Contributing to Pokémon Showdown

========================================================================

Building and running

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

The README contains most of the relevant information here.

https://github.com/smogon/pokemon-showdown/blob/master/README.md

Our build script does most of the work here: You can mostly just run `./pokemon-showdown` to start a server. (Windows users will have to replace `./whatever` with `node whatever`, every time it appears)

PS has other useful command-line invocations, which you can investigate with `./pokemon-showdown help`.

Unit tests can be run with `npm test`. You can run specific unit tests with `npx mocha -g "text"`, which will run all unit tests whose name contains "text", or you can just edit the unit test from `it` to `it.only`.

Contributing

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

In general, we welcome pull requests that fix bugs.

For feature additions and large projects, please discuss with us at https://psim.us/development an/dor https://psim.us/devdiscord first. We'd hate to have to reject a pull request that you spent a long time working on.

If you're looking for inspiration for something to do, the Ideas issue has some ideas: https://github.com/smogon/pokemon-showdown/issues/2444

Also useful is the Suggestions forum (you don't need to worry about approval if you take Approved suggestions): https://www.smogon.com/forums/forums/suggestions.517/

Also useful is the Mechanics Bugs kanban board: https://github.com/smogon/pokemon-showdown/projects/3

There's no need to worry about code standards too much (unit tests will automatically catch most of what we care about, we'll point out the rest if you make a pull request), but there here if you want them.

We try to respond to pull requests within a few days, but feel free to bump yours if it seems like we forget about it. Sometimes we did, and sometimes there might be a miscommunication in terms of who is waiting for what.

License

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

Your submitted code should be MIT licensed. The GitHub ToS (and the fact that your fork also contains our LICENSE file) ensures this, so we won't ask when you submit a pull request, but keep this in mind.

For simplicity (mostly to make relicensing easier), client code should be also be MIT licensed. The first time you make a client pull request, we'll ask you to explicitly state that you agree to MIT license it.

Commit standards

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

Commits should describe what the code \_does\_, not how it does it.

In other words:

- BAD: `Change Wonder Guard from onBeforeMove to onTryHit`

- GOOD: `Fix Mold Breaker Wonder Guard interaction`

The details of how you achieve the fix should be left for the second paragraph of the commit message.

If this is not possible because your code does not make any functionality changes, your commit summary should ideally start with the word "Refactor" (or at least it contain it in some way).

Commits should usually start with a verb in imperative mood, such as "Add", "Fix", "Refactor", etc (if the verb is there, it should be imperative, but it doesn't have to be there).

- BAD: `Adding namefilter`

- BAD: `Adds namefilter`

- GOOD: `Add namefilter`

The first line of the commit summary should be under 50 characters long.

The first letter of a commit summary should be capitalized (unless the first word starts with a number or is case-sensitive, e.g. `ls`).

The commit summary should not end in a period.

- BAD: `refactor users to use classes`

- BAD: `Refactor Users to use classes.`

- GOOD: `Refactor Users to use classes`

Your commit summary should make it clear what part of the code you're talking about. For instance, if you're editing the Trivia plugin, you might want to add "Trivia: " to the beginning of your commit summary so it's clear.

- BAD: `Ban Genesect`

- GOOD: `Monotype: Ban Genesect` (notice the uppercase "B")

OPTIONAL: If you make commits to fix commits in your pull request, you can squash/amend them into one commit. This is no longer required now that GitHub supports squash-merging.

- BAD: `Add /lock`, `Fix crash in /lock`, `Fix another crash in /lock` (if these are the same pullreq, they should be the same commit)

- GOOD: `Add /lock`

- GOOD: `Fix crash in /lock`

If you want to have more than one commit in Git master's history after merge (i.e. you want your pull request to be rebase-merged instead of squash-merged), your commits need to all make sense as separate commits, and none of your commits should be just fixing an earlier commit in your pull request (those need to be squashed/amended).

Here is a guide for squashing, if you need help with that: https://redew.github.io/rebaseguide/

If while rebasing, you somehow unintentionally break your pull request, do not close it and make a new one to replace it. Instead, you can ask in the Development chatroom for help on trying to fix it; it can almost always be fixed.

Code standards

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

We enforce most of our code standards through `eslint`. Just run `npm test` and it'll tell you if something's wrong.

Looking at your surrounding text is also a way to get a good idea of our coding style.

### Strings

The codebase currently uses a mix of `"` and `'` and `` ` `` for strings.

Our current quote convention is to use:

- `` ` `` as in `` `move-${move.id}` `` for any string that needs interpolation, otherwise:

- `` ` `` as in `` `<strong>Fire Blast</strong>` `` for code meant to be fed to an interpreter/tokenizer before being displayed to the user; i.e. protocol code and HTML

- `'` as in `'fireblast'` for any string not meant to be displayed to the user; i.e. IDs

- `"` as in `"Fire Blast"` for any string meant to be displayed verbatim to the user; i.e. names (i.e. usernames, move names, etc), most English text, and help entries of chat commands

As far as I know, we don't use strings for anything else, but if you need to use strings in a way that doesn't conform to the above three, ask Zarel in the Development chatroom to decide (and default to `` ` `` in lieu of a decision).

Unfortunately, since this is not a convention the linter can test for (and also because our older string standards predate PS), a lot of existing code is wrong on this, so you can't look at surrounding code to get an idea of what the convention should be. Refer to the above paragraph as the definitive rule.

### Optionals: `null` vs `undefined` vs `false`

PS convention is to use `null` for optionals. So a function that retrieves a possible `T` would return `T | null`.

Some old code returns `T | undefined` (our previous convention). This is a relatively common standard (ironically, TypeScript itself uses it). Feel free to convert to `T | null` where you see it.

Some even older code returns `T | false`. This is a very old PHP convention that has no place in modern PS code. Please convert to `T | null` if you see it.

### `false | null | undefined`

The simulator (code in `sim/` and `data/`) will often have functions with return signatures of the form `T | false | null | undefined`, especially in event handlers. These aren't optionals, they're different sentinel values.

Specifically:

\* `false` means "this action failed"

\* `null` means "this action failed silently; suppress any 'But it failed!' messages"

\* `undefined` means "this action should be ignored, and treated as if nothing unexpected happened"

So, if Thunder Wave hits a Ground type, the immunity checker returns `false` to indicate the immunity.

If Volt Absorb absorbs Thunder Wave, Volt Absorb's TryHit handler shows the Volt Absorb message and returns `null` to indicate that no other failure message should be shown.

If Water Absorb doesn't absorb Thunder Wave, Water Absorb's TryHit handler returns `undefined`, to show that Water Absorb does not interact with Thunder Wave.

### `??` vs `||`

We prefer using `||` instead of `??` for fallback, for a few reasons:

- `sucrase` (our TypeScript to JavaScript compiler) makes `??` rather more complicated than ideal.

- We rarely treat `0` or `''` differently from `null` (the same reason we use `!foo` instead of `foo == null` for null checks)

- TypeScript does not actually allow us to have "non-empty strings" or "positive integers" as a type, so we have to deal with those cases no matter what.

If, at a future point, TypeScript does allow us to constrain types better, we might consider using `??` for clarity. But for now, I see no reason to use `??` except in very niche situations where the difference matters.

ES5 and ES6

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

In general, use modern features; recent versions of V8 have fixed the performance problems they used to have.

- \*\*let, const: ALWAYS\*\* - Supported in Node 4+, good performance.

- \*\*for-of on Arrays: ALWAYS\*\* - Supported in Node 4+, good performance in Node 8+.

- \*\*Array#forEach: NEVER\*\* - Poor readability; we prefer `for-of`.

- \*\*for-in on Arrays: NEVER\*\* - Horrible performance, weird bugs due to string keys, poor interaction with Array prototype modification. Everyone tells you never to do it; we're no different. See `for-of`.

- \*\*Map, Set: SOMETIMES\*\* - Worse write/iteration performance, better read performance than `Object.create(null)`. Use whatever's faster for your use case.

- \*\*for-in on Objects: ALWAYS\*\* - More readable; good performance in Node 8+.

- \*\*for-of on Maps and Sets: ALWAYS\*\* - Supported in Node 4+, good performance in Node 8+.

- \*\*Map#forEach, Set#forEach: NEVER\*\* - Poor readability; we prefer `for-of`.

- \*\*Object literal functions: ALWAYS\*\* - Supported in Node 4+, good performance.

- \*\*Arrow functions: ALWAYS\*\* - Supported in Node 4+, good performance. Obviously use only for callbacks; don't use in situations where `this` shouldn't be bound.

- \*\*Promises: ALWAYS\*\* - Supported in Node 4+, poor performance but worth the readability.

- \*\*async/await: ALWAYS\*\* - Supported in Node 8+, good performance.

- \*\*Function#bind: NEVER\*\* - Horrible performance. Use arrow functions.

- \*\*classes and subclasses: ALWAYS\*\* - Supported in Node 4+ and good performance in Node 6+.

- \*\*String#includes: ALWAYS\*\* - Supported in Node 4+, poor performance, but not really noticeable and worth the better readability.

- \*\*Template strings: ALWAYS\*\* - Supported in Node 4+ and good performance in Node 6+; please start refactoring existing code over, but be careful not to use them for IDs (follow the String standards). Look at existing uses for guidance.

- \*\*Multiline template strings: NEVER\*\* - Multiline template strings are a frequent source of bugs, so it's better to be explicit with `\n`.

Take "good performance" to mean "approximately on par with ES3" and "great performance" to mean "better than ES3".

TypeScript Features

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

- \*\*Constant Enums: NEVER\*\* - Not supported by Sucrase, our current choice of transpiler. We prefer constant union types, anyway (like `type Category = 'Physical' | 'Special' | 'Status'`)

- \*\*Default Properties: SOMETIMES\*\* - Bad performance when used with Sucrase. This is fine for objects that are rarely created, but prefer setting properties directly in a constructor, for objects created in inner loops.