Instructions for Logging Issues

1. Read the FAQ

Please read the FAQ before logging new issues, even if you think you have found a bug.

Issues that ask questions answered in the FAQ will be closed without elaboration.

2. Search for Duplicates

Search the existing issues before logging a new one.

Some search tips:

- *Don't* restrict your search to only open issues. An issue with a title similar to yours may have been closed as a duplicate of one with a less-findable title.
- Check for synonyms. For example, if your bug involves an interface, it likely also occurs with type aliases or classes.
- Search for the title of the issue you're about to log. This sounds obvious but 80% of the time this is sufficient to find a duplicate when one exists.
- Read more than the first page of results. Many bugs here use the same words so relevancy sorting is not particularly strong.
- If you have a crash, search for the first few topmost function names shown in the call stack.

3. Do you have a question?

The issue tracker is for **issues**, in other words, bugs and suggestions. If you have a *question*, please use Stack Overflow, Gitter, your favorite search engine, or other resources. Due to increased traffic, we can no longer answer questions in the issue tracker.

4. Did you find a bug?

When logging a bug, please be sure to include the following:

- What version of TypeScript you're using (run `tsc --v`)
- If at all possible, an *isolated* way to reproduce the behavior
- The behavior you expect to see, and the actual behavior

You can try out the nightly build of TypeScript (`npm install typescript@next`) to see if the bug has already been fixed.

5. Do you have a suggestion?

We also accept suggestions in the issue tracker. Be sure to check the FAQ and search first.

In general, things we find useful when reviewing suggestions are:

- A description of the problem you're trying to solve
- An overview of the suggested solution
- Examples of how the suggestion would work in various places
  - Code examples showing e.g. "this would be an error, this wouldn't"
  - Code examples showing the generated JavaScript (if applicable)

- If relevant, precedent in other languages can be useful for establishing context and expected behavior

Instructions for Contributing Code

What You'll Need

0. A bug or feature you want to work on!
1. A GitHub account.
2. A copy of the TypeScript code. See the next steps for instructions.
3. Node, which runs JavaScript locally. Current or LTS will both work.
4. An editor. VS Code is the best place to start for TypeScript.
5. The gulp command line tool, for building and testing changes. See the next steps for how to install it.

Get Started

1. Install node using the version you downloaded from nodejs.org.
2. Open a terminal.
3. Make a fork—your own copy—of TypeScript on your GitHub account, then make a clone—a local copy—on your computer. (Here are some step-by-step instructions). Add `--depth=1` to the end of the `git clone` command to save time.
4. Install the gulp command line tool: `npm install -g gulp-cli`
5. Change to the TypeScript folder you made: `cd TypeScript`
6. Install dependencies: `npm ci`
7. Make sure everything builds and tests pass: `gulp runtests-parallel`
8. Open the Typescript folder in your editor.
9. Follow the directions below to add and debug a test.

Tips

Using a development container

If you prefer to develop using containers, this repository includes a development container that you can use to quickly create an isolated development environment with all the tools you need to start working on TypeScript. To get started with a dev container and VS Code, either:

- Clone the TypeScript repository locally and use the `Open Folder in Container` command.
- Use the `Clone Repository in Container Volume` command to clone the TypeScript repository into a new container.

Faster clones

The TypeScript repository is relatively large. To save some time, you might want to clone it without the repo's full history using `git clone --depth=1`.

Filename too long on Windows

You might need to run `git config --global core.longpaths true` before cloning TypeScript on Windows.

Using local builds

Run `gulp` to build a version of the compiler/language service that reflects changes you've made. You can then run `node <repo-root>/built/local/tsc.js` in place of `tsc` in your project. For example, to run `tsc --`

`watch` from within the root of the repository on a file called `test.ts`, you can run `node ./built/local/tsc.js --watch test.ts`.

Contributing bug fixes

TypeScript is currently accepting contributions in the form of bug fixes. A bug must have an issue tracking it in the issue tracker that has been approved (labelled "help wanted" or in the "Backlog milestone") by the TypeScript team. Your pull request should include a link to the bug that you are fixing. If you've submitted a PR for a bug, please post a comment in the bug to avoid duplication of effort.

Contributing features

Features (things that add new or improved functionality to TypeScript) may be accepted, but will need to first be approved (labelled "help wanted" or in the "Backlog" milestone) by a TypeScript project maintainer in the suggestion issue. Features with language design impact, or that are adequately satisfied with external tools, will not be accepted.

Legal

You will need to complete a Contributor License Agreement (CLA). Briefly, this agreement testifies that you are granting us permission to use the submitted change according to the terms of the project's license, and that the work being submitted is under appropriate copyright. Upon submitting a pull request, you will automatically be given instructions on how to sign the CLA.

Housekeeping

Your pull request should:

- Include a description of what your change intends to do
- Be based on reasonably recent commit in the **main** branch
- Include adequate tests
  - At least one test should fail in the absence of your non-test code changes. If your PR does not match this criteria, please specify why
  - Tests should include reasonable permutations of the target fix/change
  - Include baseline changes with your change
- Follow the code conventions described in Coding guidelines
- To avoid line ending issues, set `autocrlf = input` and `whitespace = cr-at-eol` in your git configuration

Contributing `lib.d.ts` fixes

There are three relevant locations to be aware of when it comes to TypeScript's library declaration files:

- `src/lib`: the location of the sources themselves.
- `lib`: the location of the last-known-good (LKG) versions of the files which are updated periodically.
- `built/local`: the build output location, including where `src/lib` files will be copied to.

Any changes should be made to src/lib. **Most** of these files can be updated by hand, with the exception of any generated files (see below).

Library files in `built/local/` are updated automatically by running the standard build task:

```
gulp
```

The files in `lib/` are used to bootstrap compilation and usually **should not** be updated unless publishing a new version or updating the LKG.

Modifying generated library files

The files `src/lib/dom.generated.d.ts` and `src/lib/webworker.generated.d.ts` both represent type declarations for the DOM and are auto-generated. To make any modifications to them, you will have to direct changes to https://github.com/Microsoft/TSJS-lib-generator

Running the Tests

To run all tests, invoke the `runtests-parallel` target using gulp:

```
gulp runtests-parallel
```

This will run all tests; to run only a specific subset of tests, use:

```
gulp runtests --tests=<regex>
```

e.g. to run all compiler baseline tests:

```
gulp runtests --tests=compiler
```

or to run a specific test: `tests\cases\compiler\2dArrays.ts`

```
gulp runtests --tests=2dArrays
```

Debugging the tests

You can debug with VS Code or Node instead with `gulp runtests -i`:

```
gulp runtests --tests=2dArrays -i
```

You can also use the provided VS Code launch configuration to launch a debug session for an open test file. Rename the file 'launch.json', open the test file of interest, and launch the debugger from the debug panel (or press F5).

Adding a Test

To add a new test case, add a `.ts` file in `tests\cases\compiler` with code that shows the bug is now fixed, or your new feature now works.

These files support metadata tags in the format `// @metaDataName: value`. The supported names and values are the same as those supported in the compiler itself, with the addition of the `fileName` flag. `fileName` tags delimit sections of a file to be used as separate compilation units. They are useful for testing modules. See below for examples.

Note that if you have a test corresponding to a specific area of spec compliance, you can put it in the appropriate subfolder of `tests\cases\conformance`. **Note** that test filenames must be distinct from all other test names, so you may have to work a bit to find a unique name if it's something common.

Tests for multiple files

When you need to mimic having multiple files in a single test to test features such as "import", use the `filename` tag:

```
// @filename: file1.ts
export function f() {
}

// @filename: file2.ts
import { f as g } from "file1";

var x = g();
```

Managing the baselines

Most tests generate "baselines" to find differences in output. As an example, compiler tests usually emit one file each for

- the `.js` and `.d.ts` output (all in the same `.js` output file),
- the errors produced by the compiler (in an `.errors.txt` file),
- the types of each expression (in a `.types` file),
- the symbols for each identifier (in a `.symbols` file), and
- the source map outputs for files if a test opts into them (in a `.js.map` file).

When a change in the baselines is detected, the test will fail. To inspect changes vs the expected baselines, use

```
git diff --diff-filter=AM --no-index ./tests/baselines/reference
./tests/baselines/local
```

Alternatively, you can set the `DIFF` environment variable and run `gulp diff`, or manually run your favorite folder diffing tool between `tests/baselines/reference` and `tests/baselines/local`. Our team largely uses Beyond Compare and WinMerge.

After verifying that the changes in the baselines are correct, run

```
gulp baseline-accept
```

This will change the files in `tests\baselines\reference`, which should be included as part of your commit. Be sure to validate the changes carefully -- apparently unrelated changes to baselines can be clues about something you didn't think of.

Localization

All strings the user may see are stored in `diagnosticMessages.json`. If you make changes to it, run `gulp generate-diagnostics` to push them to the `Diagnostic` interface in `diagnosticInformationMap.generated.ts`.

See coding guidelines on diagnostic messages.