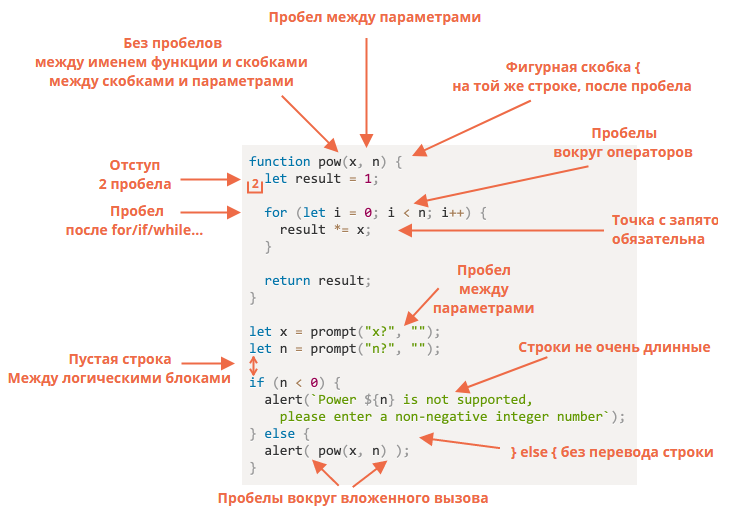
Здесь нет железных правил. Это стилевые предпочтения, а не религиозные догмы.

### Фигурные скобки

В большинстве JavaScript проектов фигурные скобки пишутся в так называемом «египетском» стиле с открывающей скобкой на той же строке, что и соответствующее ключевое слово – не на новой строке. Перед открывающей скобкой должен быть пробел, как здесь:

if (condition) {

// делай это

// ...и это

// ...и потом это

}

А что если у нас однострочная запись, типа if (condition) doSomething(), должны ли мы использовать фигурные скобки?

Вот различные варианты расстановки скобок с комментариями, посмотрите сами, какой вам кажется самым читаемым:

1. Такое иногда бывает в коде начинающих. Плохо, фигурные скобки не нужны:

if (n < 0) {alert(`Степень ${n} не поддерживается`);}

Никогда не разделяйте строки без фигурных скобок, можно ненароком сделать ошибку при добавлении строк:

if (n < 0)

alert(`Степень ${n} не поддерживается`);

В одну строку без скобок – приемлемо, если эта строка короткая:

if (n < 0) alert(`Степень ${n} не поддерживается`);

Самый лучший вариант:

if (n < 0) {

alert(`Степень ${n} не поддерживается`);

}

Для очень короткого кода допустима одна строка. Например: if (cond) return null. Но блок кода (последний вариант) обычно всё равно читается лучше.

### Длина строки

Никто не любит читать длинные горизонтальные строки кода. Лучше всего разбивать их, например:

// обратные кавычки ` позволяют разделять строку на части

let str = `

Рабочая группа TC39 организации Ecma International -

это группа JavaScript-разработчиков, теоретиков и авторов движков JavaScript,

которые вместе с сообществом занимаются поддержкой и развитием языка JavaScript.

`;

Или для if:

if (

id === 123 &&

moonPhase === 'Waning Gibbous' &&

zodiacSign === 'Libra'

) {

letTheSorceryBegin();

}

Максимальную длину строки согласовывают в команде. Обычно это 80 или 120 символов.

### Отступы

Существует два типа отступов:

* **Горизонтальные отступы: два или четыре пробела.**

Горизонтальный отступ выполняется с помощью 2 или 4 пробелов, или символа табуляции (клавиша Tab). Какой из них выбрать – это уже на ваше усмотрение. Пробелы больше распространены.

Одно из преимуществ пробелов над табуляцией заключается в том, что пробелы допускают более гибкие конфигурации отступов, чем символ табуляции.

Например, мы можем выровнять аргументы относительно открывающей скобки:

show(parameters,

aligned, // 5 пробелов слева

one,

after,

another

) {

// ...

}

**Вертикальные отступы: пустые строки для разбивки кода на «логические блоки».**

Даже одну функцию часто можно разделить на логические блоки. В примере ниже разделены инициализация переменных, основной цикл и возвращаемый результат:

function pow(x, n) {

let result = 1;

// <--

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

result \*= x;

}

// <--

return result;

}

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

### [Точка с запятой](https://learn.javascript.ru/coding-style" \l "tochka-s-zapyatoy)

Точки с запятой должны присутствовать после каждого выражения, даже если их, казалось бы, можно пропустить.

Есть языки, в которых точка с запятой необязательна и редко используется. Однако в JavaScript бывают случаи, когда перенос строки не интерпретируется, как точка с запятой, что может привести к ошибкам. Подробнее об этом – в главе [о структуре кода](https://learn.javascript.ru/structure#semicolon).

Если вы – опытный разработчик на JavaScript, то можно выбрать стиль кода без точек с запятой, например [StandardJS](https://standardjs.com/). В ином случае, лучше будет использовать точки с запятой, чтобы избежать подводных камней. Большинство разработчиков их ставят.

### [Уровни вложенности](https://learn.javascript.ru/coding-style" \l "urovni-vlozhennosti)

Уровней вложенности должно быть немного.

Например, в цикле бывает полезно использовать директиву [continue](https://learn.javascript.ru/while-for#continue), чтобы избежать лишней вложенности.

Например, вместо добавления вложенного условия if, как здесь:

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

if (cond) {

... // <- ещё один уровень вложенности

}

}

Мы можем написать:

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

if (!cond) continue;

... // <- нет лишнего уровня вложенности

}

Аналогичная ситуация – с if/else и return.

Например, две нижеследующие конструкции идентичны.

Первая:

function pow(x, n) {

if (n < 0) {

alert("Отрицательные значения 'n' не поддерживаются");

} else {

let result = 1;

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

result \*= x;

}

return result;

}

}

Вторая:

function pow(x, n) {

if (n < 0) {

alert("Отрицательные значения 'n' не поддерживаются");

return;

}

let result = 1;

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

result \*= x;

}

return result;

}

Второй вариант является более читабельным, потому что «особый случай» n < 0 обрабатывается на ранней стадии. После проверки можно переходить к «основному» потоку кода без необходимости увеличения вложенности.

## [Размещение функций](https://learn.javascript.ru/coding-style" \l "razmeschenie-funktsiy)

Если вы пишете несколько вспомогательных функций, а затем используемый ими код, то существует три способа организации функций.

1. Объявить функции перед кодом, который их вызовет:

// объявление функций

function createElement() {

...

}

function setHandler(elem) {

...

}

function walkAround() {

...

}

// код, который использует их

let elem = createElement();

setHandler(elem);

walkAround();

1. Сначала код, затем функции

// код, использующий функции

let elem = createElement();

setHandler(elem);

walkAround();

// --- вспомогательные функции ---

function createElement() {

...

}

function setHandler(elem) {

...

}

function walkAround() {

...

}

1. Смешанный стиль: функция объявляется там, где она используется впервые.

В большинстве случаев второй вариант является предпочтительным.

Это потому, что при чтении кода мы сначала хотим знать, что он делает. Если сначала идёт код, то это тут же становится понятно. И тогда, может быть, нам вообще не нужно будет читать функции, особенно если их имена хорошо подобраны.

## Руководства по стилю кода

Руководство по стилю содержит общие правила о том, как писать код, например: какие кавычки использовать, сколько пробелов отступать, максимальную длину строки и так далее – в общем, множество мелочей.

Когда все участники команды используют одно и то же руководство по стилю, код выглядит одинаково, независимо от того, кто из команды его написал.

Конечно, команда всегда может написать собственное руководство по стилю, но обычно в этом нет необходимости. Существует множество уже готовых.

Некоторые популярные руководства:

* [Google JavaScript Style Guide](https://google.github.io/styleguide/javascriptguide.xml)
* [Airbnb JavaScript Style Guide](https://github.com/airbnb/javascript) (есть [перевод](https://leonidlebedev.github.io/javascript-airbnb/))
* [Idiomatic.JS](https://github.com/rwaldron/idiomatic.js) (есть [перевод](https://github.com/rwaldron/idiomatic.js/tree/master/translations/ru_RU))
* [StandardJS](https://standardjs.com/)
* (и ещё множество других)

Если вы – начинающий разработчик, то начните со шпаргалки в начале этой главы. Как только вы освоитесь, просмотрите другие руководства, чтобы выбрать общие принципы и решить, какое вам больше подходит.

## Автоматизированные средства проверки (линтеры)

Автоматизированные средства проверки, так называемые «линтеры» – это инструменты, которые могут автоматически проверять стиль вашего кода и вносить предложения по его улучшению.

Самое замечательное в них то, что проверка стиля может также найти программные ошибки, такие как опечатки в именах переменных или функций. Из-за этой особенности использовать линтер рекомендуется, даже если вы не хотите придерживаться какого-то конкретного «стиля кода».

Вот некоторые известные инструменты для проверки:

* [JSLint](http://www.jslint.com/) – проверяет код на соответствие [стилю JSLint](http://www.jslint.com/lint.html), в онлайн-интерфейсе вверху можно ввести код, а внизу – различные настройки проверки, чтобы попробовать её в действии.
* [JSHint](http://www.jshint.com/) – больше проверок, чем в JSLint.
* [ESLint](http://eslint.org/) – пожалуй, самый современный линтер.

Большинство линтеров интегрированы со многими популярными редакторами: просто включите плагин в редакторе и настройте стиль.

Например, для ESLint вы должны выполнить следующее:

1. Установите [Node.JS](https://nodejs.org/).
2. Установите ESLint с помощью команды npm install -g eslint (npm – установщик пакетов JavaScript).
3. Создайте файл конфигурации с именем .eslintrc в корне вашего JavaScript-проекта (в папке, содержащей все ваши файлы).
4. Установите/включите плагин для вашего редактора, который интегрируется с ESLint. У большинства редакторов он есть.

Вот пример файла .eslintrc:

{

"extends": "eslint:recommended",

"env": {

"browser": true,

"node": true,

"es6": true

},

"rules": {

"no-console": 0,

"indent": ["warning", 2]

}

}

Здесь директива "extends" означает, что конфигурация основана на наборе настроек «eslint:recommended». После этого мы уточняем наши собственные.

Кроме того, возможно загрузить наборы правил стиля из сети и расширить их. Смотрите <https://eslint.org/docs/user-guide/getting-started> для получения более подробной информации об установке.

Также некоторые среды разработки имеют встроенные линтеры, возможно, удобные, но не такие гибкие в настройке, как ESLint.

# Итого

Все правила синтаксиса, описанные в этой главе (и в ссылках на руководства по стилю), направлены на повышение читаемости вашего кода. О любых можно поспорить.

Когда мы думаем о написании «лучшего» кода, мы должны задать себе вопросы: «Что сделает код более читаемым и лёгким для понимания?» и «Что может помочь избегать ошибок?». Это – основные моменты, о которых следует помнить при выборе и обсуждении стилей кода.

Чтение популярных руководств по стилю позволит вам быть в курсе лучших практик и последних идей и тенденций в стилях написания кода.