# How to contribute

Contributions to the project generally take one of three forms:

- Bug reports

- Feature requests

- Patches

This document describes the best way to contribute each of these. The

maintenance team will assign one of the following labels to document the

report's severity:

- \*P1:\* Something is throwing exceptions; broken JSHint backward compatibility.

- \*P2:\* Something is not being parsed correctly.

- \*P3:\* Features that the core team will work on once P2s and P1s are done.

- \*P4:\* Patches welcome; The request is good, but low priority.

## Bug Reports

If you believe you have identified incorrect behavior, please let the team know

by filing an issue. In order to help the team diagnose and fix the problem, the

issue report should have the following information:

- Version of JSHint being used

- Input source code (simplified to only contain details necessary to

demonstrate the problem)

- Configuration values

- Description of expected behavior

- Description of actual behavior

## Feature Requests

If there is some new functionality that you think would improve JSHint, we'd

love to hear about it! Great feature requests contain the following

information:

- Description of problem the feature solves

- Outline of the feature's intended operation

- List of any edge cases/exceptional circumstances, and how they should be

addressed

If you are capable of implementing the feature and submitting a patch, then all

the better! Please begin by making a feature request so that the maintainers

can verify that it is appropriate and help shape its design.

## Patches

The best way to make sure your issue is addressed is to submit a patch. We

accept patches through all mediums: pull requests, email, issue comment, tweet

with a link to a snippet, etc.

However, before sending a patch, please make sure that the following applies:

\* Your commit message follows the [Commit Message Guidelines](#commit-message-guidelines).

\* You have signed the [Contributor's License Agreement](https://www.clahub.com/agreements/jshint/jshint).

\* Your patch doesn't have useless merge commits.

\* Your coding style is similar to ours (see below).

\* Your patch is 100% tested. We don't accept any test regressions.

\* All tests and lint checks pass (`npm test`).

\* You understand that we're super grateful for your patch.

### Development Environment

JSHint is developed using [Node.js](http://nodejs.org/) and has a number of

dependencies specified in its `package.json` file. To install them just run the

following command from within your repo directory:

$ npm install

After that, you will be able to run the edge version of JSHint using

`bin/jshint` or build the release bundles using `bin/build`.

### Coding Style

This section describes our coding style guide. You might not agree with it and

that's fine but if you're going to send us patches treat this guide as a law.

\*\*Our main rule is simple:\*\*

> All code in any code-base should look like a single person typed it, no

> matter how many people contributed.

> —[idiomatic.js](https://github.com/rwaldron/idiomatic.js/)

#### Whitespace

\* We use two spaces everywhere.

\* Use one space after `if`, `for`, `while`, etc.

\* No spaces between `function` and `(` for anonymous functions, no space between name and `(` for named functions:

```javascript

var a = function() {};

function a() {}

```

\* Feel free to indent variable assignments or property definitions if it makes the code look better. But don't abuse that:

```javascript

// Good

var next = token.peak();

var prev = token.peak(-1);

var cur = token.current;

var scope = {

name: "(global)",

parent: parentScope,

vars: [],

uses: []

};

// Bad

var cur = token.current;

var isSemicolon = cur.isPunctuator(";");

```

\* Wrap multi-line comments with new lines on both sides.

#### Variables

\* Use one `var` per variable unless you don't assign any values to it (and it's short enough):

```javascript

var token = tokens.find(index);

var scope = scopes.current;

var next, prev, cur;

```

\* Don't be overly descriptive with your variable names but don't abuse one-letter variables either. Find a sweet spot somewhere in between.

#### Comments

\* Comment everything that is not obvious.

\* If you're adding a new check, write a comment describing why this check is important and what it checks for.

#### Misc

\* Always use strict mode.

\* Always use strict comparisons: `===` and `!==`.

\* Use semicolons.

\* Don't use comma-first notation.

\* Try not to chain stuff unless it \*\*really\*\* helps (e.g. in tests).

\* Don't short-circuit expressions if you're not assigning the result:

```javascript

// Good

token = token || tokens.find(0);

// Bad

token.isPunctuator(";") && report.addWarning("W001");

// Good

if (token.isPunctuator(";"))

report.addWarning("W001");

```

### Commit Message Guidelines

Commit messages are written in a simple format which clearly describes the purpose of a change.

The format in general should look like this:

```

[[TYPE]] <Short description>

<Blank line>

<Body / Detailed description>

<Footer>

```

Line lengths in commit messages are not strict, but good commit messages should have headers of no

more than 60 characters, and bodies/footers wrapped at 100 columns. This renders nicely on Github's

UI.

#### Header

The first line is the commit message header, which will indicate the type of change, and a general

description of the change. This should fit within 60 characters, ideally. For instance:

```

[[FIX]] Ignore "nocomma" when parsing object literals

```

The title `[[FIX]]` indicates that the change is a bugfix, while the remainder clarifies what the

change actually contains.

Several commit types are used by jshint:

1. `[[FIX]]` --- Commit fixes a bug or regression

2. `[[FEAT]]` --- Commit introduces new functionality

3. `[[DOCS]]` --- Commit modifies documentation. Docs commits should only touch comments in source code, or scripts and assets which are used to generate the documentation.

4. `[[TEST]]` --- Commit modifies tests or test infrastructure only

5. `[[CHORE]]` --- Commit affects dev-ops, CI, or package dependencies

#### Body

`<Body>` is a detailed commit message explaining exactly what has changed, and a summary of the

reason why. Lines in the body should be wrapped to 100 characters for best rendering.

For a historical example, see this [example](https://github.com/jshint/jshint/commit/5751c5ed249b7a035758a3ae876cfa1a360fd144)

#### Footer

`<Footer>` contains a description of any breaking changes, no matter how subtle, as well as a list

of issues affected or fixed by this commit. Lines in the footer should be wrapped to 100 characters

for best rendering.

For instance:

```

[[FEAT]] Enable `norecurs` option by default

Commit 124124a7f introduced an option which forbids recursion. We liked it so much, we've enabled

it by default.

BREAKING CHANGE:

This change will break the CI builds of many applications and frameworks.

In order to work around this issue, you will need to re-engineer your applications and frameworks

to avoid making recursive calls. Use Arrays as stacks rather than relying on the VM call stack.

Fixes #1000009

Closes #888888

Closes #77777

```