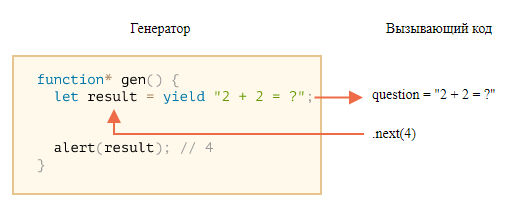
alert(result);

}

let generator = gen();

let question = generator.next().value; // <-- yield возвращает значение

generator.next(4); // --> передаём результат в генератор



1. Первый вызов generator.next() – всегда без аргумента, он начинает выполнение и возвращает результат первого yield "2+2=?". На этой точке генератор приостанавливает выполнение.
2. Затем, как показано на картинке выше, результат yield переходит во внешний код в переменную question.
3. При generator.next(4) выполнение генератора возобновляется, а 4 выходит из присваивания как результат: let result = 4.

Обратите внимание, что внешний код не обязан немедленно вызывать next(4). Ему может потребоваться время, генератор подождёт. Например:

// возобновить генератор через некоторое время

setTimeout(() => generator.next(4), 1000);

Как видно, в отличие от обычных функций, генератор может обмениваться результатами с вызывающим кодом, передавая значения в next/yield. Чтобы сделать происходящее более очевидным, вот ещё один пример с большим количеством вызовов:

function\* gen() {

let ask1 = yield "2 + 2 = ?";

alert(ask1); // 4

let ask2 = yield "3 \* 3 = ?"

alert(ask2); // 9

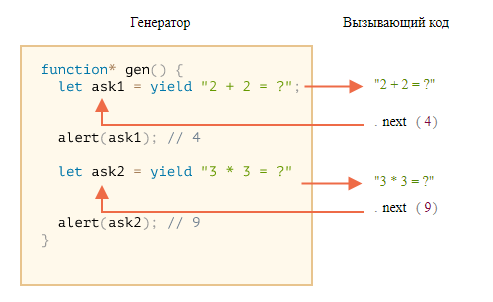
}

let generator = gen();

alert( generator.next().value ); // "2 + 2 = ?"

alert( generator.next(4).value ); // "3 \* 3 = ?"

alert( generator.next(9).done ); // true



1. Первый .next() начинает выполнение, доходит до первого yield.
2. Результат возвращается во внешний код.
3. Второй .next(4) передаёт 4 обратно в генератор как результат первого yield и возобновляет выполнение.
4. Выполнение доходит до второго yield, который станет результатом .next(4).
5. Третий next(9) передаёт 9 в генератор как результат второго yield и возобновляет выполнение, которое завершается окончанием функции, так что done: true.

Таким образом, каждый next(value) передаёт в генератор значение, которое становится результатом текущего yield, возобновляет выполнение и получает выражение из следующего yield.

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

Внешний код может передавать значение в генератор как результат yield. Но можно передать не только результат, но и инициировать (throw) ошибку. Это естественно, так как ошибка является своего рода результатом. Для того, чтобы передать ошибку в yield, нужно вызвать generator.throw(err). В этом случае, исключение err возникнет на строке с yield. Например, здесь yield "2 + 2 = ?" приведёт к ошибке:

function\* gen() {

try {

let result = yield "2 + 2 = ?"; // (1)

alert("Выполнение программы не дойдет до этой строки, потому что выше возникнет исключение");

} catch(e) {

alert(e); // покажет ошибку

}

}

let generator = gen();

let question = generator.next().value;

generator.throw(new Error("Ответ не найден в моей базе данных")); // (2)

Ошибка, которая проброшена в генератор на строке (2), приводит к исключению на строке (1) с yield. В примере выше try..catch перехватывает её и отображает. Если необходимо перехватывать её, то она, как и любое обычное исключение, будет передана из генератора во внешний код. Текущая строка вызывающего кода – это строка с generator.throw, отмечена (2). Таким образом, можно отловить её во внешнем коде, как здесь:

function\* generate() {

let result = yield "2 + 2 = ?"; // Ошибка в этой строке

}

let generator = generate();

let question = generator.next().value;

try {

generator.throw(new Error("Ответ не найден в моей базе данных"));

} catch(e) {

alert(e); // покажет ошибку

}

1. **Колбэки.**

Многие действия в JavaScript асинхронные. Например, рассмотрим функцию loadScript(src):

function loadScript(src) {

let script = document.createElement('script');

script.src = src;

document.head.append(script);

}

Эта функция загружает на страницу новый скрипт. Когда в тело документа добавится конструкция <script src="…">, браузер загрузит скрипт и выполнит его. Пример использования этой функции:

loadScript('/my/script.js');

Такие функции называют «асинхронными», потому что действие (загрузка скрипта) будет завершено не сейчас, а потом. Если после вызова loadScript(…) есть какой-то код, то он не будет ждать, пока скрипт загрузится.

loadScript('/my/script.js');

Доустпим надо использовать новый скрипт, как только он будет загружен. Например, он объявляет новую функцию, которую надо выполнить. Если просто вызовать эту функцию после loadScript(…), то ничего не выйдет:

loadScript('/my/script.js');

newFunction(); // такой функции не существует

Действительно, ведь у браузера не было времени загрузить скрипт. Сейчас функция loadScript никак не позволяет отследить момент загрузки. Скрипт загружается, а потом выполняется. Но нужно точно знать, когда это произойдет, чтобы использовать функции и переменные из этого скрипта. Давайте передадим функцию callback вторым аргументом в loadScript, чтобы вызвать ее, когда скрипт загрузится:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

Теперь, если надо вызвать функцию из скрипта, нужно делать это в колбэке:

loadScript('/my/script.js', function() {

newFunction();

...

});

Суть заключается в том, что вторым аргументом передается функция (обычно анонимная), которая выполняется по завершению действия. Возьмём для примера реальный скрипт с библиотекой функций:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(script);

document.head.append(script);

}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {

alert(`Здорово, скрипт ${script.src} загрузился`);

alert( \_ );

});

Такое написание называют асинхронным программированием с использованием колбэков. В функции, которые выполняют какие-либо асинхронные операции, передается аргумент callback – функция, которая будет вызвана по завершению асинхронного действия.

[**Колбэк в колбэке**](https://learn.javascript.ru/callbacks#kolbek-v-kolbeke)

Допустим, надо загрузить два скрипта один за другим: сначала первый, а за ним второй. Можно вызвать loadScript еще раз уже внутри колбэка:

loadScript('/my/script.js', function(script) {

alert(`Здорово, скрипт ${script.src} загрузился, загрузим еще один`);

loadScript('/my/script2.js', function(script) {

alert(`Здорово, второй скрипт загрузился`);

});

});

Когда внешняя функция loadScript выполнится, вызовется та, что внутри колбэка. Загрузим еще один скрипт:

loadScript('/my/script.js', function(script) {

loadScript('/my/script2.js', function(script) {

loadScript('/my/script3.js', function(script) {

// ...и так далее, пока все скрипты не будут загружены

});

})

});

Возникает необходимость каждое новое действие вызывать внутри колбэка. Этот вариант подойдет, когда необходимо одно-два действия, но для большего количества уже не удобно. Альтернативные подходы разберём позже.

**[Перехват ошибок](https://learn.javascript.ru/callbacks" \l "perehvat-oshibok)**

Допустим, что загрузить скрипт не удалось, тогда колбэк должен уметь реагировать на возможные проблемы. Ниже улучшенная версия loadScript, которая умеет отслеживать ошибки загрузки:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Не удалось загрузить скрипт ${src}`));

document.head.append(script);

}

Вызывается callback(null, script) в случае успешной загрузки и callback(error), если загрузить скрипт не удалось. Например:

loadScript('/my/script.js', function(error, script) {

if (error) {

// обрабатываем ошибку

} else {

// скрипт успешно загружен

}

});

Опять же, подход, который был использован в loadScript, также распространен и называется «колбэк с первым аргументом-ошибкой» («error-first callback»). Правила таковы:

1. Первый аргумент функции callback зарезервирован для ошибки. В этом случае вызов выглядит вот так: callback(err).
2. Второй и последующие аргументы — для результатов выполнения. В этом случае вызов выглядит вот так: callback(null, result1, result2…).

Одна и та же функция callback используется и для информирования об ошибке, и для передачи результатов.

[**Адская пирамида вызовов**](https://learn.javascript.ru/callbacks#adskaya-piramida-vyzovov)

На первый взгляд это рабочий способ написания асинхронного кода. Так и есть. Для одного или двух вложенных вызовов всё выглядит нормально. Но для нескольких асинхронных действий, которые нужно выполнить друг за другом, код выглядит вот так:

loadScript('1.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('2.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...

loadScript('3.js', function(error, script) {

if (error) {

handleError(error);

} else {

// ...и так далее, пока все скрипты не будут загружены (\*)

}

});

}

})

}

});

В примере выше:

1. Загружаем 1.js. Продолжаем, если нет ошибок.
2. Загружаем 2.js. Продолжаем, если нет ошибок.
3. Загружаем 3.js. Продолжаем, если нет ошибок. И так далее (\*).

Чем больше вложенных вызовов, тем код будет иметь всё большую вложенность, которую сложно поддерживать. Такой подход к написанию кода не приветствуется. Один из лучших способов – использовать промисы.

1. **Промис (Promise)**

Рассмотрим ситуацию, которая часто встречается в программировании. Есть «создающий» код, который делает что-то, что занимает время. Например, загружает данные по сети. Есть «потребляющий» код, который хочет получить результат «создающего» кода, когда он будет готов. Он может быть необходим более чем одной функции.

*Promise* (по англ. promise) – это специальный объект в JavaScript, который связывает «создающий» и «потребляющий» коды вместе. «Создающий» код может выполняться сколько потребуется, чтобы получить результат, а промис делает результат доступным для кода, который подписан на него, когда результат готов.

Синтаксис создания Promise:

let promise = new Promise(function(resolve, reject) {

});

Функция, переданная в конструкцию new Promise, называется исполнитель (executor). Когда Promise создаётся, она запускается автоматически. Она должна содержать «создающий» код, который когда-нибудь создаст результат. Её аргументы resolve и reject – это колбэки, которые предоставляет сам JavaScript. Наш код – только внутри исполнителя.

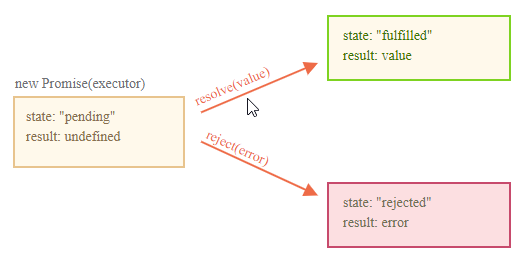
Когда он получает результат, сейчас или позже, он должен вызвать один из этих колбэков:

* resolve(value) – если работа завершилась успешно, с результатом value.
* reject(error) – если произошла ошибка, error – объект ошибки.

У объекта promise, возвращаемого конструктором new Promise, есть внутренние свойства:

* state («состояние») – вначале "pending" («ожидание»), потом меняется на "fulfilled" («выполнено успешно») при вызове resolve или на "rejected" («выполнено с ошибкой») при вызове reject.
* result («результат») – вначале undefined, далее изменяется на value при вызове resolve(value) или на error при вызове reject(error).

Так что исполнитель по итогу переводит promise одно из двух состояний:



Ниже пример конструктора Promise и простого исполнителя с кодом, дающим результат с задержкой (через setTimeout):

let promise = new Promise(function(resolve, reject) {

// функция выполнится автоматически, при вызове new Promise

// через 1 секунду сигнализировать, что задача выполнена с результатом "done"

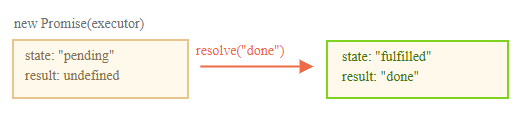
setTimeout(() => resolve("done"), 1000);

});

Можно наблюдать две вещи, запустив код выше:

1. Функция-исполнитель запускается сразу же при вызове new Promise.
2. Исполнитель получает два аргумента: resolve и reject – это функции, встроенные в JavaScript, поэтому разработчику не нужно их писать. Ему нужно лишь позаботиться, чтобы исполнитель вызвал одну из них по готовности.

Спустя одну секунду «обработки» исполнитель вызовет resolve("done"), чтобы передать результат:



Это был пример успешно выполненной задачи, в результате получен «успешно выполненный» промис.

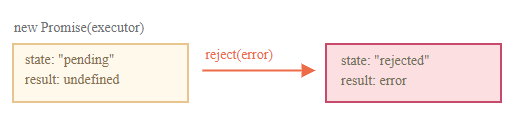
Рассмотрим пример, в котором исполнитель сообщит, что задача выполнена с ошибкой:

let promise = new Promise(function(resolve, reject) {

// спустя одну секунду будет сообщено, что задача выполнена с ошибкой

setTimeout(() => reject(new Error("Whoops!")), 1000);

});



Промис – и успешный, и отклонённый будем называть «завершённым», в отличие изначального промиса «в ожидании». Исполнитель должен вызвать что-то одно: resolve или reject. Состояние промиса может быть изменено только один раз. Все последующие вызовы resolve и reject будут проигнорированы:

let promise = new Promise(function(resolve, reject) {

resolve("done");

reject(new Error("…")); // игнорируется

setTimeout(() => resolve("…")); // игнорируется

});

Также заметим, что функция resolve/reject ожидает только один аргумент (или ни одного). Все дополнительные аргументы будут проигнорированы.

В случае, если что-то пошло не так, необходимо вызвать reject. Это можно сделать с аргументом любого типа (как и resolve), но рекомендуется использовать объект Error (или унаследованный от него).

Обычно исполнитель делает что-то асинхронное и после этого вызывает resolve/reject, то есть через какое-то время. Но это не обязательно, resolve или reject могут быть вызваны сразу:

let promise = new Promise(function(resolve, reject) {

// задача, не требующая времени

resolve(123); // мгновенно выдаст результат: 123

});

Свойства state и result – это внутренние свойства объекта Promise и к ним нет прямого доступа. Для обработки результата следует использовать методы .then/.catch/.finally.

Объект Promise служит связующим звеном между исполнителем («создающим» кодом) и функциями-потребителями, которые получат либо результат, либо ошибку. Функции-потребители могут быть зарегистрированы (подписаны) с помощью методов .then, .catch и .finally.

[**then**](https://learn.javascript.ru/promise-basics#then)

Наиболее важный и фундаментальный метод – .then. Синтаксис:

promise.then(

function(result) { /\* обработает успешное выполнение \*/ },

function(error) { /\* обработает ошибку \*/ }

);

Первый аргумент метода .then – функция, которая выполняется, когда промис переходит в состояние «выполнен успешно», и получает результат. Второй аргумент .then – функция, которая выполняется, когда промис переходит в состояние «выполнен с ошибкой», и получает ошибку. Например, реакция на успешно выполненный промис:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(

result => alert(result), // выведет "done!" через одну секунду

error => alert(error) // не будет запущена

);

Выполнилась первая функция. А в случае ошибки в промисе – выполнится вторая:

let promise = new Promise(function(resolve, reject) {

setTimeout(() => reject(new Error("Whoops!")), 1000);

});

promise.then(

result => alert(result), // не будет запущена

error => alert(error) // выведет "Error: Whoops!" спустя одну секунду

);

Если интересует только результат успешного выполнения задачи, то в then можно передать только одну функцию:

let promise = new Promise(resolve => {

setTimeout(() => resolve("done!"), 1000);

});

promise.then(alert); // выведет "done!" спустя одну секунду

[**catch**](https://learn.javascript.ru/promise-basics#catch)

Если надо обработать ошибку, то можно использовать null в качестве первого аргумента: .then(null, errorHandlingFunction). Или можно воспользоваться методом .catch(errorHandlingFunction), который сделает тоже самое:

let promise = new Promise((resolve, reject) => {

setTimeout(() => reject(new Error("Ошибка!")), 1000);

});

promise.catch(alert); // выведет "Error: Ошибка!" спустя одну секунду

Вызов .catch(f) – это сокращенный, «укороченный» вариант .then(null, f).

**[finally](https://learn.javascript.ru/promise-basics" \l "finally)**

По аналогии с блоком finally из обычного try {...} catch {...}, у промисов также есть метод finally. Вызов .finally(f) похож на .then(f, f), в том смысле, что f выполнится в любом случае, когда промис завершится: успешно или с ошибкой. Например:

new Promise((resolve, reject) => {

/\* сделать что-то, что займет время, и после вызвать resolve/reject \*/

})

// выполнится, когда промис завершится, независимо от того, успешно или нет

.finally(() => остановить индикатор загрузки)

.then(result => показать результат, err => показать ошибку)

Но это не совсем псевдоним then(f, f), как можно было подумать. Существует несколько важных отличий:

1. Обработчик, вызываемый из finally, не имеет аргументов. В finally не известно, как был завершён промис. И это нормально, потому что обычно задача состоит в том, чтобы выполнить «общие» завершающие процедуры.
2. Обработчик finally «пропускает» результат или ошибку дальше, к последующим обработчикам.

Например, здесь результат проходит через finally к then:

new Promise((resolve, reject) => {

setTimeout(() => resolve("result"), 2000)

})

.finally(() => alert("Промис завершён"))

.then(result => alert(result)); // <-- .then обработает результат

А здесь ошибка из промиса проходит через finally к catch:

new Promise((resolve, reject) => {

throw new Error("error");

})

.finally(() => alert("Промис завершён"))

.catch(err => alert(err)); // <-- .catch обработает объект ошибки

Это очень удобно, потому что finaly не предназначен для обработки результата промиса. Так что он просто пропускает его через себя дальше.

1. Последнее, но не менее значимое: вызов .finally(f) удобнее, чем.then(f, f) – не надо дублировать функции f.

Если промис в состоянии ожидания, обработчики в .then/catch/finally будут ждать его. Однако, если промис уже завершён, то обработчики выполнятся сразу:

// мгновенный перевод промиса в состояние "успешно завершён"

let promise = new Promise(resolve => resolve("готово!"));

promise.then(alert); // готово!

Рассмотрим несколько практических примеров того, как промисы могут облегчить написание асинхронного кода.

Ранее рассматривалась функция loadScript для загрузки скрипта. Реализация этой функции с использованием колбэков выглядит так:

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

}

Перепишем её, используя Promise. Новой функции loadScript будет не нужен аргумент callback. Вместо этого она будет создавать и возвращать объект Promise, который будет переходить в состояние «успешно завершён», когда загрузка закончится. Внешний код может добавлять обработчики («подписчиков»), используя .then:

function loadScript(src) {

return new Promise(function(resolve, reject) {

let script = document.createElement('script');

script.src = src;

script.onload = () => resolve(script);

script.onerror = () => reject(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

});

}

Применение:

let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");

promise.then(

script => alert(`${script.src} загружен!`),

error => alert(`Ошибка: ${error.message}`)

);

promise.then(script => alert('Ещё один обработчик...'));

Сразу заметно несколько преимуществ перед подходом с использованием колбэков:

| **Промисы** | **Колбэки** |
| --- | --- |
| Промисы позволяют выполнять действия в естественном порядке: сперва запустить loadScript(script), и затем (.then) выполнить действия с результатом. | Должна быть функция callback на момент вызова loadScript(script, callback). Другими словами, нужно знать что делать с результатом до того, как вызовется loadScript. |
| Можно вызывать .then у Promise столько раз, сколько надо, каждый раз добавляя новую функцию-подписчика. | Колбэк может быть только один. |

1. **Промисификация**

*Промисификация* – подразумевает преобразование функции, которая принимает колбэк таким образом, чтобы она вместо этого возвращала промис.

Такие преобразования часто необходимы в реальной жизни, так как многие функции и библиотеки основаны на колбэках, а использование промисов более удобно, поэтому есть смысл «промисифицировать» их. Например, ранее рассматриваемая функция loadScript(src, callback):

function loadScript(src, callback) {

let script = document.createElement('script');

script.src = src;

script.onload = () => callback(null, script);

script.onerror = () => callback(new Error(`Ошибка загрузки скрипта ${src}`));

document.head.append(script);

}

// использование:

// loadScript('path/script.js', (err, script) => {...})

Промисифицируем её. Новая функция loadScriptPromise(src) будет делать то же самое, но будет принимать только src (не callback) и возвращать промис.

let loadScriptPromise = function(src) {

return new Promise((resolve, reject) => {

loadScript(src, (err, script) => {

if (err) reject(err)

else resolve(script);

});

})

}

// использование:

// loadScriptPromise('path/script.js').then(...)

Теперь loadScriptPromise хорошо вписывается в код, основанный на промисах. Как видно, она передаёт всю работу исходной функции loadScript, предоставляя ей колбэк, по вызову которого происходит resolve/reject промиса.

На практике, как правило, есть необходимость промисифицировать не одну функцию, поэтому имеет смысл сделать для этого специальную «функцию-помощник» – promisify(f). Она будет принимать функцию для промисификации f и возвращать функцию-обёртку. Эта функция-обёртка делает то же самое, что и код выше: возвращает промис и передаёт вызов оригинальной f, отслеживая результат в своём колбэке:

function promisify(f) {

return function (...args) {

return new Promise((resolve, reject) => {

function callback(err, result) {

if (err) {

return reject(err);

} else {

resolve(result);

}

}

args.push(callback);

f.call(this, ...args);

});

};

};

// использование:

let loadScriptPromise = promisify(loadScript);

loadScriptPromise(...).then(...);

Здесь предполагается, что исходная функция ожидает колбэк с двумя аргументами (err, result). Но возможна ситуация, когда исходная f ожидает колбэк с большим количеством аргументов callback(err, res1, res2, ...). Ниже описана улучшенная функция promisify: при вызове promisify(f, true) результатом промиса будет массив результатов [res1, res2, ...]:

function promisify(f, manyArgs = false) {

return function (...args) {

return new Promise((resolve, reject) => {

function callback(err, ...results) {

if (err) {

return reject(err);

} else {

resolve(manyArgs ? results : results[0]);

}

}

args.push(callback);

f.call(this, ...args);

});

};

};

// использование:

f = promisify(f, true);

f(...).then(arrayOfResults => ..., err => ...)

Помните, промис может иметь только один результат, но колбэк технически может вызываться сколько угодно раз. Поэтому промисификация используется для функций, которые вызывают колбэк только один раз. Последующие вызовы колбэка будут проигнорированы.

1. **Async/await**

Существует специальный синтаксис для работы с промисами, который называется «async/await». Он прост для понимания и использования.

Ключевое слово *async* ставится перед функцией:

async function f() {

return 1;

}

У слова async один простой смысл: эта функция всегда возвращает промис. Значения других типов оборачиваются в завершившийся успешно промис автоматически. Например, эта функция возвратит выполненый промис с результатом 1:

async function f() {

return 1;

}

f().then(alert); // 1

Можно и явно вернуть промис, результат будет одинаковым:

async function f() {

return Promise.resolve(1);

}

f().then(alert); // 1

Так что ключевое слово async перед функцией гарантирует, что эта функция в любом случае вернёт промис.

Ключевое слово *await* можно использовать только внутри async-функций. Синтаксис:

// работает только внутри async–функций

let value = await promise;

Ключевое слово await заставит интерпретатор JavaScript ждать до тех пор, пока промис справа от await не выполнится. После чего оно вернёт его результат, и выполнение кода продолжится. В этом примере промис успешно выполнится через 1 секунду:

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("готово!"), 1000)

});

let result = await promise; // (\*)

alert(result); // "готово!"

}

f();

В данном примере выполнение функции остановится на строке (\*) до тех пор, пока промис не выполнится. Это произойдёт через секунду после запуска функции. После чего в переменную result будет записан результат выполнения промиса, и браузер отобразит alert-окно «готово!».

Обратите внимание, хотя await и заставляет JavaScript дожидаться выполнения промиса, это не отнимает ресурсов процессора. Пока промис не выполнится, JS-движок может заниматься другими задачами: выполнять прочие скрипты, обрабатывать события и т.п. По сути, это просто «синтаксический сахар» для получения результата промиса, более наглядный, чем promise.then.

Если использовать await внутри функции, объявленной без async, получим синтаксическую ошибку:

function f() {

let promise = Promise.resolve(1);

let result = await promise; // SyntaxError

}

Ошибки не будет, если указать ключевое слово async перед объявлением функции. Как было сказано раньше, await можно использовать только внутри async–функций.

await нельзя использовать на верхнем уровне вложенности (вне тела функции):

// SyntaxError на верхнем уровне вложенности

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

Можно обернуть этот код в анонимную async-функцию, тогда всё заработает:

(async () => {

let response = await fetch('/article/promise-chaining/user.json');

let user = await response.json();

...

})();

Как и promise.then, await позволяет работать с промис-совместимыми объектами. Идея в том, что если у объекта можно вызвать метод then, этого достаточно, чтобы использовать его с await. В примере ниже, экземпляры класса Thenable будут работать вместе с await:

class Thenable {

constructor(num) {

this.num = num;

}

then(resolve, reject) {

alert(resolve);

// выполнить resolve со значением this.num \* 2 через 1000мс

setTimeout(() => resolve(this.num \* 2), 1000); // (\*)

}

};

async function f() {

// код будет ждать 1 секунду,

// после чего значение result станет равным 2

let result = await new Thenable(1);

alert(result);

}

f();

Когда await получает объект с .then, не являющийся промисом, JavaScript автоматически запускает этот метод, передавая ему аргументы – встроенные функции resolve и reject. Затем await приостановит дальнейшее выполнение кода, пока любая из этих функций не будет вызвана (в примере, это строка (\*)). После чего выполнение кода продолжится с результатом resolve или reject соответственно.

Для объявления асинхронного метода достаточно написать async перед именем:

class Waiter {

async wait() {

return await Promise.resolve(1);

}

}

new Waiter()

.wait()

.then(alert); // 1

Как и с асинхронными функциями, такой метод гарантированно возвращает промис и в его теле можно использовать await.

Когда промис завершается успешно, await promise возвращает результат. Когда завершается с ошибкой – будет выброшено исключение. Как если бы на этом месте находилось выражение throw. Такой код:

async function f() {

await Promise.reject(new Error("Упс!"));

}

Делает тоже самое, что и такой:

async function f() {

throw new Error("Упс!");

}

Но есть отличие, на практике промис может завершиться с ошибкой не сразу, а через некоторое время. В этом случае будет задержка, а затем await выбросит исключение. Такие ошибки можно ловить, используя try..catch, так же как с обычным throw:

async function f() {

try {

let response = await fetch('http://no-such-url');

} catch(err) {

alert(err); // TypeError: failed to fetch

}

}

f();

В случае ошибки выполнение try прерывается и управление переходит в начало блока catch. Блоком try можно обернуть несколько строк:

async function f() {

try {

let response = await fetch('/no-user-here');

let user = await response.json();

} catch(err) {

// перехватит любую ошибку в блоке try: и в fetch, и в response.json

alert(err);

}

}

f();

Если нет try..catch, асинхронная функция будет возвращать завершившийся с ошибкой промис (в состоянии rejected). В этом случае можно использовать метод .catch промиса, чтобы обработать ошибку:

async function f() {

let response = await fetch('http://no-such-url');

}

// f() вернёт промис в состоянии rejected

f().catch(alert); // TypeError: failed to fetch // (\*)

Если не добавить .catch, то будет сгенерирована ошибка «Uncaught promise error» и информация об этом будет выведена в консоль.

При работе с async/await, .then используется нечасто, так как await автоматически ожидает завершения выполнения промиса. В этом случае, обычно (но не всегда) гораздо удобнее перехватывать ошибки используя try..catch, нежели чем .catch. Но на верхнем уровне вложенности (вне async–функций) await использовать нельзя, поэтому .then/catchдля обработки финального результата или ошибок – обычная практика. Так сделано в строке (\*) в примере выше.

Когда необходимо подождать несколько промисов одновременно, можно обернуть их в Promise.all, и затем await:

// await будет ждать массив с результатами выполнения всех промисов

let results = await Promise.all([

fetch(url1),

fetch(url2),

...

]);

В случае ошибки она будет передаваться как обычно: от завершившегося с ошибкой промиса к Promise.all. А после будет сгенерировано исключение, которое можно отловить, обернув выражение в try..catch.

Метод allSettled ведет себя иначе. Он выполняется всякий раз, когда все промисы заканчиваются, то есть выполнились успешно или выполнились с ошибкой. Он возвращает массив, который содержит как статус промиса, так и значение (или ошибку). Таким образом, allSettled никогда не отклоняется. Он либо в состоянии ожидания, либо выполнился успешно.

const promiseArray = [

Promise.resolve(200),

Promise.reject(""),

Promise.reject(new Error("Error")),

];

Promise.allSettled(promiseArray).then((result) => {

console.log("All Promises Settled", result);

});

// All Promises Settled

// [

// {status: "fulfilled", value: 200},

// {status: "rejected", reason: ""},

// {status: "rejected", reason: Error: Error at <anonymous>:4:18}

// ]

1. **Управление памятью в JavaScript.**

Управление памятью в JavaScript выполняется автоматически и незаметно. Создаются примитивы, объекты, функции. Всё это занимает память.

Основной концепцией управления памятью в JavaScript является принцип достижимости. «Достижимые» значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

1. Существует базовое множество достижимых значений, которые не могут быть удалены. Например:

* Локальные переменные и параметры текущей функции.
* Переменные и параметры других функций в текущей цепочке вложенных вызовов.
* Глобальные переменные.
* Некоторые другие внутренние значения.

Эти значения будем называть корнями.

1. Любое другое значение считается достижимым, если оно доступно из корня по ссылке или по цепочке ссылок.

Например, если в локальной переменной есть объект, и он имеет свойство, в котором хранится ссылка на другой объект, то этот объект считается достижимым. И те, на которые он ссылается, тоже достижимы. Детальные примеры на эту тему скоро последуют.

В интерпретаторе JavaScript есть фоновый процесс, который называется [сборщик мусора](https://ru.wikipedia.org/wiki/%D0%A1%D0%B1%D0%BE%D1%80%D0%BA%D0%B0_%D0%BC%D1%83%D1%81%D0%BE%D1%80%D0%B0). Он следит за всеми объектами и удаляет те, которые стали недостижимы. Простой пример:

// в user находится ссылка на объект

let user = {

name: "John"

};

Здесь стрелочка обозначает ссылку на объект. Глобальная переменная user ссылается на объект {name: "John"}. В свойстве "name" объекта John хранится примитив, поэтому оно нарисовано внутри объекта.

Если перезаписать значение user, то ссылка потеряется:

user = null;

Теперь объект John становится недостижимым. К нему нет доступа, на него нет ссылок. Сборщик мусора удалит эти данные и освободит память.

Представим, что ссылка из user  была скопирована в admin:

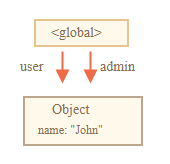
// в user находится ссылка на объект

let user = {

name: "John"

};

let admin = user;



Теперь, если перезаписать значение user:

user = null;

то объект John всё ещё достижим через глобальную переменную admin, поэтому он находится в памяти. Если также перезаписать admin, то John будет удалён.

Рассмотрим [взаимосвязанные](https://learn.javascript.ru/garbage-collection#vzaimosvyazannye-obekty) объекты:

function marry(man, woman) {

woman.husband = man;

man.wife = woman;

return {

father: man,

mother: woman

}

}

let family = marry({

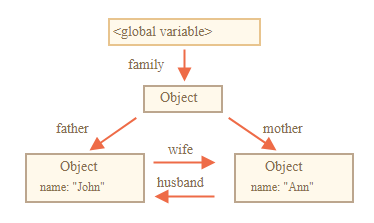
name: "John"

}, {

name: "Ann"

});

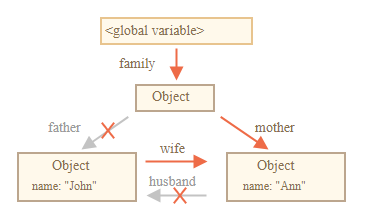
В функции marry два объекта получают ссылки друг на друга, и возвращают новый объект, содержащий ссылки на два предыдущих. В результате получается такая структура памяти:



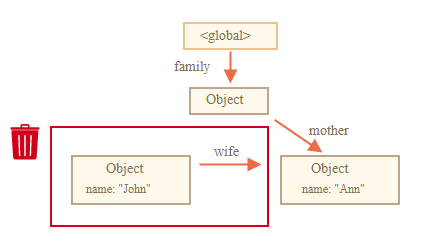
На данный момент все объекты достижимы. Удалим две ссылки:

delete family.father;

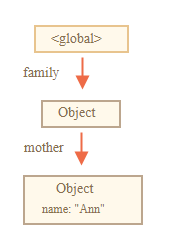
delete family.mother.husband;



Недостаточно удалить только одну из этих ссылок, потому что все объекты останутся достижимыми. Но если удалить обе, то будет видно, что у объекта John больше нет входящих ссылок:



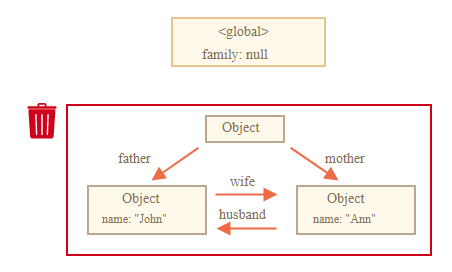
Исходящие ссылки не имеют значения. Только входящие ссылки могут сделать объект достижимым. Объект John теперь недостижим и будет удалён из памяти со всеми своими данными, которые также стали недоступны. После сборки мусора:



Вполне возможна ситуация, при которой несколько связанных объектов могут стать недостижимым и удалится из памяти. Перезапишем объект family из примера выше:

family = null;

Структура в памяти теперь станет такой:

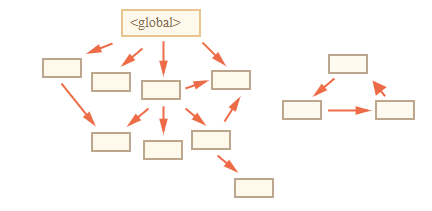


Этот пример демонстрирует, насколько важна концепция достижимости. Объекты John и Ann всё ещё связаны, оба имеют входящие ссылки, но этого недостаточно. У объекта family больше нет ссылки от корня, поэтому все эти объекты становятся недостижимым и будут удалены.

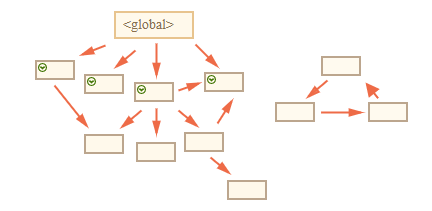
Основной алгоритм сборки мусора – «алгоритм пометок» (англ. «mark-and-sweep»). Согласно этому алгоритму, сборщик мусора регулярно выполняет следующие шаги:

* Сборщик мусора «помечает» (запоминает) все корневые объекты.
* Затем он идёт по их ссылкам и помечает все найденные объекты.
* Затем он идёт по ссылкам помеченных объектов и помечает объекты, на которые есть ссылка от них. Все объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды.
* И так далее, пока не будут посещены все ссылки (достижимые от корней).
* Все непомеченные объекты удаляются.

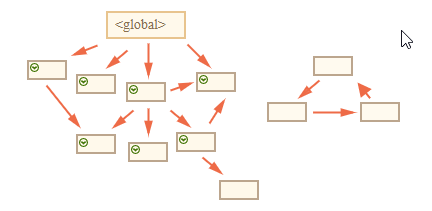
Например, пусть структура объектов выглядит так:



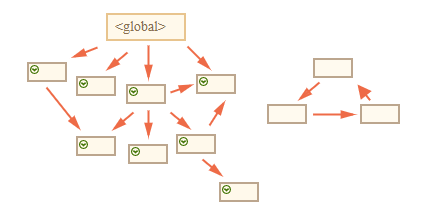
Явно видны недостижимые объекты справа. Рассмотрим, как будет работать «алгоритм пометок» сборщика мусора. На первом шаге помечаются корни:



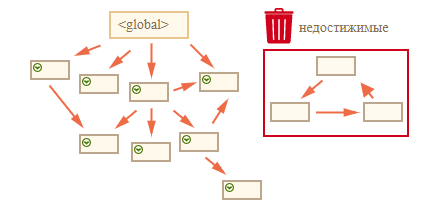
Затем помечаются объекты по их ссылкам:



Затем объекты по их ссылкам и так далее, пока это вообще возможно:



Теперь объекты, до которых не удалось дойти от корней, считаются недостижимыми и будут удалены:



Это и есть принцип работы сборки мусора.

Интерпретаторы JavaScript применяют множество оптимизаций, чтобы сборка мусора работала быстрее и не влияла на производительность.

Вот некоторые из оптимизаций:

* Сборка по поколениям (Generational collection) – объекты делятся на «новые» и «старые». Многие объекты появляются, выполняют свою задачу и быстро умирают, их можно удалять более агрессивно. Те, которые живут достаточно долго, становятся «старыми» и проверяются реже.
* Инкрементальная сборка (Incremental collection) – если объектов много, то обход всех ссылок и пометка достижимых объектов может занять значительное время и привести к видимым задержкам выполнения скрипта. Поэтому интерпретатор пытается организовать сборку мусора поэтапно. Этапы выполняются по отдельности один за другим. Это требует дополнительного учёта для отслеживания изменений между этапами, но зато теперь у нас есть много крошечных задержек вместо одной большой.
* Сборка в свободное время (Idle-time collection) – чтобы уменьшить возможное влияние на производительность, сборщик мусора старается работать только во время простоя процессора.

Существуют и другие способы оптимизации и разновидности алгоритмов сборки мусора.

Тема 4. Методы объектов и контекст вызова

1. Методы объектов, this.
2. Итерируемые объекты.
3. Преобразование объектов.
4. Создание объектов через "new".
5. Флаги и дескрипторы свойств.
6. Геттеры и сеттеры.
7. Декораторы и переадресация вызова.
8. Привязка контекста.
9. setTimeout and setInterval.
10. Асинхронные итераторы и генераторы.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. **Методы объектов, this.**

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

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. Для начала добавим в объект user функцию приветствия:

let user = {

name: "Джон",

age: 30

};

user.sayHi = function() {

alert("Привет!");

};

user.sayHi(); // Привет!

Здесь просто использовано Function Expression, чтобы создать функцию для приветствия, и присвоить её свойству user.sayHi объекта user. Затем она вызывается.

Функцию, которая является свойством объекта, называют *методом* этого объекта. Таким образом создан метод sayHi объекта user. Конечно, можно было бы заранее объявить функцию и использовать её в качестве метода, например так:

let user = {

// ...

};

function sayHi() {

alert("Привет!");

};

user.sayHi = sayHi;

user.sayHi(); // Привет!

Существует более короткий синтаксис для методов в литерале объекта:

user = {

sayHi() {

alert("Привет");

}

};

Т.е., можно пропустить ключевое слово "function" и просто написать sayHi(). Нужно отметить, что эти две записи не полностью эквивалентны. Есть тонкие различия, связанные с наследованием объектов, но на данном этапе изучения это неважно. В большинстве случаев сокращённый синтаксис предпочтителен.

В JavaScript иногда необходимо сначала проверить, существует ли объект, а затем попытаться получить одно из его свойств, например, так:

const car = null;

const color = car && car.color;

Даже если car имеет значение null, у нас нет ошибок, а color присваивается значение null. Используя оператор &&, можно пройти несколько уровней вложенностей:

const car = {}

const colorName = car && car.color && car.color.name;

Оператор *опциональной последовательности* позволяет сделать код короче:

const car = {}

const color = car?.color;

const colorName = car?.color?.name;

Если car имеет значение null или undefined, результат будет undefined.

Как правило, методу объекта необходим доступ к информации, которая хранится в объекте, чтобы выполнить с ней какие-либо действия (в соответствии с назначением метода). Например, коду внутри user.sayHi() может понадобиться имя пользователя, которое хранится в объекте user. Для доступа к информации внутри объекта метод может использовать ключевое слово this. Значение this – это объект «перед точкой», который использовался для вызова метода. Например:

let user = {

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);

}

};

Но такой код будет ненадёжным. Если скопировать ссылку на объект user в другую переменную, например, admin = user, и перезаписать 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 );

}

};

let admin = user;

user = null;

admin.sayHi(); // Ошибка!

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

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Оно может использоваться в любой функции. В этом коде нет синтаксической ошибки:

function sayHi() {

alert( this.name );

}

Значение this вычисляется во время выполнения кода и зависит от контекста. Например, здесь одна и та же функция назначена двум разным объектам и имеет различное значение «this» при вызовах:

let user = { name: "Джон" };

let admin = { name: "Админ" };

function sayHi() {

alert( this.name );

}

user.f = sayHi;

admin.f = sayHi;

user.f(); // Джон (this == user)

admin.f(); // Админ (this == admin)

admin['f'](); // Админ

Правило простое: при вызове obj.f() значение this внутри f равно obj. Так что, в приведённом примере это user или admin.

Можно вызвать функцию вовсе без использования объекта:

function sayHi() {

alert(this);

}

sayHi(); // undefined

В строгом режиме ("use strict") в таком коде значением this будет являться undefined. Если попытаться получить доступ к name, используя this.name – это вызовет ошибку.

В нестрогом режиме значением this в таком случае будет глобальный объект. Обычно подобный вызов является ошибкой программирования. Если внутри функции используется this, тогда ожидается, что она будет вызываться в контексте какого-либо объекта.

В других языках программирования this фиксировано – методы, определённые внутри объекта, всегда сохраняют в качестве значения this ссылку на свой объект (в котором был определён метод). В JavaScript this является «свободным», его значение вычисляется в момент вызова метода и не зависит от того, где этот метод был объявлен, а зависит от того, какой объект вызывает метод (какой объект стоит «перед точкой»).

Такая особенность вычисления this в момент исполнения имеет как свои плюсы, так и минусы. С одной стороны, функция может быть повторно использована в качестве метода у различных объектов (что повышает гибкость). С другой стороны, большая гибкость увеличивает вероятность ошибок.

Некоторые хитрые способы вызова метода приводят к потере значения this, например:

let user = {

name: "Джон",

hi() { alert(this.name); },

bye() { alert("Пока"); }

};

user.hi();

(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!

В последней строчке кода используется условный оператор ?, который определяет, какой будет вызван метод (user.hi или user.bye) в зависимости от выполнения условия. В данном случае будет выбран user.hi. Затем метод тут же вызывается с помощью скобок (). Но вызов не работает как положено: при вызове будет ошибка, потому что значением "this" внутри функции становится undefined (полагаем, что у нас строгий режим). Так работает (доступ к методу объекта через точку):

user.hi();

Так уже не работает (вызываемый метод вычисляется):

(user.name == "Джон" ? user.hi : user.bye)(); // Ошибка!

Чтобы понять, почему так происходит, разберёмся, как работает вызов методов (obj.method()). В выражении obj.method() сначала оператор точка '.' возвращает свойство объекта – его метод (obj.method). Затем скобки () вызывают этот метод (исполняется код метода). Если поместить эти операции в отдельные строки, то значение this, естественно, будет потеряно:

let user = {

name: "Джон",

hi() { alert(this.name); }

}

let hi = user.hi;

hi(); // Ошибка

Здесь hi = user.hi сохраняет функцию в переменной, и далее в последней строке она вызывается полностью сама по себе, без объекта, так что нет this.

Для работы вызовов типа user.hi(), JavaScript использует трюк – точка '.' возвращает не саму функцию, а специальное значение «ссылочного типа», называемого [Reference Type](https://tc39.github.io/ecma262/#sec-reference-specification-type). Этот ссылочный тип является внутренним типом. Нельзя явно использовать его, но он используется внутри языка. Значение ссылочного типа – это «триплет»: комбинация из трех значений (base, name, strict), где:

* base – это объект.
* name – это имя свойства объекта.
* strict – это режим исполнения. Является true, если действует строгий режим (use strict).

Результатом доступа к свойству user.hi является не функция, а значение ссылочного типа. Для user.hi в строгом режиме оно будет таким:

// значение ссылочного типа (Reference Type)

(user, "hi", true)

Когда скобки () применяются к значению ссылочного типа (происходит вызов), то они получают полную информацию об объекте и его методе, и могут поставить правильный this (=user в данном случае, по base).

Ссылочный тип – исключительно внутренний, промежуточный, используемый, чтобы передать информацию от точки .до вызывающих скобок (). При любой другой операции, например, присваивании hi = user.hi, ссылочный тип заменяется на собственно значение user.hi (функцию), и дальше работа уже идёт только с ней. Поэтому дальнейший вызов происходит уже без this. Таким образом, значение this передаётся правильно, только если функция вызывается напрямую с использованием синтаксиса точки obj.method() или квадратных скобок obj['method']() (они делают то же самое).

Стрелочные функции особенные: у них нет своего «собственного» this. Если использовать this внутри стрелочной функции, то его значение берётся из внешней обычной функции. Например, здесь arrow() использует значение this из внешнего метода user.sayHi():

let user = {

firstName: "Вася",

sayHi() {

let arrow = () => alert(this.firstName);

arrow();

}

};

user.sayHi(); // Вася

Это является особенностью стрелочных функций. Они полезны, когда нет необходимости иметь отдельное значение this, а надо брать его из внешнего контекста.

1. **Итерируемые объекты**.

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

Конечно же, сами массивы являются перебираемыми объектами. Но есть и много других встроенных перебираемыми объектов, например, строки.

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

let range = {

from: 1,

to: 5

};

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

1. Когда цикл for..of запускается, он вызывает этот метод один раз (или выдаёт ошибку, если метод не найден). Этот метод должен вернуть итератор – объект с методом next.
2. Дальше, for..of работает только с этим возвращённым объектом.
3. Когда for..of хочет получить следующее значение, он вызывает метод next() этого объекта.
4. Результат вызова next() должен иметь вид {done: Boolean, value: any}, где done=true означает, что итерация закончена, в противном случае value содержит очередное значение.

Вот полная реализация range с пояснениями:

let range = {

from: 1,

to: 5

};

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

range[Symbol.iterator] = function() {

// она возвращает объект итератора:

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

return {

current: this.from,

last: this.to,

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

next() {

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

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

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

} else {

return { done: true };

}

}

};

};

// теперь работает

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Обратите внимание на ключевую особенность итераторов: разделение ответственности.

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

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

let range = {

from: 1,

to: 5,

[Symbol.iterator]() {

this.current = this.from;

return this;

},

next() {

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

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

} else {

return { done: true };

}

}

};

for (let num of range) {

alert(num); // 1, затем 2, 3, 4, 5

}

Теперь range[Symbol.iterator]() возвращает сам объект range: у него есть необходимый метод next() и он запоминает текущее состояние итерации в this.current. Недостаток такого подхода в том, что теперь нельзя использовать этот объект в двух параллельных циклах for..of: у них будет общее текущее состояние итерации, потому что теперь существует лишь один итератор – сам объект. Но необходимость в двух циклах for..of, выполняемых одновременно, возникает редко, даже при наличии асинхронных операций.

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

Метод next не имеет ограничений, он может возвращать все новые и новые значения, это нормально. Конечно же, цикл for..of с таким итерируемым объектом будет бесконечным. Но всегда можно прервать его, используя break.

[**Использование генераторов для перебираемых объектов**](https://learn.javascript.ru/generators#ispolzovanie-generatorov-dlya-perebiraemyh-obektov)

Ранее рассматривался перебираемый объект range, который возвращает значения from..to:

let range = {

from: 1,

to: 5,

[Symbol.iterator]() {

return {

current: this.from,

last: this.to,

next() {

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

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

} else {

return { done: true };

}

}

};

}

};

alert([...range]); // 1,2,3,4,5

Можно использовать функцию-генератор для итерации, указав её в Symbol.iterator. Вот тот же range, но с гораздо более компактным итератором:

let range = {

from: 1,

to: 5,

\*[Symbol.iterator]() { // краткая запись для [Symbol.iterator]: function\*()

for(let value = this.from; value <= this.to; value++) {

yield value;

}

}

};

alert( [...range] ); // 1,2,3,4,5

Это работает, потому что range[Symbol.iterator]() теперь возвращает генератор, и его методы – в точности то, что ожидает for..of: у него есть метод .next(), который возвращает значения в виде {value: ..., done: true/false}.

Генераторы были добавлены в язык JavaScript, в частности, с целью упростить создание перебираемых объектов. Вариант с генератором намного короче, чем исходный вариант перебираемого range, и сохраняет те же функциональные возможности.

Среди встроенных перебираемых объектов наиболее широко используются массивы и строки. Для строки for..of перебирает символы:

for (let char of "test") {

alert( char ); // t, затем e, затем s, затем t

}

И он работает корректно даже с суррогатными парами:

let str = '𝒳😂';

for (let char of str) {

alert( char ); // 𝒳, а затем 😂

}

[**Явный вызов итератора**](https://learn.javascript.ru/iterable#yavnyy-vyzov-iteratora)

Чтобы понять устройство итераторов чуть глубже, давайте посмотрим как их использовать явно. Переберем строку в точности так же, как цикл for..of, но вручную, прямыми вызовами. Код ниже получает строковой итератор и получает из него значения:

let str = "Hello";

let iterator = str[Symbol.iterator]();

while (true) {

let result = iterator.next();

if (result.done) break;

alert(result.value);

}

Такое редко бывает необходимо, но это даёт нам больше контроля над процессом, чем for..of. Например, можно разбить процесс итерации на части: перебрать немного элементов, затем остановиться, сделать что-то ещё, и потом продолжить.

[**Итерируемые объекты и псевдомассивы**](https://learn.javascript.ru/iterable#array-like)

Есть два термина, которые очень похожи, но в то же время сильно различаются.

*Итерируемые объекты*  – это объекты, которые реализуют метод Symbol.iterator, как было описано выше.

*Псевдомассивы* – это объекты, у которых есть индексы и свойство length, то есть они выглядят как массивы.

При использовании JavaScript в браузере или других окружениях, можно встретить объекты, которые являются итерируемыми или псевдомассивами или и тем и другим. Например, строки итерируемы (для них работает for..of) и являются псевдомасссивами (они индексированы и есть length).

Но итерируемый объект может не быть псевдомассивом. И наоборот: псевдомассив может не быть итерируемым. Например, объект range из примера выше – итерируемый, но не является псевдомассивом, потому что у него нет индексированных свойств и length.

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

let arrayLike = {

0: "Hello",

1: "World",

length: 2

};

// Ошибка (отсутствует Symbol.iterator)

for (let item of arrayLike) {}

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

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

let arrayLike = {

0: "Hello",

1: "World",

length: 2

};

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

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

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

let arr = Array.from(range);

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

Полный синтаксис Array.from позволяет указать необязательную «трансформирующую» функцию:

Array.from(obj[, mapFn, thisArg])

Необязательный второй аргумент может быть функцией, которая будет применена к каждому элементу перед добавлением в массив, а thisArg позволяет установить this для этой функции. Например:

// range взят из примера выше

let arr = Array.from(range, num => num \* num);

alert(arr); // 1,4,9,16,25

Здесь используется Array.from, чтобы превратить строку в массив её элементов:

let str = '𝒳😂';

let chars = Array.from(str);

alert(chars[0]); // 𝒳

alert(chars[1]); // 😂

alert(chars.length); // 2

В отличие от str.split, этот метод в работе опирается на итеративный характер строки и поэтому, совсем как for..of, он корректно работает с суррогатными парами. Технически, это то же самое, что и код ниже, но гораздо короче:

let str = '𝒳😂';

let chars = [];

for (let char of str) {

chars.push(char);

}

alert(chars);

Мы можем даже создать slice, который поддерживает суррогатные пары:

function slice(str, start, end) {

return Array.from(str).slice(start, end).join('');

}

let str = '𝒳😂𩷶';

alert( slice(str, 1, 3) ); // 😂𩷶

// встроенный метод не поддерживает суррогатные пары

alert( str.slice(1, 3) ); // мусор

1. **Преобразование объектов.**

Если сложить два объекта obj1 + obj2, вычесть один из другого obj1 - obj2 или вывести их на экран, воспользовавшись alert(obj), то объекты сначала автоматически преобразуются в примитивы, а затем выполнится операция.

Правила преобразования объектов:

1. Все объекты в логическом контексте являются true. Существуют лишь их численные и строковые преобразования.
2. Численные преобразования происходят, когда осуществляется вычитание объектов или выполняются другие математические операции. Например, объекты Date могут вычитаться и результатом date1 - date2 будет временной отрезок между двумя датами.
3. Что касается строковых преобразований – они обычно происходят, при выводе объекта по типу alert(obj) и в подобных случаях.

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

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

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

alert(obj);

// объект в качества имени свойства

anotherObj[obj] = 123;

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

// явное преобразование

let num = Number(obj);

// математическое (исключая бинарный "+")

let n = +obj; // унарный плюс

let delta = date1 - date2;

// больше/меньше сравнения

let greater = user1 > user2;

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

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

// бинарный плюс

let total = car1 + car2;

// obj == string/number/symbol

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

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

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

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

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

1. Вызывает obj[Symbol.toPrimitive](hint) – метод с символьным ключом Symbol.toPrimitive (системный символ), если такой метод существует, и передаёт ему хинт.
2. Иначе, если хинт равен "string", пытается вызвать obj.toString(), а если его нет, то obj.valueOf(), если он существует.
3. В случае, если хинт равен "number" или "default", пытается вызвать obj.valueOf(), а если его нет, то obj.toString(), если он существует.

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

Метод используется для всех преобразований:

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

};

Для примера используем его в реализации объекта user:

let user = {

name: "John",

money: 1000,

[Symbol.toPrimitive](hint) {

alert(`hint: ${hint}`);

return hint == "string" ? `{name: "${this.name}"}` : this.money;

}

};

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

alert(+user); // hint: number -> 1000

alert(user + 500); // hint: default -> 1500

Как видно из кода, user преобразовался в информативную читаемую строку, либо в денежный счёт, в зависимости от значения хинта. Единственный метод user[Symbol.toPrimitive] смог обработать все случаи преобразований.

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

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

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

* toString – valueOf для хинта со значением «string».
* valueOf – toString – в ином случае.

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

let user = {

name: "John",

money: 1000,

// "string"

toString() {

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

},

// "number" или "default"

valueOf() {

return this.money;

}

};

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

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

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

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

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

let user = {

name: "John",

toString() {

return this.name;

}

};

alert(user); // toString -> John

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

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

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

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

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

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

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

let obj = {

toString() {

return "2";

}

};

alert(obj \* 2); // 4

* Бинарный плюс + в аналогичном случае сложит строки:

let obj = {

toString() {

return "2";

}

};

alert(obj + 2); // 22

1. **Создание объектов через "new".**

Обычный синтаксис {...} позволяет создать только один объект. Но часто 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.

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

function User(name) {

// this = {};

this.name = name;

this.isAdmin = false;

// return this;

}

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

let user = {

name: "Вася",

isAdmin: false

};

Теперь, когда необходимо будет создать других пользователей, можно 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 выполнит 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.

**new function() { … }**

Если коде большое количество строк, создающих один сложный объект, можно обернуть их в функцию-конструктор следующим образом:

let user = new function() {

this.name = "Вася";

this.isAdmin = false;

// ...

};

Такой конструктор не может быть вызван дважды, так как он нигде не сохраняется, просто создаётся и тут же вызывается. Таким образом, такой метод создания позволяет инкапсулировать код, который создаёт отдельный объект, но без возможности его повторного использования. Данный метод используется очень редко.

Используя специальное свойство new.target внутри функции, можно проверить, вызвана ли функция при помощи оператора new или без него.

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

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

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

Другими словами, return с объектом возвращает объект, в любом другом случае конструктор вернёт this. В примере ниже return возвращает объект вместо this:

function BigUser() {

this.name = "Вася";

// возвращает объект

return { name: "Godzilla" };

}

alert( new BigUser().name );

Пример с пустым return (можно поставить примитив после return, не важно):

function SmallUser() {

this.name = "Вася";

return; // возвращает this

// ...

}

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

Можно не ставить скобки после new, если вызов конструктора идёт без аргументов:

let user = new User;

// 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#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 можно добавлять не только свойства, но и методы.

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

function User(name) {

this.name = name;

this.sayHi = function() {

alert( "Меня зовут: " + this.name );

};

}

let vasya = new User("Вася");

vasya.sayHi(); // Меня зовут: Вася

/\*

vasya = {

name: "Вася",

sayHi: function() { ... }

}

\*/

1. **Флаги и дескрипторы свойств.**

**[Флаги свойств](https://learn.javascript.ru/property-descriptors" \l "flagi-svoystv)**

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

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

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

Метод [Object.getOwnPropertyDescriptor](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor) позволяет получить полную информацию о свойстве. Его синтаксис:

let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);

* obj – объект, из которого получаем информацию.
* propertyName – имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги. Например:

let user = {

name: "John"

};

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

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

/\* дескриптор свойства:

{

"value": "John",

"writable": true,

"enumerable": true,

"configurable": true

}

\*/

Чтобы изменить флаги, можно использовать метод [Object.defineProperty](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty). Его синтаксис:

Object.defineProperty(obj, propertyName, descriptor)

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

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

let user = {};

Object.defineProperty(user, "name", {

value: "John"

});

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

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

/\*

{

"value": "John",

"writable": false,

"enumerable": false,

"configurable": false

}

\*/

Сравните этот способ с user.name, который создан выше «обычным способом»: в этот раз все флаги имеют значение false. Если это не то, что нужно, надо присвоить им значения true в параметре descriptor.

Рассмотрим на примерах, что даёт использование флагов.

[**Только для чтения**](https://learn.javascript.ru/property-descriptors#tolko-dlya-chteniya)

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

let user = {

name: "John"

};

Object.defineProperty(user, "name", {

writable: false

});

user.name = "Pete"; // Ошибка

Теперь никто не сможет изменить имя пользователя, если только не обновит соответствующий флаг новым вызовом defineProperty.

Ошибки появляются только в строгом режиме, в нестрогом режиме, без use strict, не видно никаких ошибок при записи в свойства «только для чтения» и т.п. Но эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

Вот тот же пример, но свойство создано «с нуля»:

let user = { };

Object.defineProperty(user, "name", {

value: "John",

enumerable: true,

configurable: true

});

alert(user.name); // John

user.name = "Pete"; // Ошибка

[**Неперечисляемое свойство**](https://learn.javascript.ru/property-descriptors#neperechislimoe-svoystvo)

Добавим собственный метод toString к объекту user. Встроенный метод toString в объектах – неперечисляемый, его не видно в цикле for..in. Но если написать свой собственный метод toString, цикл for..in будет выводить его по умолчанию:

let user = {

name: "John",

toString() {

return this.name;

}

};

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

Если в этом нет необходимости, можно установить для свойства enumerable: false. Тогда оно перестанет появляться в цикле for..in, аналогично встроенному toString:

let user = {

name: "John",

toString() {

return this.name;

}

};

Object.defineProperty(user, "toString", {

enumerable: false

});

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

Неперечисляемые свойства также не возвращаются Object.keys:

alert(Object.keys(user)); // name

[**Неконфигурируемое свойство**](https://learn.javascript.ru/property-descriptors#nekonfiguriruemoe-svoystvo)

Флаг неконфигурируемого свойства (configurable: false) иногда предустановлен для некоторых встроенных объектов и свойств. Неконфигурируемое свойство не может быть удалено или изменено с помощью defineProperty. Например, свойство Math.PI – только для чтения, неперечисляемое и неконфигурируемое:

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

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

/\*

{

"value": 3.141592653589793,

"writable": false,

"enumerable": false,

"configurable": false

}

\*/

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

Math.PI = 3; // Ошибка

Если свойство определено как неконфигурируемое, то нельзя поменять его обратно, потому что defineProperty не работает с неконфигурируемыми свойствами. В коде ниже свойство name определено как константа:

let user = { };

Object.defineProperty(user, "name", {

value: "John",

writable: false,

configurable: false

});

Object.defineProperty(user, "name", {writable: true}); // Ошибка

В нестрогом режиме мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

**[Метод Object.defineProperties](https://learn.javascript.ru/property-descriptors" \l "metod-object-defineproperties)**

Существует метод [Object.defineProperties(obj, descriptors)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties), который позволяет определять множество свойств сразу. Его синтаксис:

Object.defineProperties(obj, {

prop1: descriptor1,

prop2: descriptor2

// ...

});

Например:

Object.defineProperties(user, {

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

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

// ...

});

Таким образом, можно определить множество свойств одной операцией.

**Метод** [**Object.getOwnPropertyDescriptors**](https://learn.javascript.ru/property-descriptors#object-getownpropertydescriptors)

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться методом [Object.getOwnPropertyDescriptors(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptors). Вместе с Object.defineProperties этот метод можно использовать для клонирования объекта вместе с его флагами:

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

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

for (let key in user) {

clone[key] = user[key]

}

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

**[Глобальное запечатывание объекта](https://learn.javascript.ru/property-descriptors" \l "globalnoe-zapechatyvanie-obekta)**

Дескрипторы свойств работают на уровне конкретных свойств. Но еще есть методы, которые ограничивают доступ ко всему объекту:

* [Object.preventExtensions(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/preventExtensions) – запрещает добавлять новые свойства в объект.
* [Object.seal(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/seal) – запрещает добавлять/удалять свойства. Устанавливает configurable: false для всех существующих свойств.
* [Object.freeze(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze) – запрещает добавлять/удалять/изменять свойства. Устанавливает configurable: false, writable: false для всех существующих свойств.

А также есть методы для их проверки:

* [Object.isExtensible(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isExtensible) – возвращает false, если добавление свойств запрещено, иначе true.
* [Object.isSealed(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isSealed) – возвращает true, если добавление/удаление свойств запрещено и для всех существующих свойств установлено configurable: false.
* [Object.isFrozen(obj)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/isFrozen) – возвращает true, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено configurable: false, writable: false.

На практике эти методы используются редко.

1. **Геттеры и сеттеры.**

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

[**Геттеры и сеттеры**](https://learn.javascript.ru/property-accessors#gettery-i-settery)

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

let obj = {

get propName() {

// геттер, срабатывает при чтении obj.propName

},

set propName(value) {

// сеттер, срабатывает при записи obj.propName = value

}

};

Геттер срабатывает, когда obj.propName читается, сеттер – когда значение назначается. Например, есть объект user со свойствами name и surname:

let user = {

name: "John",

surname: "Smith"

};

Добавим свойство объекта fullName для полного имени – "John Smith". Реализуем его при помощи аксессора:

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

}

};

alert(user.fullName); // John Smith

Снаружи свойство-аксессор выглядит как обычное свойство. В этом и заключается смысл свойств-аксессоров. user.fullName  не вызывается  как функция, а читается как обычное свойство: геттер сам вернет нужное значение.

На данный момент в примере fullName имеет только геттер. Если попытаться присвоить значение свойству user.fullName, то это вызовет ошибку:

let user = {

get fullName() {

return `...`;

}

};

user.fullName = "Тест"; // Ошибка

Давайте исправим это, добавив сеттер для user.fullName:

let user = {

name: "John",

surname: "Smith",

get fullName() {

return `${this.name} ${this.surname}`;

},

set fullName(value) {

[this.name, this.surname] = value.split(" ");

}

};

user.fullName = "Alice Cooper";

alert(user.name); // Alice

alert(user.surname); // Cooper

В итоге получим «виртуальное» свойство fullName. Его можно прочитать и изменить, но по факту его не существует.

При попытке удалить свойство-аксессор оператором delete будет ошибка.

Дескрипторы свойств-аксессоров отличаются от «обычных» свойств-данных. Свойства-аксессоры не имеют value и writable, но взамен предлагают функции get и set.

То есть, дескриптор аксессора может иметь:

* get – функция без аргументов, которая сработает при чтении свойства,
* set – функция, принимающая один аргумент, вызываемая при присвоении свойства,
* enumerable – то же самое, что и для свойств-данных,
* configurable – то же самое, что и для свойств-данных.

Например, для создания аксессора fullName при помощи defineProperty можно передать дескриптор с использованием get и set:

let user = {

name: "John",

surname: "Smith"

};

Object.defineProperty(user, 'fullName', {

get() {

return `${this.name} ${this.surname}`;

},

set(value) {

[this.name, this.surname] = value.split(" ");

}

});

alert(user.fullName); // John Smith

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

Ещё раз заметим, что свойство объекта может быть только свойством-аксессором (с методами get/set) или свойством-данных (со значением value). При попытке указать и get и value в одном дескрипторе будет ошибка:

// Error: Invalid property descriptor.

Object.defineProperty({}, 'prop', {

get() {

return 1

},

value: 2

});

Геттеры/сеттеры можно использовать как обёртки над «реальными» значениями свойств, чтобы получить больше контроля над операциями с ними. Например, если надо запретить устанавливать короткое имя для user, можно использовать сеттер name для проверки, а само значение хранить в отдельном свойстве \_name:

let user = {

get name() {

return this.\_name;

},

set name(value) {

if (value.length < 4) {

alert("Имя слишком короткое, должно быть более 4 символов");

return;

}

this.\_name = value;

}

};

user.name = "Pete";

alert(user.name); // Pete

user.name = ""; // Имя слишком короткое...

Таким образом, само имя хранится в \_name, доступ к которому производится через геттер и сеттер. Технически, внешний код всё ещё может получить доступ к имени напрямую с помощью user.\_name, но существует широко известное соглашение о том, что свойства, которые начинаются с символа "\_", являются внутренними, и к ним не следует обращаться извне пределов объекта.

Аксессоры позволяют в любой момент взять «обычное» свойство и изменить его поведение, поменяв на геттер и сеттер. Например, представим, что реализован объект user, с использованием свойств-данных имя name и возраст age:

function User(name, age) {

this.name = name;

this.age = age;

}

let john = new User("John", 25);

alert( john.age ); // 25

Но со временем взамен возраста age можно хранить дату рождения birthday, потому что так более точно и удобно:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

}

let john = new User("John", new Date(1992, 6, 1));

Чтобы не менять весь старый код, который использует свойство age можно добавить геттер для age:

function User(name, birthday) {

this.name = name;

this.birthday = birthday;

Object.defineProperty(this, "age", {

get() {

let todayYear = new Date().getFullYear();

return todayYear - this.birthday.getFullYear();

}

});

}

let john = new User("John", new Date(1992, 6, 1));

alert( john.birthday );

alert( john.age );

Теперь и старый код работает, и появилось полезное дополнительное свойство.

1. **Декораторы и переадресация вызова.**

JavaScript предоставляет исключительно гибкие возможности по работе с функциями: они могут быть переданы в другие функции, использованы как объекты. Рассмотрим как перенаправлять вызовы между ними и как их декорировать.

[**Прозрачное кеширование**](https://learn.javascript.ru/call-apply-decorators#prozrachnoe-keshirovanie)

Представим, что есть функция slow(x), выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Другими словами, для одного и того же *x* она всегда возвращает один и тот же результат. Если функция вызывается часто, то стоит кешировать (запомнить) возвращаемые ею результаты, чтобы сэкономить время на повторных вычислениях.

Вместо того, чтобы усложнять slow(x) дополнительной функциональностью заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование. Вот код с объяснениями:

function slow(x) {

// здесь могут быть CPU ресурсоёмкие вычисления

alert(`Called with ${x}`);

return x;

}

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

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

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

}

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

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

return result;

};

}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) кешируем

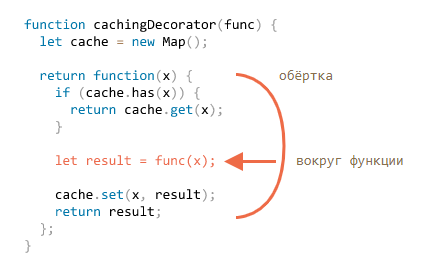
alert( "Again: " + slow(1) ); // возвращаем кеш

alert( slow(2) ); // slow(2) кешируем

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

В коде выше cachingDecorator – это декоратор, специальная функция, которая принимает другую функцию и изменяет её поведение. Идея состоит в том, что можно вызвать cachingDecorator с любой функцией, в результате чего получим кеширующую обёртку. Это удобно, т.к. может быть множество функций, использующих такой функционал, и все, что нужно сделать – это применить к ним cachingDecorator. Отделяя кеширующий код от основного кода, также сохраняем чистоту и простоту последнего.

Результат вызова cachingDecorator(func) является «обёрткой», т.е. function(x) «оборачивает» вызов func(x) в кеширующую логику:



С точки зрения внешнего кода обёрнутая функция slow по-прежнему делает то же самое. Обёртка всего добавляет к её поведению аспект кеширования.

Подводя итог, можно выделить несколько преимуществ использования отдельной cachingDecorator вместо изменения кода самой slow:

* Функцию cachingDecorator можно использовать повторно. Можно применить её к другой функции.
* Логика кеширования является отдельной, она не увеличивает сложность самой slow (если таковая была).
* При необходимости можно объединить несколько декораторов (речь о них пойдёт позже).

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

Упомянутый выше кеширующий декоратор не подходит для работы с методами объектов. Например, в приведённом ниже коде worker.slow() перестаёт работать после применения декоратора:

let worker = {

someMethod() {

return 1;

},

slow(x) {

alert("Called with " + x);

return x \* this.someMethod(); // (\*)

}

};

// тот же код, что и выше

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) {

return cache.get(x);

}

let result = func(x); // (\*\*)

cache.set(x, result);

return result;

};

}

alert( worker.slow(1) ); // оригинальный метод работает

worker.slow = cachingDecorator(worker.slow); // сделаем его кеширующим

alert( worker.slow(2) ); // Ошибка

Ошибка возникает в строке (\*). Функция пытается получить доступ к this.someMethod и завершается с ошибкой. Причина в том, что в строке (\*\*) декоратор вызывает оригинальную функцию как func(x), а в этом случае функция получает this = undefined. Т.е. декоратор передаёт вызов оригинальному методу, но без контекста.

Существует специальный встроенный метод функции [func.call(context, …args)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/call), который позволяет вызывать функцию, явно устанавливая this. Синтаксис:

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

Он запускает функцию func, предоставляя первый аргумент как this, а последующие как её аргументы. Проще говоря, эти два вызова делают почти то же самое:

func(1, 2, 3);

func.call(obj, 1, 2, 3)

Они оба вызывают func с аргументами 1, 2 и 3. Единственное отличие состоит в том, что func.call также устанавливает this в obj. Например, в приведённом ниже коде вызывается sayHi в контексте различных объектов: sayHi.call(user) запускает sayHi, передавая this=user, а следующая строка устанавливает this=admin:

function sayHi() {

alert(this.name);

}

let user = { name: "John" };

let admin = { name: "Admin" };

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

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

Здесь используется call для вызова say с заданным контекстом и фразой:

function say(phrase) {

alert(this.name + ': ' + phrase);

}

let user = { name: "John" };

say.call( user, "Hello" ); // John: Hello

В рассматриваемом примере можно использовать call в обёртке для передачи контекста в исходную функцию:

let worker = {

someMethod() {

return 1;

},

slow(x) {

alert("Called with " + x);

return x \* this.someMethod(); // (\*)

}

};

function cachingDecorator(func) {

let cache = new Map();

return function(x) {

if (cache.has(x)) {

return cache.get(x);

}

let result = func.call(this, x); // 'this' передаётся правильно

cache.set(x, result);

return result;

};

}

worker.slow = cachingDecorator(worker.slow); // сделаем её кеширующей

alert( worker.slow(2) ); // работает

alert( worker.slow(2) ); // работает

Рассмотрим подробнее, как передаётся this:

1. После декорации worker.slow становится обёрткой function (x) { ... }.
2. Так что при выполнении worker.slow(2) обёртка получает 2 в качестве аргумента, и this=worker (так как это объект перед точкой).
3. Внутри обёртки, если результат ещё не кеширован, func.call(this, x) передаёт текущий this (=worker) и текущий аргумент (=2) в оригинальную функцию.

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

Сделаем cachingDecorator ещё более универсальным. До сих пор он работал только с функциями с одним аргументом. Кешируем метод с несколькими аргументами worker.slow:

let worker = {

slow(min, max) {

return min + max; // здесь может быть тяжёлая задача

}

};

// should remember same-argument calls

worker.slow = cachingDecorator(worker.slow);

Ранее для одного аргумента *x* можно было просто сохранить результат cache.set(x, result) и вызвать cache.get(x) чтобы получить его. Но теперь нужно запомнить результат для комбинации аргументов (min,max). Встроенный Map принимает только одно значение как ключ. Есть много возможных решений:

1. Реализовать новую (или использовать стороннюю) структуру данных для коллекции, которая более универсальна чем встроенный Map, и поддерживает множественные ключи.
2. Использовать вложенные коллекции: cache.set(min) будет Map которая хранит пару (max, result). Тогда получить result можно вызвав cache.get(min).get(max).
3. Соединить два значения в одно. В нашем конкретном случае можно просто использовать строку "min,max" как ключ к Map. Для гибкости, можно позволить передавать хеширующую функцию в декоратор, которая знает, как сделать одно значение из многих.

Для многих практических применений третий вариант наиболее предпочтителен, поэтому реализуем его.

Нужно заменить func.call(this, x) на func.call(this, ...arguments), чтобы передавать все аргументы обёрнутой функции, а не только первый. Вот более мощный cachingDecorator:

let worker = {

slow(min, max) {

alert(`Called with ${min},${max}`);

return min + max;

}

};

function cachingDecorator(func, hash) {

let cache = new Map();

return function() {

let key = hash(arguments); // (\*)

if (cache.has(key)) {

return cache.get(key);

}

let result = func.call(this, ...arguments); // (\*\*)

cache.set(key, result);

return result;

};

}

function hash(args) {

return args[0] + ',' + args[1];

}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // работает

alert( "Again " + worker.slow(3, 5) ); // аналогично (из кеша)

Теперь он работает с любым количеством аргументов. Есть два изменения:

* В строке (\*) вызываем hash для создания одного ключа из arguments. Здесь используем простую функцию «объединения», которая превращает аргументы (3, 5) в ключ "3,5". В более сложных случаях могут потребоваться другие функции хеширования.
* Затем (\*\*) используем func.call(this, ...arguments) для передачи как контекста, так и всех аргументов, полученных обёрткой (независимо от их количества), в исходную функцию.

Вместо func.call(this, ...arguments) можно написать func.apply(this, arguments). Синтаксис встроенного метода [func.apply](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/apply):

func.apply(context, args)

Он выполняет func, устанавливая this=context и принимая в качестве списка аргументов псевдомассив args. Единственная разница в синтаксисе между call и apply состоит в том, что call ожидает список аргументов, в то время как apply принимает псевдомассив. Эти два вызова почти эквивалентны:

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

func.apply(context, args);

Есть только одна небольшая разница: оператор расширения ... позволяет передавать перебираемый объект args в виде списка в call, а apply принимает только псевдомассив args. Так что эти вызовы дополняют друг друга. Для перебираемых объектов сработает call, а там, где ожидается псевдомассив – apply.

Если есть, например, реальный массив, то технически можно использовать любой, но apply, вероятно, будет быстрее, потому что большинство движков JavaScript внутренне оптимизируют его лучше. Передача всех аргументов вместе с контекстом другой функции называется «перенаправлением вызова» (call forwarding). Простейший вид такого перенаправления:

let wrapper = function() {

return func.apply(this, arguments);

};

При вызове wrapper из внешнего кода его не отличить от вызова исходной функции.

[**Заимствование метода**](https://learn.javascript.ru/call-apply-decorators#method-borrowing)

Сделаем ещё одно небольшое улучшение функции хеширования:

function hash(args) {

return args[0] + ',' + args[1];

}

На данный момент он работает только для двух аргументов. Было бы лучше, если бы она могла склеить любое количество args. Естественным решением было бы использовать метод [arr.join](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Array/join):

function hash(args) {

return args.join();

}

Но это не сработает, потому что вызывается hash(arguments), а объект arguments является перебираемым и псевдомассивом, но не реальным массивом. Таким образом, вызов join для него не сработает:

function hash() {

alert( arguments.join() ); // Ошибка: arguments.join не является функцией

}

hash(1, 2);

Тем не менее, есть простой способ использовать соединение массива:

function hash() {

alert( [].join.call(arguments) ); // 1,2

}

hash(1, 2);

Этот способ называется заимствование метода. Т.е. заимствуется метод join из обычного массива [].join и используется [].join.call, чтобы выполнить его в контексте arguments. Это связано с тем, что внутренний алгоритм встроенного метода arr.join(glue) очень прост:

1. Пускай первым аргументом будет glue или, в случае отсутствия аргументов, им будет запятая ",".
2. Пускай result будет пустой строкой "".
3. Добавить this[0] к result.
4. Добавить glue и this[1].
5. Добавить glue и this[2].
6. Выполнять до тех пор, пока this.length элементов не будет склеено.
7. Вернуть result.

Таким образом, технически он принимает this и объединяет this[0], this[1]… и т.д. вместе. Он намеренно написан так, что допускает любой псевдомассив this (не случайно, многие методы следуют этой практике). Вот почему он также работает с this=arguments.

1. **Привязка контекста.**

При передаче методов объекта в качестве колбэков, например, для setTimeout, возникает известная проблема – потеря this.

**[Потеря «this»](https://learn.javascript.ru/bind" \l "poterya-this)**

Примеры потери this рассматривались ранее. Как только метод передается отдельно от объекта – this теряется. Вот как это может произойти с setTimeout:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

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

При запуске этого кода видно, что вызов this.firstName возвращает не «Вася», а undefined. Это произошло потому, что setTimeout получил функцию sayHi, отдельно от объекта user (именно здесь функция и потеряла контекст). То есть последняя строка может быть переписана как:

let f = user.sayHi;

setTimeout(f, 1000); // контекст user потеряли

Метод setTimeout в браузере имеет особенность: он устанавливает this=window для вызова функции (в Node.js thisстановится объектом таймера, но здесь это не имеет значения). Таким образом, для this.firstName он пытается получить window.firstName, которого не существует. В других подобных случаях обычно this просто становится undefined.

Задача состоит в том, что надо передать метод объекта куда-то ещё (в этом конкретном случае – в планировщик), где он будет вызван. Нужно, чтобы он был вызван в правильном контексте.

Есть несколько решений этой задачи. Одно из них – [сделать функцию-обёртку](https://learn.javascript.ru/bind#reshenie-1-sdelat-funktsiyu-obyortku), обернуть вызов в анонимную функцию, создав замыкание:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

setTimeout(function() {

user.sayHi(); // Привет, Вася!

}, 1000);

Теперь код работает корректно, так как объект user достаётся из замыкания, а затем вызывается его метод sayHi. То же самое, только короче:

setTimeout(() => user.sayHi(), 1000); // Привет, Вася!

Теперь в коде появилась небольшая уязвимость: до момента срабатывания setTimeout (задержка составляет целую секунду) в переменную user может быть записано другое значение. Тогда вызов будет совсем не тот:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

setTimeout(() => user.sayHi(), 1000);

user = { sayHi() { alert("Другой пользователь в 'setTimeout'!"); } };

Второе решение описанной выше задачи гарантирует, что такого не случится. Оно заключается в привязке [контекста с помощью bind](https://learn.javascript.ru/bind#reshenie-2-privyazat-kontekst-s-pomoschyu-bind). В современном JavaScript у функций есть встроенный метод [bind](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/bind), который позволяет зафиксировать this. Базовый синтаксис bind:

let boundFunc = func.bind(context);

Результатом вызова func.bind(context) является особый «экзотический объект» (термин взят из спецификации), который вызывается как функция и прозрачно передает вызов в func, при этом устанавливая this=context. Другими словами, вызов boundFunc подобен вызову func с фиксированным this. Например, в коде ниже funcUser передает вызов в func, фиксируя this=user:

let user = {

firstName: "Вася"

};

function func() {

alert(this.firstName);

}

let funcUser = func.bind(user);

funcUser(); // Вася

Здесь func.bind(user) – это «связанный вариант» func, с фиксированным this=user. Все аргументы передаются исходному методу func «как есть», например:

let user = {

firstName: "Вася"

};

function func(phrase) {

alert(phrase + ', ' + this.firstName);

}

// привязка this к user

let funcUser = func.bind(user);

funcUser("Привет"); // Привет, Вася

Теперь давайте попробуем с методом объекта:

let user = {

firstName: "Вася",

sayHi() {

alert(`Привет, ${this.firstName}!`);

}

};

let sayHi = user.sayHi.bind(user); // (\*)

sayHi(); // Привет, Вася!

setTimeout(sayHi, 1000); // Привет, Вася!

В строке (\*) метод user.sayHi привязываем к user. Теперь SayHi – это «связанная» функция, которая может быть вызвана отдельно или передана в setTimeout (контекст всегда будет правильным). Здесь можно видеть, что bind исправляет только this, а аргументы передаются «как есть»:

let user = {

firstName: "Вася",

say(phrase) {

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

}

};

let say = user.say.bind(user);

say("Привет"); // Привет, Вася

say("Пока"); // Пока, Вася

До сих пор речь шла только о привязывании this. Можно привязать не только this, но и аргументы. Это делается редко, но иногда может быть полезно.

Полный синтаксис bind:

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

Это позволяет привязать контекст this и начальные аргументы функции. Например, у нас есть функция умножения mul(a, b):

function mul(a, b) {

return a \* b;

}

Воспользуемся bind, чтобы создать функцию double на её основе:

function mul(a, b) {

return a \* b;

}

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

alert( double(3) ); // = mul(2, 3) = 6

alert( double(4) ); // = mul(2, 4) = 8

alert( double(5) ); // = mul(2, 5) = 10

Вызов mul.bind(null, 2) создаёт новую функцию double, которая передаёт вызов mul, фиксируя null как контекст и 2 – как первый аргумент. Следующие аргументы передаются «как есть». Это называется [частичное применение](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B8%D0%BC%D0%B5%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5) – создаётся новая функция, которая фиксирует некоторые из существующих параметров.

Обратите внимание, что в данном случае не используется this. Но для bind это обязательный параметр, так что надо передать туда что-нибудь вроде null. В следующем коде функция triple умножает значение на три:

function mul(a, b) {

return a \* b;

}

let triple = mul.bind(null, 3);

alert( triple(3) ); // = mul(3, 3) = 9

alert( triple(4) ); // = mul(3, 4) = 12

alert( triple(5) ); // = mul(3, 5) = 15

Польза от частично применённой функции в том, что можно создать независимую функцию с понятным названием (double, triple). Можно использовать её и не передавать каждый раз первый аргумент, т.к. он зафиксирован с помощью bind.

В других случаях частичное применение полезно, когда есть очень общая функция и для удобства надо создать её частный вариант. Например, есть функция send(from, to, text) и позже может возникнуть необходимость внутри объекта user использовать её частный вариант: sendTo(to, text), который отправляет текст от имени текущего пользователя.

Возможна ситуация, когда надо зафиксировать некоторые аргументы, но не контекст this. Например, для метода объекта. Встроенный bind не позволяет этого. Нельзя просто опустить контекст и перейти к аргументам. Но можно создать вспомогательную функцию partial, которая привязывает только аргументы. Вот так:

function partial(func, ...argsBound) {

return function(...args) { // (\*)

return func.call(this, ...argsBound, ...args);

}

}

let user = {

firstName: "John",

say(time, phrase) {

alert(`[${time}] ${this.firstName}: ${phrase}!`);

}

};

user.sayNow = partial(user.say, new Date().getHours() + ':' + new Date().getMinutes());

user.sayNow("Hello"); // [10:00] John: Hello!

Результатом вызова partial(func[, arg1, arg2...]) будет обёртка (\*), которая вызывает func с:

* Тем же this, который она получает (для вызова user.sayNow – это будет user)
* Затем передаёт ей ...argsBound – аргументы из вызова partial ("10:00")
* Затем передаёт ей ...args – аргументы, полученные обёрткой ("Hello")

Благодаря оператору расширения ... это реализовать легко.

1. **setTimeout and setInterval.**

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

* setTimeout позволяет вызвать функцию один раз через определённый интервал времени.
* setInterval позволяет вызывать функцию регулярно, повторяя вызов через определённый интервал времени.

Эти методы не являются частью спецификации JavaScript. Но большинство сред выполнения JS-кода имеют внутренний планировщик и предоставляют доступ к этим методам. В частности, они поддерживаются во всех браузерах и Node.js.

**Метод** [**setTimeout**](https://learn.javascript.ru/settimeout-setinterval#settimeout)

Синтаксис:

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

Параметры:

func|code – функция или строка кода для выполнения. Обычно, это функция. Можно передать и строку кода, но это не рекомендуется.

delay – задержка перед запуском в миллисекундах (1000 мс = 1 с). Значение по умолчанию – 0.

arg1, arg2… – аргументы, передаваемые в функцию (не поддерживается в IE9-)

Например, данный код вызывает sayHi() спустя одну секунду:

function sayHi() {

alert('Привет');

}

setTimeout(sayHi, 1000);

С аргументами:

function sayHi(phrase, who) {

alert( phrase + ', ' + who );

}

setTimeout(sayHi, 1000, "Привет", "Джон"); // Привет, Джон

Если первый аргумент является строкой, то JavaScript создаст из неё функцию. Это также будет работать:

setTimeout("alert('Привет')", 1000);

Но использование строк не рекомендуется. Вместо этого используйте функции. Например, так:

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

Передавайте функцию, но не запускайте её. Начинающие разработчики иногда ошибаются, добавляя скобки () после функции:

// не правильно!

setTimeout(sayHi(), 1000);

Это не работает, потому что setTimeout ожидает ссылку на функцию. Здесь sayHi() запускает выполнение функции и результат выполнения отправляется в setTimeout. В нашем случае результатом выполнения sayHi() является undefined (так как функция ничего не возвращает), поэтому ничего не планируется.

[**Отмена через clearTimeout**](https://learn.javascript.ru/settimeout-setinterval#otmena-cherez-cleartimeout)

Вызов setTimeout возвращает «идентификатор таймера» timerId, который можно использовать для отмены дальнейшего выполнения. Синтаксис для отмены:

let timerId = setTimeout(...);

clearTimeout(timerId);

В коде ниже планируем вызов функции и затем отменяем его. В результате ничего не происходит:

let timerId = setTimeout(() => alert("ничего не происходит"), 1000);

alert(timerId); // идентификатор таймера

clearTimeout(timerId);

alert(timerId); // тот же идентификатор

Как видно из вывода alert, в браузере идентификатором таймера является число. В других средах это может быть что-то ещё. Например, Node.js возвращает объект таймера с дополнительными методами.

**Метод** [**setInterval**](https://learn.javascript.ru/settimeout-setinterval#setinterval)

Метод setInterval имеет такой же синтаксис как setTimeout:

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

Все аргументы имеют такое же значение. Но отличие этого метода от setTimeout в том, что функция запускается не один раз, а периодически через указанный интервал времени. Чтобы остановить дальнейшее выполнение функции, необходимо вызвать clearInterval(timerId). Следующий пример выводит сообщение каждые 2 секунды. Через 5 секунд вывод прекращается:

// повторить с интервалом 2 секунды

let timerId = setInterval(() => alert('tick'), 2000);

// остановить вывод через 5 секунд

setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);

В большинстве браузеров, включая Chrome и Firefox внутренний счётчик продолжает тикать во время показа alert/confirm/prompt. Так что если запустить код выше и подождать с закрытием alert несколько секунд, то следующий alert будет показан сразу, как вы его закроете. Интервал времени между сообщениями alert будет короче, чем 2 секунды.

**[Рекурсивный setTimeout](https://learn.javascript.ru/settimeout-setinterval" \l "rekursivnyy-settimeout)**

Есть два способа запускать что-то регулярно. Один из них setInterval. Другим является рекурсивный setTimeout. Например:

/\* вместо:

let timerId = setInterval(() => alert('tick'), 2000);

\*/

let timerId = setTimeout(function tick() {

alert('tick');

timerId = setTimeout(tick, 2000); // (\*)

}, 2000);

Метод setTimeout выше планирует следующий вызов прямо после окончания, текущего (\*). Рекурсивный setTimeout – более гибкий метод, чем setInterval. С его помощью, последующий вызов может быть задан по-разному, в зависимости от результатов предыдущего. Например, необходимо написать сервис, который отправляет запрос для получения данных на сервер каждые 5 секунд, но если сервер перегружен, то необходимо увеличить интервал запросов до 10, 20, 40 секунд. Например:

let delay = 5000;

let timerId = setTimeout(function request() {

//...отправить запрос...

if (/\* ошибка запроса из-за перегрузки сервера \*/) {

// увеличить интервал для следующего запроса

delay \*= 2;

}

timerId = setTimeout(request, delay);

}, delay);

Рекурсивный setTimeout позволяет задать задержку между выполнениями более точно, чем setInterval. Сравним два фрагмента кода. Первый использует setInterval:

let i = 1;

setInterval(function() {

func(i);

}, 100);

Второй использует рекурсивный setTimeout:

let i = 1;

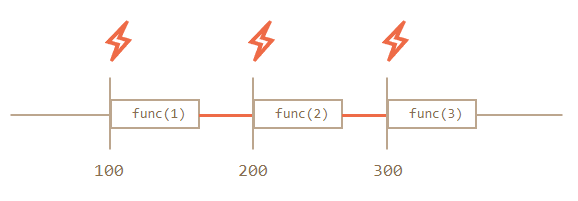
setTimeout(function run() {

func(i);

setTimeout(run, 100);

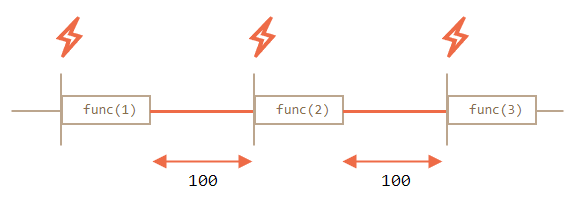
}, 100);

Для setInterval внутренний планировщик выполнит func(i) каждые 100 мс:



Реальная задержка между func для setInterval меньше, чем видно из кода. Это нормально, потому что время, затраченное на выполнение func, «потребляет» часть заданного интервала времени. Вполне возможно, что выполнение func будет дольше, чем ожидается, и займёт более 100 мс. В данном случае движок ждёт окончания выполнения func и затем проверяет планировщик и, если время истекло, немедленно запускает его снова. В крайнем случае, если функция всегда выполняется дольше, чем задержка delay, то вызовы будут выполняться без задержек вовсе.

Ниже представлено изображение, показывающее процесс работы рекурсивного setTimeout:



Рекурсивный setTimeout гарантирует фиксированную задержку (здесь 100 мс). Это потому, что новый вызов планируется в конце предыдущего.

Когда функция передаётся в setInterval/setTimeout, на неё создаётся внутренняя ссылка и сохраняется в планировщике. Это предотвращает попадание функции в сборщик муссора, даже если на неё нет других ссылок.

// функция остаётся в памяти до тех пор, пока планировщик обращается к ней

setTimeout(function() {...}, 100);

Для setInterval функция остаётся в памяти до тех, пока не будет вызван clearInterval. Есть и побочный эффект. Функция ссылается на внешнее лексическое окружение, поэтому пока она существует, внешние переменные существуют тоже. Они могут занимать больше памяти, чем сама функция. Поэтому, если регулярный вызов функции больше не нужен, то лучше отменить его, даже если функция очень маленькая.

Особый вариант использования: setTimeout(func, 0) или просто setTimeout(func). Это планирует вызов func настолько быстро, насколько это возможно. Но планировщик будет вызывать функцию только после завершения выполнения текущего кода. Так вызов функции будет запланирован сразу после выполнения текущего кода.

Например, этот код выводит «Привет» и затем сразу «Мир»:

setTimeout(() => alert("Мир"));

alert("Привет");

Первая строка «помещает вызов в календарь через 0 мс». Но планировщик «проверит календарь» после того, как текущий код завершится. Поэтому "Привет" выводится первым, а "Мир" после него.

В браузере есть ограничение, как часто внутренние счётчики могут выполняться. В [стандарте HTML5](https://www.w3.org/TR/html5/webappapis.html#timers) говорится: «после пяти вложенных таймеров интервал должен составлять не менее четырёх миллисекунд.».

Продемонстрируем в примере ниже, что это означает. Вызов setTimeout повторно вызывает себя через 0 мс. Каждый вызов запоминает реальное время от предыдущего вызова в массиве times. Посмотрим какова реальная задержка:

let start = Date.now();

let times = [];

setTimeout(function run() {

times.push(Date.now() - start);

if (start + 100 < Date.now()) alert(times);

else setTimeout(run);

});

Первый таймер запускается сразу (как и указано в спецификации) и затем начинается задержка и вывод 9, 15, 20, 24. Аналогичное происходит при использовании setInterval вместо setTimeout: setInterval(f) запускает f несколько раз с нулевой задержкой, а затем с задержкой 4+ мс. Это ограничение существует давно, многие скрипты полагаются на него, поэтому оно сохраняется по историческим причинам. Этого ограничения нет в серверном JavaScript. Там есть и другие способы планирования асинхронных задач. Например, [setImmediate](https://nodejs.org/api/timers.html) для Node.js. Так что это ограничение относится только к браузерам.

**10**.**Асинхронные итераторы и генераторы**

Асинхронные итераторы позволяют перебирать данные, поступающие асинхронно. Например, когда загружается что-то по частям по сети. Асинхронные генераторы делают такой перебор ещё удобнее. Рассмотрим простой пример, чтобы понять синтаксис, а затем – реальный практический.

[**Асинхронные итераторы**](https://learn.javascript.ru/async-iterators-generators#asinhronnye-iteratory)

Асинхронные итераторы похожи на обычные итераторы, но имеют некоторые синтаксические отличия. «Обычный» перебираемый объект выглядит так:

let range = {

from: 1,

to: 5,

// for..of вызывает этот метод один раз в самом начале

[Symbol.iterator]() {

// ...возвращает объект-итератор:

// далее for..of работает только с этим объектом, запрашивая следующее значение вызовом next()

return {

current: this.from,

last: this.to,

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

next() { // (2)

// должен возвращать значение в виде объекта {done:.., value :...}

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

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

} else {

return { done: true };

}

}

};

}

};

for(let value of range) {

alert(value); // 1 потом 2, потом 3, потом 4, потом 5

}

Чтобы сделать объект итерируемым асинхронно:

1. Используется Symbol.asyncIterator вместо Symbol.iterator.
2. next() должен возвращать промис.
3. Чтобы перебрать такой объект, используется цикл for await (let item of iterable).

Создадим итерируемый объект range, как и в предыдущем примере, но теперь он будет возвращать значения асинхронно, по одному в секунду:

let range = {

from: 1,

to: 5,

// for await..of вызывает этот метод один раз в самом начале

[Symbol.asyncIterator]() { // (1)

// ...возвращает объект-итератор:

// далее for await..of работает только с этим объектом,

// запрашивая у него следующие значения вызовом next()

return {

current: this.from,

last: this.to,

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

async next() { // (2)

// должен возвращать значение как объект {done:.., value :...}

// (автоматически оборачивается в промис с помощью async)

// можно использовать await внутри для асинхронности:

await new Promise(resolve => setTimeout(resolve, 1000)); // (3)

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

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

} else {

return { done: true };

}

}

};

}

};

(async () => {

for await (let value of range) { // (4)

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

}

})()

Как видим, структура похожа на обычные итераторы:

1. Чтобы сделать объект асинхронно итерируемым, он должен иметь метод Symbol.asyncIterator (1).
2. Этот метод должен возвращать объект с методом next(), который в свою очередь возвращает промис (2).
3. Метод next() не обязательно должен быть async, он может быть обычным методом, возвращающим промис, но async позволяет использовать await, так что это удобно. Здесь просто создаем паузу на одну секунду (3).
4. Для итерации используем for await(let value of range) (4), а именно добавляем «await» после «for». Он вызовет range[Symbol.asyncIterator]() один раз, а затем его метод next() для получения значений.

Вот небольшая шпаргалка:

|  | **Итераторы** | **Асинхронные итераторы** |
| --- | --- | --- |
| Метод для создания итерируемого объекта | Symbol.iterator | Symbol.asyncIterator |
| next() возвращает | любое значение | промис |
| для цикла используйте | for..of | for await..of |

Функции, которые требуют обычных синхронных итераторов, не работают с асинхронными. Например, оператор расширения (...) не будет работать:

alert( [...range] ); // Ошибка, нет Symbol.iterator

Это естественно, так как он ожидает Symbol.iterator, как и for..of без await. Ему не подходит Symbol.asyncIterator.

[**Асинхронные генераторы**](https://learn.javascript.ru/async-iterators-generators#asinhronnye-generatory)

Как известно генераторы являются перебираемыми. Вспомним генератор последовательности, рассматриваемый ранее. Он генерирует последовательность значений от start до end:

function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) {

yield i;

}

}

for(let value of generateSequence(1, 5)) {

alert(value); // 1, потом 2, потом 3, потом 4, потом 5

}

В обычных генераторах нельзя использовать await. Все значения должны поступать синхронно: в for..of нет места для задержки, это синхронная конструкция. Если нужно использовать await в теле генератора просто добавьте в начале async, например, вот так:

async function\* generateSequence(start, end) {

for (let i = start; i <= end; i++) {

await new Promise(resolve => setTimeout(resolve, 1000));

yield i;

}

}

(async () => {

let generator = generateSequence(1, 5);

for await (let value of generator) {

alert(value); // 1, then 2, then 3, then 4, then 5

}

})();

Получился асинхронный генератор, который можно перебирать с помощью for await ... of.

С технической точки зрения, еще одно отличие асинхронного генератора заключается в том, что его метод generator.next() теперь тоже асинхронный и возвращает промисы. Из обычного генератора можно получить значения при помощи result = generator.next(). Для асинхронного нужно добавить await, вот так:

result = await generator.next(); // result = {value: ..., done: true/false}

[**Асинхронно перебираемые объекты**](https://learn.javascript.ru/async-iterators-generators#asinhronno-perebiraemye-obekty)

Как известно, чтобы сделать объект перебираемым, нужно добавить к нему Symbol.iterator.

let range = {

from: 1,

to: 5,

[Symbol.iterator]() {

return <объект с next, чтобы сделать range перебираемым>

}

}

Обычная практика для Symbol.iterator – возвращать генератор, а не простой объект с next, как в предыдущем примере. Вспомним пример из вопроса «[Генераторы](https://learn.javascript.ru/generators)»:

let range = {

from: 1,

to: 5,

\*[Symbol.iterator]() { // сокращение для [Symbol.iterator]: function\*()

for(let value = this.from; value <= this.to; value++) {

yield value;

}

}

};

for(let value of range) {

alert(value); // 1, потом 2, потом 3, потом 4, потом 5

}

Здесь созданный объект range является перебираемым, а генератор \*[Symbol.iterator] реализует логику для перечисления значений. Если надо добавить асинхронные действия в генератор, нужно заменить Symbol.iterator на асинхронный Symbol.asyncIterator:

let range = {

from: 1,

to: 5,

async \*[Symbol.asyncIterator]() { // то же, что и [Symbol.asyncIterator]: async function\*()

for(let value = this.from; value <= this.to; value++) {

// пауза между значениями, ожидание

await new Promise(resolve => setTimeout(resolve, 1000));

yield value;

}

}

};

(async () => {

for await (let value of range) {

alert(value); // 1, потом 2, потом 3, потом 4, потом 5

}

})();

Теперь значения поступают с задержкой в одну секунду между ними.

**Тема 5. ООП в функциональном и прототипном стиле**

* 1. [Прототипное наследование](https://learn.javascript.ru/prototype-inheritance).
  2. [F.prototype](https://learn.javascript.ru/function-prototype).
  3. [Встроенные прототипы](https://learn.javascript.ru/native-prototypes).
  4. [Методы прототипов.](https://learn.javascript.ru/prototype-methods)
  5. [Классы.](https://learn.javascript.ru/class)
  6. [Наследование классов](https://learn.javascript.ru/class-inheritance).
  7. [Статические свойства и методы](https://learn.javascript.ru/static-properties-methods).
  8. [Приватные и защищённые методы и свойства](https://learn.javascript.ru/private-protected-properties-methods).
  9. [Расширение встроенных классов](https://learn.javascript.ru/extend-natives).
  10. [Проверка класса: "instanceof"](https://learn.javascript.ru/instanceof).
  11. [Примеси](https://learn.javascript.ru/mixins).
  12. Обработка ошибок, "try..catch".
  13. Пользовательские ошибки, расширение Error.
  14. Модули: введение.
  15. Модули: экспорт и импорт.
  16. Модули: динамические импорты.

Содержание данной темы включает материалы, доступные по адресу https://learn.javascript.ru.

1. [**Прототипное наследование**](https://learn.javascript.ru/prototype-inheritance)

В программировании часто возникает необходимость что-то расширить. Например, есть объект user со своими свойствами и методами, надо создать объекты admin и guest как его слегка изменённые варианты. Хотелось бы повторно использовать то, что есть у объекта user, не копировать/переопределять его методы, а просто создать новый объект на его основе.

*Прототипное наследование* — это возможность языка, которая помогает в этом.

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

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

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

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

let animal = {

eats: true

};

let rabbit = {

jumps: true

};

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

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

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

let animal = {

eats: true

};

let rabbit = {

jumps: true

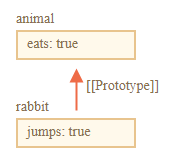
};

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

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

alert( rabbit.jumps ); // true

Здесь строка (\*) устанавливает animal как прототип для rabbit. Затем, когда alert пытается прочитать свойство rabbit.eats (\*\*), его нет в rabbit, поэтому JavaScript следует по ссылке [[Prototype]] и находит её в animal (смотрите снизу вверх):



Здесь можно сказать, что animal является прототипом rabbit или rabbit прототипно наследует от animal. Так что если у animal много полезных свойств и методов, то они автоматически становятся доступными у rabbit. Такие свойства называются *унаследованными*. Например, есть метод в animal, он может быть вызван на rabbit:

let animal = {

eats: true,

walk() {

alert("Animal walk");

}

};

let rabbit = {

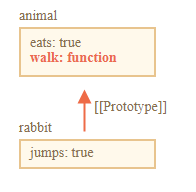
jumps: true,

\_\_proto\_\_: animal

};

rabbit.walk(); // Animal walk

Метод автоматически берётся из прототипа:



Цепочка прототипов может быть длиннее:

let animal = {

eats: true,

walk() {

alert("Animal walk");

}

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

let longEar = {

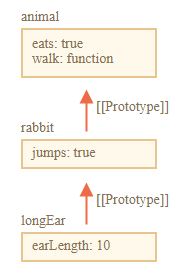
earLength: 10,

\_\_proto\_\_: rabbit

};

longEar.walk(); // Animal walk

alert(longEar.jumps); // true (для rabbit)



Есть только два ограничения:

1. Ссылки не могут идти по кругу. JavaScript выдаст ошибку, если попытаться назначить \_\_proto\_\_ по кругу.
2. Значение \_\_proto\_\_ может быть объектом или null. Другие типы игнорируются.

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

Прототип используется только для чтения свойств. Операции записи/удаления работают напрямую с объектом. В приведённом ниже примере присваивается rabbit собственный метод walk:

let animal = {

eats: true,

walk() {

/\* ... \*/

}

};

let rabbit = {

\_\_proto\_\_: animal

};

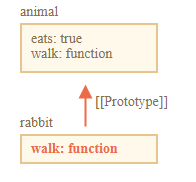
rabbit.walk = function() {

alert("Rabbit! Bounce-bounce!");

};

rabbit.walk(); // Rabbit! Bounce-bounce!

Теперь вызов rabbit.walk() находит метод непосредственно в объекте и выполняет его, не используя прототип:



Свойства-акссессоры – исключение, так как запись в него обрабатывается функцией-сеттером. То есть, это, фактически, вызов функции. По этой причине admin.fullName работает корректно в приведённом ниже коде:

let user = {

name: "John",

surname: "Smith",

set fullName(value) {

[this.name, this.surname] = value.split(" ");

},

get fullName() {

return `${this.name} ${this.surname}`;

}

};

let admin = {

\_\_proto\_\_: user,

isAdmin: true

};

alert(admin.fullName); // John Smith (\*)

// срабатывает сеттер!

admin.fullName = "Alice Cooper"; // (\*\*)

Здесь в строке (\*) свойство admin.fullName имеет геттер в прототипе user, поэтому вызывается он. В строке (\*\*) свойство также имеет сеттер в прототипе, который и будет вызван.

[**Значение «this»**](https://learn.javascript.ru/prototype-inheritance#znachenie-this)

Прототипы никак не влияют на this. Неважно, где находится метод: в объекте или его прототипе. При вызове метода this – всегда объект перед точкой. Таким образом, вызов сеттера admin.fullName в качестве this использует admin, а не user.

Это на самом деле очень важная деталь, потому что может быть большой объект со множеством методов, от которого можно наследовать. Затем наследущие объекты могут вызывать его методы, но они будут изменять состояние этих объектов, а не большого. Например, здесь animal представляет собой «хранилище методов», и rabbit использует его. Вызов rabbit.sleep() устанавливает this.isSleeping для объекта rabbit:

let animal = {

walk() {

if (!this.isSleeping) {

alert(`I walk`);

}

},

sleep() {

this.isSleeping = true;

}

};

let rabbit = {

name: "White Rabbit",

\_\_proto\_\_: animal

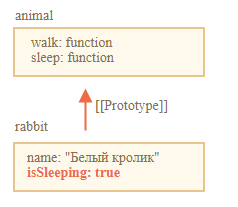
};

rabbit.sleep();

alert(rabbit.isSleeping); // true

alert(animal.isSleeping); // undefined (нет такого свойства в прототипе)

Картинка с результатом:



Если бы были другие объекты, такие как bird, snake и т.д., унаследованные от animal, они также получили бы доступ к методам animal. Но this при вызове каждого метода будет соответствовать объекту, на котором происходит вызов (перед точкой), а не animal. Поэтому, когда записываются данные в this, они сохраняются в этих объектах. В результате методы являются общими, а состояние объекта — нет.

[**Цикл for…in**](https://learn.javascript.ru/prototype-inheritance#tsikl-for-in)

Цикл for..in проходит не только по собственным, но и по унаследованным свойствам объекта. Например:

let animal = {

eats: true

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

alert(Object.keys(rabbit)); // jumps

for(let prop in rabbit) alert(prop); // jumps, then eats

Если унаследованные свойства не нужны, то можно отфильтровать их при помощи встроенного метода [obj.hasOwnProperty(key)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty): он возвращает true, если у obj есть собственное, не унаследованное, свойство с именем key. Пример такой фильтрации:

let animal = {

eats: true

};

let rabbit = {

jumps: true,

\_\_proto\_\_: animal

};

for(let prop in rabbit) {

let isOwn = rabbit.hasOwnProperty(prop);

if (isOwn) {

alert(`Our: ${prop}`); // Our: jumps

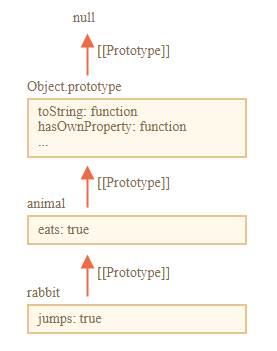
} else {

alert(`Inherited: ${prop}`); // Inherited: eats

}

}

В этом примере цепочка наследования выглядит так: rabbit наследует от animal, который наследует от Object.prototype (так как animal – литеральный объект {...}, это по умолчанию), а затем null на самом верху:



Стоит отметить следующее: метод rabbit.hasOwnProperty явно не определен. Если посмотреть на цепочку прототипов, то видно, что он берётся из Object.prototype.hasOwnProperty. То есть, он унаследован, но не появляется в цикле for..in, в отличие от eats и jumps. Дело в том, что это свойство не перечислимо. То есть, у него внутренний флаг enumerable стоит false, как и у других свойств Object.prototype. Поэтому оно и не появляется в цикле.

Почти все методы, получающие ключи/значения, такие как Object.keys, Object.values и другие – игнорируют унаследованные свойства. Они учитывают только свойства самого объекта, не его прототипа.

1. [**F.prototype**](https://learn.javascript.ru/function-prototype)

Как известно, новые объекты могут быть созданы с помощью функции-конструктора, new F(). Если в F.prototype содержится объект, оператор new устанавливает его в качестве [[Prototype]] для нового объекта.

JavaScript использовал прототипное наследование с момента своего появления. Это одна из основных особенностей языка.

Но раньше, прямого доступа к прототипу объекта не было. Надёжно работало только свойство "prototype" функции-конструктора. Поэтому оно используется во многих скриптах. Обратите внимание, что F.prototype означает обычное свойство с именем "prototype" для F. Это ещё не «прототип объекта», а обычное свойство F с таким именем. Приведём пример:

let animal = {

eats: true

};

function Rabbit(name) {

this.name = name;

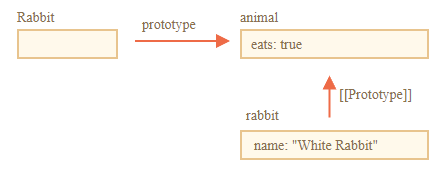
}

Rabbit.prototype = animal;

let rabbit = new Rabbit("White Rabbit"); // rabbit.\_\_proto\_\_ == animal

alert( rabbit.eats ); // true

Установка Rabbit.prototype = animal буквально говорит интерпретатору следующее: "При создании объекта через new Rabbit() запиши ему animal в [[Prototype]]". Результат будет выглядеть так:



На изображении: "prototype" – горизонтальная стрелка, обозначающая обычное свойство для "F", а [[Prototype]]– вертикальная, обозначающая наследование rabbit от animal.

F.prototype используется только при вызове new F() и присваивается в качестве свойства [[Prototype]]нового объекта. После этого F.prototype и новый объект ничего не связывает. После создания F.prototype может измениться, и новые объекты, созданные с помощью new F(), будут иметь другой объект в качестве [[Prototype]], но уже существующие объекты сохранят старый.

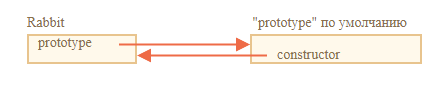
У каждой функции по умолчанию уже есть свойство "prototype". По умолчанию "prototype" – объект с единственным свойством constructor, которое ссылается на функцию-конструктор. Вот такой:

function Rabbit() {}

/\* прототип по умолчанию

Rabbit.prototype = { constructor: Rabbit };

\*/



Проверим это:

function Rabbit() {}

// по умолчанию:

// Rabbit.prototype = { constructor: Rabbit }

alert( Rabbit.prototype.constructor == Rabbit ); // true

Соответственно, если ничего не меняется, то свойство constructor будет доступно всем кроликам через [[Prototype]]:

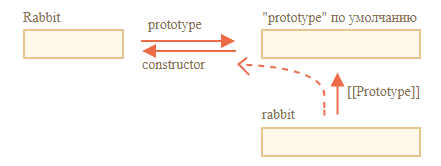
function Rabbit() {}

// по умолчанию:

// Rabbit.prototype = { constructor: Rabbit }

let rabbit = new Rabbit(); // наследует от {constructor: Rabbit}

alert(rabbit.constructor == Rabbit); // true (свойство получено из прототипа)



Можно использовать свойство constructor существующего объекта для создания нового. Пример:

function Rabbit(name) {

this.name = name;

alert(name);

}

let rabbit = new Rabbit("White Rabbit");

let rabbit2 = new rabbit.constructor("Black Rabbit");

Это удобно, когда есть объект, но неизвестно какой конструктор использовался для его создания (например, он был взят из сторонней библиотеки), а необходимо создать ещё один такой объект.

Самое важное о свойстве "constructor" это то, что JavaScript сам по себе не гарантирует правильное значение свойства "constructor". Оно является свойством по умолчанию в "prototype" у функций, но что будет с ним позже – зависит только от разработчика. В частности, если заменить прототип по умолчанию на другой объект, свойства "constructor" в нём не будет. Например:

function Rabbit() {}

Rabbit.prototype = {

jumps: true

};

let rabbit = new Rabbit();

alert(rabbit.constructor === Rabbit); // false

Таким образом, чтобы сохранить верное свойство "constructor", надо добавлять/удалять/изменять свойства у прототипа по умолчанию вместо того, чтобы перезаписывать его целиком. В примере ниже Rabbit.prototype не перезаписывается полностью, а добавляется к нему свойство. Прототип по умолчанию сохраняется, и сохраняется доступ к Rabbit.prototype.constructor.

function Rabbit() {}

Rabbit.prototype.jumps = true

Или можно заново создать свойство constructor:

Rabbit.prototype = {

jumps: true,

constructor: Rabbit

};

1. [**Встроенные прототипы**](https://learn.javascript.ru/native-prototypes)

Свойство "prototype" широко используется в внутри самого языка JavaScript. Все встроенные функции-конструкторы используют его.

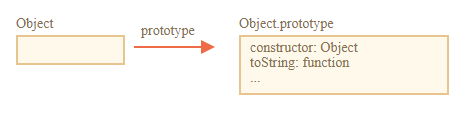
[**Object.prototype**](https://learn.javascript.ru/native-prototypes#object-prototype)

Выведем пустой объект:

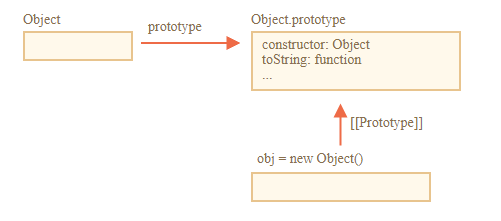
let obj = {};

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

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



Когда вызывается new Object() (или создаётся объект с помощью литерала {...}), свойство [[Prototype]] этого объекта устанавливается на Object.prototype по правилам, которые рассматривались в предыдущем вопросе:



Таким образом, когда вызывается obj.toString(), метод берётся из Object.prototype. Можно проверить это так:

let obj = {};

alert(obj.\_\_proto\_\_ === Object.prototype); // true

// obj.toString === obj.\_\_proto\_\_.toString == Object.prototype.toString

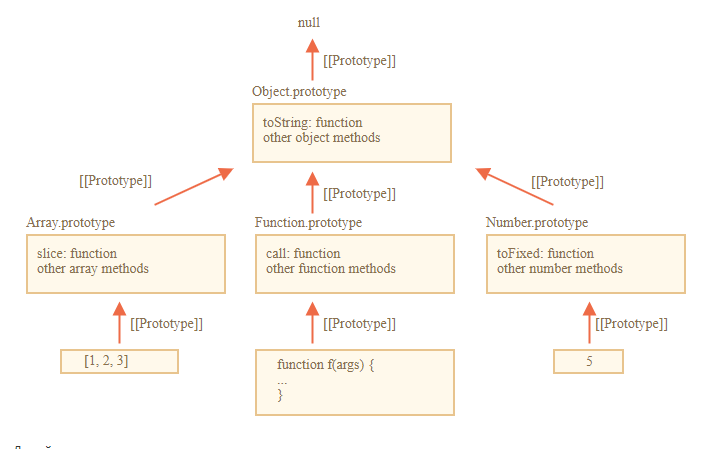
Обратим внимание, что выше Object.prototype по цепочке прототипов больше нет [[Prototype]]:

alert(Object.prototype.\_\_proto\_\_); // null

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

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

Вот более полная картина (для 3 встроенных объектов):



Проверим прототипы:

let arr = [1, 2, 3];

// наследует от Array.prototype?

alert( arr.\_\_proto\_\_ === Array.prototype ); // true

// затем от Object.prototype?

alert( arr.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype ); // true

// и null на вершине иерархии

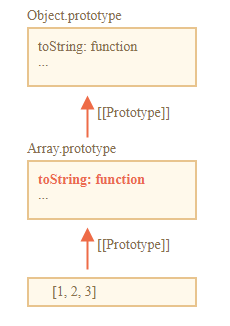
alert( arr.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ ); // null

Некоторые методы в прототипах могут пересекаться, например, у Array.prototype есть свой метод toString, который выводит элементы массива через запятую:

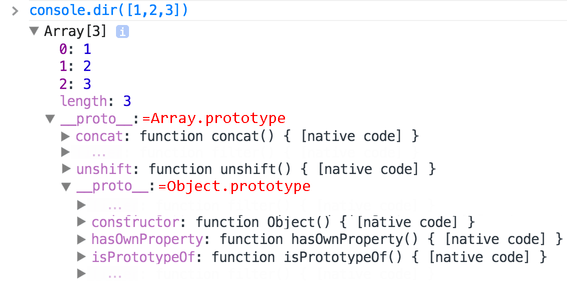
let arr = [1, 2, 3]

alert(arr); // 1,2,3 <-- результат Array.prototype.toString

Как известно, у Object.prototype есть свой метод toString, но так как Array.prototype ближе в цепочке прототипов, то берётся именно вариант для массивов:



В браузерных инструментах, таких как консоль разработчика, можно посмотреть цепочку наследования (возможно, потребуется использовать console.dir для встроенных объектов):



Другие встроенные объекты устроены аналогично. Даже функции – объекты встроенного конструктора Function, и все их методы (call/apply и другие) берутся из Function.prototype. Также у функций есть свой метод toString.

function f() {}

alert(f.\_\_proto\_\_ == Function.prototype); // true

alert(f.\_\_proto\_\_.\_\_proto\_\_ == Object.prototype); // true, наследует от Object

[**Примитивы**](https://learn.javascript.ru/native-prototypes#primitivy)

Самое сложное происходит со строкам, числами и булевым типом. Как известно, они не объекты. Но если попытаться получить доступ к их свойствам, тогда будет создан временный объект-обёртка с использованием встроенных конструкторов String, Number, Boolean, который предоставит методы и после чего исчезнет. Эти объекты создаются невидимо для нас, и большая часть движков оптимизирует этот процесс, но спецификация описывает это именно таким образом. Методы этих объектов также находятся в прототипах, доступных как String.prototype, Number.prototype и Boolean.prototype.

Специальные значения null и undefined не имеют объектов-обёрток, так что методы и свойства им недоступны. Также у них нет соответствующих прототипов.

[**Изменение встроенных прототипов**](https://learn.javascript.ru/native-prototypes#native-prototype-change)

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

String.prototype.show = function() {

alert(this);

};

"BOOM!".show(); // BOOM!

В течение процесса разработки могут возникнуть идеи о новых встроенных методах, которые хотелось бы иметь и добавить их во встроенные прототипы. Это плохая идея. Прототипы глобальны, поэтому очень легко могут возникнуть конфликты. Если две библиотеки добавляют метод String.prototype.show, то одна из них перепишет метод другой.

В современном программировании есть только один случай, в котором одобряется изменение встроенных прототипов. Это создание полифилов. Полифил – это термин, который означает замену метода, который существует в спецификации JavaScript, но он ещё не поддерживается текущим движком JavaScript. Тогда можно реализовать его и добавить его во встроенный прототип. Например:

if (!String.prototype.repeat) { // Если такого метода нет

// добавляем его в прототип

String.prototype.repeat = function(n) {

// повторить строку n раз

return new Array(n + 1).join(this);

};

}

alert( "La".repeat(3) ); // LaLaLa

[**Заимствование у прототипов**](https://learn.javascript.ru/native-prototypes#zaimstvovanie-u-prototipov)

Ранее рассматривалось заимствовании методов. Это когда метод из одного объекта и копируется в другой. Некоторые методы встроенных прототипов часто одалживают. Например, если создать объект, похожий на массив (псевдомассив), можно скопировать некоторые методы из Array в этот объект. Пример:

let obj = {

0: "Hello",

1: "world!",

length: 2,

};

obj.join = Array.prototype.join;

alert( obj.join(',') ); // Hello,world!

Это работает, потому что для внутреннего алгоритма встроенного метода join важна только корректность индексов и свойства length, он не проверяет является ли объект на самом деле массивом. И многие встроенные методы работают так же. Альтернативная возможность –можно унаследовать от массива, установив obj.\_\_proto\_\_ как Array.prototype, таким образом все методы Array станут автоматически доступны в obj. Но это будет невозможно, если obj уже наследует от другого объекта, ведь можно наследовать только от одного объекта одновременно.

Заимствование методов – гибкий способ, позволяющий смешивать функциональность разных объектов по необходимости.

1. [**Методы прототипов**](https://learn.javascript.ru/prototype-methods)

Ранее упоминалось, что существуют современные методы работы с прототипами. Свойство \_\_proto\_\_ считается устаревшим, и по стандарту должно поддерживаться только браузерами. Современные методы это:

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

Эти методы нужно использовать вместо \_\_proto\_\_. Например:

let animal = {

eats: true

};

let rabbit = Object.create(animal);

alert(rabbit.eats); // true

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

Object.setPrototypeOf(rabbit, {});

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

let animal = {

eats: true

};

let rabbit = Object.create(animal, {

jumps: {

value: true

}

});

alert(rabbit.jumps); // true

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

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

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

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

JavaScript движки хорошо оптимизированы для этого. Изменение прототипа «на лету» с помощью Object.setPrototypeOf или obj.\_\_proto\_\_= – очень медленная операция, которая ломает внутренние оптимизации для операций доступа к свойствам объекта. Так что лучше избегайте этого, кроме тех случаев, когда знаете, что делаете, либо скорость JavaScript для вас не имеет никакого значения.

Как известно, объекты можно использовать как ассоциативные массивы для хранения пар ключ/значение. Но если попробовать хранить созданные пользователями ключи (например, словари с пользовательским вводом), можно заметить интересный сбой: все ключи работают как ожидается, за исключением "\_\_proto\_\_". Например:

let obj = {};

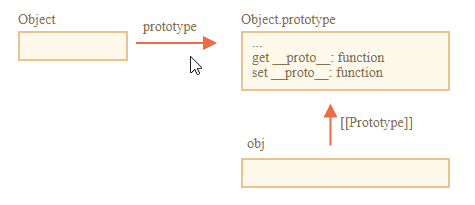
let key = prompt("What's the key?", "\_\_proto\_\_");

obj[key] = "some value";

alert(obj[key]); // [object Object], не "some value"

Если пользователь введёт \_\_proto\_\_, присвоение проигнорируется, так как свойство \_\_proto\_\_ должно быть либо объектом, либо null, а строка не может стать прототипом.

Свойство \_\_proto\_\_ – не обычное, а аксессор, заданный в Object.prototype:



Так что при чтении или установке obj.\_\_proto\_\_ вызывается соответствующий геттер/сеттер из прототипа obj, и именно он устанавливает/получает свойство [[Prototype]].

Поните, что \_\_proto\_\_ – это способ доступа к свойству [[Prototype]], это не само свойство [[Prototype]].

Теперь, если надо использовать объект как ассоциативный массив, можно сделать это следующим образом:

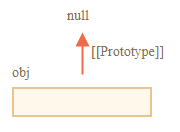
let obj = Object.create(null);

let key = prompt("What's the key?", "\_\_proto\_\_");

obj[key] = "some value";

alert(obj[key]); // "some value"

Object.create(null) создаёт пустой объект без прототипа ([[Prototype]] будет null):



Таким образом не будет унаследованного геттера/сеттера для \_\_proto\_\_. Теперь это свойство обрабатывается как обычное свойство, и приведённый выше пример работает правильно. Можно назвать такой объект «простейшим» или «чистым словарным объектом», потому что он ещё проще чем обычный объект {...}. Недостаток в том, что у таких объектов не будет встроенных методов объекта, таких как toString:

let obj = Object.create(null);

alert(obj); // Error (no toString)

Но обычно это нормально для ассоциативных массивов. Обратите внимание, что большая часть методов, связанных с объектами, имеют вид Object.something(...). К примеру, Object.keys(obj) не находятся в прототипе, так что они продолжат работать для таких объектов:

let chineseDictionary = Object.create(null);

chineseDictionary.hello = "你好";

chineseDictionary.bye = "再见";

alert(Object.keys(chineseDictionary)); // hello,bye

1. [**Классы**](https://learn.javascript.ru/class)

В объектно-ориентированном программировании класс – это расширяемый шаблон кода для создания объектов, который устанавливает в них начальные значения (свойства) и реализацию поведения (методы).

На практике часто надо создавать много объектов одного вида, например пользователей, товары или что-то еще. Как известно, с этим может помочь new function. Но в современном JavaScript есть и более продвинутая конструкция class, которая предоставляет новые возможности, полезные для объектно-ориентированного программирования.

[**Синтаксис «class»**](https://learn.javascript.ru/class#sintaksis-class)

Базовый синтаксис выглядит так:

class MyClass {

// методы класса

constructor() { ... }

method1() { ... }

method2() { ... }

method3() { ... }

...

}

Затем используйте вызов new MyClass() для создания нового объекта со всеми перечисленными методами. При этом автоматически вызывается метод constructor(), в нём можно инициализовать объект. Например:

class User {

constructor(name) {

this.name = name;

}

sayHi() {

alert(this.name);

}

}

let user = new User("Иван");

user.sayHi();

Когда вызывается new User("Иван"):

1. Создаётся новый объект.
2. constructor запускается с заданным аргументом и сохраняет его в this.name.

Затем можно вызывать методы объекта, такие как user.sayHi().

Методы в классе не разделяются запятой. Это приводит к синтаксической ошибке.

В JavaScript класс – это разновидность функции. Рассмотрим пример:

class User {

constructor(name) { this.name = name; }

sayHi() { alert(this.name); }

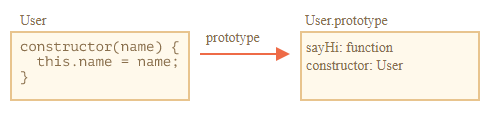
}

alert(typeof User); // function

Вот что на самом деле делает конструкция class User {...}:

1. Создает функцию с именем User, которая становится результатом объявления класса. Код функции берется из метода constructor (она будет пустой, если такого метода нет).
2. Сохраняет все методы, такие как sayHi, в User.prototype.

Затем, при вызове метода на новых объектах new User, он возьмётся из прототипа. Таким образом, объект new User имеет доступ к методам класса. На картинке показан результат объявления class User:



Как видно из кода ниже, класс – это функция или, если точнее, это метод constructor, методы находятся в User.prototype.

class User {

constructor(name) { this.name = name; }

sayHi() { alert(this.name); }

}

alert(typeof User); // function

alert(User === User.prototype.constructor); // true

alert(User.prototype.sayHi); // alert(this.name);

alert(Object.getOwnPropertyNames(User.prototype)); // constructor, sayHi

Иногда говорят, что class – это просто «синтаксический сахар» в JavaScript (синтаксис для улучшения читаемости кода, но не делающий ничего принципиально нового), потому что можно сделать все то же самое без конструкции class. Например:

// перепишем класс User с помощью функций

// 1. Создаём функцию constructor

function User(name) {

this.name = name;

}

// 2. Добавляем метод в прототип

User.prototype.sayHi = function() {

alert(this.name);

};

let user = new User("Иван");

user.sayHi();

Результат этого кода очень похож на предыдущий. Поэтому, class можно считать синтаксическим сахаром для определения конструктора вместе с методами прототипа. Однако есть важные отличия:

1. Во-первых, функция, созданная с помощью class, помечена специальным внутренним свойством [[FunctionKind]]:"classConstructor". Поэтому это не совсем то же самое, что создавать её вручную.

В отличие от обычных функций, конструктор класса не может быть вызван без new:

class User {

constructor() {}

}

alert(typeof User); // function

User(); // Error: Class constructor User cannot be invoked without 'new'

Кроме того, строковое представление конструктора класса в большинстве движков JavaScript начинается с «class …».

class User {

constructor() {}

}

alert(User); // class User { ... }

1. Методы класса являются неперечислимыми. Определение класса устанавливает флаг enumerable в false для всех методов в "prototype".
2. Классы всегда используют use strict. Весь код внутри класса автоматически находится в строгом режиме.

[**Class Expression**](https://learn.javascript.ru/class#class-expression)

Как и функции, классы можно определять внутри другого выражения, передавать, возвращать, присваивать и т.д. Пример Class Expression (по аналогии с Function Expression):

let User = class {

sayHi() {

alert("Привет");

}

};

Как и Named Function Expressions, выражения классов могут иметь имя, которое видно только внутри класса. Если у Class Expression есть имя, то оно видно только внутри класса:

let User = class MyClass {

sayHi() {

alert(MyClass);

}

};

new User().sayHi(); // работает

alert(MyClass); // ошибка

Можно динамически создавать классы «по-запросу»:

function makeClass(phrase) {

// объявляем класс и возвращаем его

return class {

sayHi() {

alert(phrase);

};

};

}

// Создаем новый класс

let User = makeClass("Привет");

new User().sayHi(); // Привет

Как и в литеральных объектах, в классах можно объявлять генераторы, вычисляемые свойства, геттеры/сеттеры и т.д. Пример user.name, реализованного с использованием get/set:

class User {

constructor(name) {

// вызывает сеттер

this.name = name;

}

get name() {

return this.\_name;

}

set name(value) {

if (value.length < 4) {

alert("Имя слишком короткое.");

return;

}

this.\_name = value;

}

}

let user = new User("Иван");

alert(user.name); // Иван

user = new User(""); // Имя слишком короткое.

При объявлении класса геттеры/сеттеры создаются на User.prototype:

Object.defineProperties(User.prototype, {

name: {

get() {

return this.\_name

},

set(name) {

// ...

}

}

});

Пример с вычисляемым свойством в скобках [...]:

class User {

['say' + 'Hi']() {

alert("Привет");

}

}

new User().sayHi();

Для методов-генераторов добавьте перед именем \*.

[**Свойства классов**](https://learn.javascript.ru/class#svoystva-klassov)

Свойства классов добавлены в язык недавно. Старым браузерам может понадобиться полифил. В приведённом выше примере у класса User были только методы. Добавим свойство:

class User {

name = "Аноним";

sayHi() {

alert(`Привет, ${this.name}!`);

}

}

new User().sayHi();

Свойство name не устанавливается в User.prototype. Вместо этого оно создаётся оператором new перед запуском конструктора, это именно свойство объекта.

1. [**Наследование классов**](https://learn.javascript.ru/class-inheritance)

Допустим, у нас есть два класса. Класс Animal:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed += speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

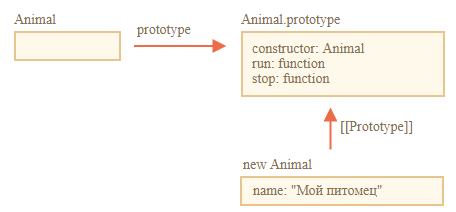
this.speed = 0;

alert(`${this.name} стоит.`);

}

}

let animal = new Animal("Мой питомец");



Класс Rabbit:

class Rabbit {

constructor(name) {

this.name = name;

}

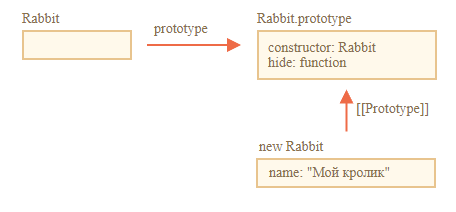
hide() {

alert(`${this.name} прячется!`);

}

}

let rabbit = new Rabbit("Мой кролик");



Для того, чтобы наследовать класс от другого, мы должны использовать ключевое слово "extends" и указать название родительского класса перед {..}. Ниже Rabbit наследует от Animal:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed += speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} стоит.`);

}

}

// Наследуем от Animal указывая "extends Animal"

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

}

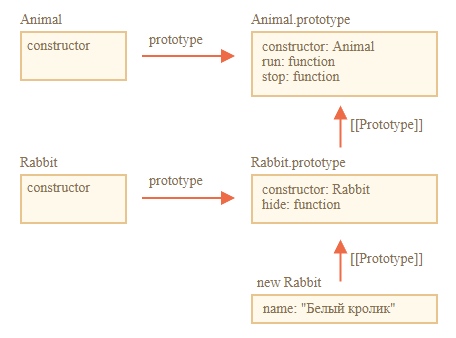
let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.hide(); // Белый кролик прячется!

Теперь код Rabbit стал короче, так как используется конструктор класса Animal по умолчанию и кролик может использовать метод run как и все животные.

На самом деле ключевое слово extends добавляет ссылку на [[Prototype]] из Rabbit.prototype в Animal.prototype:



Если метод не найден в Rabbit.prototype, JavaScript возьмёт его из Animal.prototype.

Синтаксис создания класса допускает указывать после extends не только класс, но любое выражение. Пример вызова функции, которая генерирует родительский класс:

function f(phrase) {

return class {

sayHi() { alert(phrase) }

}

}

class User extends f("Привет") {}

new User().sayHi(); // Привет

Здесь class User наследует от результата вызова f("Привет"). Это может быть полезно для продвинутых приёмов проектирования, где можно использовать функции для генерации классов в зависимости от многих условий и затем наследовать их.

[**Переопределение методов**](https://learn.javascript.ru/class-inheritance#pereopredelenie-metodov)

Сейчас Rabbit наследует от Animal метод stop, который устанавливает this.speed = 0. Если определить свой метод stop в классе Rabbit, то он будет использоваться взамен родительского:

class Rabbit extends Animal {

stop() {

// ...будет использован для rabbit.stop()

}

}

Обычно нет необходимости полностью заменять родительский метод, а только сделать новый на его основе, изменяя или расширяя его функциональность. Для этого надо определить новый метод, добавив нужный функционал, и вызывать родительский метод до/после или в процессе.

У классов есть ключевое слово "super" для таких случаев:

* super.method(...) вызывает родительский метод.
* super(...) вызывает родительский конструктор (работает только внутри нашего конструктора).

Пусть наш кролик автоматически прячется при остановке:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

run(speed) {

this.speed += speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

stop() {

this.speed = 0;

alert(`${this.name} стоит.`);

}

}

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

stop() {

super.stop(); // вызываем родительский метод stop

this.hide(); // и затем hide

}

}

let rabbit = new Rabbit("Белый кролик");

rabbit.run(5); // Белый кролик бежит со скоростью 5.

rabbit.stop(); // Белый кролик стоит. Белый кролик прячется!

Теперь у класса Rabbit есть метод stop, который вызывает родительский super.stop() в процессе выполнения.

У стрелочных функций нет super. При обращении к super стрелочной функции он берётся из внешней функции:

class Rabbit extends Animal {

stop() {

setTimeout(() => super.stop(), 1000); // вызывает родительский stop после 1 секунды

}

}

В примере super в стрелочной функции тот же самый, что и в stop(), поэтому метод отрабатывает как и ожидается. Если указать здесь «обычную» функцию, была бы ошибка:

// Unexpected super

setTimeout(function() { super.stop() }, 1000);

Согласно [спецификации](https://tc39.github.io/ecma262/" \l "sec-runtime-semantics-classdefinitionevaluation), если класс расширяет другой класс и не имеет конструктора, то автоматически создаётся такой «пустой» конструктор. Например, у Rabbit нет своего конструктора.

class Rabbit extends Animal {

constructor(...args) {

super(...args);

}

}

Как видно, он просто вызывает конструктор родительского класса. Так будет происходить, пока не будет создан собственный конструктор. Добавим конструктор для Rabbit. Он будет устанавливать earLength в дополнение к name:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

this.speed = 0;

this.name = name;

this.earLength = earLength;

}

// ...

}

let rabbit = new Rabbit("Белый кролик", 10); // Error: this is not defined.

Ошибка возникла из-за того, что в классах-потомках конструктор обязан вызывать super(...) и делать это перед использованием this. Дело в том, что в JavaScript существует различие между «функцией-конструктором наследующего класса» и всеми остальными. В наследующем классе соответствующая функция-конструктор помечена специальным внутренним свойством [[ConstructorKind]]:"derived". Разница в следующем:

* Когда выполняется обычный конструктор, он создаёт пустой объект и присваивает его this.
* Когда запускается конструктор унаследованного класса, он этого не делает. Вместо этого он ждёт, что это сделает конструктор родительского класса.

Поэтому, если создать собственный конструктор, то надо вызвать super, в противном случае объект для this не будет создан, и возникнет ошибка. Чтобы конструктор Rabbit работал, он должен вызвать super() до того, как использовать this, чтобы не было ошибки:

class Animal {

constructor(name) {

this.speed = 0;

this.name = name;

}

// ...

}

class Rabbit extends Animal {

constructor(name, earLength) {

super(name);

this.earLength = earLength;

}

// ...

}

let rabbit = new Rabbit("Белый кролик", 10);

alert(rabbit.name); // Белый кролик

alert(rabbit.earLength); // 10

**[[[HomeObject]]](https://learn.javascript.ru/class-inheritance" \l "homeobject)**

В JavaScript для функций добавлено специальное внутреннее свойство: [[HomeObject]]. Когда функция объявлена как метод внутри класса или объекта, её свойство [[HomeObject]] становится равно этому объекту. Затем super использует его, чтобы получить прототип родителя и его методы. Посмотрим, как это работает – опять же, используя простые объекты:

let animal = {

name: "Животное",

eat() { // animal.eat.[[HomeObject]] == animal

alert(`${this.name} eats.`);

}

};

let rabbit = {

\_\_proto\_\_: animal,

name: "Кролик",

eat() { // rabbit.eat.[[HomeObject]] == rabbit

super.eat();

}

};

let longEar = {

\_\_proto\_\_: rabbit,

name: "Длинноух",

eat() { // longEar.eat.[[HomeObject]] == longEar

super.eat();

}

};

longEar.eat(); // Длинноух ест.

Это работает как задумано благодаря [[HomeObject]]. Метод, такой как longEar.eat, знает свой [[HomeObject]] и получает метод родителя из его прототипа. Вообще без использования this.

[**Методы не «свободны»**](https://learn.javascript.ru/class-inheritance#metody-ne-svobodny)

Как известно, функции в JavaScript «свободны», не привязаны к объектам. Их можно копировать между объектами и вызывать с любым this. Но само существование [[HomeObject]] нарушает этот принцип, так как методы запоминают свои объекты. [[HomeObject]] нельзя изменить, эта связь – навсегда. Единственное место в языке, где используется [[HomeObject]] – это super. Поэтому если метод не использует super, то все ещё можно считать его свободным и копировать между объектами. А вот если super в коде есть, то возможны побочные эффекты. Вот пример неверного результата super после копирования:

let animal = {

sayHi() {

console.log(`Я животное`);

}

};

// rabbit наследует от animal

let rabbit = {

\_\_proto\_\_: animal,

sayHi() {

super.sayHi();

}

};

let plant = {

sayHi() {

console.log("Я растение");

}

};

// tree наследует от plant

let tree = {

\_\_proto\_\_: plant,

sayHi: rabbit.sayHi // (\*)

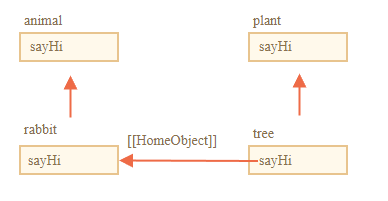
};

tree.sayHi(); // Я животное

Вызов tree.sayHi() показывает «Я животное». Определённо неверно. Причина проста:

* В строке (\*), метод tree.sayHi скопирован из rabbit.
* Его [[HomeObject]] – это rabbit, ведь он был создан в rabbit. Свойство [[HomeObject]] никогда не меняется.
* В коде tree.sayHi() есть вызов super.sayHi(). Он идёт вверх от rabbit и берёт метод из animal.

Вот диаграмма происходящего:



Свойство [[HomeObject]] определено для методов как классов, так и обычных объектов. Но для объектов методы должны быть объявлены именно как method(), а не "method: function()". В приведённом ниже примере используется синтаксис не метода, свойства-функции. Поэтому у него нет [[HomeObject]], и наследование не работает:

let animal = {

eat: function() { // должен быть короткий синтаксис: eat() {...}

// ...

}

};

let rabbit = {

\_\_proto\_\_: animal,

eat: function() {

super.eat();

}

};

rabbit.eat(); // Ошибка вызова super

1. [**Статические свойства и методы**](https://learn.javascript.ru/static-properties-methods)

Можно присвоить метод самой функции-классу, а не её "prototype". Такие методы называются статическими. В классе такие методы обозначаются ключевым словом static, например:

class User {

static staticMethod() {

alert(this === User);

}

}

User.staticMethod(); // true

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

class User() { }

User.staticMethod = function() {

alert(this === User);

};

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

Обычно статические методы используются для реализации функций, принадлежащих классу, но не к каким-то конкретным его объектам. Например, есть объекты статей Article, и нужна функция для их сравнения. Естественное решение – сделать для этого метод Article.compare:

class Article {

constructor(title, date) {

this.title = title;

this.date = date;

}

static compare(articleA, articleB) {

return articleA.date - articleB.date;

}

}

let articles = [

new Article("HTML", new Date(2019, 1, 1)),

new Article("CSS", new Date(2019, 0, 1)),

new Article("JavaScript", new Date(2019, 11, 1))

];

articles.sort(Article.compare);

alert( articles[0].title ); // CSS

Здесь метод Article.compare стоит «над» статьями, как способ их сравнения. Это метод не отдельной статьи, а всего класса. Другим примером может быть так называемый «фабричный» метод. Представим, что нужно создавать статьи различными способами:

1. Создание через заданные параметры (title, date и т. д.).
2. Создание пустой статьи с сегодняшней датой и др.

Первый способ может быть реализован через конструктор. А для второго можно использовать статический метод класса. Такой как Article.createTodays() в следующем примере:

class Article {

constructor(title, date) {

this.title = title;

this.date = date;

}

static createTodays() {

// this = Article

return new this("Сегодняшний дайджест", new Date());

}

}

let article = Article.createTodays();

alert( article.title ); // Сегодняшний дайджест

Теперь каждый раз, когда нужно создать сегодняшний дайджест, нужно вызывать Article.createTodays(). Ещё раз, это не метод одной статьи, а метод всего класса. Статические методы также используются в классах, относящихся к базам данных, для поиска/сохранения/удаления вхождений в базу данных. Например, предположим, что Article - это специальный класс для управления статьями статический метод для удаления статьи:

Article.remove({id: 12345});

[**Статические свойства**](https://learn.javascript.ru/static-properties-methods#staticheskie-svoystva)

Эта возможность была добавлена в язык недавно. Примеры работают в последнем Chrome. Статические свойства также возможны, они выглядят как свойства класса, но с static в начале:

class Article {

static publisher = "Иван Иванов";

}

alert( Article.publisher ); // Иван Иванов

Это то же самое, что и прямое присваивание Article:

Article.publisher = "Иван Иванов";

Статические свойства и методы наследуются. Например, метод Animal.compare в коде ниже наследуется и доступен как Rabbit.compare:

class Animal {

constructor(name, speed) {

this.speed = speed;

this.name = name;

}

run(speed = 0) {

this.speed += speed;

alert(`${this.name} бежит со скоростью ${this.speed}.`);

}

static compare(animalA, animalB) {

return animalA.speed - animalB.speed;

}

}

// Наследует от Animal

class Rabbit extends Animal {

hide() {

alert(`${this.name} прячется!`);

}

}

let rabbits = [

new Rabbit("Белый кролик", 10),

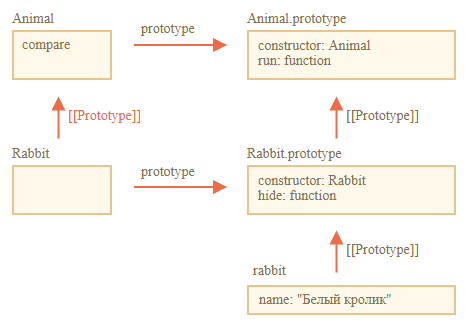
new Rabbit("Чёрный кролик", 5)

];

rabbits.sort(Rabbit.compare);

rabbits[0].run(); // Чёрный кролик бежит со скоростью 5.

Можно вызвать Rabbit.compare, при этом будет вызван унаследованный Animal.compare. Это работает с использованием прототипов. Extends даёт Rabbit ссылку [[Prototype]] на Animal.



Так что Rabbit extends Animal создаёт две ссылки на прототип:

1. Функция Rabbit прототипно наследует от Animal function.
2. Rabbit.prototype прототипно наследует от Animal.prototype.

В результате наследование работает как для обычных, так и для статических методов:

class Animal {}

class Rabbit extends Animal {}

// для статики

alert(Rabbit.\_\_proto\_\_ === Animal); // true

// для обычных методов

alert(Rabbit.prototype.\_\_proto\_\_ === Animal.prototype);

1. [**Приватные и защищённые методы и свойства**](https://learn.javascript.ru/private-protected-properties-methods)

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

* *Внутренний интерфейс* – методы и свойства, доступные из других методов класса, но не снаружи класса.
* *Внешний интерфейс* – методы и свойства, доступные снаружи класса.

Внутренний интерфейс используется для работы объекта, его методы, свойства используют друг друга. Всё, что нужно для использования объекта, это знать его внешний интерфейс. Совершенно не обязательно знать, его внутреннюю структуру и логику.

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

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

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

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

Создадим простой класс для описания работы кофеварки:

class CoffeeMachine {

waterAmount = 0; // количество воды внутри

constructor(power) {

this.power = power;

alert( `Создана кофеварка, мощность: ${power}` );

}

}

// создаём кофеварку

let coffeeMachine = new CoffeeMachine(100);

// добавляем воды

coffeeMachine.waterAmount = 200;

Сейчас свойства waterAmount и power публичные. Можно легко получать и устанавливать им любое значение извне. Изменим свойство waterAmount на защищённое, чтобы иметь больше контроля над ним. Например, не надо, чтобы кто-либо устанавливал его ниже нуля.

Защищённые свойства обычно начинаются с префикса \_. Это не синтаксис языка: есть хорошо известное соглашение между программистами, что такие свойства и методы не должны быть доступны извне. Большинство программистов следуют этому соглашению. Пожтому свойство будет называться \_waterAmount:

class CoffeeMachine {

\_waterAmount = 0;

set waterAmount(value) {

if (value < 0) throw new Error("Отрицательное количество воды");

this.\_waterAmount = value;

}

get waterAmount() {

return this.\_waterAmount;

}

constructor(power) {

this.\_power = power;

}

}

// создаём новую кофеварку

let coffeeMachine = new CoffeeMachine(100);

// устанавливаем количество воды

coffeeMachine.waterAmount = -10; // Error: Отрицательное количество воды

Теперь доступ под контролем, поэтому указать воду ниже нуля не удалось.

Сделаем свойство power доступным только для чтения. Иногда нужно, чтобы свойство устанавливалось только при создании объекта и после этого никогда не изменялось. Это как раз требуется для кофеварки: мощность никогда не меняется. Для этого нужно создать только геттер, но не сеттер:

class CoffeeMachine {

// ...

constructor(power) {

this.\_power = power;

}

get power() {

return this.\_power;

}

}

let coffeeMachine = new CoffeeMachine(100);

alert(`Мощность: ${coffeeMachine.power}W`); // Мощность: 100W

coffeeMachine.power = 25; // Error (no setter)

Здесь использовался синтаксис геттеров/сеттеров. Но в большинстве случаев использование функций get.../set... предпочтительнее:

class CoffeeMachine {

\_waterAmount = 0;

setWaterAmount(value) {

if (value < 0) throw new Error("Отрицательное количество воды");

this.\_waterAmount = value;

}

getWaterAmount() {

return this.\_waterAmount;

}

}

new CoffeeMachine().setWaterAmount(100);

Это выглядит немного длиннее, но функции более гибкие. Они могут принимать несколько аргументов.

Если унаследовать class MegaMachine extends CoffeeMachine, ничто не помешает нам обращаться к this.\_waterAmount или this.\_power из методов нового класса. Таким образом защищённые методы, конечно же, наследуются. В отличие от приватных полей.

**[Приватное свойство «#waterLimit»](https://learn.javascript.ru/private-protected-properties-methods" \l "privatnoe-svoystvo-waterlimit)**

Поддержка приватных свойств и методов была добавлена в язык недавно. В движках JavaScript пока не поддерживается или поддерживается частично, нужен полифилл. Приватные свойства и методы должны начинаться со знака #. Они доступны только внутри класса. Например, в классе ниже есть приватное свойство #waterLimit и приватный метод #checkWater для проверки количества воды:

class CoffeeMachine {

#waterLimit = 200;

#checkWater(value) {

if (value < 0) throw new Error("Отрицательный уровень воды");

if (value > this.#waterLimit) throw new Error("Слишком много воды");

}

}

let coffeeMachine = new CoffeeMachine();

// снаружи нет доступа к приватным методам класса

coffeeMachine.#checkWater(); // Error

coffeeMachine.#waterLimit = 1000; // Error

На уровне языка # является специальным символом, который означает, что поле приватное. Нельзя получить к нему доступ извне или из наследуемых классов. Приватные поля не конфликтуют с публичными. Может быть два поля одновременно – приватное #waterAmount и публичное waterAmount.

Например, давайте сделаем аксессор waterAmount для #waterAmount:

class CoffeeMachine {

#waterAmount = 0;

get waterAmount() {

return this.#waterAmount;

}

set waterAmount(value) {

if (value < 0) throw new Error("Отрицательный уровень воды");

this.#waterAmount = value;

}

}

let machine = new CoffeeMachine();

machine.waterAmount = 100;

alert(machine.#waterAmount); // Error

В отличие от защищённых, функционал приватных полей обеспечивается самим языком. Но если унаследовать от CoffeeMachine, то не будет прямого доступа к #waterAmount, только через геттер/сеттер waterAmount:

class MegaCoffeeMachine extends CoffeeMachine() {

method() {

alert( this.#waterAmount ); // Error: can only access from CoffeeMachine

}

}

Во многих случаях такое ограничение неудобно. Если расширяется CoffeeMachine, то должен быть доступ к внутренним методам и свойствам. Поэтому защищённые свойства используются чаще, хоть они и не поддерживаются синтаксисом языка.

Как известно, обычно можно получить доступ к полям объекта с помощью this[name]:

class User {

...

sayHi() {

let fieldName = "name";

alert(`Hello, ${this[fieldName]}`);

}

}

С приватными свойствами такое невозможно: this['#name'] не работает. Это ограничение синтаксиса сделано для обеспечения приватности.

1. [**Расширение встроенных классов**](https://learn.javascript.ru/extend-natives)

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

class PowerArray extends Array {

isEmpty() {

return this.length === 0;

}

}

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

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

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

alert(filteredArr); // 10, 50

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

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

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

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

class PowerArray extends Array {

isEmpty() {

return this.length === 0;

}

// встроенные методы массива будут использовать этот метод как конструктор

static get [Symbol.species]() {

return Array;

}

}

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

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

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

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

// filteredArr не является PowerArray, это Array

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

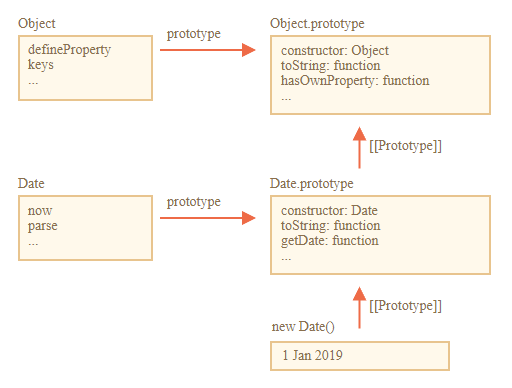
Теперь .filter возвращает Array. Расширенная функциональность не будет передаваться далее.

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

[**Отсутствие статического наследования встроенных классов**](https://learn.javascript.ru/extend-natives#otsutstvie-staticheskogo-nasledovaniya-vstroennyh-klassov)

У встроенных объектов есть собственные статические методы, например, Object.keys, Array.isArray и т. д. Как известно, встроенные классы расширяют друг друга. Обычно, когда один класс наследует другому, то наследуются и статические методы. Но встроенные классы – исключение. Они не наследуют статические методы друг друуга. Например, и Array и Date наследуют от Object, так что в их экземплярах доступны методы из Object.prototype. Но Array.[[Prototype]] не ссылается на Object, поэтому нет методов Array.keys() или Date.keys().

Ниже представлена структура Date и Object:



Как видно, нет связи между Date и Object. Они независимы, только Date.prototype наследует от Object.prototype. В этом важное отличие наследования встроенных объектов от тех, которые получаются с использованием extends.

1. [**Проверка класса: "instanceof"**](https://learn.javascript.ru/instanceof)

Оператор instanceof позволяет проверить, к какому классу принадлежит объект, с учётом наследования. Такая проверка может потребоваться во многих случаях. В рассматриваемых примерах она используется для создания полиморфной функции, которая интерпретирует аргументы по-разному в зависимости от их типа.

[**Оператор instanceof**](https://learn.javascript.ru/instanceof#ref-instanceof)

Синтаксис:

obj instanceof Class

Оператор вернёт true, если obj принадлежит классу Class или наследующему от него. Например:

class Rabbit {}

let rabbit = new Rabbit();

alert( rabbit instanceof Rabbit ); // true

Также это работает с функциями-конструкторами:

function Rabbit() {}

alert( new Rabbit() instanceof Rabbit ); // true

И для встроенных классов, таких как Array:

let arr = [1, 2, 3];

alert( arr instanceof Array ); // true

alert( arr instanceof Object ); // true

Обратите внимание, что arr также принадлежит классу Object, потому что Array наследует от Object.

Обычно оператор instanceof просматривает для проверки цепочку прототипов. Но это поведение может быть изменено при помощи статического метода Symbol.hasInstance.

Алгоритм работы obj instanceof Class работает примерно так:

1. Если имеется статический метод Symbol.hasInstance, тогда вызвать его: Class[Symbol.hasInstance](obj). Он должен вернуть либо true, либо false. Это как раз и есть возможность ручной настройки instanceof.

Пример:

class Animal {

static [Symbol.hasInstance](obj) {

if (obj.canEat) return true;

}

}

let obj = { canEat: true };

alert(obj instanceof Animal); // true: вызван Animal[Symbol.hasInstance](obj)

1. Большая часть классов не имеет метода Symbol.hasInstance. В этом случае используется стандартная логика: проверяется, равен ли Class.prototype одному из прототипов в прототипной цепочке obj.

Другими словами, сравнивается:

obj.\_\_proto\_\_ === Class.prototype?

obj.\_\_proto\_\_.\_\_proto\_\_ === Class.prototype?

obj.\_\_proto\_\_.\_\_proto\_\_.\_\_proto\_\_ === Class.prototype?

...

// если какой-то из ответов true - возвратить true

// если дошли до конца цепочки - false

В примере выше rabbit.\_\_proto\_\_ === Rabbit.prototype, так что результат будет получен немедленно. В случае с наследованием, совпадение будет на втором шаге:

class Animal {}

class Rabbit extends Animal {}

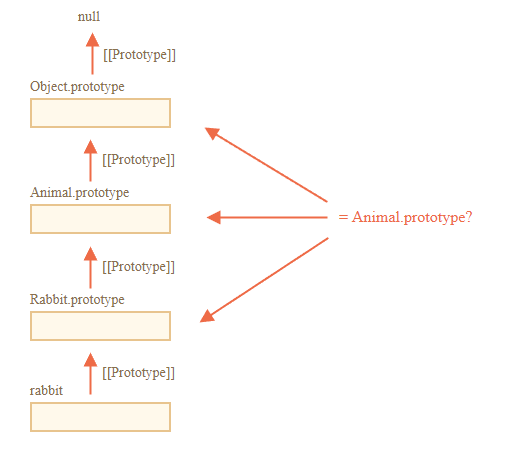
let rabbit = new Rabbit();

alert(rabbit instanceof Animal); // true

// rabbit.\_\_proto\_\_ === Rabbit.prototype

// rabbit.\_\_proto\_\_.\_\_proto\_\_ === Animal.prototype

Вот иллюстрация того как rabbit instanceof Animal сравнивается с Animal.prototype:



Есть метод [objA.isPrototypeOf(objB)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/object/isPrototypeOf), который возвращает true, если объект objA есть где-то в прототипной цепочке объекта objB. Так что obj instanceof Class можно перефразировать как Class.prototype.isPrototypeOf(obj). Сам конструктор Class не участвует в процессе проверки. Важна только цепочка прототипов Class.prototype. Это может приводить к интересным последствиям при изменении свойства prototype после создания объекта. Как, например, тут:

function Rabbit() {}

let rabbit = new Rabbit();

// заменяем прототип

Rabbit.prototype = {};

alert( rabbit instanceof Rabbit ); // false

Известно, что обычные объекты преобразуется к строке как [object Object]:

let obj = {};

alert(obj); // [object Object]

alert(obj.toString()); // то же самое

Так работает реализация метода toString. Но у toString имеются скрытые возможности, которые делают метод гораздо более мощным. Можно использовать его как расширенную версию typeof и как альтернативу instanceof. Согласно [спецификации](https://tc39.github.io/ecma262/#sec-object.prototype.tostring) встроенный метод toString может бы позаимствован у объекта и вызван в контексте любого другого значения. И результат зависит от типа этого значения.

* для числа это будет [object Number];
* для булева типа это будет [object Boolean];
* для null: [object Null];
* для undefined: [object Undefined];
* для массивов: [object Array] и т.д.

Например:

let objectToString = Object.prototype.toString;

let arr = [];

alert( objectToString.call(arr) ); // [object Array]

В примере использовался [call](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Function/call), чтобы выполнить функцию objectToString в контексте this=arr. Внутри, алгоритм метода toString анализирует контекст вызова this и возвращает соответствующий результат. Пример:

let s = Object.prototype.toString;

alert( s.call(123) ); // [object Number]

alert( s.call(null) ); // [object Null]

alert( s.call(alert) ); // [object Function]

**Свойство** [**Symbol.toStringTag**](https://learn.javascript.ru/instanceof#symbol-tostringtag)

Поведение метода объектов toString можно настраивать, используя специальное свойство объекта Symbol.toStringTag. Например:

let user = {

[Symbol.toStringTag]: "User"

};

alert( {}.toString.call(user) ); // [object User]

Такое свойство есть у большей части объектов, специфичных для определённых окружений. Вот несколько примеров для браузера:

alert( window[Symbol.toStringTag]); // window

alert( XMLHttpRequest.prototype[Symbol.toStringTag] ); // XMLHttpRequest

alert( {}.toString.call(window) ); // [object Window]

alert( {}.toString.call(new XMLHttpRequest()) ); // [object XMLHttpRequest]

Как видно, результат – это значение Symbol.toStringTag (если он имеется) обёрнутое в [object ...]. В итоге получили typeof, который не только работает с примитивными типами данных, но также и со встроенными объектами, и даже может быть настроен.

Можно использовать {}.toString.call вместо instanceof для встроенных объектов, когда надо получить тип в виде строки, а не просто сделать проверку.

1. [**Примеси**](https://learn.javascript.ru/mixins)

В JavaScript можно наследовать только от одного объекта. Объект имеет единственный [[Prototype]]. И класс может расширить только один другой класс. Иногда это ограничивает разработчика. Например, есть класс User, который реализует пользователей, и класс EventEmitter, реализующий события. Надо добавить функционал класса EventEmitter к User, чтобы пользователи могли легко генерировать события.

Для таких случаев существуют «примеси». По определению из Википедии, [примесь](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D0%BC%D0%B5%D1%81%D1%8C_(%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 класс, методы которого предназначены для использования в других классах, причём без наследования от примеси. Другими словами, примесь определяет методы, которые реализуют определённое поведение. Примесь не используется сама по себе, а используется, чтобы добавить функционал другим классам.

Простейший способ реализовать примесь в JavaScript – это создать объект с полезными методами, которые затем могут быть легко добавлены в прототип любого класса. В примере ниже примесь sayHiMixin имеет методы для придания объектам класса User возможности вести разговор:

let sayHiMixin = {

sayHi() {

alert(`Привет, ${this.name}`);

},

sayBye() {

alert(`Пока, ${this.name}`);

}

};

// использование:

class User {

constructor(name) {

this.name = name;

}

}

// копируем методы

Object.assign(User.prototype, sayHiMixin);

new User("Вася").sayHi(); // Привет, Вася!

Это не наследование, а просто копирование методов. Таким образом, класс User может наследовать от другого класса, но при этом также включать в себя примеси, «подмешивающие» другие методы, например:

class User extends Person {

// ...

}

Object.assign(User.prototype, sayHiMixin);

Примеси могут наследовать друг друга. В примере ниже sayHiMixin наследует от sayMixin:

let sayMixin = {

say(phrase) {

alert(phrase);

}

};

let sayHiMixin = {

\_\_proto\_\_: sayMixin,

sayHi() {

// вызываем метод родителя

super.say(`Привет, ${this.name}`); // (\*)

},

sayBye() {

super.say(`Пока, ${this.name}`); // (\*)

}

};

class User {

constructor(name) {

this.name = name;

}

}

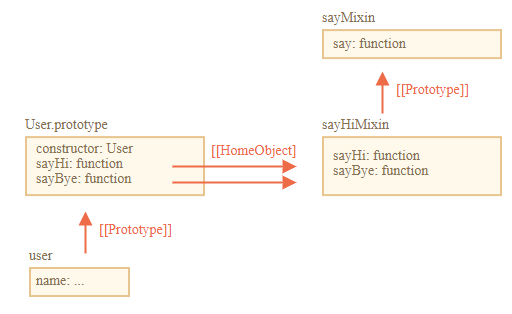
// копируем методы

Object.assign(User.prototype, sayHiMixin);

new User("Вася").sayHi(); // Привет, Вася!

Обратите внимание, что при вызове родительского метода super.say() из sayHiMixin (строки, помеченные (\*)) этот метод ищется в прототипе самой примеси, а не класса.

Диаграмма:



Это связано с тем, что методы sayHi и sayBye были изначально созданы в объекте sayHiMixin. Несмотря на то, что они скопированы, их внутреннее свойство [[HomeObject]] ссылается на sayHiMixin, как показано на картинке выше. Так как super ищет родительские методы в [[HomeObject]].[[Prototype]], это означает sayHiMixin.[[Prototype]], а не User.[[Prototype]].

[**EventMixin**](https://learn.javascript.ru/mixins#eventmixin)

Многие объекты в браузерной разработке (и не только) обладают важной способностью – они могут генерировать события. События – отличный способ передачи информации всем, кто в ней заинтересован. Создадим примесь, которая позволит легко добавлять функционал по работе с событиями любым классам/объектам.

* Примесь добавит метод .trigger(name, [data]) для генерации события. Аргумент name – это имя события, за которым могут следовать другие аргументы с данными для события.
* Также будет добавлен метод .on(name, handler), который назначает обработчик для события с заданным именем. Обработчик будет вызван, когда произойдёт событие с указанным именем name, и получит данные из .trigger.
* Будет добавлен метод .off(name, handler), который удаляет обработчик указанного события.

После того, как все методы примеси будут добавлены, объект user сможет сгенерировать событие "login" после входа пользователя в личный кабинет. А другой объект, к примеру, calendar сможет использовать это событие, чтобы показывать зашедшему пользователю актуальный для него календарь. Или menu может генерировать событие "select", когда элемент меню выбран, а другие объекты могут назначать обработчики, чтобы реагировать на это событие, и т.п. Код примеси:

let eventMixin = {

/\*\*

\* Подписаться на событие, использование:

\* menu.on('select', function(item) { ... }

\*/

on(eventName, handler) {

if (!this.\_eventHandlers) this.\_eventHandlers = {};

if (!this.\_eventHandlers[eventName]) {

this.\_eventHandlers[eventName] = [];

}

this.\_eventHandlers[eventName].push(handler);

},

/\*\*

\* Отменить подписку, использование:

\* menu.off('select', handler)

\*/

off(eventName, handler) {

let handlers = this.\_eventHandlers && this.\_eventHandlers[eventName];

if (!handlers) return;

for (let i = 0; i < handlers.length; i++) {

if (handlers[i] === handler) {

handlers.splice(i--, 1);

}

}

},

/\*\*

\* Сгенерировать событие с указанным именем и данными

\* this.trigger('select', data1, data2);

\*/

trigger(eventName, ...args) {

if (!this.\_eventHandlers || !this.\_eventHandlers[eventName]) {

return; // обработчиков для этого события нет

}

// вызов обработчиков

this.\_eventHandlers[eventName].forEach(handler => handler.apply(this, args));

}

};

Использование:

class Menu {

choose(value) {

this.trigger("select", value);

}

}

Object.assign(Menu.prototype, eventMixin);

let menu = new Menu();

// Добавить обработчик, который будет вызван при событии "select":

menu.on("select", value => alert(`Выбранное значение: ${value}`));

// Генерирует событие => обработчик выше запускается и выводит:

menu.choose("123"); // Выбранное значение: 123

Теперь если есть код, заинтересованный в событии "select", то он может слушать его с помощью menu.on(...). А eventMixin позволяет легко добавить такое поведение в любой класс без вмешательства в цепочку наследования.

1. **Обработка ошибок, "try..catch"**

В JavaScript есть синтаксическая конструкция try..catch, которая позволяет обрабатывать ошибки, в случае их возникновения.

[**Синтаксис «try…catch»**](https://learn.javascript.ru/try-catch#sintaksis-try-catch)

Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

try {

// код...

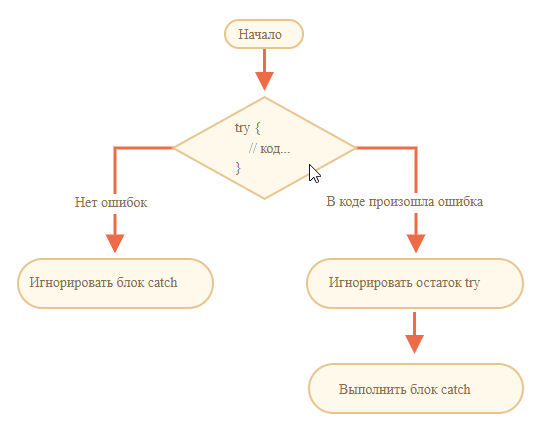
} catch (err) {

// обработка ошибки

}

Работает она так:

1. Сначала выполняется код внутри блока try {...}.
2. Если в нём нет ошибок, то блок catch(err) игнорируется: выполнение доходит до конца try и потом далее, полностью пропуская catch.
3. Если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). Переменная err (можно использовать любое имя) содержит объект ошибки с подробной информацией о произошедшем.



Таким образом, при ошибке в блоке try {…} скрипт не «падает», и есть возможность обработать ошибку внутри catch. Рассмотрим примеры.

Пример без ошибок: выведет alert (1) и (2):

try {

alert('Начало блока try'); // (1) <--

// код без ошибок

alert('Конец блока try'); // (2) <--

} catch(err) {

alert('Catch игнорируется, так как нет ошибок'); // (3)

}

Пример с ошибками: выведет (1) и (3):

try {

alert('Начало блока try'); // (1) <--

lalala; // ошибка, переменная не определена

alert('Конец блока try (никогда не выполнится)'); // (2)

} catch(err) {

alert(`Возникла ошибка!`); // (3) <--

}

Чтобы try..catch работал, код должен быть выполнимым. Другими словами, это должен быть корректный JavaScript-код. Он не сработает, если код синтаксически неверен, например, содержит несовпадающее количество фигурных скобок:

try {

{{{{{{{{{{{{

} catch(e) {

alert("Движок не может понять этот код, он не корректен");

}

JavaScript-движок сначала читает код, а затем исполняет его. Ошибки, которые возникают во время фазы чтения, называются ошибками парсинга. Их нельзя обработать (изнутри этого кода), потому что движок не понимает код. Таким образом, try..catch может обрабатывать только ошибки, которые возникают в корректном коде. Такие ошибки называют «ошибками во время выполнения», а иногда «исключениями».

Исключение, которое произойдёт в коде, запланированном «на будущее», например в setTimeout, try..catch не поймает:

try {

setTimeout(function() {

noSuchVariable; // скрипт «упадёт» тут

}, 1000);

} catch (e) {

alert( "не сработает" );

}

Это потому, что функция выполняется позже, когда движок уже покинул конструкцию try..catch. Чтобы поймать исключение внутри запланированной функции, try..catch должен находиться внутри самой этой функции:

setTimeout(function() {

try {

noSuchVariable; // try..catch обрабатывает ошибку

} catch {

alert( "ошибка поймана!" );

}

}, 1000);

[**Объект ошибки**](https://learn.javascript.ru/try-catch#obekt-oshibki)

Когда возникает ошибка, JavaScript генерирует объект, содержащий её детали. Затем этот объект передаётся как аргумент в блок catch:

try {

// ...

} catch(err) { // объект ошибки, можно использовать другое название вместо err

// ...

}

Для всех встроенных ошибок этот объект имеет два основных свойства:

* name – имя ошибки. Например, для неопределённой переменной это "ReferenceError".
* message – текстовое сообщение о деталях ошибки.

В большинстве окружений доступны и другие, нестандартные свойства. Одно из самых широко используемых и поддерживаемых – это stack – текущий стек вызова: строка, содержащая информацию о последовательности вложенных вызовов, которые привели к ошибке. Используется в целях отладки.

Например:

try {

lalala; // ошибка, переменная не определена

} catch(err) {

alert(err.name); // ReferenceError

alert(err.message); // lalala is not defined

alert(err.stack); // ReferenceError: lalala is not defined at (...стек вызовов)

// Ошибка приводится к строке вида "name: message"

alert(err); // ReferenceError: lalala is not defined

}

[**Блок «catch» без переменной**](https://learn.javascript.ru/try-catch#blok-catch-bez-peremennoy)

Эта возможность была добавлена в язык недавно. В старых браузерах может понадобиться полифилл. Если не нужны детали ошибки, в catch можно её пропустить:

try {

// ...

} catch { // без (err)

// ...

}

Рассмотрим реальные случаи использования try..catch. Как известно, JavaScript поддерживает метод [JSON.parse(str)](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse) для чтения JSON. Обычно он используется для декодирования данных, полученных по сети, от сервера или из другого источника:

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

let user = JSON.parse(json);

alert( user.name ); // John

alert( user.age ); // 30

Если json некорректен, JSON.parse генерирует ошибку, то есть скрипт «падает». Это значит, что если вдруг что-то не так с данными, то посетитель никогда (если, конечно, не откроет консоль) об этом не узнает. Используем try..catch для обработки ошибки:

let json = "{ некорректный JSON }";

try {

let user = JSON.parse(json); // ошибка

alert( user.name ); // не сработает

} catch (e) {

// выполнение «прыгает» сюда

alert( "Извините, в данных ошибка, мы попробуем получить их ещё раз." );

alert( e.name );

alert( e.message );

}

Здесь использовался блок catch только для вывода сообщения, но также можно сделать гораздо больше: отправить новый сетевой запрос, предложить посетителю альтернативный способ, отослать информацию об ошибке на сервер для логирования,

[**Генерация собственных ошибок**](https://learn.javascript.ru/try-catch#generatsiya-sobstvennyh-oshibok)

Возможна ситуация, когда json синтаксически корректен, но не содержит необходимого свойства name. Например, так:

let json = '{ "age": 30 }';

try {

let user = JSON.parse(json);

alert( user.name ); // нет свойства name

} catch (e) {

alert( "не выполнится" );

}

Здесь JSON.parse выполнится без ошибок, но на самом деле отсутствие свойства name это ошибка. Для того, чтобы унифицировать обработку ошибок, необходимо воспользоваться оператором throw.

[**Оператор «throw»**](https://learn.javascript.ru/try-catch#operator-throw)

Оператор throw генерирует ошибку.

Синтаксис:

throw <объект ошибки>

Технически в качестве объекта ошибки можно передать что угодно. Это может быть даже примитив, число или строка, но всё же лучше, чтобы это был объект, желательно со свойствами name и message (для совместимости со встроенными ошибками).

В JavaScript есть множество встроенных конструкторов для стандартных ошибок: Error, SyntaxError, ReferenceError, TypeError и другие. Можно использовать и их для создания объектов ошибки. Их синтаксис:

let error = new Error(message);

// или

let error = new SyntaxError(message);

let error = new ReferenceError(message);

// ...

Для встроенных ошибок (не для любых объектов, только для ошибок), свойство name – это в точности имя конструктора. А свойство message берётся из аргумента. Например:

let error = new Error(" Ого, ошибка! o\_O");

alert(error.name); // Error

alert(error.message); // Ого, ошибка! o\_O

Посмотрим, какую ошибку генерирует JSON.parse:

try {

JSON.parse("{ bad json o\_O }");

} catch(e) {

alert(e.name); // SyntaxError

alert(e.message); // Unexpected token o in JSON at position 0

}

Как видно, это SyntaxError. В случае отсутствие свойства name – это ошибка, ведь пользователи должны иметь имена. Сгенерируем её:

let json = '{ "age": 30 }';

try {

let user = JSON.parse(json);

if (!user.name) {

throw new SyntaxError("Данные неполны: нет имени"); // (\*)

}

alert( user.name );

} catch(e) {

alert( "JSON Error: " + e.message ); // JSON Error: Данные неполны: нет имени

}

В строке (\*) оператор throw генерирует ошибку SyntaxError с сообщением message. Точно такого же вида, как генерирует сам JavaScript. Выполнение блока try немедленно останавливается, и поток управления прыгает в catch. Теперь блок catch становится единственным местом для обработки всех ошибок: и для JSON.parse и для других случаев.

**[Проброс исключения](https://learn.javascript.ru/try-catch" \l "probros-isklyucheniya)**

В примере выше использовалась конструкция try..catch для обработки некорректных данных. Предположим, что в блоке try {...}возникнет другая неожиданная ошибка, например, программная (неопределённая переменная) или какая-то ещё, а не ошибка, связанная с некорректными данными. Пример:

let json = '{ "age": 30 }';

try {

user = JSON.parse(json); // забыл добавить "let" перед user

// ...

} catch(err) {

alert("JSON Error: " + err); // JSON Error: ReferenceError: user is not defined

// (не JSON ошибка на самом деле)

}

В рассматриваемом примере catch получает из try ошибку, не связанную с данными, но показывает то же самое сообщение "JSON Error". Это неправильно и затрудняет отладку кода. Есть способ выяснить, какая ошибка была получена, например, по её свойству name:

try {

user = { /\*...\*/ };

} catch(e) {

alert(e.name); // "ReferenceError" из-за неопределённой переменной

}

Есть простое правило: блок catch должен обрабатывать только те ошибки, которые ему известны, и «пробрасывать» все остальные.

Техника «проброс исключения» выглядит так:

1. Блок catch получает все ошибки.
2. В блоке catch(err) {...} анализируется объект ошибки err.
3. Если неизвестно как её обработать, тогда генерируется throw err.

В коде ниже используется проброс исключения, catch обрабатывает только SyntaxError:

let json = '{ "age": 30 }';

try {

let user = JSON.parse(json);

if (!user.name) {

throw new SyntaxError("Данные неполны: нет имени");

}

blabla(); // неожиданная ошибка

alert( user.name );

} catch(e) {

if (e.name == "SyntaxError") {

alert( "JSON Error: " + e.message );

} else {

throw e; // rethrow (\*)

}

}

Ошибка в строке (\*) из блока catch «выпадает наружу» и может быть поймана другой внешней конструкцией try..catch (если есть), или «убьёт» скрипт. Таким образом, блок catch фактически обрабатывает только те ошибки, с которыми он знает, как справляться, и пропускает остальные. Пример ниже демонстрирует, как такие ошибки могут быть пойманы с помощью ещё одного уровня try..catch:

function readData() {

let json = '{ "age": 30 }';

try {

// ...

blabla(); // ошибка

} catch (e) {

// ...

if (e.name != 'SyntaxError') {

throw e; // проброс исключения

}

}

}

try {

readData();

} catch (e) {

alert( "Внешний catch поймал: " + e ); // поймал

}

В рассматриваемом примере в readData обрабатываются только SyntaxError, тогда как внешний блок try..catch обрабатывает всё.

**Конструкция** [**try…catch…finally**](https://learn.javascript.ru/try-catch#try-catch-finally)

Конструкция try..catch может содержать ещё одну секцию: finally. Если секция есть, то она выполняется в любом случае:

* после try, если не было ошибок,
* после catch, если ошибки были.

Расширенный синтаксис выглядит следующим образом:

try {

//... пробуем выполнить код...

} catch(e) {

//... обрабатываем ошибки ...

} finally {

//... выполняем всегда ...

}

Рассмотрим следующий код:

try {

alert( 'try' );

if (confirm('Сгенерировать ошибку?')) BAD\_CODE();

} catch (e) {

alert( 'catch' );

} finally {

alert( 'finally' );

}

У рассматриваемого кода есть два пути выполнения:

1. Если ответить на вопрос «Сгенерировать ошибку?» утвердительно, то try -> catch -> finally.
2. Если ответить отрицательно, то try -> finally.

Секцию finally часто используют, если надо что-то выполнить вне зависимости от того, будет ошибка или нет. Например, надо измерить время, которое занимает функция чисел Фибоначчи fib(n). Естественно, можно начать измерения до того, как функция начнёт выполняться и закончить после. Но если при вызове функции возникла ошибка, (в частности, реализация fib(n) в коде ниже возвращает ошибку для отрицательных и для нецелых чисел), то следует использовать секцию finally, которая завершит измерения несмотря ни на что.

Здесь finally гарантирует, что время будет измерено корректно в обеих ситуациях – и в случае успешного завершения fib и в случае ошибки:

let num = +prompt("Введите положительное целое число?", 35)

let diff, result;

function fib(n) {

if (n < 0 || Math.trunc(n) != n) {

throw new Error("Должно быть целое неотрицательное число");

}

return n <= 1 ? n : fib(n - 1) + fib(n - 2);

}

let start = Date.now();

try {

result = fib(num);

} catch (e) {

result = 0;

} finally {

diff = Date.now() - start;

}

alert(result || "возникла ошибка");

alert( `Выполнение заняло ${diff}ms` );

Это можно проверить, запустив этот код и введя 35 в prompt – код завершится нормально, finally выполнится после try. Затем ввести -1 – незамедлительно произойдёт ошибка, выполнение займёт 0ms. Оба измерения выполняются корректно. Другими словами, неважно как завершилась функция: через return или throw. Секция finally срабатывает в обоих случаях.

Переменные внутри try..catch..finally локальны. Обратите внимание, что переменные result и diff в коде выше объявлены до try..catch. Если переменную объявить в блоке, например, в try, то она не будет доступна после него.

Блок finally срабатывает при любом выходе из try..catch, в том числе и return. В примере ниже из try происходит return, но finally получает управление до того, как контроль возвращается во внешний код.

function func() {

try {

return 1;

} catch (e) {

/\* ... \*/

} finally {

alert( 'finally' );

}

}

alert( func() ); // сначала срабатывает alert из finally, а затем этот код

Конструкция try..finally без секции catch также полезна. Стоит применять её, когда нет необходимости обрабатывать ошибки (пусть выпадут), но надо убедиться, что начатые процессы завершились.

function func() {

// начать делать что-то, что требует завершения

try {

// ...

} finally {

// завершить это, даже если все упадёт

}

}

В приведённом выше коде ошибка всегда выпадает наружу, потому что тут нет блока catch. Но finally отрабатывает до того, как поток управления выйдет из функции.

1. **Пользовательские ошибки, расширение Error**

Разработчикам часто необходимы собственные классы ошибок для разных задач. Для ошибок при работе с сетью может понадобиться HttpError, для операций с базой данных DbError, для поиска – NotFoundError и т.д. Ошибки должны поддерживать базовые свойства, такие как message, name и, желательно, stack. Но также они могут иметь свои собственные свойства. Например, объекты HttpError могут иметь свойство statusCode со значениями 404, 403 или 500.

JavaScript позволяет вызывать throw с любыми аргументами, то есть технически пользовательские классы ошибок не нуждаются в наследовании от Error. Но если использовать наследование, то появляется возможность идентификации объектов ошибок посредством obj instanceof Error. Так что лучше применять наследование.

По мере роста приложения, наши собственные ошибки образуют иерархию, например, HttpTimeoutError может наследовать от HttpError и так далее.

[**Расширение Error**](https://learn.javascript.ru/custom-errors#rasshirenie-error)

В качестве примера рассмотрим функцию readUser(json), которая должна читать данные пользователя в формате JSON. Пример того, как может выглядеть корректный json:

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

Внутри используется JSON.parse. При получении некорректного json он будет генерировать ошибку SyntaxError. Но даже если json синтаксически верен, то это не значит, что это будет корректный пользователь. Могут быть пропущены необходимые данные. Например, могут отсутствовать свойства name и age, которые являются необходимыми для наших пользователей. Функция readUser(json) будет не только читать JSON-данные, но и проверять их («валидировать»). Если необходимые поля отсутствуют или данные в неверном формате, то это будет ошибкой. Но не синтаксической ошибкой SyntaxError, потому что данные синтаксически корректны. Это будет другая ошибка.

Назовем её ошибкой валидации ValidationError и создадим для неё класс. Ошибка этого вида должна содержать информацию о поле, которое является источником ошибки. Класс ValidationError должен наследовать от встроенного класса Error. Класс Error встроенный, вот его примерный код, чтобы понимать, что расширяется:

class Error {

constructor(message) {

this.message = message;

this.name = "Error";

this.stack = <стек вызовов>;

}

}

Теперь унаследуем от него ValidationError и попробуем новый класс в действии:

class ValidationError extends Error {

constructor(message) {

super(message); // (1)

this.name = "ValidationError"; // (2)

}

}

function test() {

throw new ValidationError("Упс!");

}

try {

test();

} catch(err) {

alert(err.message);

alert(err.name); // ValidationError

alert(err.stack); // список вложенных вызовов с номерами строк для каждого

}

Обратите внимание: в строке (1) вызывается родительский конструктор. JavaScript требует вызов super в дочернем конструкторе, так что это обязательно. Родительский конструктор устанавливает свойство message. Родительский конструктор также устанавливает свойство name для "Error", поэтому в строке (2) оно сбрасывается на правильное значение. Попробуем использовать его в readUser(json):

class ValidationError extends Error {

constructor(message) {

super(message);

this.name = "ValidationError";

}

}

// Использование

function readUser(json) {

let user = JSON.parse(json);

if (!user.age) {

throw new ValidationError("Нет поля: age");

}

if (!user.name) {

throw new ValidationError("Нет поля: name");

}

return user;

}

// Рабочий пример с try..catch

try {

let user = readUser('{ "age": 25 }');

} catch (err) {

if (err instanceof ValidationError) {

alert("Некорректные данные: " + err.message); // Некорректные данные: Нет поля: name

} else if (err instanceof SyntaxError) { // (\*)

alert("JSON Ошибка Синтаксиса: " + err.message);

} else {

throw err; // неизвестная ошибка, пробросить исключение (\*\*)

}

}

Блок try..catch в коде выше обрабатывает и нашу ValidationError, и встроенную SyntaxError из JSON.parse. Обратите внимание, как используется instanceof для проверки конкретного типа ошибки в строке (\*). Можно также проверить тип, используя err.name:

// ...

// вместо (err instanceof SyntaxError)

} else if (err.name == "SyntaxError") { // (\*)

// ...

Версия с instanceof гораздо лучше, потому что в будущем можно расширить ValidationError, сделав его подтипы, такие как PropertyRequiredError. И проверка instanceof продолжит работать для новых наследованных классов. Также важно, что если catch встречает неизвестную ошибку, то он пробрасывает её в строке (\*\*). Блок catch знает, только как обрабатывать ошибки валидации и синтаксические ошибки, а другие виды ошибок (из-за опечаток в коде и др.) он должен пропустить.

Класс ValidationError является общим и не учитывает ряд ошибок. Например, отсутствие свойства или неверный формат (например, строка как значение возраста age). Поэтому для отсутствующих свойств сделаем более конкретный класс PropertyRequiredError. Он будет нести дополнительную информацию о свойстве, которое отсутствует.

class ValidationError extends Error {

constructor(message) {

super(message);

this.name = "ValidationError";

}

}

class PropertyRequiredError extends ValidationError {

constructor(property) {

super("Нет свойства: " + property);

this.name = "PropertyRequiredError";

this.property = property;

}

}

// Применение

function readUser(json) {

let user = JSON.parse(json);

if (!user.age) {

throw new PropertyRequiredError("age");

}

if (!user.name) {

throw new PropertyRequiredError("name");

}

return user;

}

// Рабочий пример с try..catch

try {

let user = readUser('{ "age": 25 }');

} catch (err) {

if (err instanceof ValidationError) {

alert("Неверные данные: " + err.message); // Неверные данные: Нет свойства: name

alert(err.name); // PropertyRequiredError

alert(err.property); // name

} else if (err instanceof SyntaxError) {

alert("Ошибка синтаксиса JSON: " + err.message);

} else {

throw err; // неизвестная ошибка, повторно выбросит исключение

}

}

Новый класс PropertyRequiredError очень просто использовать: необходимо указать только имя свойства new PropertyRequiredError(property). Сообщение для пользователя message генерируется конструктором. Обратите внимание, что свойство this.name в конструкторе PropertyRequiredError снова присвоено вручную. Правда, немного утомительно – присваивать this.name = <class name> в каждом классе пользовательской ошибки. Можно этого избежать, если сделать собственный «базовый» класс ошибки MyError, который будет ставить this.name = this.constructor.name. И затем наследовать все ошибки уже от него. Вот упрощённый код с MyError и другими пользовательскими классами ошибок:

class MyError extends Error {

constructor(message) {

super(message);

this.name = this.constructor.name;

}

}

class ValidationError extends MyError { }

class PropertyRequiredError extends ValidationError {

constructor(property) {

super("Нет свойства: " + property);

this.property = property;

}

}

// name корректное

alert( new PropertyRequiredError("field").name ); // PropertyRequiredError

Теперь пользовательские ошибки стали намного короче, особенно ValidationError, так как нет строки "this.name = ..." в конструкторе.

[**Обёртывание исключений**](https://learn.javascript.ru/custom-errors#obyortyvanie-isklyucheniy)

Назначение функции readUser в приведенном выше коде – это «чтение данных пользователя». В процессе могут возникнуть различные виды ошибок. Сейчас есть SyntaxError и ValidationError, но в будущем функция readUser может расшириться и, возможно, генерировать другие виды ошибок. Код, который вызывает readUser, должен обрабатывать эти ошибки.

Сейчас в нём используются проверки if в блоке catch, которые проверяют класс и обрабатывают известные ошибки и пробрасывают дальше неизвестные. Но если функция readUser генерирует несколько видов ошибок, не стоит проверять все типы ошибок поодиночке во всех местах в коде, где вызывается readUser. Лучше обработать какую-то обобщённую ошибку чтения данных.

Создадим новый класс ReadError для представления таких ошибок. Если ошибка возникает внутри readUser, её надо перехватить и сгенерировать ReadError. Также ссылка на исходную ошибку будет сохранена в свойстве cause. Тогда внешний код должен будет только проверить наличие ReadError. Этот код определяет ошибку ReadError и демонстрирует её использование в readUserи try..catch:

class ReadError extends Error {

constructor(message, cause) {

super(message);

this.cause = cause;

this.name = 'ReadError';

}

}

class ValidationError extends Error { /\*...\*/ }

class PropertyRequiredError extends ValidationError { /\* ... \*/ }

function validateUser(user) {

if (!user.age) {

throw new PropertyRequiredError("age");

}

if (!user.name) {

throw new PropertyRequiredError("name");

}

}

function readUser(json) {

let user;

try {

user = JSON.parse(json);

} catch (err) {

if (err instanceof SyntaxError) {

throw new ReadError("Синтаксическая ошибка", err);

} else {

throw err;

}

}

try {

validateUser(user);

} catch (err) {

if (err instanceof ValidationError) {

throw new ReadError("Ошибка валидация", err);

} else {

throw err;

}

}

}

try {

readUser('{bad json}');

} catch (e) {

if (e instanceof ReadError) {

alert(e);

// Исходная ошибка: SyntaxError:Unexpected token b in JSON at position 1

alert("Исходная ошибка: " + e.cause);

} else {

throw e;

}

}

В приведённом выше коде readUser работает так, как описано – функция распознаёт синтаксические ошибки и ошибки валидации и выдаёт вместо них ошибки ReadError (неизвестные ошибки, как обычно, пробрасываются). Внешний код проверяет только instanceof ReadError. Не нужно перечислять все возможные типы ошибок

Этот подход называется «обёртывание исключений», потому что «исключения низкого уровня» «оборачиваются» в ReadError, который является более абстрактным и более удобным для использования в вызывающем коде. Такой подход широко используется в объектно-ориентированном программировании.

1. **Модули: введение.**

По мере роста приложения, обычно возникает необходимость разделить его на много файлов, так называемых «модулей». Модуль обычно содержит класс или библиотеку с функциями. Долгое время в JavaScript отсутствовал синтаксис модулей на уровне языка. Это не было проблемой, потому что первые скрипты были маленькими и простыми. В модулях не было необходимости. Но со временем скрипты становились всё более и более сложными, поэтому сообщество придумало несколько вариантов организации кода в модули. Появились библиотеки для динамической подгрузки модулей. Например:

* [AMD](https://ru.wikipedia.org/wiki/Asynchronous_module_definition) – одна из самых старых модульных систем, изначально реализована библиотекой [require.js](http://requirejs.org/).
* [CommonJS](http://wiki.commonjs.org/wiki/Modules/1.1) – модульная система, созданная для сервера Node.js.
* [UMD](https://github.com/umdjs/umd) – ещё одна модульная система, предлагается как универсальная, совместима с AMD и CommonJS.

Теперь все они постепенно становятся частью истории, хотя их и можно найти в старых скриптах.

Система модулей на уровне языка появилась в стандарте JavaScript в 2015 году и постепенно эволюционировала. На данный момент она поддерживается большинством браузеров и Node.js.

Модуль – это файл с кодом. Один скрипт – это один модуль. Модули могут загружать друг друга и использовать директивы export и import, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого:

* export отмечает переменные и функции, которые должны быть доступны вне текущего модуля.
* import позволяет импортировать функциональность из других модулей.

Например, если есть файл sayHi.js, который экспортирует функцию:

//sayHi.js

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

Тогда другой файл может импортировать её и использовать:

// main.js

import {sayHi} from './sayHi.js';

alert(sayHi); // function...

sayHi('John'); // Hello, John!

Директива import загружает модуль по пути ./sayHi.js относительно текущего файла и записывает эксортированную функцию sayHi в соответствующую переменную. Так как модули поддерживают ряд специальных ключевых слов, и у них есть ряд особенностей, то необходимо явно сказать браузеру, что скрипт является модулем, при помощи атрибута <script type="module">. Вот так:

// say.js

export function sayHi(user) {

return `Hello, ${user}!`;

}

// index.html

<!doctype html>

<script type="module">

import {sayHi} from './say.js';

document.body.innerHTML = sayHi('John');

</script>

Браузер автоматически загрузит и запустит импортированный модуль (и те, которые он импортирует, если надо), а затем запустит скрипт.



[**Основные возможности модулей**](https://learn.javascript.ru/modules-intro#osnovnye-vozmozhnosti-moduley)

Есть основные возможности и особенности, работающие как в браузере, так и в серверном JavaScript.

В модулях всегда используется режим use strict. Например, присваивание к необъявленной переменной вызовет ошибку.

<script type="module">

a = 5; // ошибка

</script>

Каждый модуль имеет свою собственную область видимости. Другими словами, переменные и функции, объявленные в модуле, не видны в других скриптах.

В следующем примере импортированы 2 скрипта, и hello.js пытается использовать переменную user, объявленную в user.js. В итоге ошибка:



// hello.js

alert(user);

// user.js

let user = "John";

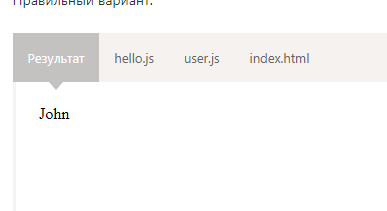
// index.html

<!doctype html>

<script type="module" src="user.js"></script>

<script type="module" src="hello.js"></script>

Модули должны экспортировать функционал, предназначенный для использования извне. А другие модули могут его импортировать. Так что надо импортировать user.js в hello.js и взять из него нужный функционал, вместо того чтобы полагаться на глобальные переменные. Правильный вариант:



// hello.js

import {user} from './user.js';

document.body.innerHTML = user; // John

// user.js

export let user = "John";

// index.html

import {user} from './user.js';

document.body.innerHTML = user; // John

В браузере также существует независимая область видимости для каждого скрипта <script type="module">:

<script type="module">

// Переменная доступна только в этом модуле

let user = "John";

</script>

<script type="module">

alert(user); // Error: user is not defined

</script>

Если нужно сделать глобальную переменную уровня всей страницы, можно явно присвоить её объекту window, тогда получить значение переменной можно обратившись к window.user. Но это должно быть исключением, требующим веской причины.

Если один и тот же модуль используется в нескольких местах, то его код выполнится только один раз, после чего экспортируемая функциональность передаётся всем импортёрам. Это очень важно для понимания работы модулей. Рассмотрим примеры.

Во-первых, если при запуске модуля возникают побочные эффекты, например, выдаётся сообщение, то импорт модуля в нескольких местах покажет его только один раз – при первом импорте:

// alert.js

alert("Модуль выполнен!");

// Импорт одного и того же модуля в разных файлах

// 1.js

import `./alert.js`; // Модуль выполнен!

// 2.js

import `./alert.js`; // (ничего не покажет)

На практике, задача кода модуля – это обычно инициализация, создание внутренних структур данных, а если надо, чтобы что-то можно было использовать много раз, то экспортируем это.

Рассмотрим более сложный пример. Представим, что модуль экспортирует объект:

// admin.js

export let admin = {

name: "John"

};

Если модуль импортируется в нескольких файлах, то код модуля будет выполнен только один раз, объект admin будет создан и в дальнейшем будет передан всем импортёрам. Все импортёры получат один-единственный объект admin:

// 1.js

import {admin} from './admin.js';

admin.name = "Pete";

// 2.js

import {admin} from './admin.js';

alert(admin.name); // Pete

В примере оба файла, 1.js и 2.js, импортируют один и тот же объект. Изменения, сделанные в 1.js, будут видны в 2.js. Если что-то изменится в объекте admin, то другие модули тоже увидят эти изменения. Такое поведение позволяет конфигурировать модули при первом импорте. Можно установить его свойства один раз, и в дальнейших импортах он будет уже настроенным.

Например, модуль admin.js предоставляет определённую функциональность, но ожидает передачи учётных данных в объект admin извне:

// admin.js

export let admin = { };

export function sayHi() {

alert(`Ready to serve, ${admin.name}!`);

}

В init.js, первом скрипте рассматриваемого приложения, установим admin.name. Тогда все это увидят, включая вызовы, сделанные из самого admin.js:

// init.js

import {admin} from './admin.js';

admin.name = "Pete";

Другой модуль тоже увидит admin.name:

// other.js

import {admin, sayHi} from './admin.js';

alert(admin.name); // Pete

sayHi(); // Ready to serve, Pete!

Объект import.meta содержит информацию о текущем модуле. Содержимое зависит от окружения. В браузере он содержит ссылку на скрипт или ссылку на текущую веб-страницу, если модуль встроен в HTML:

<script type="module">

alert(import.meta.url); // ссылка на html страницу для встроенного скрипта

</script>

В модуле на верхнем уровне this не определён (undefined). Это незначительная особенность, но для полноты картины нужно упомянуть об этом. Сравним с немодульными скриптами, там this – глобальный объект:

<script>

alert(this); // window

</script>

<script type="module">

alert(this); // undefined

</script>

Есть и несколько других, именно браузерных особенностей скриптов с type="module" по сравнению с обычными скриптами.

Модули всегда выполняются в отложенном (deferred) режиме, точно так же, как скрипты с атрибутом defer. Это верно и для внешних и встроенных скриптов-модулей. Другими словами:

* загрузка внешних модулей, таких как <script type="module" src="...">, не блокирует обработку HTML.
* модули, даже если загрузились быстро, ожидают полной загрузки HTML документа, и только затем выполняются.
* сохраняется относительный порядок скриптов: скрипты, которые идут раньше в документе, выполняются раньше.

Как побочный эффект, модули всегда видят полностью загруженную HTML-страницу, включая элементы под ними. Например:

<script type="module">

alert(typeof button); // object: скрипт может 'видеть' кнопку под ним

</script>

Сравним с обычным скриптом ниже:

<script>

alert(typeof button); // Ошибка: кнопка не определена, скрипт не видит элементы под ним

</script>

<button id="button">Кнопка</button>

Второй скрипт выполнится раньше, чем первый. Поэтому сначала будет выведено undefined, а потом object. Это потому, что модули начинают выполняться после полной загрузки страницы. Обычные скрипты запускаются сразу же, поэтому сообщение из обычного скрипта видно первым.

При использовании модулей стоит иметь в виду, что HTML-страница будет показана браузером до того, как выполнятся модули и JavaScript-приложение будет готово к работе. Некоторые функции могут ещё не работать. Следует разместить «индикатор загрузки» или что-то ещё, чтобы не смутить этим посетителя.

[Атрибут async работает во встроенных скриптах](https://learn.javascript.ru/modules-intro" \l "atribut-async-rabotaet-vo-vstroennyh-skriptah). Для немодульных скриптов атрибут async работает только на внешних скриптах. Скрипты с ним запускаются сразу по готовности, они не ждут другие скрипты или HTML-документ.

Для модулей атрибут async работает на любых скриптах. Например, в скрипте ниже есть async, поэтому он выполнится сразу после загрузки, не ожидая других скриптов. Скрипт выполнит импорт (загрузит ./analytics.js) и сразу запустится, когда будет готов, даже если HTML документ ещё не будет загружен, или если другие скрипты ещё загружаются. Это очень полезно, когда модуль ни с чем не связан, например, для счётчиков, рекламы, обработчиков событий.

<script async type="module">

import {counter} from './analytics.js';

counter.count();

</script>

[**Внешние скрипты**](https://learn.javascript.ru/modules-intro#vneshnie-skripty)

Внешние скрипты с атрибутом type="module" имеют два отличия:

1. Внешние скрипты с одинаковым атрибутом src запускаются только один раз. В примере ниже скрипт my.js загрузится и будет выполнен только один раз:

<script type="module" src="my.js"></script>

<script type="module" src="my.js"></script>

1. Внешний скрипт, который загружается с другого домена, требует указания заголовков [CORS](https://developer.mozilla.org/ru/docs/Web/HTTP/CORS). Другими словами, если модульный скрипт загружается с другого домена, то удалённый сервер должен установить заголовок Access-Control-Allow-Origin означающий, что загрузка скрипта разрешена.

<script type="module" src="http://another-site.com/their.js"></script>

Это обеспечивает лучшую безопасность по умолчанию.

В браузере import должен содержать относительный или абсолютный путь к модулю. Модули без пути называются «голыми» (bare). Они не разрешены в import. Например, этот import неправильный:

import {sayHi} from 'sayHi';

Другие окружения, например, Node.js, допускают использование «голых» модулей, без путей, так как в них есть свои правила, как работать с такими модулями и где их искать. Но браузеры пока не поддерживают «голые» модули.

Старые браузеры не понимают атрибут type="module". Скрипты с неизвестным атрибутом type просто игнорируются. Можно сделать для них «резервный» скрипт при помощи атрибута nomodule:

<script type="module">

alert("Работает в современных браузерах");

</script>

<script nomodule>

alert("Современные браузеры понимают оба атрибута - и type=module, и nomodule, поэтому пропускают этот тег script")

alert("Старые браузеры игнорируют скрипты с неизвестным атрибутом type=module, но выполняют этот.");

</script>

[Инструменты сборки](https://learn.javascript.ru/modules-intro#instrumenty-sborki)

В реальной жизни модули в браузерах редко используются как есть. Обычно, они объединяются вместе, специальными инструментами, например [Webpack](https://webpack.js.org/) и после выкладывается код на рабочий сервер.

Одно из преимуществ использования сборщика – он предоставляет больший контроль над тем, как модули ищутся, позволяет использовать «голые» модули и многое другое «своё», например, CSS/HTML-модули. Сборщик делает следующее:

1. Берёт «основной» модуль, который необходимо поместить в <script type="module"> в HTML.
2. Анализирует зависимости (импорты, импорты импортов и так далее).
3. Собирает один файл со всеми модулями (или несколько файлов, это можно настроить), перезаписывает встроенный import функцией импорта от сборщика, чтобы всё работало. «Специальные» типы модулей, такие как HTML/CSS тоже поддерживаются.
4. В процессе могут происходить и другие трансформации, и оптимизации кода:

* недоступный код удаляется;
* неиспользуемые экспорты удаляются («tree-shaking»);
* специфические операторы для разработки, такие как console и debugger, удаляются;
* современный синтаксис JavaScript также может быть трансформирован в предыдущий стандарт, с похожей функциональностью;
* полученный файл можно минимизировать (удалить пробелы, заменить названия переменных на более короткие и т.д.).

Если используются инструменты сборки, то они объединяют модули вместе в один или несколько файлов, и заменяют import/export на свои вызовы. Поэтому итоговую сборку (в примере ниже – bundle.js) можно подключать и без атрибута type="module", как обычный скрипт:

<script src="bundle.js"></script>

Хотя и «как есть» модули тоже можно использовать, а сборщик настроить позже при необходимости.

1. **Модули: экспорт и импорт**

Директивы экспорт и импорт имеют несколько вариантов вызова.

[**Экспорт до объявления**](https://learn.javascript.ru/import-export#eksport-do-obyavleniya)

Можно пометить любое объявление как экспортируемое, разместив export перед ним, будь то переменная, функция или класс. Например, все следующие экспорты допустимы:

// экспорт массива

export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// экспорт константы

export const MODULES\_BECAME\_STANDARD\_YEAR = 2015;

// экспорт класса

export class User {

constructor(name) {

this.name = name;

}

}

Обратите внимание, что export перед классом или функцией не делает их [функциональным выражением](https://learn.javascript.ru/function-expressions-arrows). Это всё также объявление функции, хотя и экспортируемое. Большинство руководств по стилю кода в JavaScript не рекомендуют ставить точку с запятой после объявлений функций или классов. Поэтому в конце export class и export function не нужна точка с запятой:

export function sayHi(user) {

alert(`Hello, ${user}!`);

} // без ; в конце

[**Экспорт отдельно от объявления**](https://learn.javascript.ru/import-export#eksport-otdelno-ot-obyavleniya)

Также можно написать export отдельно. Здесь сначала объявляются функции, а затем экспортируются:

// say.js

function sayHi(user) {

alert(`Hello, ${user}!`);

}

function sayBye(user) {

alert(`Bye, ${user}!`);

}

export {sayHi, sayBye}; // список экспортируемых переменных

Технически также можно расположить export выше функций.

[**Импорт \***](https://learn.javascript.ru/import-export#import)

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

// main.js

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

sayHi('John'); // Hello, John!

sayBye('John'); // Bye, John!

Но если импортировать нужно много чего, то можно импортировать всё сразу в виде объекта, используя import \* as <obj>. Например:

// main.js

import \* as say from './say.js';

say.sayHi('John');

say.sayBye('John');

На первый взгляд «импортировать всё» выглядит очень удобно, не надо писать лишнего. Но иногда надо явно перечислять список того, что нужно импортировать , тому есть несколько причин:

1. Современные инструменты сборки ([webpack](http://webpack.github.io/) и другие) собирают модули вместе и оптимизируют их, ускоряя загрузку и удаляя неиспользуемый код.

Предположим, в проект была добавлена сторонняя библиотеку say.js с множеством функций:

// say.js

export function sayHi() { ... }

export function sayBye() { ... }

export function becomeSilent() { ... }

Теперь, если из этой библиотеки в проекте использовать только одну функцию:

// main.js

import {sayHi} from './say.js';

тогда оптимизатор увидит, что другие функции не используются, и удалит остальные из собранного кода, тем самым делая код меньше. Это называется «tree-shaking».

1. Явно перечисляя то, что надо импортировать, можно получить более короткие имена функций: sayHi() вместо say.sayHi().
2. Явное перечисление импортов делает код более понятным, позволяет увидеть, что именно и где используется. Это упрощает поддержку и рефакторинг кода.

**[Импорт «как»](https://learn.javascript.ru/import-export" \l "import-kak)**

Также мжно использовать as, чтобы импортировать под другими именами. Например, для краткости импортируем sayHi в локальную переменную hi, а sayBye импортируем как bye:

// main.js

import {sayHi as hi, sayBye as bye} from './say.js';

hi('John'); // Hello, John!

bye('John'); // Bye, John!

[**Экспортировать «как»**](https://learn.javascript.ru/import-export#eksportirovat-kak)

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

// say.js

export {sayHi as hi, sayBye as bye};

Теперь hi и bye – официальные имена для внешнего кода, их нужно использовать при импорте:

// main.js

import \* as say from './say.js';

say.hi('John'); // Hello, John!

say.bye('John'); // Bye, John!

[**Экспорт по умолчанию**](https://learn.javascript.ru/import-export#eksport-po-umolchaniyu)

На практике модули встречаются в основном одного из двух типов:

1. Модуль, содержащий библиотеку или набор функций, как say.js выше.
2. Модуль, который объявляет что-то одно, например, модуль user.js экспортирует только class User.

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

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

// user.js

export default class User {

constructor(name) {

this.name = name;

}

}

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

// main.js

import User from './user.js'; // не {User}, просто User

new User('John');

Импорты без фигурных скобок выглядят красивее. Обычная ошибка начинающих: забывать про фигурные скобки. Запомним: фигурные скобки необходимы в случае именованных экспортов, для export default они не нужны.

| **Именованный экспорт** | **Экспорт по умолчанию** |
| --- | --- |
| export class User {...} | export default class User {...} |
| import {User} from ... | import User from ... |

Технически в одном модуле может быть, как экспорт по умолчанию, так и именованные экспорты, но на практике обычно их не смешивают. То есть, в модуле находятся либо именованные экспорты, либо один экспорт по умолчанию.

Так как в файле может быть максимум один export default, то экспортируемая сущность не обязана иметь имя. Например, всё это – полностью корректные экспорты по умолчанию:

export default class { // у класса нет имени

constructor() { ... }

}

export default function(user) { // у функции нет имени

alert(`Hello, ${user}!`);

}

// экспортируем значение, не создавая переменную

export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

Это нормально, потому что может быть только один export default в файле, так что import без фигурных скобок всегда знает, что импортировать. Без default такой экспорт выдал бы ошибку, так как необходимо имя класса, если это не экспорт по умолчанию:

export class { // Ошибка

constructor() {}

}

[**Имя «default»**](https://learn.javascript.ru/import-export#imya-default)

В некоторых ситуациях для обозначения экспорта по умолчанию в качестве имени используется default. Например, чтобы экспортировать функцию отдельно от её объявления:

function sayHi(user) {

alert(`Hello, ${user}!`);

}

export {sayHi as default};

Представим следующую ситуацию: модуль user.js экспортирует одну сущность «по умолчанию» и несколько именованных:

// user.js

export default class User {

constructor(name) {

this.name = name;

}

}

export function sayHi(user) {

alert(`Hello, ${user}!`);

}

Вот как импортировать экспорт по умолчанию вместе с именованным экспортом:

// main.js

import {default as User, sayHi} from './user.js';

new User('John');

Если импортируется всё как объект import \*, тогда его свойство default – как раз и будет экспортом по умолчанию:

// main.js

import \* as user from './user.js';

let User = user.default; // экспорт по умолчанию

new User('John');

[Однако, у экспорта по умолчанию](https://learn.javascript.ru/import-export#dovod-protiv-eksportov-po-umolchaniyu) есть недостатки. Именованные экспорты «включают в себя» своё имя. Эта информация является частью модуля, она сообщает, что именно экспортируется. Именованные экспорты вынуждают использовать правильное имя при импорте:

// import {MyUser} не сработает, должно быть именно имя {User}

import {User} from './user.js';

В то время как для экспорта по умолчанию можно выбрать любое имя при импорте:

import User from './user.js'; // сработает

import MyUser from './user.js'; // сработает

Так что члены команды разработчиков могут использовать разные имена для импорта одной и той же вещи, и это не очень хорошо. Обычно, чтобы избежать этого и соблюсти единообразие кода, есть правило: имена импортируемых переменных должны соответствовать именам файлов. Вот так:

import User from './user.js';

import LoginForm from './loginForm.js';

import func from '/path/to/func.js';

Тем не менее, в некоторых командах это считают серьёзным доводом против экспортов по умолчанию и предпочитают использовать именованные экспорты везде. Даже если экспортируется только одна вещь, она всё равно экспортируется с именем, без использования default. Это также немного упрощает реэкспорт.

[**Реэкспорт**](https://learn.javascript.ru/import-export#reeksport)

Синтаксис «реэкспорта» export ... from ... позволяет импортировать что-то и тут же экспортировать, даже под другим именем:

export {sayHi} from './say.js'; // реэкспортировать sayHi

export {default as User} from './user.js'; // реэкспортировать default

Рассмотрим практический пример использования. Представим, что разрабатывается «пакет»: папка со множеством модулей, из которой часть функционала экспортируется наружу (инструменты вроде NPM позволяют публиковать и распространять такие пакеты), а многие модули – просто вспомогательные, для внутреннего использования в других модулях пакета. Структура файлов может быть такой:

auth/

index.js

user.js

helpers.js

tests/

login.js

providers/

github.js

facebook.js

...

Надо сделать функционал пакета доступным через единую точку входа: «главный файл» auth/index.js. Чтобы можно было использовать его следующим образом:

import {login, logout} from 'auth/index.js'

Идея в том, что внешние разработчики, которые будут использовать пакет, не должны разбираться с его внутренней структурой. Всё, что нужно, надо экспортировать в auth/index.js, а остальное скрыть от разработчиков. Так как нужный функционал может быть разбросан по модулям пакета, то надо импортировать их в auth/index.js и тут же экспортировать наружу.

// auth/index.js

// импортировать login/logout и тут же экспортировать

import {login, logout} from './helpers.js';

export {login, logout};

// импортировать экспорт по умолчанию как User и тут же экспортировать

import User from './user.js';

export {User};

Теперь пользователи пакета могут писать import {login} from "auth/index.js". Запись export ... from ...– это просто более короткий вариант такого импорта-экспорта:

// auth/index.js

// импортировать login/logout и тут же экспортировать

export {login, logout} from './helpers.js';

// импортировать экспорт по умолчанию как User и тут же экспортировать

export {default as User} from './user.js';

[**Реэкспорт экспорта по умолчанию**](https://learn.javascript.ru/import-export#reeksport-eksporta-po-umolchaniyu)

При реэкспорте экспорт по умолчанию нужно обрабатывать особым образом. Например, есть user.js, из которого надо реэкспортировать класс User:

// user.js

export default class User {

// ...

}

1. export User from './user.js' не будет работать, возникнет синтаксическая ошибка.

Чтобы реэкспортировать экспорт по умолчанию, надо написать export {default as User}, как в примере выше.

1. export \* from './user.js' реэкспортирует только именованные экспорты, исключая экспорт по умолчанию.

Если надо реэкспортировать и именованные экспорты, и экспорт по умолчанию, то понадобятся две инструкции:

export \* from './user.js'; // для реэкспорта именованных экспортов

export {default} from './user.js'; // для реэкспорта по умолчанию

Такое особое поведение реэкспорта с экспортом по умолчанию – одна из причин того, почему их неудобно использовать.

1. **Модули: динамические импорты**

Инструкции экспорта и импорта, которые рассматривались в предыдущем вопросе, называются «статическими». Синтаксис у них весьма простой и строгий.

Во-первых, нельзя динамически задавать никакие из параметров import. Путь к модулю должен быть строковым примитивом и не может быть вызовом функции. Вот так работать не будет:

import ... from getModuleName(); // Ошибка, должна быть строка

Во-вторых, нельзя делать импорт в зависимости от условий или в процессе выполнения.

if(...) {

import ...; // Ошибка, запрещено

}

{

import ...; // Ошибка, нельзя ставить импорт в блок

}

Всё это следствие того, что цель директив import/export – задать костяк структуры кода. Благодаря им она может быть проанализирована, модули могут быть собраны в один файл специальными инструментами, а неиспользуемые экспорты удалены. Это возможно только благодаря тому, что всё статично.

[**Выражение import()**](https://learn.javascript.ru/modules-dynamic-imports#vyrazhenie-import)

Выражение import(module) загружает модуль и возвращает промис, результатом которого становится объект модуля, содержащий все его экспорты. Использовать его можно динамически в любом месте кода, например, так:

let modulePath = prompt("Какой модуль загружать?");

import(modulePath)

.then(obj => <объект модуля>)

.catch(err => <ошибка загрузки, например если нет такого модуля>)

Или если внутри асинхронной функции, то можно let module = await import(modulePath). Например, если у нас есть такой модуль say.js:

// say.js

export function hi() {

alert(`Привет`);

}

export function bye() {

alert(`Пока`);

}

То динамический импорт может выглядеть так:

let {hi, bye} = await import('./say.js');

hi();

bye();

А если в say.js указан экспорт по умолчанию:

// say.js

export default function() {

alert("Module loaded (export default)!");

}

То для доступа к нему следует взять свойство default объекта модуля:

let obj = await import('./say.js');

let say = obj.default;

// или, одной строкой: let {default: say} = await import('./say.js');

say();

Вот полный пример:



// say.js

export function hi() {

alert(`Привет`);

}

export function bye() {

alert(`Пока`);

}

export default function() {

alert("Модуль загружен (экспорт по умолчанию)!");

}

// index.html

<!doctype html>

<script>

async function load() {

let say = await import('./say.js');

say.hi(); // Привет!

say.bye(); // Пока!

say.default(); // Модуль загружен (экспорт по умолчанию)!

}

</script>

<button onclick="load()">Нажми меня</button>

Динамический импорт работает в обычных скриптах, он не требует указания script type="module".

Хотя import() и выглядит похоже на вызов функции, на самом деле это специальный синтаксис, так же, как, например, super().

Так что нельзя копировать import в другую переменную или вызвать при помощи .call/apply. Это не функция.