**Привет, мир!**

В этой статье мы создадим простой скрипт и посмотрим, как он работает.

**[Тег SCRIPT](https://learn.javascript.ru/hello-world" \l "тег-script)**

**А побыстрее?**

В том (и только в том!) случае, если читатель нетерпелив и уже разрабатывал на JavaScript или имеет достаточно опыта в другом языке программировании, он может не читать каждую статью этого раздела, а перепрыгнуть сразу к главе [Всё вместе: особенности JavaScript](https://learn.javascript.ru/javascript-specials). Там будет кратко самое основное.

Если же у вас есть достаточно времени и желание начать с азов, то читайте дальше :)

Программы на языке JavaScript можно вставить в любое место HTML при помощи тега SCRIPT. Например:

<!DOCTYPE HTML>

<html>

<head>

<!-- Тег meta для указания кодировки -->

<meta charset="utf-8">

</head>

<body>

<p>Начало документа...</p>

<script>

alert( 'Привет, Мир!' );

</script>

<p>...Конец документа</p>

</body>

</html>

Этот пример использует следующие элементы:

**<script> ... </script>**

Тег script содержит исполняемый код. Предыдущие стандарты HTML требовали обязательного указания атрибута type, но сейчас он уже не нужен. Достаточно просто <script>.

Браузер, когда видит <script>:

1. Начинает отображать страницу, показывает часть документа до script
2. Встретив тег script, переключается в JavaScript-режим и не показывает, а исполняет его содержимое.
3. Закончив выполнение, возвращается обратно в HTML-режим и *только тогда* отображает оставшуюся часть документа.

Попробуйте этот пример в действии, и вы сами всё увидите.

**alert(сообщение)**

Отображает окно с сообщением и ждёт, пока посетитель не нажмёт «Ок».

**Кодировка и тег META**

При попытке сделать такой же файл у себя на диске и запустить, вы можете столкнуться с проблемой – выводятся «кракозяблы», «квадратики» и «вопросики» вместо русского текста.

Чтобы всё было хорошо, нужно:

1. Убедиться, что в HEAD есть строка <meta charset="utf-8">. Если вы будете открывать файл с диска, то именно он укажет браузеру кодировку.
2. Убедиться, что редактор сохранил файл именно в кодировке UTF-8, а не, скажем, в windows-1251.

Указание кодировки – часть обычного HTML, я упоминаю об этом «на всякий случай», чтобы не было сюрпризов при запуске примеров локально.

**[Современная разметка для SCRIPT](https://learn.javascript.ru/hello-world" \l "современная-разметка-для-script)**

В старых скриптах оформление тега SCRIPT было немного сложнее. В устаревших руководствах можно встретить следующие элементы:

**Атрибут <script type=…>**

В отличие от HTML5, стандарт HTML 4 требовал обязательного указания этого атрибута. Выглядел он так: type="text/javascript". Если указать другое значение type, то скрипт выполнен не будет.

В современной разработке атрибут type не обязателен.

**Атрибут <script language=…>**

Этот атрибут предназначен для указания языка, на котором написан скрипт. По умолчанию, язык – JavaScript, так что и этот атрибут ставить не обязательно.

**Комментарии до и после скриптов**

В совсем старых руководствах и книгах иногда рекомендуют использовать HTML-комментарии внутри SCRIPT, чтобы спрятать Javascript от браузеров, которые не поддерживают его.

Выглядит это примерно так:

<script type="text/javascript"><!--

...

//--></script>

Браузер, для которого предназначались такие трюки, очень старый Netscape, давно умер. Поэтому в этих комментариях нет нужды.

Итак, для вставки скрипта мы просто пишем <script>, без дополнительных атрибутов и комментариев.

[**Задачи**](https://learn.javascript.ru/hello-world#tasks)

**[Выведите alert](https://learn.javascript.ru/hello-world" \l "выведите-alert)**

важность: 5

Сделайте страницу, которая выводит «Я – JavaScript!».

Создайте её на диске, откройте в браузере, убедитесь, что всё работает.

[Демо в новом окне](https://learn.javascript.ru/task/hello-alert/solution/)

**Внешние скрипты, порядок исполнения**

Если JavaScript-кода много – его выносят в отдельный файл, который подключается в HTML:

<script src="/path/to/script.js"></script>

Здесь /path/to/script.js – это абсолютный путь к файлу, содержащему скрипт (из корня сайта).

Браузер сам скачает скрипт и выполнит.

Можно указать и полный URL, например:

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

Вы также можете использовать путь относительно текущей страницы. Например, src="lodash.js"обозначает файл из текущей директории.

Чтобы подключить несколько скриптов, используйте несколько тегов:

<script src="/js/script1.js"></script>

<script src="/js/script2.js"></script>

...

**На заметку:**

Как правило, в HTML пишут только самые простые скрипты, а сложные выносят в отдельный файл.

Браузер скачает его только первый раз и в дальнейшем, при правильной настройке сервера, будет брать из своего [кеша](http://ru.wikipedia.org/wiki/%D0%9A%D1%8D%D1%88).

Благодаря этому один и тот же большой скрипт, содержащий, к примеру, библиотеку функций, может использоваться на разных страницах без полной перезагрузки с сервера.

**Если указан атрибут src, то содержимое тега игнорируется.**

В одном теге SCRIPT нельзя одновременно подключить внешний скрипт и указать код.

Вот так не сработает:

<script src="file.js">

alert(1); // так как указан src, то внутренняя часть тега игнорируется

</script>

Нужно выбрать: либо SCRIPT идёт с src, либо содержит код. Тег выше следует разбить на два: один – с src, другой – с кодом, вот так:

<script src="file.js"></script>

<script>

alert( 1 );

</script>

**[Асинхронные скрипты: defer/async](https://learn.javascript.ru/external-script" \l "асинхронные-скрипты-defer-async)**

Браузер загружает и отображает HTML постепенно. Особенно это заметно при медленном интернет-соединении: браузер не ждёт, пока страница загрузится целиком, а показывает ту часть, которую успел загрузить.

Если браузер видит тег <script>, то он по стандарту обязан сначала выполнить его, а потом показать оставшуюся часть страницы.

Например, в примере ниже – пока все кролики не будут посчитаны – нижний <p> не будет показан:

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

</head>

<body>

<p>Начинаем считать:</p>

<script>

alert( 'Первый кролик!' );

alert( 'Второй кролик!' );

alert( 'Третий кролик!' );

</script>

<p>Кролики посчитаны!</p>

</body>

</html>

Такое поведение называют «синхронным». Как правило, оно вполне нормально, но есть важное следствие.

**Если скрипт – внешний, то пока браузер не выполнит его, он не покажет часть страницы под ним.**

То есть, в таком документе, пока не загрузится и не выполнится big.js, содержимое <body> будет скрыто:

<html>

<head>

<script src="big.js"></script>

</head>

<body>

Этот текст не будет показан, пока браузер не выполнит big.js.

</body>

</html>

И здесь вопрос – действительно ли мы этого хотим? То есть, действительно ли оставшуюся часть страницы нельзя показывать до загрузки скрипта?

Есть ситуации, когда мы не только НЕ хотим такой задержки, но она даже опасна.

Например, если мы подключаем внешний скрипт, который показывает рекламу или вставляет счётчик посещений, а затем идёт наша страница. Конечно, неправильно, что пока счётчик или реклама не подгрузятся – оставшаяся часть страницы не показывается. Счётчик посещений не должен никак задерживать отображение страницы сайта. Реклама тоже не должна тормозить сайт и нарушать его функционал.

А что, если сервер, с которого загружается внешний скрипт, перегружен? Посетитель в этом случае может ждать очень долго!

Вот пример, с подобным скриптом (стоит искусственная задержка загрузки):

<p>Важная информация не покажется, пока не загрузится скрипт.</p>

<script src="https://js.cx/hello/ads.js?speed=0"></script>

<p>...Важная информация!</p>

Что делать?

Можно поставить все подобные скрипты в конец страницы – это уменьшит проблему, но не избавит от неё полностью, если скриптов несколько. Допустим, в конце страницы 3 скрипта, и первый из них тормозит – получается, другие два его будут ждать – тоже нехорошо.

Кроме того, браузер дойдёт до скриптов, расположенных в конце страницы, они начнут грузиться только тогда, когда вся страница загрузится. А это не всегда правильно. Например, счётчик посещений наиболее точно сработает, если загрузить его пораньше.

Поэтому «расположить скрипты внизу» – не лучший выход.

Кардинально решить эту проблему помогут атрибуты async или defer:

**Атрибут async**

Поддерживается всеми браузерами, кроме IE9-. Скрипт выполняется полностью асинхронно. То есть, при обнаружении <script async src="..."> браузер не останавливает обработку страницы, а спокойно работает дальше. Когда скрипт будет загружен – он выполнится.

**Атрибут defer**

Поддерживается всеми браузерами, включая самые старые IE. Скрипт также выполняется асинхронно, не заставляет ждать страницу, но есть два отличия от async.

Первое – браузер гарантирует, что относительный порядок скриптов с defer будет сохранён.

То есть, в таком коде (с async) первым сработает тот скрипт, который раньше загрузится:

<script src="1.js" async></script>

<script src="2.js" async></script>

А в таком коде (с defer) первым сработает всегда 1.js, а скрипт 2.js, даже если загрузился раньше, будет его ждать.

<script src="1.js" defer></script>

<script src="2.js" defer></script>

Поэтому атрибут defer используют в тех случаях, когда второй скрипт 2.js зависит от первого 1.js, к примеру – использует что-то, описанное первым скриптом.

Второе отличие – скрипт с defer сработает, когда весь HTML-документ будет обработан браузером.

Например, если документ достаточно большой…

<script src="async.js" async></script>

<script src="defer.js" defer></script>

Много много много букв

…То скрипт async.js выполнится, как только загрузится – возможно, до того, как весь документ готов. А defer.js подождёт готовности всего документа.

Это бывает удобно, когда мы в скрипте хотим работать с документом, и должны быть уверены, что он полностью получен.

**async вместе с defer**

При одновременном указании async и defer в современных браузерах будет использован только async, в IE9- – только defer (не понимает async).

**Атрибуты async/defer – только для внешних скриптов**

Атрибуты async/defer работают только в том случае, если назначены на внешние скрипты, т.е. имеющие src.

При попытке назначить их на обычные скрипты <script>…</script>, они будут проигнороированы.

Тот же пример с async:

<p>Важная информация теперь не ждёт, пока загрузится скрипт...</p>

<script async src="https://js.cx/hello/ads.js?speed=0"></script>

<p>...Важная информация!</p>

При запуске вы увидите, что вся страница отобразилась тут же, а alert из внешнего скрипта появится позже, когда загрузится скрипт.

**Эти атрибуты давно «в ходу»**

Большинство современных систем рекламы и счётчиков знают про эти атрибуты и используют их.

Перед вставкой внешнего тега <script> понимающий программист всегда проверит, есть ли у него подобный атрибут. Иначе медленный скрипт может задержать загрузку страницы.

**Забегая вперёд**

Для продвинутого читателя, который знает, что теги <script> можно добавлять на страницу в любой момент при помощи самого javascript, заметим, что скрипты, добавленные таким образом, ведут себя так же, как async. То есть, выполняются как только загрузятся, без сохранения относительного порядка.

Если же нужно сохранить порядок выполнения, то есть добавить несколько скриптов, которые выполнятся строго один за другим, то используется свойство script.async = false.

Выглядит это примерно так:

function addScript(src){

var script = document.createElement('script');

script.src = src;

script.async = false; // чтобы гарантировать порядок

document.head.appendChild(script);

}

addScript('1.js'); // загружаться эти скрипты начнут сразу

addScript('2.js'); // выполнятся, как только загрузятся

addScript('3.js'); // но, гарантированно, в порядке 1 -> 2 -> 3

Более подробно работу со страницей мы разберём во второй части учебника.

**[Итого](https://learn.javascript.ru/external-script" \l "итого)**

* Скрипты вставляются на страницу как текст в теге <script>, либо как внешний файл через <script src="путь"></script>
* Специальные атрибуты async и defer используются для того, чтобы пока грузится внешний скрипт – браузер показал остальную (следующую за ним) часть страницы. Без них этого не происходит.
* Разница между async и defer: атрибут defer сохраняет относительную последовательность скриптов, а async – нет. Кроме того, defer всегда ждёт, пока весь HTML-документ будет готов, а async – нет.

Очень важно не только читать учебник, но делать что-то самостоятельно.

Решите задачки, чтобы удостовериться, что вы всё правильно поняли.

[**Задачи**](https://learn.javascript.ru/external-script#tasks)

**[Вывести alert внешним скриптом](https://learn.javascript.ru/external-script" \l "вывести-alert-внешним-скриптом)**

важность: 5

Возьмите решение предыдущей задачи [Выведите alert](https://learn.javascript.ru/task/hello-alert) и вынесите скрипт во внешний файл alert.js, который расположите в той же директории.

Откройте страницу и проверьте, что вывод сообщения всё ещё работает.

**Структура кода**

В этой главе мы рассмотрим общую структуру кода, команды и их разделение.

**[Команды](https://learn.javascript.ru/structure" \l "команды)**

Раньше мы уже видели пример команды: alert('Привет, мир!') выводит сообщение.

Для того, чтобы добавить в код ещё одну команду – можно поставить её после точки с запятой.

Например, вместо одного вызова alert сделаем два:

alert('Привет'); alert('Мир');

Как правило, каждая команда пишется на отдельной строке – так код лучше читается:

alert('Привет');

alert('Мир');

**[Точка с запятой](https://learn.javascript.ru/structure" \l "semicolon)**

Точку с запятой *во многих случаях* можно не ставить, если есть переход на новую строку.

Так тоже будет работать:

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, важно то, что «во многих случаях» не означает «всегда»!**

Например, запустите этот код:

alert(3 +

1

+ 2);

Выведет 6.

То есть, точка с запятой не ставится. Почему? Интуитивно понятно, что здесь дело в «незавершённом выражении», конца которого JavaScript ждёт с первой строки и поэтому не ставит точку с запятой. И здесь это, пожалуй, хорошо и приятно.

**Но в некоторых важных ситуациях JavaScript «забывает» вставить точку с запятой там, где она нужна.**

Таких ситуаций не так много, но ошибки, которые при этом появляются, достаточно сложно обнаруживать и исправлять.

Чтобы не быть голословным, вот небольшой пример.

Такой код работает:

[1, 2].forEach(alert)

Он выводит по очереди 1, 2. Почему он работает – сейчас не важно, позже разберёмся.

Важно, что вот такой код уже работать не будет:

alert("Сейчас будет ошибка")

[1, 2].forEach(alert)

Выведется только первый alert, а дальше – ошибка. Потому что перед квадратной скобкой JavaScript точку с запятой не ставит, а как раз здесь она нужна (упс!).

Если её поставить, то всё будет в порядке:

alert( "Сейчас будет ошибка" );

[1, 2].forEach(alert)

**Поэтому в JavaScript рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт, которому следуют все большие проекты.**

**[Комментарии](https://learn.javascript.ru/structure" \l "комментарии)**

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

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' );

**Используйте горячие клавиши!**

В большинстве редакторов комментарий можно поставить горячей клавишей, обычно это Ctrl+/для однострочных и что-то вроде Ctrl+Shift+/ – для многострочных комментариев (нужно выделить блок и нажать сочетание клавиш). Детали смотрите в руководстве по редактору.

**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.

На следующих занятиях мы поговорим о переменных, блоках и других структурных элементах программы на JavaScript.

**Современный стандарт, "use strict"**

Очень долго язык JavaScript развивался без потери совместимости. Новые возможности добавлялись в язык, но старые – никогда не менялись, чтобы не «сломать» уже существующие HTML/JS-страницы с их использованием.

Однако, это привело к тому, что любая ошибка в дизайне языка становилась «вмороженной» в него навсегда.

Так было до появления стандарта ECMAScript 5 (ES5), который одновременно добавил новые возможности и внёс в язык ряд исправлений, которые могут привести к тому, что старый код, который был написан до его появления, перестанет работать.

Чтобы этого не случилось, решили, что по умолчанию эти опасные изменения будут выключены, и код будет работать по-старому. А для того, чтобы перевести код в режим полного соответствия современному стандарту, нужно указать специальную директиву use strict.

Эта директива не поддерживается IE9-.

**[Директива use strict](https://learn.javascript.ru/strict-mode" \l "директива-use-strict)**

Директива выглядит как строка "use strict"; или 'use strict'; и ставится в начале скрипта.

Например:

"use strict";

// этот код будет работать по современному стандарту ES5

...

**Отменить действие use strict никак нельзя**

Не существует директивы no use strict или подобной, которая возвращает в старый режим.

Если уж вошли в современный режим, то это дорога в один конец.

**use strict для функций**

Через некоторое время мы будем проходить [функции](https://learn.javascript.ru/function-basics). На будущее заметим, что use strict также можно указывать в начале функций, тогда строгий режим будет действовать только внутри функции.

В следующих главах мы будем подробно останавливаться на отличиях в работе языка при use strict и без него.

**[Нужен ли мне «use strict»?](https://learn.javascript.ru/strict-mode" \l "нужен-ли-мне-use-strict)**

Если говорить абстрактно, то – да, нужен. В строгом режиме исправлены некоторые ошибки в дизайне языка, и вообще, современный стандарт – это хорошо.

Однако, есть и две проблемы.

**Поддержка браузеров IE9-, которые игнорируют "use strict".**

Предположим, что мы, используя "use strict", разработали код и протестировали его в браузере Chrome. Всё работает… Однако, вероятность ошибок при этом в IE9- выросла! Он-то всегда работает по старому стандарту, а значит, иногда по-другому. Возникающие ошибки придётся отлаживать уже в IE9-, и это намного менее приятно, нежели в Chrome.

Впрочем, проблема не так страшна. Несовместимостей мало. И если их знать (а в учебнике мы будем останавливаться на них) и писать правильный код, то всё будет в порядке и "use strict" станет нашим верным помощником.

**Библиотеки, написанные без учёта "use strict".**

Некоторые библиотеки, которые написаны без "use strict", не всегда корректно работают, если вызывающий код содержит "use strict".

В первую очередь имеются в виду сторонние библиотеки, которые писали не мы, и которые не хотелось бы переписывать или править.

Таких библиотек мало, но при переводе давно существующих проектов на "use strict" эта проблема возникает с завидной регулярностью.

Вывод?

**Писать код с use strict следует лишь в том случае, если вы уверены, что описанных выше проблем не будет.**

Конечно же, весь код, который находится в этом учебнике, корректно работает в режиме "use strict".

**[ES5-shim](https://learn.javascript.ru/strict-mode" \l "es5-shim)**

Браузер IE8 поддерживает только совсем старую версию стандарта JavaScript, а именно ES3.

К счастью, многие возможности современного стандарта можно добавить в этот браузер, подключив библиотеку [ES5 shim](https://github.com/es-shims/es5-shim), а именно – скрипты es5-shim.js и es5-sham.js из неё.

[**Итого**](https://learn.javascript.ru/strict-mode#итого)

В этой главе мы познакомились с понятием «строгий режим».

Далее мы будем предполагать, что разработка ведётся либо в современном браузере, либо в IE8- с подключённым [ES5 shim](https://github.com/es-shims/es5-shim). Это позволит нам использовать большинство возможностей современного JavaScript во всех браузерах.

Очень скоро, буквально в следующей главе, мы увидим особенности строгого режима на конкретных примерах.

**Переменные**

В зависимости от того, для чего вы делаете скрипт, понадобится работать с информацией.

Если это электронный магазин – то это товары, корзина. Если чат – посетители, сообщения и так далее.

Чтобы хранить информацию, используются *переменные*.

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

*Переменная* состоит из имени и выделенной области памяти, которая ему соответствует.

Для *объявления* или, другими словами, *создания переменной* используется ключевое слово var:

var message;

После объявления, можно записать в переменную данные:

var message;

message = 'Hello'; // сохраним в переменной строку

Эти данные будут сохранены в соответствующей области памяти и в дальнейшем доступны при обращении по имени:

var message;

message = 'Hello!';

alert( message ); // выведет содержимое переменной

Для краткости можно совместить объявление переменной и запись данных:

var message = 'Hello!';

Можно даже объявить несколько переменных сразу:

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

**[Аналогия из жизни](https://learn.javascript.ru/variables" \l "аналогия-из-жизни)**

Проще всего понять переменную, если представить её как «коробку» для данных, с уникальным именем.

Например, переменная message – это коробка, в которой хранится значение "Hello!":



В коробку можно положить любое значение, а позже – поменять его. Значение в переменной можно изменять сколько угодно раз:

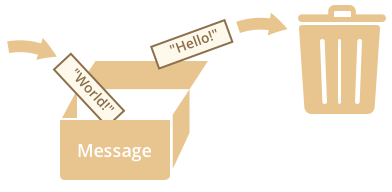
var message;

message = 'Hello!';

message = 'World!'; // заменили значение

alert( message );

При изменении значения старое содержимое переменной удаляется.



Можно объявить две переменные и копировать данные из одной в другую:

var hello = 'Hello world!';

var message;

// скопировали значение

message = hello;

alert( hello ); // Hello world!

alert( message ); // Hello world!

**На заметку:**

Существуют [функциональные](http://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%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%D1%8F) языки программирования, в которых значение переменной менять нельзя. Например, [Scala](http://www.scala-lang.org/) или [Erlang](http://www.erlang.org/).

В таких языках положил один раз значение в коробку – и оно хранится там вечно, ни удалить ни изменить. А нужно что-то другое сохранить – изволь создать новую коробку (объявить новую переменную), повторное использование невозможно.

С виду – не очень удобно, но, как ни странно, и на таких языках вполне можно успешно программировать. Более того, оказывается, что в ряде областей, например в распараллеливании вычислений, они имеют преимущества. Изучение какого-нибудь функционального языка рекомендуется для расширения кругозора.

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

На имя переменной в JavaScript наложены всего два ограничения.

1. Имя может состоять из: букв, цифр, символов $ и \_
2. Первый символ не должен быть цифрой.

Например:

var myName;

var test123;

**Что особенно интересно – доллар '$' и знак подчеркивания '\_' являются такими же обычными символами, как буквы:**

var $ = 1; // объявили переменную с именем '$'

var \_ = 2; // переменная с именем '\_'

alert( $ + \_ ); // 3

А такие переменные были бы неправильными:

var 1a; // начало не может быть цифрой

var my-name; // дефис '-' не является разрешенным символом

**Регистр букв имеет значение**

Переменные apple и AppLE – две разные переменные.

**Русские буквы допустимы, но не рекомендуются**

В названии переменных можно использовать и русские буквы, например:

var имя = "Вася";

alert( имя ); // "Вася"

Технически, ошибки здесь нет, но на практике сложилась традиция использовать в именах только английские буквы.

**Зарезервированные имена**

Существует список зарезервированных слов, которые нельзя использовать для переменных, так как они используются самим языком, например: var, class, return, export и др.

Например, такой пример выдаст синтаксическую ошибку:

var return = 5; // ошибка

alert(return);

**[Важность директивы var](https://learn.javascript.ru/variables" \l "важность-директивы-var)**

В старом стандарте JavaScript разрешалось создавать переменную и без var, просто присвоив ей значение:

num = 5; // переменная num будет создана, если ее не было

В режиме "use strict" так делать уже нельзя.

Следующий код выдаст ошибку:

"use strict";

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

Обратим внимание, директиву use strict нужно ставить до кода, иначе она не сработает:

var something;

"use strict"; // слишком поздно

num = 5; // ошибки не будет, так как строгий режим не активирован

**Ошибка в IE8- без var**

Если же вы собираетесь поддерживать IE8-, то у меня для вас ещё одна причина всегда использовать var.

Следущий документ в IE8- ничего не выведет, будет ошибка:

<div id="test"></div>

<script>

test = 5; // здесь будет ошибка!

alert( test ); // не сработает

</script>

Это потому, что переменная test не объявлена через var и совпадает с id элемента <div>. Даже не спрашивайте почему – это ошибка в браузере IE до версии 9.

Самое «забавное» то, что такая ошибка присвоения значений будет только в IE8- и только если на странице присутствует элемент с совпадающим с именем id.

Такие ошибки особенно «весело» исправлять и отлаживать.

Вывод простой – всегда объявляем переменные через var, и сюрпризов не будет. Даже в старых IE.

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

*Константа* – это переменная, которая никогда не меняется. Как правило, их называют большими буквами, через подчёркивание. Например:

var COLOR\_RED = "#F00";

var COLOR\_GREEN = "#0F0";

var COLOR\_BLUE = "#00F";

var COLOR\_ORANGE = "#FF7F00";

var color = COLOR\_ORANGE;

alert( color ); // #FF7F00

Технически, константа является обычной переменной, то есть её *можно* изменить. Но мы *договариваемся* этого не делать.

Зачем нужны константы? Почему бы просто не писать var color = "#FF7F00"?

1. Во-первых, константа COLOR\_ORANGE – это понятное имя. По присвоению var color="#FF7F00"непонятно, что цвет – оранжевый. Иными словами, константа COLOR\_ORANGE является «понятным псевдонимом» для значения #FF7F00.
2. Во-вторых, опечатка в строке, особенно такой сложной как #FF7F00, может быть не замечена, а в имени константы её допустить куда сложнее.

**Константы используют вместо строк и цифр, чтобы сделать программу понятнее и избежать ошибок.**

[**Итого**](https://learn.javascript.ru/variables#итого)

* В JavaScript можно объявлять переменные для хранения данных. Это делается при помощи var.
* Технически, можно просто записать значение и без объявления переменной, однако по ряду причин это не рекомендуется.
* Вместе с объявлением можно сразу присвоить значение: var x = 10.
* Переменные, которые названы БОЛЬШИМИ\_БУКВАМИ, являются константами, то есть никогда не меняются. Как правило, они используются для удобства, чтобы было меньше ошибок.

[**Задачи**](https://learn.javascript.ru/variables#tasks)

**[Работа с переменными](https://learn.javascript.ru/variables" \l "работа-с-переменными)**

важность: 2

1. Объявите две переменные: admin и name.
2. Запишите в name строку "Василий".
3. Скопируйте значение из name в admin.
4. Выведите admin (должно вывести «Василий»).

**Правильный выбор имени переменной**

Правильный выбор имени переменной – одна из самых важных и сложных вещей в программировании, которая отличает начинающего от гуру.

Дело в том, что большинство времени мы тратим не на изначальное написание кода, а на его развитие.

Возможно, эти слова не очевидны, если вы пока что ничего большого не писали или пишете код «только для чтения» (написал 5 строк, отдал заказчику и забыл). Но чем более серьёзные проекты вы будете делать, тем более актуальны они будут для вас.

Что такое это «развитие»? Это когда я вчера написал код, а сегодня (или спустя неделю) прихожу и хочу его поменять. Например, вывести сообщение не так, а эдак… Обработать товары по-другому, добавить функционал… А где у меня там сообщение хранится? А где товар?..

Гораздо проще найти нужные данные, если они правильно помечены, то есть когда переменная названа *правильно*.

**[Правила именования](https://learn.javascript.ru/variable-names" \l "правила-именования)**

* **Правило 1.**

**Никакого транслита. Только английский.**

Неприемлемы:

var moiTovari;

var cena;

var ssilka;

Подойдут:

var myGoods;

var price;

var link;

Чем плох транслит?

Во-первых, среди разработчиков всего мира принято использовать английский язык для имён переменных. И если ваш код потом попадёт к кому-то другому, например вы будете в команде больше чем из одного человека, то велик шанс, что транслит ему не понравится.

Во-вторых, русский транслит хуже читается и длиннее, чем названия на английском.

В-третьих, в проектах вы наверняка будете применять библиотеки, написанные другими людьми. Многое уже готово, в распоряжении современного разработчика есть масса инструментов, все они используют названия переменных и функций на английском языке, и вы, конечно, будете их использовать. А от кода, где транслит перемешан с английским – волосы могут встать дыбом, и не только на голове.

Если вы вдруг не знаете английский – самое время выучить.

* **Правило 2.**

**Использовать короткие имена только для переменных «местного значения».**

Называть переменные именами, не несущими смысловой нагрузки, например a, e, p, mg – можно только в том случае, если они используются в небольшом фрагменте кода и их применение очевидно.

Вообще же, название переменной должно быть понятным. Иногда для этого нужно использовать несколько слов.

* **Правило 3.**

**Переменные из нескольких слов пишутся вместеВотТак.**

Например:

var borderLeftWidth;

Этот способ записи называется «верблюжьей нотацией» или, по-английски, «camelCase».

Существует альтернативный стандарт, когда несколько слов пишутся через знак подчеркивания '\_':

var border\_left\_width;

Преимущественно в JavaScript используется вариант borderLeftWidth, в частности во встроенных языковых и браузерных функциях. Поэтому целесообразно остановиться на нём.

Ещё одна причина выбрать «верблюжью нотацию» – запись в ней немного короче, чем с подчеркиванием, т.к. не нужно вставлять '\_'.

* **Правило последнее, главное.**

**Имя переменной должно максимально чётко соответствовать хранимым в ней данным.**

Придумывание таких имен – одновременно коротких и точных, при которых всегда понятно, что где лежит, приходит с опытом, но только если сознательно стремиться к этому.

Позвольте поделиться одним небольшим секретом, который очень прост, но позволит улучшить названия переменных и сэкономит время.

Бывает так, что, написав код, мы через некоторое время к нему возвращаемся, надо что-то поправить. И мы примерно помним, что переменная, в которой хранится нужное вам значение, называется… Ну, скажем, borderLeftWidth. Мы ищем её в коде, не находим, но, разобравшись, обнаруживаем, что на самом деле переменная называлась вот так: leftBorderWidth.

**Если мы ищем переменную с одним именем, а находим – с другим, то зачастую самый лучший ход – это *переименовать* переменную, чтобы имя было тем, которое вы искали.**

То есть, в коде leftBorderWidth, а мы её переименуем на ту, которую искали: borderLeftWidth.

Зачем? Дело в том, что в следующий раз, когда вы захотите что-то поправить, то вы будете искать по тому же самому имени. Соответственно, это сэкономит вам время.

Есть причина и поважнее. Поскольку именно это имя переменной пришло в голову – скорее всего, оно больше соответствует хранимым там данным, чем то, которое было мы придумали изначально. Исключения бывают, но в любом случае – такое несовпадение – это повод задуматься.

Чтобы удобно переименовывать переменную, нужно использовать [хороший редактор JavaScript](https://learn.javascript.ru/editor), тогда этот процесс будет очень простым и быстрым.

**Если коротко…**

Смысл имени переменной – это «имя на коробке», по которому мы сможем максимально быстро находить нужные нам данные.

**Не нужно бояться переименовывать переменные, если вы придумали имя получше.**

Современные редакторы позволяют делать это очень удобно и быстро. Это в конечном счете сэкономит вам время.

**Храните в переменной то, что следует**

Бывают ленивые программисты, которые, вместо того чтобы объявить новую переменную, используют существующую.

В результате получается, что такая переменная – как коробка, в которую кидают то одно, то другое, то третье, при этом не меняя название. Что в ней лежит сейчас? А кто его знает… Нужно подойти, проверить.

Сэкономит такой программист время на объявлении переменной – потеряет в два раза больше на отладке кода.

**«Лишняя» переменная – добро, а не зло.**

[**Задачи**](https://learn.javascript.ru/variable-names#tasks)

**[Объявление переменных](https://learn.javascript.ru/variable-names" \l "объявление-переменных)**

важность: 3

1. Создайте переменную для названия нашей планеты и присвойте ей значение "Земля". *Правильное* имя выберите сами.
2. Создайте переменную для имени посетителя со значением "Петя". Имя также на ваш вкус.

**Шесть типов данных, typeof**

В JavaScript существует несколько основных типов данных.

В этой главе мы получим о них общее представление, а позже, в соответствующих главах подробно познакомимся с использованием каждого типа в отдельности.

**[Число «number»](https://learn.javascript.ru/types-intro" \l "число-number)**

var n = 123;

n = 12.345;

Единый тип *число* используется как для целых, так и для дробных чисел.

Существуют специальные числовые значения Infinity (бесконечность) и NaN (ошибка вычислений).

Например, бесконечность Infinity получается при делении на ноль:

alert( 1 / 0 ); // Infinity

Ошибка вычислений NaN будет результатом некорректной математической операции, например:

alert( "нечисло" \* 2 ); // NaN, ошибка

Эти значения формально принадлежат типу «число», хотя, конечно, числами в их обычном понимании не являются.

Особенности работы с числами в JavaScript разобраны в главе [Числа](https://learn.javascript.ru/number).

**[Строка «string»](https://learn.javascript.ru/types-intro" \l "строка-string)**

var str = "Мама мыла раму";

str = 'Одинарные кавычки тоже подойдут';

**В JavaScript одинарные и двойные кавычки равноправны.** Можно использовать или те или другие.

**Тип *символ* не существует, есть только *строка*.**

В некоторых языках программирования есть специальный тип данных для одного символа. Например, в языке С это char. В JavaScript есть только тип «строка» string. Что, надо сказать, вполне удобно.

Более подробно со строками мы познакомимся в главе [Строки](https://learn.javascript.ru/string).

**[Булевый (логический) тип «boolean»](https://learn.javascript.ru/types-intro" \l "булевый-логический-тип-boolean)**

У него всего два значения: true (истина) и false (ложь).

Как правило, такой тип используется для хранения значения типа да/нет, например:

var checked = true; // поле формы помечено галочкой

checked = false; // поле формы не содержит галочки

О нём мы поговорим более подробно, когда будем обсуждать логические вычисления и условные операторы.

**[Специальное значение «null»](https://learn.javascript.ru/types-intro" \l "специальное-значение-null)**

Значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null:

var age = null;

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

В частности, код выше говорит о том, что возраст age неизвестен.

**[Специальное значение «undefined»](https://learn.javascript.ru/types-intro" \l "специальное-значение-undefined)**

Значение undefined, как и null, образует свой собственный тип, состоящий из одного этого значения. Оно имеет смысл «значение не присвоено».

Если переменная объявлена, но в неё ничего не записано, то её значение как раз и есть undefined:

var x;

alert( x ); // выведет "undefined"

Можно присвоить undefined и в явном виде, хотя это делается редко:

var x = 123;

x = undefined;

alert( x ); // "undefined"

В явном виде undefined обычно не присваивают, так как это противоречит его смыслу. Для записи в переменную «пустого» или «неизвестного» значения используется null.

**[Объекты «object»](https://learn.javascript.ru/types-intro" \l "объекты-object)**

Первые 5 типов называют *«примитивными»*.

Особняком стоит шестой тип: *«объекты»*.

Он используется для коллекций данных и для объявления более сложных сущностей.

Объявляются объекты при помощи фигурных скобок {...}, например:

var user = { name: "Вася" };

Мы подробно разберём способы объявления объектов и, вообще, работу с объектами, позже, в главе [Объекты как ассоциативные массивы](https://learn.javascript.ru/object).

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

Оператор typeof возвращает тип аргумента.

У него есть два синтаксиса: со скобками и без:

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

Работают они одинаково, но первый синтаксис короче.

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

typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

typeof {} // "object"

typeof null // "object" (1)

typeof function(){} // "function" (2)

Последние две строки помечены, потому что typeof ведет себя в них по-особому.

1. Результат typeof null == "object" – это официально признанная ошибка в языке, которая сохраняется для совместимости. На самом деле null – это не объект, а отдельный тип данных.
2. Функции мы пройдём чуть позже. Пока лишь заметим, что функции не являются отдельным базовым типом в JavaScript, а подвидом объектов. Но typeof выделяет функции отдельно, возвращая для них "function". На практике это весьма удобно, так как позволяет легко определить функцию.

К работе с типами мы также вернёмся более подробно в будущем, после изучения основных структур данных.

[**Итого**](https://learn.javascript.ru/types-intro#итого)

Есть 5 «примитивных» типов: number, string, boolean, null, undefined и 6-й тип – объекты object.

Очень скоро мы изучим их во всех деталях.

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

**Основные операторы**

Для работы с переменными, со значениями, JavaScript поддерживает все стандартные операторы, большинство которых есть и в других языках программирования.

Несколько операторов мы знаем со школы – это обычные сложение +, умножение \*, вычитание и так далее.

В этой главе мы сконцентрируемся на операторах, которые в курсе математики не проходят, и на их особенностях в JavaScript.

**[Термины: «унарный», «бинарный», «операнд»](https://learn.javascript.ru/operators" \l "термины-унарный-бинарный-операнд)**

У операторов есть своя терминология, которая используется во всех языках программирования.

Прежде, чем мы двинемся дальше – несколько терминов, чтобы понимать, о чём речь.

* *Операнд* – то, к чему применяется оператор. Например: 5 \* 2 – оператор умножения с левым и правым операндами. Другое название: «аргумент оператора».
* *Унарным* называется оператор, который применяется к одному выражению. Например, оператор унарный минус "-" меняет знак числа на противоположный:
* var 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:
* var x = 1, y = 3;

alert( y - x ); // 2, бинарный минус

**[Сложение строк, бинарный +](https://learn.javascript.ru/operators" \l "сложение-строк-бинарный)**

Обычно при помощи плюса '+' складывают числа.

Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну:

var a = "моя" + "строка";

alert( a ); // моястрока

Иначе говорят, что «плюс производит конкатенацию (сложение) строк».

**Если хотя бы один аргумент является строкой, то второй будет также преобразован к строке!**

Причем не важно, справа или слева находится операнд-строка, в любом случае нестроковый аргумент будет преобразован. Например:

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

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

**Это приведение к строке – особенность исключительно бинарного оператора "+".**

Остальные арифметические операторы работают только с числами и всегда приводят аргументы к числу.

Например:

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

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

**[Преобразование к числу, унарный плюс +](https://learn.javascript.ru/operators" \l "преобразование-к-числу-унарный-плюс)**

Унарный, то есть применённый к одному значению, плюс ничего не делает с числами:

alert( +1 ); // 1

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

Как видно, плюс ничего не изменил в выражениях. Результат – такой же, как и без него.

Тем не менее, он широко применяется, так как его «побочный эффект» – преобразование значения в число.

Например, когда мы получаем значения из HTML-полей или от пользователя, то они обычно в форме строк.

А что, если их нужно, к примеру, сложить? Бинарный плюс сложит их как строки:

var apples = "2";

var oranges = "3";

alert( apples + oranges ); // "23", так как бинарный плюс складывает строки

Поэтому используем унарный плюс, чтобы преобразовать к числу:

var apples = "2";

var oranges = "3";

alert( +apples + +oranges ); // 5, число, оба операнда предварительно преобразованы в числа

С точки зрения математики такое изобилие плюсов может показаться странным. С точки зрения программирования – никаких разночтений: сначала выполнятся унарные плюсы, приведут строки к числам, а затем – бинарный '+' их сложит.

Почему унарные плюсы выполнились до бинарного сложения? Как мы сейчас увидим, дело в их приоритете.

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

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется *приоритетом*.

Из школы мы знаем, что умножение в выражении 2 \* 2 + 1 выполнится раньше сложения, т.к. его *приоритет* выше, а скобки явно задают порядок выполнения. Но в JavaScript – гораздо больше операторов, поэтому существует целая [таблица приоритетов](https://developer.mozilla.org/en/JavaScript/Reference/operators/operator_precedence).

Она содержит как уже пройденные операторы, так и те, которые мы еще не проходили. В ней каждому оператору задан числовой приоритет. Тот, у кого число больше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.

Отрывок из таблицы:

| **Приоритет** | **Название** | **Обозначение** |
| --- | --- | --- |
| … | … | … |
| 15 | унарный плюс | + |
| 15 | унарный минус | - |
| 14 | умножение | \* |
| 14 | деление | / |
| 13 | сложение | + |
| 13 | вычитание | - |
| … | … | … |
| 3 | присваивание | = |
| … | … | … |

Так как «унарный плюс» имеет приоритет 15, выше, чем 13 у обычного «сложения», то в выражении +apples + +oranges сначала сработали плюсы у apples и oranges, а затем уже обычное сложение.

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

Обратим внимание, в таблице приоритетов также есть оператор присваивания =.

У него – один из самых низких приоритетов: 3.

Именно поэтому, когда переменную чему-либо присваивают, например, x = 2 \* 2 + 1 сначала выполнится арифметика, а уже затем – произойдёт присваивание =.

var x = 2 \* 2 + 1;

alert( x ); // 5

**Возможно присваивание по цепочке:**

var a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4

alert( b ); // 4

alert( c ); // 4

Такое присваивание работает справа-налево, то есть сначала вычислятся самое правое выражение 2+2, присвоится в c, затем выполнится b = c и, наконец, a = b.

**Оператор "=" возвращает значение**

Все операторы возвращают значение. Вызов x = выражение не является исключением.

Он записывает выражение в x, а затем возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения:

var a = 1;

var b = 2;

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

alert( a ); // 3

alert( c ); // 0

В примере выше результатом (a = b + 1) является значение, которое записывается в a (т.е. 3). Оно используется для вычисления c.

Забавное применение присваивания, не так ли?

Знать, как это работает – стоит обязательно, а вот писать самому – только если вы уверены, что это сделает код более читаемым и понятным.

**[Взятие остатка %](https://learn.javascript.ru/operators" \l "взятие-остатка)**

Оператор взятия остатка % интересен тем, что, несмотря на обозначение, никакого отношения к процентам не имеет.

Его результат a % b – это 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

**[Инкремент/декремент: ++, --](https://learn.javascript.ru/operators" \l "инкремент-декремент)**

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:
* var i = 2;
* i++; // более короткая запись для i = i + 1.

alert(i); // 3

* **Декремент** -- уменьшает на 1:
* var i = 2;
* i--; // более короткая запись для i = i - 1.

alert(i); // 1

**Важно:**

Инкремент/декремент можно применить только к переменной. Код 5++ даст ошибку.

Вызывать эти операторы можно не только после, но и перед переменной: i++ (называется «постфиксная форма») или ++i («префиксная форма»).

Обе эти формы записи делают одно и то же: увеличивают на 1.

Тем не менее, между ними существует разница. Она видна только в том случае, когда мы хотим не только увеличить/уменьшить переменную, но и использовать результат в том же выражении.

Например:

var i = 1;

var a = ++i; // (\*)

alert(a); // 2

В строке (\*) вызов ++i увеличит переменную, а *затем* вернёт ее значение в a. Так что в a попадёт значение i *после* увеличения.

**Постфиксная форма i++ отличается от префиксной ++i тем, что возвращает старое значение, бывшее до увеличения.**

В примере ниже в a попадёт старое значение i, равное 1:

var i = 1;

var a = i++; // (\*)

alert(a); // 1

* Если результат оператора не используется, а нужно только увеличить/уменьшить переменную – без разницы, какую форму использовать:
* var i = 0;
* i++;
* ++i;

alert( i ); // 2

* Если хочется тут же использовать результат, то нужна префиксная форма:
* var i = 0;

alert( ++i ); // 1

* Если нужно увеличить, но нужно значение переменной *до увеличения* – постфиксная форма:
* var i = 0;

alert( i++ ); // 0

**Инкремент/декремент можно использовать в любых выражениях**

При этом он имеет более высокий приоритет и выполняется раньше, чем арифметические операции:

var i = 1;

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

var i = 1;

alert( 2 \* i++ ); // 2, выполнился раньше но значение вернул старое

При этом, нужно с осторожностью использовать такую запись, потому что в более длинной строке при быстром «вертикальном» чтении кода легко пропустить такой i++, и будет неочевидно, что переменая увеличивается.

Три строки, по одному действию в каждой – длиннее, зато нагляднее:

var i = 1;

alert( 2 \* i );

i++;

**[Побитовые операторы](https://learn.javascript.ru/operators" \l "побитовые-операторы)**

Побитовые операторы рассматривают аргументы как 32-разрядные целые числа и работают 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, поэтому вынесены в отдельную главу [Побитовые операторы](https://learn.javascript.ru/bitwise-operators).

**[Сокращённая арифметика с присваиванием](https://learn.javascript.ru/operators" \l "сокращённая-арифметика-с-присваиванием)**

Часто нужно применить оператор к переменной и сохранить результат в ней же, например:

var n = 2;

n = n + 5;

n = n \* 2;

Эту запись можно укоротить при помощи совмещённых операторов, вот так:

var 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

Так можно сделать для операторов +,-,\*,/,% и бинарных <<,>>,>>>,&,|,^.

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

var n = 2;

n \*= 3 + 5;

alert( n ); // 16 (n = 2 \* 8)

**[Оператор запятая](https://learn.javascript.ru/operators" \l "оператор-запятая)**

Один из самых необычных операторов – запятая ','.

Его можно вызвать явным образом, например:

var a = (5, 6);

alert( a );

Запятая позволяет перечислять выражения, разделяя их запятой ','. Каждое из них – вычисляется и отбрасывается, за исключением последнего, которое возвращается.

Запятая – единственный оператор, приоритет которого ниже присваивания. В выражении a = (5,6) для явного задания приоритета использованы скобки, иначе оператор '=' выполнился бы до запятой ',', получилось бы (a=5), 6.

Зачем же нужен такой странный оператор, который отбрасывает значения всех перечисленных выражений, кроме последнего?

Обычно он используется 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++) {

...

}

Такие трюки используются во многих JavaScript-фреймворках для укорачивания кода.

[**Задачи**](https://learn.javascript.ru/operators#tasks)

**[Инкремент, порядок срабатывания](https://learn.javascript.ru/operators" \l "инкремент-порядок-срабатывания)**

важность: 5

Посмотрите, понятно ли вам, почему код ниже работает именно так?

var a = 1, b = 1, c, d;

c = ++a; alert(c); // 2

d = b++; alert(d); // 1

c = (2+ ++a); alert(c); // 5

d = (2+ b++); alert(d); // 4

alert(a); // 3

alert(b); // 3

решение

**[Результат присваивания](https://learn.javascript.ru/operators" \l "результат-присваивания)**

важность: 3

Чему будет равен x в примере ниже?

var a = 2;

var x = 1 + (a \*= 2);

решение

**Операторы сравнения и логические значения**

В этом разделе мы познакомимся с операторами сравнения и с логическими значениями, которые такие операторы возвращают.

Многие операторы сравнения знакомы нам из математики:

* Больше/меньше: a > b, a < b.
* Больше/меньше или равно: a >= b, a <= b.
* Равно a == b. Для сравнения используется два символа равенства '='. Один символ a = b означал бы присваивание.
* «Не равно». В математике он пишется как ≠, в JavaScript – знак равенства с восклицательным знаком перед ним !=.

**[Логические значения](https://learn.javascript.ru/comparison" \l "логические-значения)**

Как и другие операторы, сравнение возвращает значение. Это значение имеет *логический* тип.

Существует всего два логических значения:

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

Например:

alert( 2 > 1 ); // true, верно

alert( 2 == 1 ); // false, неверно

alert( 2 != 1 ); // true

Логические значения можно использовать и напрямую, присваивать переменным, работать с ними как с любыми другими:

var a = true; // присваивать явно

var b = 3 > 4; // или как результат сравнения

alert( b ); // false

alert( a == b ); // (true == false) неверно, выведет false

**[Сравнение строк](https://learn.javascript.ru/comparison" \l "сравнение-строк)**

Строки сравниваются побуквенно:

alert( 'Б' > 'А' ); // true

**Осторожно, Unicode!**

Аналогом «алфавита» во внутреннем представлении строк служит кодировка, у каждого символа – свой номер (код). JavaScript использует кодировку [Unicode](http://ru.wikipedia.org/wiki/%D0%AE%D0%BD%D0%B8%D0%BA%D0%BE%D0%B4).

При этом сравниваются *численные коды символов*. В частности, код у символа Б больше, чем у А, поэтому и результат сравнения такой.

**В кодировке Unicode обычно код у строчной буквы больше, чем у прописной.**

Поэтому регистр имеет значение:

alert( 'а' > 'Я' ); // true, строчные буквы больше прописных

Для корректного сравнения символы должны быть в одинаковом регистре.

Если строка состоит из нескольких букв, то сравнение осуществляется как в телефонной книжке или в словаре. Сначала сравниваются первые буквы, потом вторые, и так далее, пока одна не будет больше другой.

Иными словами, больше – та строка, которая в телефонной книге была бы на большей странице.

Например:

* Если первая буква первой строки больше – значит первая строка больше, независимо от остальных символов:

alert( 'Банан' > 'Аят' );

* Если одинаковы – сравнение идёт дальше. Здесь оно дойдёт до третьей буквы:

alert( 'Вася' > 'Ваня' ); // true, т.к. 'с' > 'н'

* При этом любая буква больше отсутствия буквы:

alert( 'Привет' > 'Прив' ); // true, так как 'е' больше чем "ничего".

Такое сравнение называется *лексикографическим*.

**Важно:**

Обычно мы получаем значения от посетителя в виде строк. Например, prompt возвращает *строку*, которую ввел посетитель.

Числа, полученные таким образом, в виде строк сравнивать нельзя, результат будет неверен. Например:

alert( "2" > "14" ); // true, неверно, ведь 2 не больше 14

В примере выше 2 оказалось больше 14, потому что строки сравниваются посимвольно, а первый символ '2' больше '1'.

Правильно было бы преобразовать их к числу явным образом. Например, поставив перед ними +:

alert( +"2" > +"14" ); // false, теперь правильно

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

При сравнении значений разных типов, используется числовое преобразование. Оно применяется к обоим значениям.

Например:

alert( '2' > 1 ); // true, сравнивается как 2 > 1

alert( '01' == 1 ); // true, сравнивается как 1 == 1

alert( false == 0 ); // true, false становится числом 0

alert( true == 1 ); // true, так как true становится числом 1.

Тема преобразований типов будет продолжена далее, в главе [Преобразование типов для примитивов](https://learn.javascript.ru/types-conversion).

**[Строгое равенство](https://learn.javascript.ru/comparison" \l "строгое-равенство)**

В обычном операторе == есть «проблема» – он не может отличить 0 от false:

alert( 0 == false ); // true

Та же ситуация с пустой строкой:

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

Это естественное следствие того, что операнды разных типов преобразовались к числу. Пустая строка, как и false, при преобразовании к числу дают 0.

Что же делать, если всё же нужно отличить 0 от false?

**Для проверки равенства без преобразования типов используются операторы строгого равенства === (тройное равно) и !==.**

Если тип разный, то они всегда возвращают false:

alert( 0 === false ); // false, т.к. типы различны

Строгое сравнение предпочтительно, если мы хотим быть уверены, что «сюрпризов» не будет.

**[Сравнение с null и undefined](https://learn.javascript.ru/comparison" \l "сравнение-с-null-и-undefined)**

Проблемы со специальными значениями возможны, когда к переменной применяется операция сравнения > < <= >=, а у неё может быть как численное значение, так и null/undefined.

**Интуитивно кажется, что null/undefined эквивалентны нулю, но это не так.**

Они ведут себя по-другому.

1. Значения null и undefined равны == друг другу и не равны чему бы то ни было ещё. Это жёсткое правило буквально прописано в спецификации языка.
2. При преобразовании в число null становится 0, а undefined становится NaN.

Посмотрим забавные следствия.

**[Некорректный результат сравнения null с 0](https://learn.javascript.ru/comparison" \l "некорректный-результат-сравнения-null-с-0)**

Сравним null с нулём:

alert( null > 0 ); // false

alert( null == 0 ); // false

Итак, мы получили, что null не больше и не равен нулю. А теперь…

alert(null >= 0); // true

Как такое возможно? Если нечто *«больше или равно нулю»*, то резонно полагать, что оно либо *больше*, либо *равно*. Но здесь это не так.

Дело в том, что алгоритмы проверки равенства == и сравнения >= > < <= работают по-разному.

Сравнение честно приводит к числу, получается ноль. А при проверке равенства значения null и undefined обрабатываются особым образом: они равны друг другу, но не равны чему-то ещё.

В результате получается странная с точки зрения здравого смысла ситуация, которую мы видели в примере выше.

**[Несравнимый undefined](https://learn.javascript.ru/comparison" \l "несравнимый-undefined)**

Значение undefined вообще нельзя сравнивать:

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

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

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

* Сравнения (1) и (2) дают false потому, что undefined при преобразовании к числу даёт NaN. А значение NaN по стандарту устроено так, что сравнения ==, <, >, <=, >= и даже === с ним возвращают false.
* Проверка равенства (3) даёт false, потому что в стандарте явно прописано, что undefined равно лишь null и ничему другому.

**Вывод: любые сравнения с undefined/null, кроме точного ===, следует делать с осторожностью.**

Желательно не использовать сравнения >= > < <= с ними, во избежание ошибок в коде.

[**Итого**](https://learn.javascript.ru/comparison#итого)

* В JavaScript есть логические значения true (истина) и false (ложь). Операторы сравнения возвращают их.
* Строки сравниваются побуквенно.
* Значения разных типов приводятся к числу при сравнении, за исключением строгого равенства ===(!==).
* Значения null и undefined равны == друг другу и не равны ничему другому. В других сравнениях (с участием >,<) их лучше не использовать, так как они ведут себя не как 0.

Мы ещё вернёмся к теме сравнения позже, когда лучше изучим различные типы данных в JavaScript.

**Побитовые операторы**

Побитовые операторы интерпретируют операнды как последовательность из 32 битов (нулей и единиц). Они производят операции, используя двоичное представление числа, и возвращают новую последовательность из 32 бит (число) в качестве результата.

Эта глава требует дополнительных знаний в программировании и не очень важная, при первом чтении вы можете пропустить её и вернуться потом, когда захотите понять, как побитовые операторы работают.

**[Формат 32-битного целого числа со знаком](https://learn.javascript.ru/bitwise-operators" \l "signed-format)**

Побитовые операторы в JavaScript работают с 32-битными целыми числами в их двоичном представлении.

Это представление называется «32-битное целое со знаком, старшим битом слева и дополнением до двойки».

Разберём, как устроены числа внутри подробнее, это необходимо знать для битовых операций с ними.

* Что такое [двоичная система счисления](http://ru.wikipedia.org/wiki/%C4%E2%EE%E8%F7%ED%E0%FF_%F1%E8%F1%F2%E5%EC%E0_%F1%F7%E8%F1%EB%E5%ED%E8%FF), вам, надеюсь, уже известно. При разборе побитовых операций мы будем обсуждать именно двоичное представление чисел, из 32 бит.
* *Старший бит слева* – это научное название для самого обычного порядка записи цифр (от большего разряда к меньшему). При этом, если больший разряд отсутствует, то соответствующий бит равен нулю.

Примеры представления чисел в двоичной системе:

a = 0; // 00000000000000000000000000000000

a = 1; // 00000000000000000000000000000001

a = 2; // 00000000000000000000000000000010

a = 3; // 00000000000000000000000000000011

a = 255;// 00000000000000000000000011111111

Обратите внимание, каждое число состоит ровно из 32-битов.

* *Дополнение до двойки* – это название способа поддержки отрицательных чисел.

**Двоичный вид числа, обратного данному (например, 5 и -5) получается путём обращения всех битов с прибавлением 1.**

То есть, нули заменяются на единицы, единицы – на нули и к числу прибавляется 1. Получается внутреннее представление того же числа, но со знаком минус.

Например, вот число 314:

00000000000000000000000100111010

Чтобы получить -314, первый шаг – обратить биты числа: заменить 0 на 1, а 1 на 0:

11111111111111111111111011000101

Второй шаг – к полученному двоичному числу прибавить единицу, обычным двоичным сложением: 11111111111111111111111011000101 + 1 = 11111111111111111111111011000110.

Итак, мы получили:

-314 = 11111111111111111111111011000110

Принцип дополнения до двойки делит все двоичные представления на два множества: если крайний-левый бит равен 0 – число положительное, если 1 – число отрицательное. Поэтому этот бит называется *знаковым битом*.

**[Список операторов](https://learn.javascript.ru/bitwise-operators" \l "список-операторов)**

В следующей таблице перечислены все побитовые операторы. Далее операторы разобраны более подробно.

|  |  |  |
| --- | --- | --- |
| **Оператор** | **Использование** | **Описание** |
| Побитовое И (AND) | a & b | Ставит 1 на бит результата, для которого соответствующие биты операндов равны 1. |
| Побитовое ИЛИ (OR) | a | b | Ставит 1 на бит результата, для которого хотя бы один из соответствующих битов операндов равен 1. |
| Побитовое исключающее ИЛИ (XOR) | a ^ b | Ставит 1 на бит результата, для которого только один из соответствующих битов операндов равен 1 (но не оба). |
| Побитовое НЕ (NOT) | ~a | Заменяет каждый бит операнда на противоположный. |
| Левый сдвиг | `a << b` | Сдвигает двоичное представление a на b битов влево, добавляя справа нули. |
| Правый сдвиг, переносящий знак | `a >> b` | Сдвигает двоичное представление a на b битов вправо, отбрасывая сдвигаемые биты. |
| Правый сдвиг с заполнением нулями | `a >>> b` | Сдвигает двоичное представление a на b битов вправо, отбрасывая сдвигаемые биты и добавляя нули слева. |

Побитовые операторы работают следующим образом:

1. Операнды преобразуются в 32-битные целые числа, представленные последовательностью битов. Дробная часть, если она есть, отбрасывается.
2. Для бинарных операторов – каждый бит в первом операнде рассматривается вместе с соответствующим битом второго операнда: первый бит с первым, второй со вторым и т.п. Оператор применяется к каждой паре бит, давая соответствующий бит результата.
3. Получившаяся в результате последовательность бит интерпретируется как обычное число.

Посмотрим, как работают операторы, на примерах.

**Вспомогательные функции parseInt, toString**

Для удобной работы с примерами в этой статье, если вы захотите протестировать что-то в консоли, пригодятся две функции.

* parseInt("11000", 2) – переводит строку с двоичной записью числа в число.
* n.toString(2) – получает для числа n запись в 2-ной системе в виде строки.

Например:

var access = parseInt("11000", 2); // получаем число из строки

alert( access ); // 24, число с таким 2-ным представлением

var access2 = access.toString(2); // обратно двоичную строку из числа

alert( access2 ); // 11000

Без них перевод в двоичную систему и обратно был бы куда менее удобен. Более подробно они разбираются в главе [Числа](https://learn.javascript.ru/number).

**[& (Побитовое И)](https://learn.javascript.ru/bitwise-operators" \l "побитовое-и)**

Выполняет операцию И над каждой парой бит.

Результат a & b равен единице только когда оба бита a и b равны единице.

Таблица истинности для &:

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **a & b** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Пример:

9 (по осн. 10)

= 00000000000000000000000000001001 (по осн. 2)

14 (по осн. 10)

= 00000000000000000000000000001110 (по осн. 2)

--------------------------------

14 & 9 (по осн. 10)

= 00000000000000000000000000001000 (по осн. 2)

= 8 (по осн. 10)

**[| (Побитовое ИЛИ)](https://learn.javascript.ru/bitwise-operators" \l "побитовое-или)**

Выполняет операцию ИЛИ над каждой парой бит. Результат a | b равен 1, если хотя бы один бит из a,bравен 1.

Таблица истинности для |:

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **a | b** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Пример:

9 (по осн. 10)

= 00000000000000000000000000001001 (по осн. 2)

14 (по осн. 10)

= 00000000000000000000000000001110 (по осн. 2)

--------------------------------

14 | 9 (по осн. 10)

= 00000000000000000000000000001111 (по осн. 2)

= 15 (по осн. 10)

**[^ (Исключающее ИЛИ)](https://learn.javascript.ru/bitwise-operators" \l "исключающее-или)**

Выполняет операцию «Исключающее ИЛИ» над каждой парой бит.

a Исключающее ИЛИ b равно 1, если только a=1 или только b=1, но не оба одновременно a=b=1.

Таблица истинности для исключающего ИЛИ:

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **a ^ b** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Как видно, оно даёт 1, если ЛИБО слева 1, ЛИБО справа 1, но не одновременно. Поэтому его и называют «исключающее ИЛИ».

Пример:

9 (по осн. 10)

= 00000000000000000000000000001001 (по осн. 2)

14 (по осн. 10)

= 00000000000000000000000000001110 (по осн. 2)

--------------------------------

14 ^ 9 (по осн. 10)

= 00000000000000000000000000000111 (по осн. 2)

= 7 (по осн. 10)

**Исключающее ИЛИ в шифровании**

Исключающее или можно использовать для шифрования, так как эта операция полностью обратима. Двойное применение исключающего ИЛИ с тем же аргументом даёт исходное число.

Иначе говоря, верна формула: a ^ b ^ b == a.

Пускай Вася хочет передать Пете секретную информацию data. Эта информация заранее превращена в число, например строка интерпретируется как последовательность кодов символов.

Вася и Петя заранее договариваются о числовом ключе шифрования key.

Алгоритм:

* Вася берёт двоичное представление data и делает операцию data ^ key. При необходимости data бьётся на части, равные по длине key, чтобы можно было провести побитовое ИЛИ ^для каждой части. В JavaScript оператор ^ работает с 32-битными целыми числами, так что data нужно разбить на последовательность таких чисел.
* Результат data ^ key отправляется Пете, это шифровка.

Например, пусть в data очередное число равно 9, а ключ key равен 1220461917.

Данные: 9 в двоичном виде

00000000000000000000000000001001

Ключ: 1220461917 в двоичном виде

01001000101111101100010101011101

Результат операции 9 ^ key:

01001000101111101100010101010100

Результат в 10-ной системе (шифровка):

1220461908

* Петя, получив очередное число шифровки 1220461908, применяет к нему такую же операцию ^ key.
* Результатом будет исходное число data.

В нашем случае:

Полученная шифровка в двоичной системе:

9 ^ key = 1220461908

01001000101111101100010101010100

Ключ: 1220461917 в двоичном виде:

01001000101111101100010101011101

Результат операции 1220461917 ^ key:

00000000000000000000000000001001

Результат в 10-ной системе (исходное сообщение):

9

Конечно, такое шифрование поддаётся частотному анализу и другим методам дешифровки, поэтому современные алгоритмы используют операцию XOR ^ как одну из важных частей более сложной многоступенчатой схемы.

**[~ (Побитовое НЕ)](https://learn.javascript.ru/bitwise-operators" \l "побитовое-не)**

Производит операцию НЕ над каждым битом, заменяя его на обратный ему.

Таблица истинности для НЕ:

|  |  |
| --- | --- |
| **a** | **~a** |
| 0 | 1 |
| 1 | 0 |

Пример:

9 (по осн. 10)

= 00000000000000000000000000001001 (по осн. 2)

--------------------------------

~9 (по осн. 10)

= 11111111111111111111111111110110 (по осн. 2)

= -10 (по осн. 10)

Из-за внутреннего представления отрицательных чисел получается так, что ~n == -(n+1).

Например:

alert( ~3 ); // -4

alert( ~-1 ); // 0

**[<< (Битовый сдвиг влево)](https://learn.javascript.ru/bitwise-operators" \l "битовый-сдвиг-влево)**

Операторы битового сдвига принимают два операнда. Первый – это число для сдвига, а второй – количество битов, которые нужно сдвинуть в первом операнде.

Оператор << сдвигает первый операнд на указанное число битов влево. Лишние биты отбрасываются, справа добавляются нулевые биты.

Например, 9 << 2 даст 36:

9 (по осн.10)

= 00000000000000000000000000001001 (по осн.2)

--------------------------------

9 << 2 (по осн.10)

= 00000000000000000000000000100100 (по осн.2)

= 36 (по осн.10)

Операция << 2 сдвинула и отбросила два левых нулевых бита и добавила справа два новых нулевых.

**Левый сдвиг почти равен умножению на 2**

Битовый сдвиг << N обычно имеет тот же эффект, что и умножение на два N раз, например:

alert( 3 << 1 ); // 6, умножение на 2

alert( 3 << 2 ); // 12, умножение на 2 два раза

alert( 3 << 3 ); // 24, умножение на 2 три раза

Конечно, следует иметь в виду, что побитовые операторы работают только с 32-битными числами, поэтому верхний порог такого «умножения» ограничен:

alert(10000000000 << 1); // -1474836480, отброшен крайний-левый бит

alert(10000000000 \* 2); // 20000000000, обычное умножение

**[>> (Правый битовый сдвиг, переносящий знак)](https://learn.javascript.ru/bitwise-operators" \l "правый-битовый-сдвиг-переносящий-знак)**

Этот оператор сдвигает биты вправо, отбрасывая лишние. При этом слева добавляется *копия* крайнего-левого бита.

Знак числа (представленный крайним-левым битом) при этом не меняется, так как новый крайний-левый бит имеет то же значение, что и исходном числе.

Поэтому он назван «переносящим знак».

Например, 9 >> 2 даст 2:

9 (по осн.10)

= 00000000000000000000000000001001 (по осн.2)

--------------------------------

9 >> 2 (по осн.10)

= 00000000000000000000000000000010 (по осн.2)

= 2 (по осн.10)

Операция >> 2 сдвинула вправо и отбросила два правых бита 01 и добавила слева две копии первого бита 00.

Аналогично, -9 >> 2 даст -3:

-9 (по осн.10)

= 11111111111111111111111111110111 (по осн.2)

--------------------------------

-9 >> 2 (по осн.10)

= 11111111111111111111111111111101 (по осн.2) = -3 (по осн.10)

Здесь операция >> 2 сдвинула вправо и отбросила два правых бита 11 и добавила слева две копии первого бита 11. , Знак числа сохранён, так как крайний-левый (знаковый) бит сохранил значение 1.

**Правый сдвиг почти равен целочисленному делению на 2**

Битовый сдвиг >> N обычно имеет тот же результат, что и целочисленное деление на два N раз:

alert( 100 >> 1 ); // 50, деление на 2

alert( 100 >> 2 ); // 25, деление на 2 два раза

alert( 100 >> 3 ); // 12, деление на 2 три раза, целая часть от результата

**[>>> (Правый сдвиг с заполнением нулями)](https://learn.javascript.ru/bitwise-operators" \l "правый-сдвиг-с-заполнением-нулями)**

Этот оператор сдвигает биты первого операнда вправо. Лишние биты справа отбрасываются. Слева добавляются нулевые биты.

Знаковый бит становится равным 0, поэтому результат всегда положителен.

**Для неотрицательных чисел правый сдвиг с заполнением нулями >>> и правый сдвиг с переносом знака >> дадут одинаковый результат, т.к в обоих случаях слева добавятся нули.**

Для отрицательных чисел – результат работы разный. Например, -9 >>> 2 даст 1073741821, в отличие от -9 >> 2 (дает -3):

-9 (по осн.10)

= 11111111111111111111111111110111 (по осн.2)

--------------------------------

-9 >>> 2 (по осн.10)

= 00111111111111111111111111111101 (по осн.2)

= 1073741821 (по осн.10)

**[Применение побитовых операторов](https://learn.javascript.ru/bitwise-operators" \l "применение-побитовых-операторов)**

Побитовые операторы используются редко, но всё же используются.

Случаи применения побитовых операторов, которые мы здесь разберём, составляют большинство всех использований в JavaScript.

**Осторожно, приоритеты!**

В JavaScript побитовые операторы ^, &, | выполняются после сравнений ==.

Например, в сравнении a == b^0 будет сначала выполнено сравнение a == b, а потом уже операция ^0, как будто стоят скобки (a == b)^0.

Обычно это не то, чего мы хотим. Чтобы гарантировать желаемый порядок, нужно ставить скобки: a == (b^0).

**[Маска](https://learn.javascript.ru/bitwise-operators" \l "маска)**

Для этого примера представим, что наш скрипт работает с пользователями.

У них могут быть различные роли в проекте:

* Гость
* Редактор
* Админ

Каждой роли соответствует ряд доступов к статьям и функционалу сайта.

Например, Гость может лишь просматривать статьи сайта, а Редактор – ещё и редактировать их, и тому подобное.

Что-то в таком духе:

| **Пользователь** | **Просмотр статей** | **Изменение статей** | **Просмотр товаров** | **Изменение товаров** | **Управление правами** |
| --- | --- | --- | --- | --- | --- |
| Гость | Да | Нет | Да | Нет | Нет |
| Редактор | Да | Да | Да | Да | Нет |
| Админ | Да | Да | Да | Да | Да |

Если вместо «Да» поставить 1, а вместо «Нет» – 0, то каждый набор доступов описывается числом:

| **Пользователь** | **Просмотр статей** | **Изменение статей** | **Просмотр товаров** | **Изменение товаров** | **Управление правами** | **В 10-ной системе** |
| --- | --- | --- | --- | --- | --- | --- |
| Гость | 1 | 0 | 1 | 0 | 0 | = 20 |
| Редактор | 1 | 1 | 1 | 1 | 0 | = 30 |
| Админ | 1 | 1 | 1 | 1 | 1 | = 31 |

В последней колонке находится десятичное число, которое получится, если прочитать строку доступов в двоичном виде.

Например, доступ гостя 10100 = 20.

Такая интерпретация доступов позволяет «упаковать» много информации в одно число. Это экономит память, а кроме этого – это удобно, поскольку в дополнение к экономии – по такому значению очень легко проверить, имеет ли посетитель заданную *комбинацию доступов*!

Для этого посмотрим, как в 2-ной системе представляется каждый доступ в отдельности.

* Доступ, соответствующий только управлению правами: 00001 (=1) (все нули кроме 1 на позиции, соответствующей этому доступу).
* Доступ, соответствующий только изменению товаров: 00010 (=2).
* Доступ, соответствующий только просмотру товаров: 00100 (=4).
* Доступ, соответствующий только изменению статей: 01000 (=8).
* Доступ, соответствующий только просмотру статей: 10000 (=16).

Доступ одновременно на просмотр и изменение статей – это двоичное число с 1 на соответствующих позициях, то есть access = 11000.

Как правило, доступы задаются в виде констант:

var ACCESS\_ADMIN = 1; // 00001

var ACCESS\_GOODS\_EDIT = 2; // 00010

var ACCESS\_GOODS\_VIEW = 4; // 00100

var ACCESS\_ARTICLE\_EDIT = 8; // 01000

var ACCESS\_ARTICLE\_VIEW = 16; // 10000

Из этих констант получить нужную комбинацию доступов можно при помощи операции |.

var guest = ACCESS\_ARTICLE\_VIEW | ACCESS\_GOODS\_VIEW; // 10100

var editor = guest | ACCESS\_ARTICLE\_EDIT | ACCESS\_GOODS\_EDIT; // 11110

var admin = editor | ACCESS\_ADMIN; // 11111

Теперь, чтобы понять, есть ли в доступе editor нужный доступ, например управление правами – достаточно применить к нему побитовый оператор И (&) с соответствующей константой.

Ненулевой результат будет означать, что доступ есть:

alert(editor & ACCESS\_ADMIN); // 0, доступа нет

alert(editor & ACCESS\_ARTICLE\_EDIT); // 8, доступ есть

Такая проверка работает, потому что оператор И ставит 1 на те позиции результата, на которых в обоих операндах стоит 1.

**Можно проверить один из нескольких доступов.**

Например, проверим, есть ли права на просмотр ИЛИ изменение товаров. Соответствующие права задаются битом 1 на втором и третьем месте с конца, что даёт число 00110 (=6 в 10-ной системе).

var check = ACCESS\_GOODS\_VIEW | ACCESS\_GOODS\_EDIT; // 6, 00110

alert( admin & check ); // не 0, значит есть доступ к просмотру ИЛИ изменению

*Битовой маской* называют как раз комбинацию двоичных значений (check в примере выше), которая используется для проверки и выборки единиц на нужных позициях.

Маски могут быть весьма удобны.

В частности, их используют в функциях, чтобы одним параметром передать несколько «флагов», т.е. однобитных значений.

Пример вызова функции с маской:

// найти пользователей с правами на изменение товаров или администраторов

findUsers(ACCESS\_GOODS\_EDIT | ACCESS\_ADMIN);

Это довольно-таки коротко и элегантно, но, вместе с тем, применение масок налагает определённые ограничения. В частности, побитовые операторы в JavaScript работают только с 32-битными числами, а значит, к примеру, 33 доступа уже в число не упакуешь. Да и работа с двоичной системой счисления – как ни крути, менее удобна, чем с десятичной или с обычными логическими значениями true/false.

Поэтому основная сфера применения масок – это быстрые вычисления, экономия памяти, низкоуровневые операции, связанные с рисованием из JavaScript (3d-графика), интеграция с некоторыми функциями ОС (для серверного JavaScript), и другие ситуации, когда уже существуют функции, требующие битовую маску.

**[Округление](https://learn.javascript.ru/bitwise-operators" \l "округление)**

Так как битовые операции отбрасывают десятичную часть, то их можно использовать для округления. Достаточно взять любую операцию, которая не меняет значение числа.

Например, двойное НЕ (~):

alert( ~~12.345 ); // 12

Подойдёт и Исключающее ИЛИ (^) с нулём:

alert( 12.345 ^ 0 ); // 12

Последнее даже более удобно, поскольку отлично читается:

alert(12.3 \* 14.5 ^ 0); // (=178) "12.3 умножить на 14.5 и округлить"

У побитовых операторов достаточно низкий приоритет, он меньше чем у остальной арифметики:

alert( 1.1 + 1.2 ^ 0 ); // 2, сложение выполнится раньше округления

**[Проверка на&#8209;1](https://learn.javascript.ru/bitwise-operators" \l "проверка-на-1)**

[Внутренний формат](https://learn.javascript.ru/bitwise-operators#signed-format) 32-битных чисел устроен так, что для смены знака нужно все биты заменить на противоположные («обратить») и прибавить 1.

Обращение битов – это побитовое НЕ (~). То есть, при таком формате представления числа -n = ~n + 1. Или, если перенести единицу: ~n = -(n+1).

Как видно из последнего равенства, ~n == 0 только если n == -1. Поэтому можно легко проверить равенство n == -1:

var n = 5;

if (~n) { // сработает, т.к. ~n = -(5+1) = -6

alert( "n не -1" ); // выведет!

}

var n = -1;

if (~n) { // не сработает, т.к. ~n = -(-1+1) = 0

alert( "...ничего не выведет..." );

}

Проверка на -1 пригождается, например, при поиске символа в строке. Вызов str.indexOf("подстрока") возвращает позицию подстроки в str, или -1 если не нашёл.

var str = "Проверка";

if (~str.indexOf("верка")) { // Сочетание "if (~...indexOf)" читается как "если найдено"

alert( 'найдено!' );

}

**[Умножение и деление на степени 2](https://learn.javascript.ru/bitwise-operators" \l "умножение-и-деление-на-степени-2)**

Оператор a << b, сдвигая биты, по сути умножает a на 2 в степени b.

Например:

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

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

alert( 3 << 3 ); // 3\*(2\*2\*2) = 24

При этом следует иметь в виду, что максимальный верхний порог такого умножения меньше, чем обычно, так как побитовый оператор оперирует 32-битными целыми, в то время как обычные операторы оперируют числами длиной 64 бита.

Оператор сдвига в другую сторону a >> b, производит обратную операцию – целочисленное деление aна 2b.

alert( 8 >> 2 ); // 2 = 8/4, убрали 2 нуля в двоичном представлении

alert( 11 >> 2 ); // 2, целочисленное деление (менее значимые биты просто отброшены)

[**Итого**](https://learn.javascript.ru/bitwise-operators#итого)

* Бинарные побитовые операторы: & | ^ << >> >>>.
* Унарный побитовый оператор один: ~.

Как правило, битовое представление числа используется для:

* Округления числа: (12.34^0) = 12.
* Проверки на равенство -1: if (~n) { n не -1 }.
* Упаковки нескольких битововых значений («флагов») в одно значение. Это экономит память и позволяет проверять наличие комбинации флагов одним оператором &.
* Других ситуаций, когда нужны битовые маски.

[**Задачи**](https://learn.javascript.ru/bitwise-operators#tasks)

**[Побитовый оператор и значение](https://learn.javascript.ru/bitwise-operators" \l "побитовый-оператор-и-значение)**

важность: 5

Почему побитовые операции в примерах ниже не меняют число? Что они делают внутри?

alert( 123 ^ 0 ); // 123

alert( 0 ^ 123 ); // 123

alert( ~~123 ); // 123

решение

**[Проверка, целое ли число](https://learn.javascript.ru/bitwise-operators" \l "проверка-целое-ли-число)**

важность: 3

Напишите функцию isInteger(num), которая возвращает true, если num – целое число, иначе false.

Например:

alert( isInteger(1) ); // true

alert( isInteger(1.5) ); // false

alert( isInteger(-0.5) ); // false

решение

**[Симметричны ли операции ^, |, &?](https://learn.javascript.ru/bitwise-operators" \l "симметричны-ли-операции)**

важность: 5

Верно ли, что для любых a и b выполняются равенства ниже?

* a ^ b == b ^ a
* a & b == b & a
* a | b == b | a

Иными словами, при перемене мест – всегда ли результат остаётся тем же?

решение

**[Почему результат разный?](https://learn.javascript.ru/bitwise-operators" \l "почему-результат-разный)**

важность: 5

Почему результат второго alert'а такой странный?

alert( 123456789 ^ 0 ); // 123456789

alert( 12345678912345 ^ 0 ); // 1942903641

решение

**Взаимодействие с пользователем: alert, prompt, confirm**

В этом разделе мы рассмотрим базовые UI операции: alert, prompt и confirm, которые позволяют работать с данными, полученными от пользователя.

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

**Safari 5.1+ не возвращает null**

Единственный браузер, который не возвращает null при отмене ввода – это Safari. При отсутствии ввода он возвращает пустую строку. Предположительно, это ошибка в браузере.

Если нам важен этот браузер, то пустую строку нужно обрабатывать точно так же, как и null, т.е. считать отменой ввода.

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 IE вставит в диалог значение по умолчанию "undefined".

Запустите этот код в IE, чтобы понять о чём речь:

var test = prompt("Тест");

Поэтому рекомендуется *всегда* указывать второй аргумент:

var test = prompt("Тест", ''); // <-- так лучше

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

**[Особенности встроенных функций](https://learn.javascript.ru/uibasic" \l "особенности-встроенных-функций)**

Конкретное место, где выводится модальное окно с вопросом – обычно это центр браузера, и внешний вид окна выбирает браузер. Разработчик не может на это влиять.

С одной стороны – это недостаток, так как нельзя вывести окно в своем, особо красивом, дизайне.

С другой стороны, преимущество этих функций по сравнению с другими, более сложными методами взаимодействия, которые мы изучим в дальнейшем – как раз в том, что они очень просты.

Это самый простой способ вывести сообщение или получить информацию от посетителя. Поэтому их используют в тех случаях, когда простота важна, а всякие «красивости» особой роли не играют.

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

* alert выводит сообщение.
* prompt выводит сообщение и ждёт, пока пользователь введёт текст, а затем возвращает введённое значение или null, если ввод отменён (CANCEL/Esc).
* confirm выводит сообщение и ждёт, пока пользователь нажмёт «OK» или «CANCEL» и возвращает true/false.

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

**[Простая страница](https://learn.javascript.ru/uibasic" \l "простая-страница)**

важность: 4

Создайте страницу, которая спрашивает имя и выводит его.

[Запустить демо](https://learn.javascript.ru/uibasic)решение

**Условные операторы: if, '?'**

Иногда, в зависимости от условия, нужно выполнить различные действия. Для этого используется оператор if.

Например:

var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');

if (year != 2011) alert( 'А вот и неправильно!' );

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

Оператор if («если») получает условие, в примере выше это year != 2011. Он вычисляет его, и если результат – true, то выполняет команду.

Если нужно выполнить более одной команды – они оформляются блоком кода в фигурных скобках:

if (year != 2011) {

alert( 'А вот..' );

alert( '..и неправильно!' );

}

**Рекомендуется использовать фигурные скобки всегда, даже когда команда одна.**

Это улучшает читаемость кода.

**[Преобразование к логическому типу](https://learn.javascript.ru/ifelse" \l "преобразование-к-логическому-типу)**

Оператор if (...) вычисляет и преобразует выражение в скобках к логическому типу.

В логическом контексте:

* Число 0, пустая строка "", null и undefined, а также NaN являются false,
* Остальные значения – true.

Например, такое условие никогда не выполнится:

if (0) { // 0 преобразуется к false

...

}

…А такое – выполнится всегда:

if (1) { // 1 преобразуется к true

...

}

Можно и просто передать уже готовое логическое значение, к примеру, заранее вычисленное в переменной:

var cond = (year != 2011); // true/false

if (cond) {

...

}

**[Неверное условие, else](https://learn.javascript.ru/ifelse" \l "неверное-условие-else)**

Необязательный блок else («иначе») выполняется, если условие неверно:

var year = prompt('Введите год появления стандарта ECMA-262 5.1', '');

if (year == 2011) {

alert( 'Да вы знаток!' );

} else {

alert( 'А вот и неправильно!' ); // любое значение, кроме 2011

}

**[Несколько условий, else if](https://learn.javascript.ru/ifelse" \l "несколько-условий-else-if)**

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

var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');

if (year < 2011) {

alert( 'Это слишком рано..' );

} else if (year > 2011) {

alert( 'Это поздновато..' );

} else {

alert( 'Да, точно в этом году!' );

}

В примере выше JavaScript сначала проверит первое условие, если оно ложно – перейдет ко второму – и так далее, до последнего else.

**[Оператор вопросительный знак „?“](https://learn.javascript.ru/ifelse" \l "оператор-вопросительный-знак)**

Иногда нужно в зависимости от условия присвоить переменную. Например:

var access;

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

if (age > 14) {

access = true;

} else {

access = false;

}

alert(access);

Оператор вопросительный знак '?' позволяет делать это короче и проще.

Он состоит из трех частей:

условие ? значение1 : значение2

Проверяется условие, затем если оно верно – возвращается значение1, если неверно – значение2, например:

access = (age > 14) ? true : false;

Оператор '?' выполняется позже большинства других, в частности – позже сравнений, поэтому скобки можно не ставить:

access = age > 14 ? true : false;

…Но когда скобки есть – код лучше читается. Так что рекомендуется их писать.

**На заметку:**

В данном случае можно было бы обойтись и без оператора '?', т.к. сравнение само по себе уже возвращает true/false:

access = age > 14;

**«Тернарный оператор»**

Вопросительный знак – единственный оператор, у которого есть аж три аргумента, в то время как у обычных операторов их один-два. Поэтому его называют *«тернарный оператор»*.

**[Несколько операторов „?“](https://learn.javascript.ru/ifelse" \l "несколько-операторов)**

Последовательность операторов '?' позволяет вернуть значение в зависимости не от одного условия, а от нескольких.

Например:

var age = prompt('возраст?', 18);

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

(age < 18) ? 'Привет!' :

(age < 100) ? 'Здравствуйте!' :

'Какой необычный возраст!';

alert( message );

Поначалу может быть сложно понять, что происходит. Однако, внимательно приглядевшись, мы замечаем, что это обычная последовательная проверка!

Вопросительный знак проверяет сначала age < 3, если верно – возвращает 'Здравствуй, малыш!', если нет – идет за двоеточие и проверяет age < 18. Если это верно – возвращает 'Привет!', иначе проверка age < 100 и 'Здравствуйте!'… И наконец, если ничего из этого не верно, то 'Какой необычный возраст!'.

То же самое через 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!';

}

**[Нетрадиционное использование „?“](https://learn.javascript.ru/ifelse" \l "нетрадиционное-использование)**

Иногда оператор вопросительный знак '?' используют как замену if:

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

(company == 'Netscape') ?

alert('Да, верно') : alert('Неправильно');

Работает это так: в зависимости от условия, будет выполнена либо первая, либо вторая часть после '?'.

Результат выполнения не присваивается в переменную, так что пропадёт (впрочем, alert ничего не возвращает).

**Рекомендуется не использовать вопросительный знак таким образом.**

Несмотря на то, что с виду такая запись короче if, она является существенно менее читаемой.

Вот, для сравнения, то же самое с if:

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

if (company == 'Netscape') {

alert('Да, верно');

} else {

alert('Неправильно');

}

При чтении кода глаз идёт вертикально и конструкции, занимающие несколько строк, с понятной вложенностью, воспринимаются гораздо легче. Возможно, вы и сами почувствуете, пробежавшись глазами, что синтаксис с if более прост и очевиден чем с оператором '?'.

Смысл оператора '?' – вернуть то или иное значение, в зависимости от условия. Пожалуйста, используйте его по назначению, а для выполнения разных веток кода есть if.

[**Задачи**](https://learn.javascript.ru/ifelse#tasks)

**[if (строка с нулём)](https://learn.javascript.ru/ifelse" \l "if-строка-с-нулём)**

важность: 5

Выведется ли alert?

if ("0") {

alert( 'Привет' );

}

решение

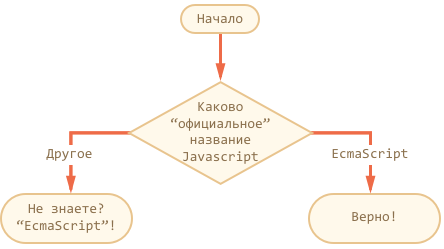
**[Проверка стандарта](https://learn.javascript.ru/ifelse" \l "проверка-стандарта)**

важность: 2

Используя конструкцию if..else, напишите код, который будет спрашивать: «Каково «официальное» название JavaScript?».

Если посетитель вводит «ECMAScript», то выводить «Верно!», если что-то другое – выводить «Не знаете? «ECMAScript»!».

Блок-схема:



[Демо в новом окне](https://learn.javascript.ru/task/check-standard/ifelse_task2/)решение

**[Получить знак числа](https://learn.javascript.ru/ifelse" \l "получить-знак-числа)**

важность: 2

Используя конструкцию if..else, напишите код, который получает значение prompt, а затем выводит alert:

* 1, если значение больше нуля,
* -1, если значение меньше нуля,
* 0, если значение равно нулю.

[Демо в новом окне](https://learn.javascript.ru/task/sign/if_sign/)решение

**[Проверка логина](https://learn.javascript.ru/ifelse" \l "проверка-логина)**

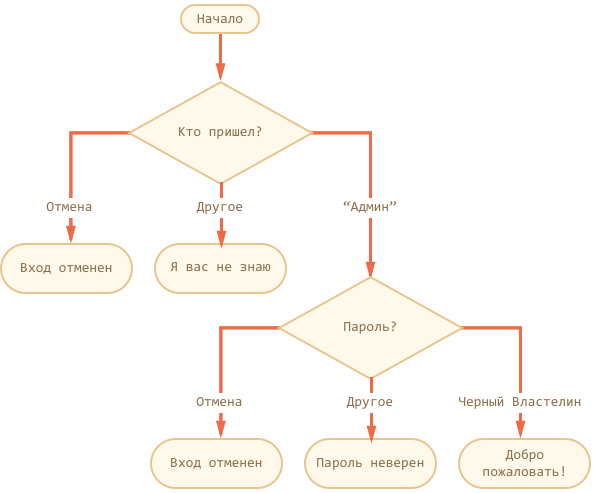
важность: 3

Напишите код, который будет спрашивать логин (prompt).

Если посетитель вводит «Админ», то спрашивать пароль, если нажал отмена (escape) – выводить «Вход отменён», если вводит что-то другое – «Я вас не знаю».

Пароль проверять так. Если введён пароль «Чёрный Властелин», то выводить «Добро пожаловать!», иначе – «Пароль неверен», при отмене – «Вход отменён».

Блок-схема:



Для решения используйте вложенные блоки if. Обращайте внимание на стиль и читаемость кода.

[Запустить демо](https://learn.javascript.ru/ifelse)решение

**[Перепишите 'if' в '?'](https://learn.javascript.ru/ifelse" \l "перепишите-if-в)**

важность: 5

Перепишите if с использованием оператора '?':

if (a + b < 4) {

result = 'Мало';

} else {

result = 'Много';

}

решение

**[Перепишите 'if..else' в '?'](https://learn.javascript.ru/ifelse" \l "перепишите-if-else-в)**

важность: 5

Перепишите if..else с использованием нескольких операторов '?'.

Для читаемости – оформляйте код в несколько строк.

var message;

if (login == 'Вася') {

message = 'Привет';

} else if (login == 'Директор') {

message = 'Здравствуйте';

} else if (login == '') {

message = 'Нет логина';

} else {

message = '';

}

решение

**Логические операторы**

Для операций над логическими значениями в JavaScript есть || (ИЛИ), && (И) и ! (НЕ).

Хоть они и называются *«логическими»*, но в JavaScript могут применяться к значениям любого типа и возвращают также значения любого типа.

**[|| (ИЛИ)](https://learn.javascript.ru/logical-ops" \l "или)**

Оператор ИЛИ выглядит как двойной символ вертикальной черты:

result = a || b;

Логическое ИЛИ в классическом программировании работает следующим образом: "если *хотя бы один* из аргументов true, то возвращает true, иначе – false". В JavaScript, как мы увидим далее, это не совсем так, но для начала рассмотрим только логические значения.

Получается следующая «таблица результатов»:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false ); // true

alert( false || false ); // false

Если значение не логического типа – то оно к нему приводится в целях вычислений. Например, число 1будет воспринято как true, а 0 – как false:

if (1 || 0) { // сработает как if( true || false )

alert( 'верно' );

}

Обычно оператор ИЛИ используется в if, чтобы проверить, выполняется ли хотя бы одно из условий, например:

var hour = 9;

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

alert( 'Офис до 10 или после 18 закрыт' );

}

Можно передать и больше условий:

var hour = 12,

isWeekend = true;

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

alert( 'Офис до 10 или после 18 или в выходной закрыт' );

}

**[Короткий цикл вычислений](https://learn.javascript.ru/logical-ops" \l "короткий-цикл-вычислений)**

JavaScript вычисляет несколько ИЛИ слева направо. При этом, чтобы экономить ресурсы, используется так называемый *«короткий цикл вычисления»*.

Допустим, вычисляются несколько ИЛИ подряд: a || b || c || .... Если первый аргумент – true, то результат заведомо будет true (хотя бы одно из значений – true), и остальные значения игнорируются.

Это особенно заметно, когда выражение, переданное в качестве второго аргумента, имеет *сторонний эффект* – например, присваивает переменную.

При запуске примера ниже присвоение x не произойдёт:

var x;

true || (x = 1);

alert(x); // undefined, x не присвоен

…А в примере ниже первый аргумент – false, так что ИЛИ попытается вычислить второй, запустив тем самым присваивание:

var x;

false || (x = 1);

alert(x); // 1

**[Значение ИЛИ](https://learn.javascript.ru/logical-ops" \l "значение-или)**

*|| запинается на «правде»,  
&& запинается на «лжи».*

Илья Канатов, участник курса JavaScript

Итак, как мы видим, оператор ИЛИ вычисляет ровно столько значений, сколько необходимо – до первого true.

При этом оператор ИЛИ возвращает то значение, на котором остановились вычисления. Причём, не преобразованное к логическому типу.

Например:

alert( 1 || 0 ); // 1

alert( true || 'неважно что' ); // true

alert( null || 1 ); // 1

alert( undefined || 0 ); // 0

Это используют, в частности, чтобы выбрать первое «истинное» значение из списка:

var undef; // переменная не присвоена, т.е. равна undefined

var zero = 0;

var emptyStr = "";

var msg = "Привет!";

var result = undef || zero || emptyStr || msg || 0;

alert( result ); // выведет "Привет!" - первое значение, которое является true

Если все значения «ложные», то || возвратит последнее из них:

alert( undefined || '' || false || 0 ); // 0

Итак, оператор || вычисляет операнды слева направо до первого «истинного» и возвращает его, а если все ложные – то последнее значение.

Иначе можно сказать, что "|| запинается на правде".

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

Оператор И пишется как два амперсанда &&:

result = a && b;

В классическом 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:

var hour = 12,

minute = 30;

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

alert( 'Время 12:30' );

}

Как и в ИЛИ, в И допустимы любые значения:

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

alert( 'не сработает, т.к. условие ложно' );

}

К И применим тот же принцип «короткого цикла вычислений», но немного по-другому, чем к ИЛИ.

Если левый аргумент – false, оператор И возвращает его и заканчивает вычисления. Иначе – вычисляет и возвращает правый аргумент.

Например:

// Первый аргумент - true,

// Поэтому возвращается второй аргумент

alert( 1 && 0 ); // 0

alert( 1 && 5 ); // 5

// Первый аргумент - false,

// Он и возвращается, а второй аргумент игнорируется

alert( null && 5 ); // null

alert( 0 && "не важно" ); // 0

Можно передать и несколько значений подряд, при этом возвратится первое «ложное» (на котором остановились вычисления), а если его нет – то последнее:

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

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

Итак, оператор && вычисляет операнды слева направо до первого «ложного» и возвращает его, а если все истинные – то последнее значение.

Иначе можно сказать, что "&& запинается на лжи".

**Приоритет у && больше, чем у ||**

Приоритет оператора И && больше, чем ИЛИ ||, так что он выполняется раньше.

Поэтому в следующем коде сначала будет вычислено правое И: 1 && 0 = 0, а уже потом – ИЛИ.

alert( 5 || 1 && 0 ); // 5

**Не используйте && вместо if**

Оператор && в простых случаях можно использовать вместо if, например:

var x = 1;

(x > 0) && alert( 'Больше' );

Действие в правой части && выполнится только в том случае, если до него дойдут вычисления. То есть, alert сработает, если в левой части будет true.

Получился аналог:

var x = 1;

if (x > 0) {

alert( 'Больше' );

}

Однако, как правило, вариант с if лучше читается и воспринимается. Он более очевиден, поэтому лучше использовать его. Это, впрочем, относится и к другим неочевидным применениям возможностей языка.

**[! (НЕ)](https://learn.javascript.ru/logical-ops" \l "не)**

Оператор НЕ – самый простой. Он получает один аргумент. Синтаксис:

var result = !value;

Действия !:

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

Например:

alert( !true ); // false

alert( !0 ); // true

**В частности, двойное НЕ используют для преобразования значений к логическому типу:**

alert( !!"строка" ); // true

alert( !!null ); // false

[**Задачи**](https://learn.javascript.ru/logical-ops#tasks)

[**Что выведет alert (ИЛИ)?**](https://learn.javascript.ru/logical-ops#что-выведет-alert-или)

важность: 5

Что выведет код ниже?

alert( null || 2 || undefined );

решение

**[Что выведет alert (ИЛИ)?](https://learn.javascript.ru/logical-ops" \l "что-выведет-alert-или)**

важность: 3

Что выведет код ниже?

alert( alert(1) || 2 || alert(3) );

решение

[**Что выведет alert (И)?**](https://learn.javascript.ru/logical-ops#что-выведет-alert-и)

важность: 5

Что выведет код ниже?

alert( 1 && null && 2 );

решение

**[Что выведет alert (И)?](https://learn.javascript.ru/logical-ops" \l "что-выведет-alert-и)**

важность: 3

Что выведет код ниже?

alert( alert(1) && alert(2) );

решение

**[Что выведет этот код?](https://learn.javascript.ru/logical-ops" \l "что-выведет-этот-код)**

важность: 5

Что выведет код ниже?

alert( null || 2 && 3 || 4 );

решение

**[Проверка if внутри диапазона](https://learn.javascript.ru/logical-ops" \l "проверка-if-внутри-диапазона)**

важность: 3

Напишите условие if для проверки того факта, что переменная age находится между 14 и 90включительно.

«Включительно» означает, что концы промежутка включены, то есть age может быть равна 14 или 90.

решение

**[Проверка if вне диапазона](https://learn.javascript.ru/logical-ops" \l "проверка-if-вне-диапазона)**

важность: 3

Напишите условие if для проверки того факта, что age НЕ находится между 14 и 90 включительно.

Сделайте два варианта условия: первый с использованием оператора НЕ !, второй – без этого оператора.

решение

**[Вопрос про "if"](https://learn.javascript.ru/logical-ops" \l "вопрос-про-if)**

важность: 5

Какие из этих if верны, т.е. выполнятся?

Какие конкретно значения будут результатами выражений в условиях if(...)?

if (-1 || 0) alert( 'первое' );

if (-1 && 0) alert( 'второе' );

if (null || -1 && 1) alert( 'третье' );

решение

**Преобразование типов для примитивов**

Система преобразования типов в JavaScript очень проста, но отличается от других языков. Поэтому она часто служит «камнем преткновения» для приходящих из других языков программистов.

Всего есть три преобразования:

1. Строковое преобразование.
2. Числовое преобразование.
3. Преобразование к логическому значению.

**Эта глава описывает преобразование только примитивных значений, объекты разбираются далее.**

**[Строковое преобразование](https://learn.javascript.ru/types-conversion" \l "строковое-преобразование)**

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. Например, его производит функция alert.

var a = true;

alert( a ); // "true"

Можно также осуществить преобразование явным вызовом String(val):

alert( String(null) === "null" ); // true

Как видно из примеров выше, преобразование происходит наиболее очевидным способом, «как есть»: false становится "false", null – "null", undefined – "undefined" и т.п.

Также для явного преобразования применяется оператор "+", у которого один из аргументов строка. В этом случае он приводит к строке и другой аргумент, например:

alert( true + "test" ); // "truetest"

alert( "123" + undefined ); // "123undefined"

**[Численное преобразование](https://learn.javascript.ru/types-conversion" \l "численное-преобразование)**

Численное преобразование происходит в математических функциях и выражениях, а также при сравнении данных различных типов (кроме сравнений ===, !==).

Для преобразования к числу в явном виде можно вызвать Number(val), либо, что короче, поставить перед выражением унарный плюс "+":

var a = +"123"; // 123

var a = Number("123"); // 123, тот же эффект

| **Значение** | **Преобразуется в...** |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true / false | 1 / 0 |
| Строка | Пробельные символы по краям обрезаются. Далее, если остаётся пустая строка, то 0, иначе из непустой строки "считывается" число, при ошибке результат NaN. |

Например:

// после обрезания пробельных символов останется "123"

alert( +" \n 123 \n \n" ); // 123

Ещё примеры:

* Логические значения:
* alert( +true ); // 1

alert( +false ); // 0

* Сравнение разных типов – значит численное преобразование:

alert( "\n0 " == 0 ); // true

При этом строка "\n0" преобразуется к числу, как указано выше: начальные и конечные пробелы обрезаются, получается строка "0", которая равна 0.

* С логическими значениями:
* alert( "\n" == false );

alert( "1" == true );

Здесь сравнение "==" снова приводит обе части к числу. В первой строке слева и справа получается 0, во второй 1.

**[Специальные значения](https://learn.javascript.ru/types-conversion" \l "специальные-значения)**

Посмотрим на поведение специальных значений более внимательно.

**Интуитивно, значения null/undefined ассоциируются с нулём, но при преобразованиях ведут себя иначе.**

Специальные значения преобразуются к числу так:

|  |  |
| --- | --- |
| **Значение** | **Преобразуется в...** |
| undefined | NaN |
| null | 0 |

Это преобразование осуществляется при арифметических операциях и сравнениях > >= < <=, но не при проверке равенства ==. Алгоритм проверки равенства для этих значений в спецификации прописан отдельно (пункт [11.9.3](http://es5.github.com/x11.html#x11.9.3)). В нём считается, что null и undefined равны "==" между собой, но эти значения не равны никакому другому значению.

Это ведёт к забавным последствиям.

Например, null не подчиняется законам математики – он «больше либо равен нулю»: null>=0, но не больше и не равен:

alert( null >= 0 ); // true, т.к. null преобразуется к 0

alert( null > 0 ); // false (не больше), т.к. null преобразуется к 0

alert( null == 0 ); // false (и не равен!), т.к. == рассматривает null особо.

Значение undefined вообще «несравнимо»:

alert( undefined > 0 ); // false, т.к. undefined -> NaN

alert( undefined == 0 ); // false, т.к. это undefined (без преобразования)

alert( undefined < 0 ); // false, т.к. undefined -> NaN

**Для более очевидной работы кода и во избежание ошибок лучше не давать специальным значениям участвовать в сравнениях > >= < <=.**

Используйте в таких случаях переменные-числа или приводите к числу явно.

**[Логическое преобразование](https://learn.javascript.ru/types-conversion" \l "логическое-преобразование)**

Преобразование к true/false происходит в логическом контексте, таком как if(value), и при применении логических операторов.

Все значения, которые интуитивно «пусты», становятся false. Их несколько: 0, пустая строка, null, undefined и NaN.

Остальное, в том числе и любые объекты – true.

Полная таблица преобразований:

|  |  |
| --- | --- |
| **Значение** | **Преобразуется в...** |
| undefined, null | false |
| Числа | Все true, кроме 0, NaN -- false. |
| Строки | Все true, кроме пустой строки "" -- false |
| Объекты | Всегда true |

**Для явного преобразования используется двойное логическое отрицание !!value или вызов Boolean(value).**

**Обратите внимание: строка "0" становится true**

В отличие от многих языков программирования (например PHP), "0" в JavaScript является true, как и строка из пробелов:

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

alert( !!" " ); // любые непустые строки, даже из пробелов - true!

Логическое преобразование интересно тем, как оно сочетается с численным.

**Два значения могут быть равны, но одно из них в логическом контексте true, другое – false**.

Например, равенство в следующем примере верно, так как происходит численное преобразование:

alert( 0 == "\n0\n" ); // true

…А в логическом контексте левая часть даст false, правая – true:

if ("\n0\n") {

alert( "true, совсем не как 0!" );

}

С точки зрения преобразования типов в JavaScript это совершенно нормально. При сравнении с помощью «==» – численное преобразование, а в if – логическое, только и всего.

[**Итого**](https://learn.javascript.ru/types-conversion#итого)

В JavaScript есть три преобразования:

1. Строковое: String(value) – в строковом контексте или при сложении со строкой. Работает очевидным образом.
2. Численное: Number(value) – в численном контексте, включая унарный плюс +value. Происходит при сравнении разных типов, кроме строгого равенства.
3. Логическое: Boolean(value) – в логическом контексте, можно также сделать двойным НЕ: !!value.

Точные таблицы преобразований даны выше в этой главе.

Особым случаем является проверка равенства с null и undefined. Они равны друг другу, но не равны чему бы то ни было ещё, этот случай прописан особо в спецификации.

[**Задачи**](https://learn.javascript.ru/types-conversion#tasks)

**[Вопросник по преобразованиям, для примитивов](https://learn.javascript.ru/types-conversion" \l "вопросник-по-преобразованиям-для-примитивов)**

важность: 5

Подумайте, какой результат будет у выражений ниже. Тут не только преобразования типов. Когда закончите – сверьтесь с решением.

"" + 1 + 0

"" - 1 + 0

true + false

6 / "3"

"2" \* "3"

4 + 5 + "px"

"$" + 4 + 5

"4" - 2

"4px" - 2

7 / 0

" -9\n" + 5

" -9\n" - 5

5 && 2

2 && 5

5 || 0

0 || 5

null + 1

undefined + 1

null == "\n0\n"

+null == +"\n0\n"

решение

**Циклы while, for**

При написании скриптов зачастую встает 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" \l "цикл-while)**

Цикл while имеет вид:

while (условие) {

// код, тело цикла

}

Пока условие верно – выполняется код из тела цикла.

Например, цикл ниже выводит i пока i < 3:

var i = 0;

while (i < 3) {

alert( i );

i++;

}

**Повторение цикла по-научному называется *«итерация»*. Цикл в примере выше совершает три итерации.**

Если бы i++ в коде выше не было, то цикл выполнялся бы (в теории) вечно. На практике, браузер выведет сообщение о «зависшем» скрипте и посетитель его остановит.

**Бесконечный цикл** можно сделать и проще:

while (true) {

// ...

}

**Условие в скобках интерпретируется как логическое значение, поэтому вместо while (i!=0)обычно пишут while (i)**:

var i = 3;

while (i) { // при i, равном 0, значение в скобках будет false и цикл остановится

alert( i );

i--;

}

**[Цикл do…while](https://learn.javascript.ru/while-for" \l "цикл-do-while)**

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

do {

// тело цикла

} while (условие);

Цикл, описанный, таким образом, сначала выполняет тело, а затем проверяет условие.

Например:

var i = 0;

do {

alert( i );

i++;

} while (i < 3);

Синтаксис do..while редко используется, т.к. обычный while нагляднее – в нём не приходится искать глазами условие и ломать голову, почему оно проверяется именно в конце.

**[Цикл for](https://learn.javascript.ru/while-for" \l "цикл-for)**

Чаще всего применяется цикл for. Выглядит он так:

for (начало; условие; шаг) {

// ... тело цикла ...

}

Пример цикла, который выполняет alert(i) для i от 0 до 2 включительно (до 3):

var i;

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

alert( i );

}

Здесь:

* **Начало:** i=0.
* **Условие:** i<3.
* **Шаг:** i++.
* **Тело:** alert(i), т.е. код внутри фигурных скобок (они не обязательны, если только одна операция)

Цикл выполняется так:

1. Начало: i=0 выполняется один-единственный раз, при заходе в цикл.
2. Условие: i<3 проверяется перед каждой итерацией и при входе в цикл, если оно нарушено, то происходит выход.
3. Тело: alert(i).
4. Шаг: i++ выполняется после *тела* на каждой итерации, но перед проверкой условия.
5. Идти на шаг 2.

Иными словами, поток выполнения: начало → (если условие → тело → шаг) → (если условие → тело→ шаг) → … и так далее, пока верно условие.

**На заметку:**

В цикле также можно определить переменную:

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

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

}

Эта переменная будет видна и за границами цикла, в частности, после окончания цикла i станет равно 3.

**[Пропуск частей for](https://learn.javascript.ru/while-for" \l "пропуск-частей-for)**

Любая часть for может быть пропущена.

Например, можно убрать начало. Цикл в примере ниже полностью идентичен приведённому выше:

var i = 0;

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

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

}

Можно убрать и шаг:

var i = 0;

for (; i < 3;) {

alert( i );

// цикл превратился в аналог while (i<3)

}

А можно и вообще убрать всё, получив бесконечный цикл:

for (;;) {

// будет выполняться вечно

}

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

**for..in**

Существует также специальная конструкция for..in для перебора свойств объекта.

Мы познакомимся с ней позже, когда будем [говорить об объектах](https://learn.javascript.ru/object-for-in#for..in).

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

Выйти из цикла можно не только при проверке условия но и, вообще, в любой момент. Эту возможность обеспечивает директива break.

Например, следующий код подсчитывает сумму вводимых чисел до тех пор, пока посетитель их вводит, а затем – выдаёт:

var sum = 0;

while (true) {

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

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

sum += value;

}

alert( 'Сумма: ' + sum );

Директива break в строке (\*), если посетитель ничего не ввёл, полностью прекращает выполнение цикла и передаёт управление на строку за его телом, то есть на alert.

Вообще, сочетание «бесконечный цикл + break» – отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале-конце цикла, а посередине.

**[Следующая итерация: continue](https://learn.javascript.ru/while-for" \l "continue)**

Директива continue прекращает выполнение *текущей итерации* цикла.

Она – в некотором роде «младшая сестра» директивы break: прерывает не весь цикл, а только текущее выполнение его тела, как будто оно закончилось.

Её используют, если понятно, что на текущем повторе цикла делать больше нечего.

Например, цикл ниже использует continue, чтобы не выводить чётные значения:

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

if (i % 2 == 0) continue;

alert(i);

}

Для чётных i срабатывает continue, выполнение тела прекращается и управление передаётся на следующий проход for.

**Директива continue позволяет обойтись без скобок**

Цикл, который обрабатывает только нечётные значения, мог бы выглядеть так:

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

if (i % 2) {

alert( i );

}

}

С технической точки зрения он полностью идентичен. Действительно, вместо continue можно просто завернуть действия в блок if. Однако, мы получили дополнительный уровень вложенности фигурных скобок. Если код внутри if более длинный, то это ухудшает читаемость, в отличие от варианта с continue.

**Нельзя использовать break/continue справа от оператора „?“**

Обычно мы можем заменить if на оператор вопросительный знак '?'.

То есть, запись:

if (условие) {

a();

} else {

b();

}

…Аналогична записи:

условие ? a() : b();

В обоих случаях в зависимости от условия выполняется либо a() либо b().

Но разница состоит в том, что оператор вопросительный знак '?', использованный во второй записи, возвращает значение.

**Синтаксические конструкции, которые не возвращают значений, нельзя использовать в операторе '?'.**

К таким относятся большинство конструкций и, в частности, break/continue.

Поэтому такой код приведёт к ошибке:

(i > 5) ? alert(i) : continue;

Впрочем, как уже говорилось ранее, оператор вопросительный знак '?' не стоит использовать таким образом. Это – всего лишь ещё одна причина, почему для проверки условия предпочтителен if.

**[Метки для break/continue](https://learn.javascript.ru/while-for" \l "метки-для-break-continue)**

Бывает нужно выйти одновременно из нескольких уровней цикла.

Например, внутри цикла по i находится цикл по j, и при выполнении некоторого условия мы бы хотели выйти из обоих циклов сразу:

outer: for (var i = 0; i < 3; i++) {

for (var j = 0; j < 3; j++) {

var input = prompt('Значение в координатах '+i+','+j, '');

// если отмена ввода или пустая строка -

// завершить оба цикла

if (!input) break outer; // (\*)

}

}

alert('Готово!');

В коде выше для этого использована *метка*.

Метка имеет вид "имя:", имя должно быть уникальным. Она ставится перед циклом, вот так:

outer: for (var i = 0; i < 3; i++) { ... }

Можно также выносить её на отдельную строку:

outer:

for (var i = 0; i < 3; i++) { ... }

Вызов break outer ищет ближайший внешний цикл с такой меткой и переходит в его конец.

В примере выше это означает, что будет разорван самый внешний цикл и управление перейдёт на alert.

Директива continue также может быть использована с меткой, в этом случае управление перепрыгнет на следующую итерацию цикла с меткой.

[**Итого**](https://learn.javascript.ru/while-for#итого)

JavaScript поддерживает три вида циклов:

* while – проверка условия перед каждым выполнением.
* do..while – проверка условия после каждого выполнения.
* for – проверка условия перед каждым выполнением, а также дополнительные настройки.

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

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

Обе этих директивы поддерживают «метки», которые ставятся перед циклом. Метки – единственный способ для break/continue повлиять на выполнение внешнего цикла.

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

[**Задачи**](https://learn.javascript.ru/while-for#tasks)

**[Последнее значение цикла](https://learn.javascript.ru/while-for" \l "последнее-значение-цикла)**

важность: 3

Какое последнее значение выведет этот код? Почему?

var i = 3;

while (i) {

alert( i-- );

}

решение

**[Какие значения i выведет цикл while?](https://learn.javascript.ru/while-for" \l "какие-значения-i-выведет-цикл-while)**

важность: 4

Для каждого цикла запишите, какие значения он выведет. Потом сравните с ответом.

1. Префиксный вариант
2. var i = 0;

while (++i < 5) alert( i );

1. Постфиксный вариант
2. var i = 0;

while (i++ < 5) alert( i );

решение

**[Какие значения i выведет цикл for?](https://learn.javascript.ru/while-for" \l "какие-значения-i-выведет-цикл-for)**

важность: 4

Для каждого цикла запишите, какие значения он выведет. Потом сравните с ответом.

1. Постфиксная форма:

for (var i = 0; i < 5; i++) alert( i );

1. Префиксная форма:

for (var i = 0; i < 5; ++i) alert( i );

решение

**[Выведите чётные числа](https://learn.javascript.ru/while-for" \l "выведите-чётные-числа)**

важность: 5

При помощи цикла for выведите чётные числа от 2 до 10.

[Запустить демо](https://learn.javascript.ru/while-for)решение

**[Замените for на while](https://learn.javascript.ru/while-for" \l "замените-for-на-while)**

важность: 5

Перепишите код, заменив цикл for на while, без изменения поведения цикла.

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

alert( "номер " + i + "!" );

}

решение

**[Повторять цикл, пока ввод неверен](https://learn.javascript.ru/while-for" \l "повторять-цикл-пока-ввод-неверен)**

важность: 5

Напишите цикл, который предлагает prompt ввести число, большее 100. Если посетитель ввёл другое число – попросить ввести ещё раз, и так далее.

Цикл должен спрашивать число пока либо посетитель не введёт число, большее 100, либо не нажмёт кнопку Cancel (ESC).

Предполагается, что посетитель вводит только числа. Предусматривать обработку нечисловых строк в этой задаче необязательно.

[Запустить демо](https://learn.javascript.ru/while-for)решение

**[Вывести простые числа](https://learn.javascript.ru/while-for" \l "вывести-простые-числа)**

важность: 3

Натуральное число, большее 1, называется *простым*, если оно ни на что не делится, кроме себя и 1.

Другими словами, n>1 – простое, если при делении на любое число от 2 до n-1 есть остаток.

**Создайте код, который выводит все простые числа из интервала от 2 до 10.** Результат должен быть: 2,3,5,7.

P.S. Код также должен легко модифицироваться для любых других интервалов.

решение

**Конструкция switch**

Конструкция switch заменяет собой сразу несколько if.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

**[Синтаксис](https://learn.javascript.ru/switch" \l "синтаксис)**

Выглядит она так:

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.

При этом case называют *вариантами switch*.

**[Пример работы](https://learn.javascript.ru/switch" \l "пример-работы)**

Пример использования switch (сработавший код выделен):

var a = 2 + 2;

switch (a) {

case 3:

alert( 'Маловато' );

break;

case 4:

alert( 'В точку!' );

break;

case 5:

alert( 'Перебор' );

break;

default:

alert( 'Я таких значений не знаю' );

}

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:

var a = 2 + 2;

switch (a) {

case 3:

alert( 'Маловато' );

case 4:

alert( 'В точку!' );

case 5:

alert( 'Перебор' );

default:

alert( 'Я таких значений не знаю' );

}

В примере выше последовательно выполнятся три alert:

alert( 'В точку!' );

alert( 'Перебор' );

alert( 'Я таких значений не знаю' );

В case могут быть любые выражения, в том числе включающие в себя переменные и функции.

Например:

var a = 1;

var b = 0;

switch (a) {

case b + 1:

alert( 1 );

break;

default:

alert('нет-нет, выполнится вариант выше')

}

**[Группировка case](https://learn.javascript.ru/switch" \l "группировка-case)**

Несколько значений case можно группировать.

В примере ниже case 3 и case 5 выполняют один и тот же код:

var a = 2+2;

switch (a) {

case 4:

alert('Верно!');

break;

case 3: // (\*)

case 5: // (\*\*)

alert('Неверно!');

alert('Немного ошиблись, бывает.');

break;

default:

alert('Странный результат, очень странный');

}

При case 3 выполнение идёт со строки (\*), при case 5 – со строки (\*\*).

**[Тип имеет значение](https://learn.javascript.ru/switch" \l "тип-имеет-значение)**

Следующий пример принимает значение от посетителя.

var arg = prompt("Введите arg?")

switch (arg) {

case '0':

case '1':

alert( 'Один или ноль' );

case '2':

alert( 'Два' );

break;

case 3:

alert( 'Никогда не выполнится' );

default:

alert('Неизвестное значение: ' + arg)

}

Что оно выведет при вводе числа 0? Числа 1? 2? 3?

Подумайте, выпишите свои ответы, исходя из текущего понимания работы switch и *потом* читайте дальше…

* При вводе 0 выполнится первый alert, далее выполнение продолжится вниз до первого break и выведет второй alert('Два'). Итого, два вывода alert.
* При вводе 1 произойдёт то же самое.
* При вводе 2, switch перейдет к case '2', и сработает единственный alert('Два').
* **При вводе 3, switch перейдет на default.** Это потому, что prompt возвращает строку '3', а не число. Типы разные. Оператор switch предполагает строгое равенство ===, так что совпадения не будет.

[**Задачи**](https://learn.javascript.ru/switch#tasks)

**[Напишите "if", аналогичный "switch"](https://learn.javascript.ru/switch" \l "напишите-if-аналогичный-switch)**

важность: 5

Напишите if..else, соответствующий следующему switch:

switch (browser) {

case 'IE':

alert( 'О, да у вас IE!' );

break;

case 'Chrome':

case 'Firefox':

case 'Safari':

case 'Opera':

alert( 'Да, и эти браузеры мы поддерживаем' );

break;

default:

alert( 'Мы надеемся, что и в вашем браузере все ок!' );

}

решение

**[Переписать if'ы в switch](https://learn.javascript.ru/switch" \l "переписать-if-ы-в-switch)**

важность: 4

Перепишите код с использованием одной конструкции switch:

var a = +prompt('a?', '');

if (a == 0) {

alert( 0 );

}

if (a == 1) {

alert( 1 );

}

if (a == 2 || a == 3) {

alert( '2,3' );

}

решение

**Функции**

Зачастую нам надо повторять одно и то же действие во многих частях программы.

Например, красиво вывести сообщение необходимо при приветствии посетителя, при выходе посетителя с сайта, ещё где-нибудь.

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

Примеры встроенных функций вы уже видели – это alert(message), prompt(message, default) и confirm(question). Но можно создавать и свои.

**[Объявление](https://learn.javascript.ru/function-basics" \l "объявление)**

Пример объявления функции:

function showMessage() {

alert( 'Привет всем присутствующим!' );

}

Вначале идет ключевое слово function, после него *имя функции*, затем *список параметров* в скобках (в примере выше он пустой) и *тело функции* – код, который выполняется при её вызове.

Объявленная функция доступна по имени, например:

function showMessage() {

alert( 'Привет всем присутствующим!' );

}

showMessage();

showMessage();

Этот код выведет сообщение два раза. Уже здесь видна **главная цель создания функций: избавление от дублирования кода**.

Если понадобится поменять сообщение или способ его вывода – достаточно изменить его в одном месте: в функции, которая его выводит.

**[Локальные переменные](https://learn.javascript.ru/function-basics" \l "локальные-переменные)**

Функция может содержать *локальные* переменные, объявленные через var. Такие переменные видны только внутри функции:

function showMessage() {

var message = 'Привет, я - Вася!'; // локальная переменная

alert( message );

}

showMessage(); // 'Привет, я - Вася!'

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

**Блоки if/else, switch, for, while, do..while не влияют на область видимости переменных.**

При объявлении переменной в таких блоках, она всё равно будет видна во всей функции.

Например:

function count() {

// переменные i,j не будут уничтожены по окончании цикла

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

var j = i \* 2;

}

alert( i ); // i=3, последнее значение i, при нём цикл перестал работать

alert( j ); // j=4, последнее значение j, которое вычислил цикл

}

**Неважно, где именно в функции и сколько раз объявляется переменная. Любое объявление срабатывает один раз и распространяется на всю функцию.**

Объявления переменных в примере выше можно передвинуть вверх, это ни на что не повлияет:

function count() {

var i, j; // передвинули объявления var в начало

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

j = i \* 2;

}

alert( i ); // i=3

alert( j ); // j=4

}

**[Внешние переменные](https://learn.javascript.ru/function-basics" \l "внешние-переменные)**

Функция может обратиться ко внешней переменной, например:

var userName = 'Вася';

function showMessage() {

var message = 'Привет, я ' + userName;

alert(message);

}

showMessage(); // Привет, я Вася

Доступ возможен не только на чтение, но и на запись. При этом, так как переменная внешняя, то изменения будут видны и снаружи функции:

var userName = 'Вася';

function showMessage() {

userName = 'Петя'; // (1) присвоение во внешнюю переменную

var message = 'Привет, я ' + userName;

alert( message );

}

showMessage();

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

Конечно, если бы внутри функции, в строке (1), была бы объявлена своя локальная переменная var userName, то все обращения использовали бы её, и внешняя переменная осталась бы неизменной.

**Переменные, объявленные на уровне всего скрипта, называют *«глобальными переменными»*.**

В примере выше переменная userName – глобальная.

Делайте глобальными только те переменные, которые действительно имеют общее значение для вашего проекта, а нужные для решения конкретной задачи – пусть будут локальными в соответствующей функции.

**Внимание: неявное объявление глобальных переменных!**

В старом стандарте JavaScript существовала возможность неявного объявления переменных присвоением значения.

Например:

function showMessage() {

message = 'Привет'; // без var!

}

showMessage();

alert( message ); // Привет

В коде выше переменная message нигде не объявлена, а сразу присваивается. Скорее всего, программист просто забыл поставить var.

При use strict такой код привёл бы к ошибке, но без него переменная будет создана автоматически, причём в примере выше она создаётся не в функции, а на уровне всего скрипта.

Избегайте этого.

Здесь опасность даже не в автоматическом создании переменной, а в том, что глобальные переменные должны использоваться тогда, когда действительно нужны «общескриптовые» параметры.

Забыли var в одном месте, потом в другом – в результате одна функция неожиданно поменяла глобальную переменную, которую использует другая. И поди разберись, кто и когда её поменял, не самая приятная ошибка для отладки.

В будущем, когда мы лучше познакомимся с основами JavaScript, в главе [Замыкания, функции изнутри](https://learn.javascript.ru/closures), мы более детально рассмотрим внутренние механизмы работы переменных и функций.

**[Параметры](https://learn.javascript.ru/function-basics" \l "параметры)**

При вызове функции ей можно передать данные, которые та использует по своему усмотрению.

Например, этот код выводит два сообщения:

function showMessage(from, text) { // параметры from, text

from = "\*\* " + from + " \*\*"; // здесь может быть сложный код оформления

alert(from + ': ' + text);

}

showMessage('Маша', 'Привет!');

showMessage('Маша', 'Как дела?');

**Параметры копируются в локальные переменные функции**.

Например, в коде ниже есть внешняя переменная from, значение которой при запуске функции копируется в параметр функции с тем же именем. Далее функция работает уже с параметром:

function showMessage(from, text) {

from = '\*\*' + from + '\*\*'; // меняем локальную переменную from

alert( from + ': ' + text );

}

var from = "Маша";

showMessage(from, "Привет");

alert( from ); // старое значение from без изменений, в функции была изменена копия

**[Аргументы по умолчанию](https://learn.javascript.ru/function-basics" \l "аргументы-по-умолчанию)**

Функцию можно вызвать с любым количеством аргументов.

Если параметр не передан при вызове – он считается равным undefined.

Например, функцию показа сообщения showMessage(from, text) можно вызвать с одним аргументом:

showMessage("Маша");

При этом можно проверить, и если параметр не передан – присвоить ему значение «по умолчанию»:

function showMessage(from, text) {

if (text === undefined) {

text = 'текст не передан';

}

alert( from + ": " + text );

}

showMessage("Маша", "Привет!"); // Маша: Привет!

showMessage("Маша"); // Маша: текст не передан

**При объявлении функции необязательные аргументы, как правило, располагают в конце списка.**

Для указания значения «по умолчанию», то есть, такого, которое используется, если аргумент не указан, используется два способа:

1. Можно проверить, равен ли аргумент undefined, и если да – то записать в него значение по умолчанию. Этот способ продемонстрирован в примере выше.
2. Использовать оператор ||:
3. function showMessage(from, text) {
4. text = text || 'текст не передан';
5. ...

}

Второй способ считает, что аргумент отсутствует, если передана пустая строка, 0, или вообще любое значение, которое в логическом контексте является false.

Если аргументов передано больше, чем надо, например showMessage("Маша", "привет", 1, 2, 3), то ошибки не будет. Но, чтобы получить такие «лишние» аргументы, нужно будет прочитать их из специального объекта arguments, который мы рассмотрим в главе [Псевдомассив аргументов "arguments"](https://learn.javascript.ru/arguments-pseudoarray).

**[Возврат значения](https://learn.javascript.ru/function-basics" \l "возврат-значения)**

Функция может возвратить результат, который будет передан в вызвавший её код.

Например, создадим функцию calcD, которая будет возвращать дискриминант квадратного уравнения по формуле b2 – 4ac:

function calcD(a, b, c) {

return b\*b - 4\*a\*c;

}

var test = calcD(-4, 2, 1);

alert(test); // 20

**Для возврата значения используется директива return.**

Она может находиться в любом месте функции. Как только до неё доходит управление – функция завершается и значение передается обратно.

Вызовов return может быть и несколько, например:

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Родители разрешили?');

}

}

var age = prompt('Ваш возраст?');

if (checkAge(age)) {

alert( 'Доступ разрешен' );

} else {

alert( 'В доступе отказано' );

}

Директива return может также использоваться без значения, чтобы прекратить выполнение и выйти из функции.

Например:

function showMovie(age) {

if (!checkAge(age)) {

return;

}

alert( "Фильм не для всех" ); // (\*)

// ...

}

В коде выше, если сработал if, то строка (\*) и весь код под ней никогда не выполнится, так как returnзавершает выполнение функции.

**Значение функции без return и с пустым return**

В случае, когда функция не вернула значение или return был без аргументов, считается что она вернула undefined:

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

alert( doNothing() ); // undefined

Обратите внимание, никакой ошибки нет. Просто возвращается undefined.

Ещё пример, на этот раз с return без аргумента:

function doNothing() {

return;

}

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

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

Имя функции следует тем же правилам, что и имя переменной. Основное отличие – оно должно быть глаголом, т.к. функция – это действие.

Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение.

Функции, которые начинаются с "show" – что-то показывают:

showMessage(..) // префикс show, "показать" сообщение

Функции, начинающиеся с "get" – получают, и т.п.:

getAge(..) // get, "получает" возраст

calcD(..) // calc, "вычисляет" дискриминант

createForm(..) // create, "создает" форму

checkPermission(..) // check, "проверяет" разрешение, возвращает true/false

Это очень удобно, поскольку взглянув на функцию – мы уже примерно представляем, что она делает, даже если функцию написал совсем другой человек, а в отдельных случаях – и какого вида значение она возвращает.

**Одна функция – одно действие**

Функция должна делать только то, что явно подразумевается её названием. И это должно быть одно действие.

Если оно сложное и подразумевает поддействия – может быть имеет смысл выделить их в отдельные функции? Зачастую это имеет смысл, чтобы лучше структурировать код.

**…Но самое главное – в функции не должно быть ничего, кроме самого действия и поддействий, неразрывно связанных с ним.**

Например, функция проверки данных (скажем, "validate") не должна показывать сообщение об ошибке. Её действие – проверить.

**Сверхкороткие имена функций**

Имена функций, которые используются *очень часто*, иногда делают сверхкороткими.

Например, во фреймворке [jQuery](http://jquery.com/) есть функция $, во фреймворке [Prototype](http://prototypejs.org/) – функция $$, а в библиотеке [LoDash](http://lodash.com/) очень активно используется функция с названием из одного символа подчеркивания \_.

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

Объявление функции имеет вид:

function имя(параметры, через, запятую) {

код функции

}

* Передаваемые значения копируются в параметры функции и становятся локальными переменными.
* Параметры функции копируются в её локальные переменные.
* Можно объявить новые локальные переменые при помощи var.
* Значение возвращается оператором return ....
* Вызов return тут же прекращает функцию.
* Если return; вызван без значения, или функция завершилась без return, то её результат равен undefined.

При обращении к необъявленной переменной функция будет искать внешнюю переменную с таким именем, но лучше, если функция использует только локальные переменные:

* Это делает очевидным общий поток выполнения – что передаётся в функцию и какой получаем результат.
* Это предотвращает возможные конфликты доступа, когда две функции, возможно написанные в разное время или разными людьми, неожиданно друг для друга меняют одну и ту же внешнюю переменную.

Именование функций:

* Имя функции должно понятно и чётко отражать, что она делает. Увидев её вызов в коде, вы должны тут же понимать, что она делает.
* Функция – это действие, поэтому для имён функций, как правило, используются глаголы.

Функции являются основными строительными блоками скриптов. Мы будем неоднократно возвращаться к ним и изучать все более и более глубоко.

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

**[Обязателен ли "else"?](https://learn.javascript.ru/function-basics" \l "обязателен-ли-else)**

важность: 4

Следующая функция возвращает true, если параметр age больше 18. В ином случае она задаёт вопрос посредством вызова confirm и возвращает его результат.

function checkAge(age) {

if (age > 18) {

return true;

} else {

// ...

return confirm('Родители разрешили?');

}

}

Будет ли эта функция работать как-то иначе, если убрать else?

function checkAge(age) {

if (age > 18) {

return true;

}

// ...

return confirm('Родители разрешили?');

}

Есть ли хоть одно отличие в поведении этого варианта?

решение

**[Перепишите функцию, используя оператор '?' или '||'](https://learn.javascript.ru/function-basics" \l "перепишите-функцию-используя-оператор-или)**

важность: 4

Следующая функция возвращает true, если параметр age больше 18. В ином случае она задаёт вопрос confirm и возвращает его результат.

function checkAge(age) {

if (age > 18) {

return true;

} else {

return confirm('Родители разрешили?');

}

}

Перепишите функцию, чтобы она делала то же самое, но без if, в одну строку. Сделайте два варианта функции checkAge:

1. Используя оператор '?'
2. Используя оператор ||

решение

**[Функция min](https://learn.javascript.ru/function-basics" \l "функция-min)**

важность: 1

Задача «Hello World» для функций :)

Напишите функцию min(a,b), которая возвращает меньшее из чисел a,b.

Пример вызовов:

min(2, 5) == 2

min(3, -1) == -1

min(1, 1) == 1

решение

**[Функция pow(x,n)](https://learn.javascript.ru/function-basics" \l "функция-pow-x-n)**

важность: 4

Напишите функцию pow(x,n), которая возвращает x в степени n. Иначе говоря, умножает x на себя nраз и возвращает результат.

pow(3, 2) = 3 \* 3 = 9

pow(3, 3) = 3 \* 3 \* 3 = 27

pow(1, 100) = 1 \* 1 \* ...\*1 = 1

Создайте страницу, которая запрашивает x и n, а затем выводит результат pow(x,n).

[Запустить демо](https://learn.javascript.ru/function-basics)

P.S. В этой задаче функция обязана поддерживать только натуральные значения n, т.е. целые от 1 и выше.

решение

**Функциональные выражения**

В JavaScript функция является значением, таким же как строка или число.

Как и любое значение, объявленную функцию можно вывести, вот так:

function sayHi() {

alert( "Привет" );

}

alert( sayHi ); // выведет код функции

Обратим внимание на то, что в последней строке после sayHi нет скобок. То есть, функция не вызывается, а просто выводится на экран.

**Функцию можно скопировать в другую переменную:**

function sayHi() { // (1)

alert( "Привет" );

}

var func = sayHi; // (2)

func(); // Привет // (3)

sayHi = null;

sayHi(); // ошибка (4)

1. Объявление (1) как бы говорит интерпретатору "создай функцию и помести её в переменную sayHi
2. В строке (2) мы копируем функцию в новую переменную func. Ещё раз обратите внимание: после sayHi нет скобок. Если бы они были, то вызов var func = sayHi() записал бы в func *результат* работы sayHi() (кстати, чему он равен? правильно, undefined, ведь внутри sayHi нет return).
3. На момент (3) функцию можно вызывать и как sayHi() и как func()
4. …Однако, в любой момент значение переменной можно поменять. При этом, если оно не функция, то вызов (4) выдаст ошибку.

Обычные значения, такие как числа или строки, представляют собой *данные*. А функцию можно воспринимать как *действие*.

Это действие можно запустить через скобки (), а можно и скопировать в другую переменную, как было продемонстрировано выше.

**[Объявление Function Expression](https://learn.javascript.ru/function-declaration-expression" \l "function-expression)**

Существует альтернативный синтаксис для объявления функции, который ещё более наглядно показывает, что функция – это всего лишь разновидность значения переменной.

Он называется «Function Expression» (функциональное выражение) и выглядит так:

var f = function(параметры) {

// тело функции

};

Например:

var sayHi = function(person) {

alert( "Привет, " + person );

};

sayHi('Вася');

**[Сравнение с Function Declaration](https://learn.javascript.ru/function-declaration-expression" \l "сравнение-с-function-declaration)**

«Классическое» объявление функции, о котором мы говорили до этого, вида function имя(параметры) {...}, называется в спецификации языка «Function Declaration».

* *Function Declaration* – функция, объявленная в основном потоке кода.
* *Function Expression* – объявление функции в контексте какого-либо выражения, например присваивания.

Несмотря на немного разный вид, по сути две эти записи делают одно и то же:

// Function Declaration

function sum(a, b) {

return a + b;

}

// Function Expression

var sum = function(a, b) {

return a + b;

}

Оба этих объявления говорят интерпретатору: "объяви переменную sum, создай функцию с указанными параметрами и кодом и сохрани её в sum".

**Основное отличие между ними: функции, объявленные как Function Declaration, создаются интерпретатором до выполнения кода.**

Поэтому их можно вызвать *до* объявления, например:

sayHi("Вася"); // Привет, Вася

function sayHi(name) {

alert( "Привет, " + name );

}

А если бы это было объявление Function Expression, то такой вызов бы не сработал:

sayHi("Вася"); // ошибка!

var sayHi = function(name) {

alert( "Привет, " + name );

}

Это из-за того, что JavaScript перед запуском кода ищет в нём Function Declaration (их легко найти: они не являются частью выражений и начинаются со слова function) и обрабатывает их.

А Function Expression создаются в процессе выполнении выражения, в котором созданы, в данном случае – функция будет создана при операции присваивания sayHi = function...

Как правило, возможность Function Declaration вызвать функцию до объявления – это удобно, так как даёт больше свободы в том, как организовать свой код.

Можно расположить функции внизу, а их вызов – сверху или наоборот.

**[Условное объявление функции](https://learn.javascript.ru/function-declaration-expression" \l "bad-conditional-declaration)**

В некоторых случаях «дополнительное удобство» Function Declaration может сослужить плохую службу.

Например, попробуем, в зависимости от условия, объявить функцию sayHi по-разному:

var age = +prompt("Сколько вам лет?", 20);

if (age >= 18) {

function sayHi() {

alert( 'Прошу вас!' );

}

} else {

function sayHi() {

alert( 'До 18 нельзя' );

}

}

sayHi();

Function Declaration при use strict видны только внутри блока, в котором объявлены. Так как код в учебнике выполняется в режиме use strict, то будет ошибка.

А что, если использовать Function Expression?

var age = prompt('Сколько вам лет?');

var sayHi;

if (age >= 18) {

sayHi = function() {

alert( 'Прошу Вас!' );

}

} else {

sayHi = function() {

alert( 'До 18 нельзя' );

}

}

sayHi();

Или даже так:

var age = prompt('Сколько вам лет?');

var sayHi = (age >= 18) ?

function() { alert('Прошу Вас!'); } :

function() { alert('До 18 нельзя'); };

sayHi();

Оба этих варианта работают правильно, поскольку, в зависимости от условия, создаётся именно та функция, которая нужна.

**[Анонимные функции](https://learn.javascript.ru/function-declaration-expression" \l "анонимные-функции)**

Взглянем ещё на один пример – функцию ask(question, yes, no) с тремя параметрами:

**question**

Строка-вопрос

**yes**

Функция

**no**

Функция

Она выводит вопрос на подтверждение question и, в зависимости от согласия пользователя, вызывает функцию yes() или no():

function ask(question, yes, no) {

if (confirm(question)) yes()

else no();

}

function showOk() {

alert( "Вы согласились." );

}

function showCancel() {

alert( "Вы отменили выполнение." );

}

// использование

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

Какой-то очень простой код, не правда ли? Зачем, вообще, может понадобиться такая ask?

…Оказывается, при работе со страницей такие функции как раз очень востребованы, только вот спрашивают они не простым confirm, а выводят более красивое окно с вопросом и могут интеллектуально обработать ввод посетителя. Но это всё потом, когда перейдём к работе с интерфейсом.

Здесь же обратим внимание на то, что то же самое можно написать более коротко:

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 объявлены прямо внутри вызова ask(...), даже без присвоения им имени.

**Функциональное выражение, которое не записывается в переменную, называют**[**анонимной функцией**](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%BE%D0%BD%D0%B8%D0%BC%D0%BD%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F)**.**

Действительно, зачем нам записывать функцию в переменную, если мы не собираемся вызывать её ещё раз? Можно просто объявить непосредственно там, где функция нужна.

Такого рода код возникает естественно, он соответствует «духу» JavaScript.

**[new Function](https://learn.javascript.ru/function-declaration-expression" \l "new-function)**

Существует ещё один способ создания функции, который используется очень редко, но упомянем и его для полноты картины.

Он позволяет создавать функцию полностью «на лету» из строки, вот так:

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

var result = sum(1, 2);

alert( result ); // 3

То есть, функция создаётся вызовом new Function(params, code):

**params**

Параметры функции через запятую в виде строки.

**code**

Код функции в виде строки.

Таким образом можно конструировать функцию, код которой неизвестен на момент написания программы, но строка с ним генерируется или подгружается динамически во время её выполнения.

Пример использования – динамическая компиляция шаблонов на JavaScript, мы встретимся с ней позже, при работе с интерфейсами.

[**Итого**](https://learn.javascript.ru/function-declaration-expression#итого)

Функции в JavaScript являются значениями. Их можно присваивать, передавать, создавать в любом месте кода.

* Если функция объявлена в *основном потоке кода*, то это Function Declaration.
* Если функция создана как *часть выражения*, то это Function Expression.

Между этими двумя основными способами создания функций есть следующие различия:

|  |  |  |
| --- | --- | --- |
|  | **Function Declaration** | **Function Expression** |
| Время создания | До выполнения первой строчки кода. | Когда управление достигает строки с функцией. |
| Можно вызвать до объявления | Да (т.к. создаётся заранее) | Нет |
| Условное объявление в if | Не работает | Работает |

Иногда в коде начинающих разработчиков можно увидеть много Function Expression. Почему-то, видимо, не очень понимая происходящее, функции решают создавать как var func = function(), но в большинстве случаев обычное объявление функции – лучше.

**Если нет явной причины использовать Function Expression – предпочитайте Function Declaration.**

Сравните по читаемости:

// Function Expression

var f = function() { ... }

// Function Declaration

function f() { ... }

Function Declaration короче и лучше читается. Дополнительный бонус – такие функции можно вызывать до того, как они объявлены.

Используйте Function Expression только там, где это действительно нужно и удобно.

**Рекурсия, стек**

В теле функции могут быть вызываны другие функции для выполнения подзадач.

Частный случай подвызова – когда функция вызывает сама себя. Это называется *рекурсией*.

Рекурсия используется для ситуаций, когда выполнение одной сложной задачи можно представить как некое действие в совокупности с решением той же задачи в более простом варианте.

Сейчас мы посмотрим примеры.

Рекурсия – общая тема программирования, не относящаяся напрямую к JavaScript. Если вы разрабатывали на других языках или изучали программирование раньше в ВУЗе, то наверняка уже знаете, что это такое.

Эта глава предназначена для читателей, которые пока с этой темой незнакомы и хотят лучше разобраться в том, как работают функции.

**[Степень pow(x, n) через рекурсию](https://learn.javascript.ru/recursion" \l "степень-pow-x-n-через-рекурсию)**

В качестве первого примера использования рекурсивных вызовов – рассмотрим задачу возведения числа x в натуральную степень n.

Её можно представить как совокупность более простого действия и более простой задачи того же типа вот так:

pow(x, n) = x \* pow(x, n - 1)

То есть, xn = x \* xn-1.

Например, вычислим pow(2, 4), последовательно переходя к более простой задаче:

1. pow(2, 4) = 2 \* pow(2, 3)
2. pow(2, 3) = 2 \* pow(2, 2)
3. pow(2, 2) = 2 \* pow(2, 1)
4. pow(2, 1) = 2

На шаге 1 нам нужно вычислить pow(2,3), поэтому мы делаем шаг 2, дальше нам нужно pow(2,2), мы делаем шаг 3, затем шаг 4, и на нём уже можно остановиться, ведь очевидно, что результат возведения числа в степень 1 – равен самому числу.

Далее, имея результат на шаге 4, он подставляется обратно в шаг 3, затем имеем pow(2,2) – подставляем в шаг 2 и на шаге 1 уже получаем результат.

Этот алгоритм на JavaScript:

function pow(x, n) {

if (n != 1) { // пока n != 1, сводить вычисление pow(x,n) к pow(x,n-1)

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

} else {

return x;

}

}

alert( pow(2, 3) ); // 8

Говорят, что «функция pow *рекурсивно вызывает сама себя*» до n == 1.

Значение, на котором рекурсия заканчивается называют *базисом рекурсии*. В примере выше базисом является 1.

Общее количество вложенных вызовов называют *глубиной рекурсии*. В случае со степенью, всего будет nвызовов.

Максимальная глубина рекурсии в браузерах ограничена, точно можно рассчитывать на 10000вложенных вызовов, но некоторые интерпретаторы допускают и больше.

Итак, рекурсию используют, когда вычисление функции можно свести к её более простому вызову, а его – ещё к более простому, и так далее, пока значение не станет очевидно.

**[Контекст выполнения, стек](https://learn.javascript.ru/recursion" \l "контекст-выполнения-стек)**

Теперь мы посмотрим, как работают рекурсивные вызовы. Для этого мы рассмотрим, как вообще работают функции, что происходит при вызове.

**У каждого вызова функции есть свой «контекст выполнения» (execution context).**

Контекст выполнения – это служебная информация, которая соответствует текущему запуску функции. Она включает в себя локальные переменные функции и конкретное место в коде, на котором находится интерпретатор.

Например, для вызова pow(2, 3) из примера выше будет создан контекст выполнения, который будет хранить переменные x = 2, n = 3. Мы схематично обозначим его так:

* **Контекст: { x: 2, n: 3, строка 1 }**

Далее функция pow начинает выполняться. Вычисляется выражение n != 1 – оно равно true, ведь в текущем контексте n=3. Поэтому задействуется первая ветвь if :

function pow(x, n) {

if (n != 1) { // пока n != 1 сводить вычисление pow(x,n) к pow(x,n-1)

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

} else {

return x;

}

}

Чтобы вычислить выражение x \* pow(x, n-1), требуется произвести запуск pow с новыми аргументами.

**При любом вложенном вызове JavaScript запоминает текущий контекст выполнения в специальной внутренней структуре данных – «стеке контекстов».**

Затем интерпретатор приступает к выполнению вложенного вызова.

В данном случае вызывается та же pow, однако это абсолютно неважно. Для любых функций процесс одинаков.

Для нового вызова создаётся свой контекст выполнения, и управление переходит в него, а когда он завершён – старый контекст достаётся из стека и выполнение внешней функции возобновляется.

Разберём происходящее с контекстами более подробно, начиная с вызова (\*):

function pow(x, n) {

if (n != 1) { // пока n!=1 сводить вычисление pow(..n) к pow(..n-1)

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

} else {

return x;

}

}

alert( pow(2, 3) ); // (\*)

**pow(2, 3)**

Запускается функция pow, с аргументами x=2, n=3. Эти переменные хранятся в контексте выполнения, схематично изображённом ниже:

* **Контекст: { x: 2, n: 3, строка 1 }**

Выполнение в этом контексте продолжается, пока не встретит вложенный вызов в строке 3.

**pow(2, 2)**

В строке 3 происходит вложенный вызов pow с аргументами x=2, n=2. Текущий контекст сохраняется в стеке, а для вложеннного вызова создаётся новый контекст (выделен жирным ниже):

* Контекст: { x: 2, n: 3, строка 3 }
* **Контекст: { x: 2, n: 2, строка 1 }**

Обратим внимание, что контекст включает в себя не только переменные, но и место в коде, так что когда вложенный вызов завершится -- можно будет легко вернуться назад.

Слово «строка» здесь условно, на самом деле, конечно, запомнено более точное место в цепочке команд.

**pow(2, 1)**

Опять вложенный вызов в строке 3, на этот раз – с аргументами x=2, n=1. Создаётся новый текущий контекст, предыдущий добавляется в стек:

* Контекст: { x: 2, n: 3, строка 3 }
* Контекст: { x: 2, n: 2, строка 3 }
* **Контекст: { x: 2, n: 1, строка 1 }**

На текущий момент в стеке уже два старых контекста.

**Выход из pow(2, 1).**

При выполнении pow(2, 1), в отличие от предыдущих запусков, выражение n != 1 будет равно false, поэтому сработает вторая ветка if..else:

function pow(x, n) {

if (n != 1) {

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

} else {

return x; // первая степень числа равна самому числу

}

}

Здесь вложенных вызовов нет, так что функция заканчивает свою работу, возвращая 2. Текущий контекст больше не нужен и удаляется из памяти, из стека восстанавливается предыдущий:

* Контекст: { x: 2, n: 3, строка 3 }
* **Контекст: { x: 2, n: 2, строка 3 }**

Возобновляется обработка внешнего вызова `pow(2, 2)`.

**Выход из pow(2, 2).**

…И теперь уже pow(2, 2) может закончить свою работу, вернув 4. Восстанавливается контекст предыдущего вызова:

* **Контекст: { x: 2, n: 3, строка 3 }**

Возобновляется обработка внешнего вызова `pow(2, 3)`.

**Выход из pow(2, 3).**

Самый внешний вызов заканчивает свою работу, его результат: pow(2, 3) = 8.

Глубина рекурсии в данном случае составила: **3**.

Как видно из иллюстраций выше, глубина рекурсии равна максимальному числу контекстов, одновременно хранимых в стеке.

Обратим внимание на требования к памяти. Рекурсия приводит к хранению всех данных для неоконченных внешних вызовов в стеке, в данном случае это приводит к тому, что возведение в степень n хранит в памяти n различных контекстов.

Реализация возведения в степень через цикл гораздо более экономна:

function pow(x, n) {

var result = x;

for (var i = 1; i < n; i++) {

result \*= x;

}

return result;

}

У такой функции pow будет один контекст, в котором будут последовательно меняться значения i и result.

**Любая рекурсия может быть переделана в цикл. Как правило, вариант с циклом будет эффективнее.**

Но переделка рекурсии в цикл может быть нетривиальной, особенно когда в функции, в зависимости от условий, используются различные рекурсивные подвызовы, когда ветвление более сложное.

[**Итого**](https://learn.javascript.ru/recursion#итого)

Рекурсия – это когда функция вызывает сама себя, как правило, с другими аргументами.

Существуют много областей применения рекурсивных вызовов. Здесь мы посмотрели на один из них – решение задачи путём сведения её к более простой (с меньшими аргументами), но также рекурсия используется для работы с «естественно рекурсивными» структурами данных, такими как HTML-документы, для «глубокого» копирования сложных объектов.

Есть и другие применения, с которыми мы встретимся по мере изучения JavaScript.

Здесь мы постарались рассмотреть происходящее достаточно подробно, однако, если пожелаете, допустимо временно забежать вперёд и открыть главу [Отладка в браузере Chrome](https://learn.javascript.ru/debugging-chrome), с тем чтобы при помощи отладчика построчно пробежаться по коду и посмотреть стек на каждом шаге. Отладчик даёт к нему доступ.

[**Задачи**](https://learn.javascript.ru/recursion#tasks)

**[Вычислить сумму чисел до данного](https://learn.javascript.ru/recursion" \l "вычислить-сумму-чисел-до-данного)**

важность: 5

Напишите функцию sumTo(n), которая для данного n вычисляет сумму чисел от 1 до n, например:

sumTo(1) = 1

sumTo(2) = 2 + 1 = 3

sumTo(3) = 3 + 2 + 1 = 6

sumTo(4) = 4 + 3 + 2 + 1 = 10

...

sumTo(100) = 100 + 99 + ... + 2 + 1 = 5050

Сделайте три варианта решения:

1. С использованием цикла.
2. Через рекурсию, т.к. sumTo(n) = n + sumTo(n-1) для n > 1.
3. С использованием формулы для суммы [арифметической прогрессии](http://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%B8%D1%84%D0%BC%D0%B5%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B5%D1%81%D1%81%D0%B8%D1%8F).

Пример работы вашей функции:

function sumTo(n) { /\*... ваш код ... \*/ }

alert( sumTo(100) ); // 5050

**Какой вариант решения самый быстрый? Самый медленный? Почему?**

**Можно ли при помощи рекурсии посчитать sumTo(100000)? Если нет, то почему?**

решение

**[Вычислить факториал](https://learn.javascript.ru/recursion" \l "вычислить-факториал)**

важность: 4

*Факториа́л числа* – это число, умноженное на «себя минус один», затем на «себя минус два» и так далее, до единицы. Обозначается n!

Определение факториала можно записать как:

n! = n \* (n - 1) \* (n - 2) \* ...\*1

Примеры значений для разных n:

1! = 1

2! = 2 \* 1 = 2

3! = 3 \* 2 \* 1 = 6

4! = 4 \* 3 \* 2 \* 1 = 24

5! = 5 \* 4 \* 3 \* 2 \* 1 = 120

Задача – написать функцию factorial(n), которая возвращает факториал числа n!, используя рекурсивный вызов.

alert( factorial(5) ); // 120

Подсказка: обратите внимание, что n! можно записать как n \* (n-1)!. Например: 3! = 3\*2! = 3\*2\*1! = 6

решение

**[Числа Фибоначчи](https://learn.javascript.ru/recursion" \l "числа-фибоначчи)**

важность: 5

Последовательность [чисел Фибоначчи](http://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%B0_%D0%A4%D0%B8%D0%B1%D0%BE%D0%BD%D0%B0%D1%87%D1%87%D0%B8) имеет формулу Fn = Fn-1 + Fn-2. То есть, следующее число получается как сумма двух предыдущих.

Первые два числа равны 1, затем 2(1+1), затем 3(1+2), 5(2+3) и так далее: 1, 1, 2, 3, 5, 8, 13, 21....

Числа Фибоначчи тесно связаны с [золотым сечением](http://ru.wikipedia.org/wiki/%D0%97%D0%BE%D0%BB%D0%BE%D1%82%D0%BE%D0%B5_%D1%81%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) и множеством природных явлений вокруг нас.

Напишите функцию fib(n), которая возвращает n-е число Фибоначчи. Пример работы:

function fib(n) { /\* ваш код \*/ }

alert( fib(3) ); // 2

alert( fib(7) ); // 13

alert( fib(77)); // 5527939700884757

**Все запуски функций из примера выше должны срабатывать быстро.**

решение

**Именованные функциональные выражения**

Специально для работы с рекурсией в JavaScript существует особое расширение функциональных выражений, которое называется «Named Function Expression» (сокращённо NFE) или, по-русски, *«именованное функциональное выражение»*.

**[Named Function Expression](https://learn.javascript.ru/named-function-expression" \l "functions-nfe)**

Обычное функциональное выражение:

var f = function(...) { /\* тело функции \*/ };

Именованное с именем sayHi:

var f = function sayHi(...) { /\* тело функции \*/ };

Что же это за имя, которое идёт в дополнение к f, и зачем оно?

Имя функционального выражения (sayHi) имеет особый смысл. Оно доступно только изнутри самой функции.

Это ограничение видимости входит в стандарт JavaScript и поддерживается всеми браузерами, кроме IE8-.

Например:

var f = function sayHi(name) {

alert( sayHi ); // изнутри функции - видно (выведет код функции)

};

alert( sayHi ); // снаружи - не видно (ошибка: undefined variable 'sayHi')

Кроме того, имя NFE нельзя перезаписать:

var test = function sayHi(name) {

sayHi = "тест"; // попытка перезаписи

alert( sayHi ); // function... (перезапись не удалась)

};

test();

В режиме use strict код выше выдал бы ошибку.

Как правило, имя NFE используется для единственной цели – позволить изнутри функции вызвать саму себя.

**[Пример использования](https://learn.javascript.ru/named-function-expression" \l "пример-использования)**

NFE используется в первую очередь в тех ситуациях, когда функцию нужно передавать в другое место кода или перемещать из одной переменной в другую.

**Внутреннее имя позволяет функции надёжно обращаться к самой себе, где бы она ни находилась.**

Вспомним, к примеру, функцию-факториал из задачи [Вычислить факториал](https://learn.javascript.ru/task/factorial):

function f(n) {

return n ? n \* f(n - 1) : 1;

};

alert( f(5) ); // 120

Попробуем перенести её в другую переменную g:

function f(n) {

return n ? n \* f(n - 1) : 1;

};

var g = f;

f = null;

alert( g(5) ); // запуск функции с новым именем - ошибка при выполнении!

Ошибка возникла потому что функция из своего кода обращается к своему старому имени f. А этой функции уже нет, f = null.

Для того, чтобы функция всегда надёжно работала, объявим её как Named Function Expression:

var f = function factorial(n) {

return n ? n\*factorial(n-1) : 1;

};

var g = f; // скопировали ссылку на функцию-факториал в g

f = null;

alert( g(5) ); // 120, работает!

**В браузере IE8- создаются две функции**

Как мы говорили выше, в браузере IE до 9 версии имя NFE видно везде, что является ошибкой с точки зрения стандарта.

…Но на самом деле ситуация ещё забавнее. Старый IE создаёт в таких случаях целых две функции: одна записывается в переменную f, а вторая – в переменную factorial.

Например:

var f = function factorial(n) { /\*...\*/ };

// в IE8- false

// в остальных браузерах ошибка, т.к. имя factorial не видно

alert( f === factorial );

Все остальные браузеры полностью поддерживают именованные функциональные выражения.

**Устаревшее специальное значение arguments.callee**

Если вы давно работаете с JavaScript, то, возможно, знаете, что раньше для этой цели также служило специальное значение arguments.callee.

Если это название вам ни о чём не говорит – всё в порядке, читайте дальше, мы обязательно обсудим его [в отдельной главе](https://learn.javascript.ru/arguments-pseudoarray#arguments-callee).

Если же вы в курсе, то стоит иметь в виду, что оно официально исключено из современного стандарта. А NFE – это наше настоящее.

[**Итого**](https://learn.javascript.ru/named-function-expression#итого)

Если функция задана как Function Expression, ей можно дать имя.

Оно будет доступно только внутри функции (кроме IE8-).

Это имя предназначено для надёжного рекурсивного вызова функции, даже если она записана в другую переменную.

Обратим внимание, что с Function Declaration так поступить нельзя. Такое «специальное» внутреннее имя функции задаётся только в синтаксисе Function Expression.

[**Задачи**](https://learn.javascript.ru/named-function-expression#tasks)

**[Проверка на NFE](https://learn.javascript.ru/named-function-expression" \l "проверка-на-nfe)**

важность: 5

Каков будет результат выполнения кода?

function g() { return 1; }

alert(g);

А такого? Будет ли разница, если да – почему?

(function g() { return 1; });

alert(g);

решение

**Всё вместе: особенности JavaScript**

В этой главе приводятся основные особенности JavaScript, на уровне базовых конструкций, типов, синтаксиса.

Она будет особенно полезна, если ранее вы программировали на другом языке, ну или как повторение важных моментов раздела.

Всё очень компактно, со ссылками на развёрнутые описания.

**[Структура кода](https://learn.javascript.ru/javascript-specials" \l "структура-кода)**

Операторы разделяются точкой с запятой:

alert('Привет'); alert('Мир');

Как правило, перевод строки тоже подразумевает точку с запятой. Так тоже будет работать:

alert('Привет')

alert('Мир')

…Однако, иногда JavaScript не вставляет точку с запятой. Например:

var a = 2

+3

alert(a); // 5

Бывают случаи, когда это ведёт к ошибкам, которые достаточно трудно найти и исправить, например:

alert("После этого сообщения будет ошибка")

[1, 2].forEach(alert)

Детали того, как работает код выше (массивы [...] и forEach) мы скоро изучим, здесь важно то, что при установке точки с запятой после alert он будет работать корректно.

**Поэтому в JavaScript рекомендуется точки с запятой ставить. Сейчас это, фактически, общепринятый стандарт.**

Поддерживаются однострочные комментарии // ... и многострочные /\* ... \*/:

Подробнее: [Структура кода](https://learn.javascript.ru/structure).

**[Переменные и типы](https://learn.javascript.ru/javascript-specials" \l "переменные-и-типы)**

* Объявляются директивой var. Могут хранить любое значение:
* var x = 5;

x = "Петя";

* Есть 5 «примитивных» типов и объекты:
* x = 1; // число
* x = "Тест"; // строка, кавычки могут быть одинарные или двойные
* x = true; // булево значение true/false
* x = null; // спец. значение (само себе тип)

x = undefined; // спец. значение (само себе тип)

Также есть специальные числовые значения Infinity (бесконечность) и NaN.

Значение NaN обозначает ошибку и является результатом числовой операции, если она некорректна.

* **Значение null не является «ссылкой на нулевой адрес/объект» или чем-то подобным. Это просто специальное значение.**

Оно присваивается, если мы хотим указать, что значение переменной неизвестно.

Например:

var age = null; // возраст неизвестен

* **Значение undefined означает «переменная не присвоена».**

Например:

var x;

alert( x ); // undefined

Можно присвоить его и явным образом: x = undefined, но так делать не рекомендуется.

Про объекты мы поговорим в главе [Объекты как ассоциативные массивы](https://learn.javascript.ru/object), они в JavaScript сильно отличаются от большинства других языков.

* В имени переменной могут быть использованы любые буквы или цифры, но цифра не может быть первой. Символы доллар $ и подчёркивание \_ допускаются наравне с буквами.

Подробнее: [Переменные](https://learn.javascript.ru/variables), [Шесть типов данных, typeof](https://learn.javascript.ru/types-intro).

**[Строгий режим](https://learn.javascript.ru/javascript-specials" \l "строгий-режим)**

Для того, чтобы интерпретатор работал в режиме максимального соответствия современному стандарту, нужно начинать скрипт директивой 'use strict';

'use strict';

...

Эта директива может также указываться в начале функций. При этом функция будет выполняться в режиме соответствия, а на внешний код такая директива не повлияет.

Одно из важных изменений в современном стандарте – все переменные нужно объявлять через var. Есть и другие, которые мы изучим позже, вместе с соответствующими возможностями языка.

**[Взаимодействие с посетителем](https://learn.javascript.ru/javascript-specials" \l "взаимодействие-с-посетителем)**

Простейшие функции для взаимодействия с посетителем в браузере:

[**«prompt(вопрос[, по\_умолчанию])»**](https://developer.mozilla.org/en/DOM/window.prompt)

Задать вопрос и возвратить введённую строку, либо null, если посетитель нажал «Отмена».

[**«confirm(вопрос)»**](https://developer.mozilla.org/en/DOM/window.confirm)

Задать вопрос и предложить кнопки «Ок», «Отмена». Возвращает, соответственно, true/false.

[**«alert(сообщение)»**](https://developer.mozilla.org/en/DOM/window.alert)

Вывести сообщение на экран.

Все эти функции являются *модальными*, т.е. не позволяют посетителю взаимодействовать со страницей до ответа.

Например:

var userName = prompt("Введите имя?", "Василий");

var isTeaWanted = confirm("Вы хотите чаю?");

alert( "Посетитель: " + userName );

alert( "Чай: " + isTeaWanted );

Подробнее: [Взаимодействие с пользователем: alert, prompt, confirm](https://learn.javascript.ru/uibasic).

**[Особенности операторов](https://learn.javascript.ru/javascript-specials" \l "особенности-операторов)**

* **Для сложения строк используется оператор +.**

Если хоть один аргумент – строка, то другой тоже приводится к строке:

alert( 1 + 2 ); // 3, число

alert( '1' + 2 ); // '12', строка

alert( 1 + '2' ); // '12', строка

* **Сравнение === проверяет точное равенство, включая одинаковый тип.** Это самый очевидный и надёжный способ сравнения.
* **Остальные сравнения == < <= > >= осуществляют числовое приведение типа:**
* alert( 0 == false ); // true

alert( true > 0 ); // true

Исключение – сравнение двух строк, которое осуществляется лексикографически (см. далее).

Также: значения null и undefined при == равны друг другу и не равны ничему ещё. А при операторах больше/меньше происходит приведение null к 0, а undefined к NaN.

Такое поведение может привести к неочевидным результатам, поэтому лучше всего использовать для сравнения с null/undefined оператор ===. Оператор == тоже можно, если не хотите отличать nullот undefined.

Например, забавное следствие этих правил для null:

alert( null > 0 ); // false, т.к. null преобразовано к 0

alert( null >= 0 ); // true, т.к. null преобразовано к 0

alert( null == 0 ); // false, в стандарте явно указано, что null равен лишь undefined

С точки зрения здравого смысла такое невозможно. Значение null не равно нулю и не больше, но при этом null >= 0 возвращает true!

* **Сравнение строк – лексикографическое, символы сравниваются по своим unicode-кодам.**

Поэтому получается, что строчные буквы всегда больше, чем прописные:

alert( 'а' > 'Я' ); // true

Подробнее: [Основные операторы](https://learn.javascript.ru/operators), [Операторы сравнения и логические значения](https://learn.javascript.ru/comparison).

**[Логические операторы](https://learn.javascript.ru/javascript-specials" \l "логические-операторы)**

В JavaScript есть логические операторы: И (обозначается &&), ИЛИ (обозначается ||) и НЕ (обозначается !). Они интерпретируют любое значение как логическое.

Не стоит путать их с [побитовыми операторами](https://learn.javascript.ru/bitwise-operators) И, ИЛИ, НЕ, которые тоже есть в JavaScript и работают с числами на уровне битов.

Как и в большинстве других языков, в логических операторах используется «короткий цикл» вычислений. Например, вычисление выражения 1 && 0 && 2 остановится после первого И &&, т.к. понятно что результат будет ложным (ноль интерпретируется как false).

**Результатом логического оператора служит последнее значение в коротком цикле вычислений.**

Можно сказать и по-другому: значения хоть и интерпретируются как логические, но то, которое в итоге определяет результат, возвращается без преобразования.

Например:

alert( 0 && 1 ); // 0

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

alert( null || 1 || 2 ); // 1

Подробнее: [Логические операторы](https://learn.javascript.ru/logical-ops).

**[Циклы](https://learn.javascript.ru/javascript-specials" \l "циклы)**

* Поддерживаются три вида циклов:
* // 1
* while (условие) {
* ...
* }
* // 2
* do {
* ...
* } while (условие);
* // 3
* for (var i = 0; i < 10; i++) {
* ...

}

* Переменную можно объявлять прямо в цикле, но видна она будет и за его пределами.
* Поддерживаются директивы break/continue для выхода из цикла/перехода на следующую итерацию.

Для выхода одновременно из нескольких уровней цикла можно задать метку.

Синтаксис: «имя\_метки:», ставится она только перед циклами и блоками, например:

outer:

for(;;) {

...

for(;;) {

...

break outer;

}

}

Переход на метку возможен только изнутри цикла, и только на внешний блок по отношению к данному циклу. В произвольное место программы перейти нельзя.

Подробнее: [Циклы while, for](https://learn.javascript.ru/while-for).

**[Конструкция switch](https://learn.javascript.ru/javascript-specials" \l "конструкция-switch)**

При сравнениях в конструкции switch используется оператор ===.

Например:

var age = prompt('Ваш возраст', 18);

switch (age) {

case 18:

alert( 'Никогда не сработает' ); // результат prompt - строка, а не число

case "18": // вот так - сработает!

alert( 'Вам 18 лет!' );

break;

default:

alert( 'Любое значение, не совпавшее с case' );

}

Подробнее: [Конструкция switch](https://learn.javascript.ru/switch).

**[Функции](https://learn.javascript.ru/javascript-specials" \l "функции)**

Синтаксис функций в JavaScript:

// function имя(список параметров) { тело }

function sum(a, b) {

var result = a + b;

return result;

}

// использование:

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

* sum – имя функции, ограничения на имя функции – те же, что и на имя переменной.
* Переменные, объявленные через var внутри функции, видны везде внутри этой функции, блоки if, for и т.п. на видимость не влияют.
* Параметры копируются в локальные переменные a, b.
* Функция без return считается возвращающей undefined. Вызов return без значения также возвращает undefined:
* function f() { }

alert( f() ); // undefined

Подробнее: [Функции](https://learn.javascript.ru/function-basics).

**[Function Declaration и Expression](https://learn.javascript.ru/javascript-specials" \l "function-declaration-и-expression)**

Функция в JavaScript является обычным значением.

Её можно создать в любом месте кода и присвоить в переменную, вот так:

var sum = function(a, b) {

var result = a + b;

return result;

}

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

Такой синтаксис, при котором функция объявляется в контексте выражения (в данном случае, выражения присваивания), называется Function Expression, а обычный синтаксис, при котором функция объявляется в основном потоке кода – Function Declaration.

Функции, объявленные через Function Declaration, отличаются от Function Expression тем, что интерпретатор создаёт их при входе в область видимости (в начале выполнения скрипта), так что они работают до объявления.

Обычно это удобно, но может быть проблемой, если нужно объявить функцию в зависимости от условия. В этом случае, а также в других ситуациях, когда хочется создать функцию «здесь и сейчас», используют Function Expression.

Детали: [Функциональные выражения](https://learn.javascript.ru/function-declaration-expression).

**[Named Function Expression](https://learn.javascript.ru/javascript-specials" \l "named-function-expression)**

Если объявление функции является частью какого-либо выражения, например var f = function... или любого другого, то это Function Expression.

В этом случае функции можно присвоить «внутреннее» имя, указав его после function. Оно будет видно только внутри этой функции и позволяет обратиться к функции изнутри себя. Обычно это используется для рекурсивных вызовов.

Например, создадим функцию для вычисления факториала как Function Expression и дадим ей имя me:

var factorial = function me(n) {

return (n == 1) ? n : n \* me(n - 1);

}

alert( factorial(5) ); // 120

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

Ограничение видимости для имени не работает в IE8-, но вызов с его помощью работает во всех браузерах.

Более развёрнуто: [Именованные функциональные выражения](https://learn.javascript.ru/named-function-expression).

[**Итого**](https://learn.javascript.ru/javascript-specials#итого)

В этой главе мы повторили основные особенности JavaScript, знание которых необходимо для обхода большинства «граблей», да и просто для написания хорошего кода.

Это, конечно, лишь основы. Дальше вы узнаете много других особенностей и приёмов программирования на этом языке.