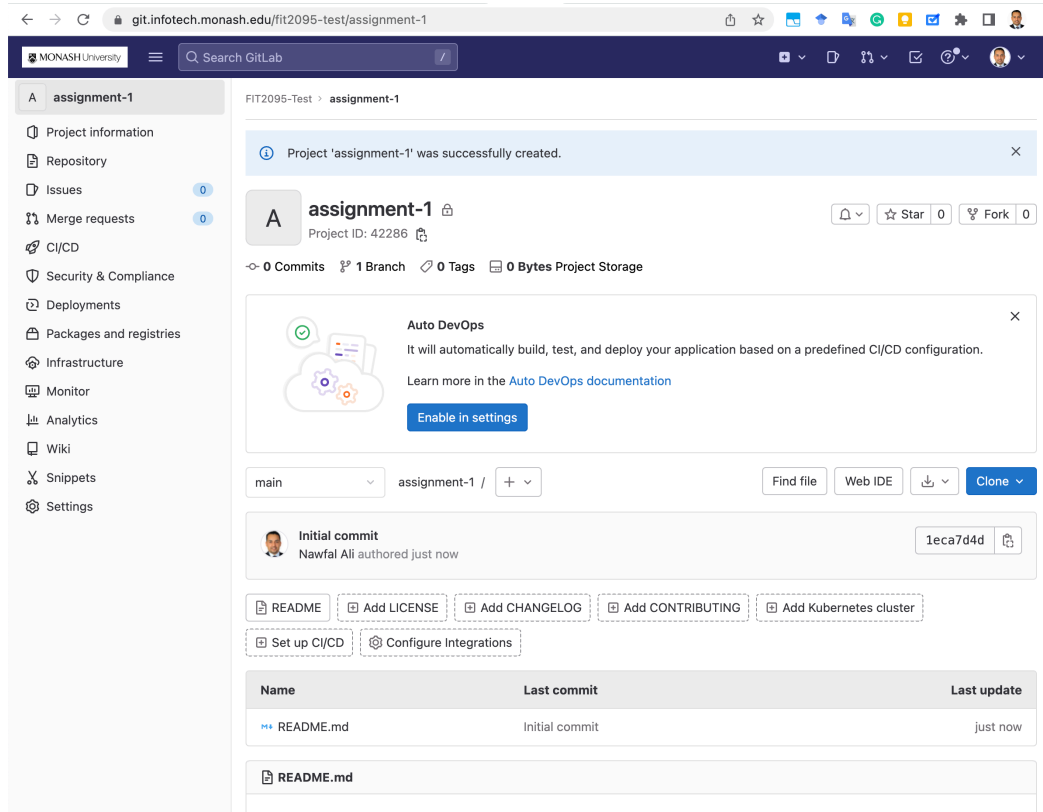


Assignments Control Panel

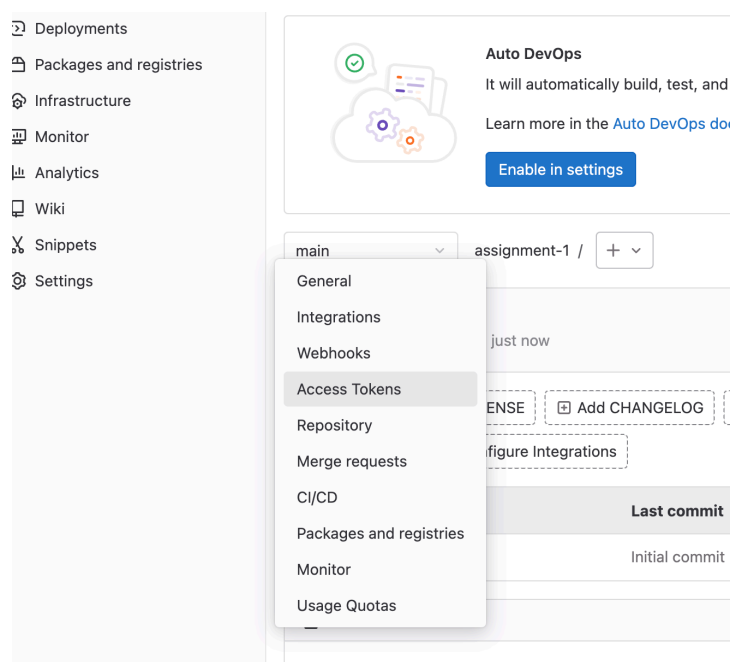
Assessments Timeline

GitLab

- Ensure you have access to <https://git.infotech.monash.edu/>
- Locate your repo.



- Generate the access token (it is your password to access your repo)
- From Settings, select Access Tokens



- Give a name, change the role to 'Maintainer' and select the API option as shown below:

Project Access Tokens

Generate project access tokens scoped to this project for your applications that need access to the GitLab API. You can also use project access tokens with Git to authenticate over HTTP(S). [Learn more](#).

Add a project access token

Enter the name of your application, and we'll return a unique project access token.

Token name

For example, the application using the token or the purpose of the token. Do not give sensitive information for the name of the token, as it will be visible to all project members.

Expiration date

Select a role

Select scopes

Scopes set the permission levels granted to the token. [Learn more](#).

☒

api

Grants complete read and write access to the scoped project API, including the Package Registry.

☐

read_api

Grants read access to the scoped project API, including the Package Registry.

☐

read_repository

Grants read access (pull) to the repository.

☐

write_repository

Grants read and write access (pull and push) to the repository.

- Copy the new access token and save it in a safe place.

Your new project access token has been created.

Project Access Tokens

Generate project access tokens scoped to this project for your applications that need access to the GitLab API. You can also use project access tokens with Git to authenticate over HTTP(S). [Learn more](#).

Your new project access token

Make sure you save it - you won't be able to access it again.

Add a project access token

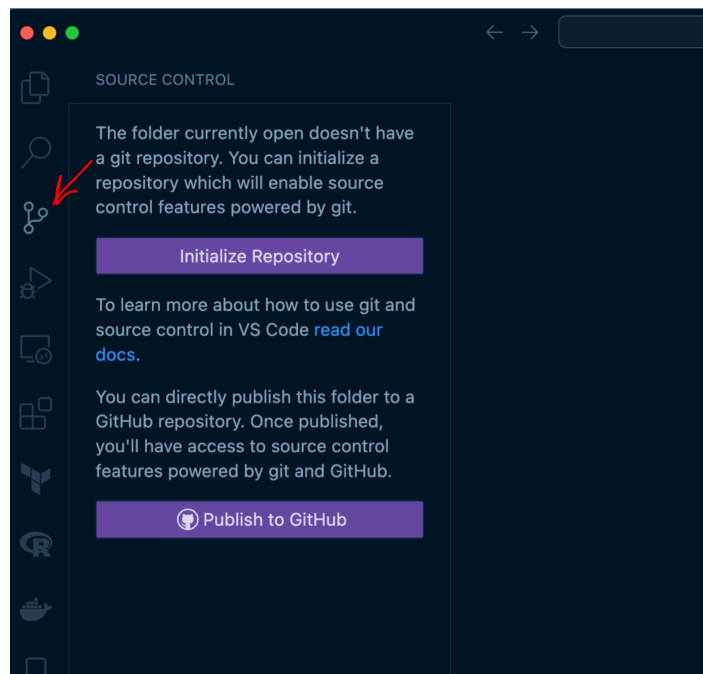
Enter the name of your application, and we'll return a unique project access token.

Token name

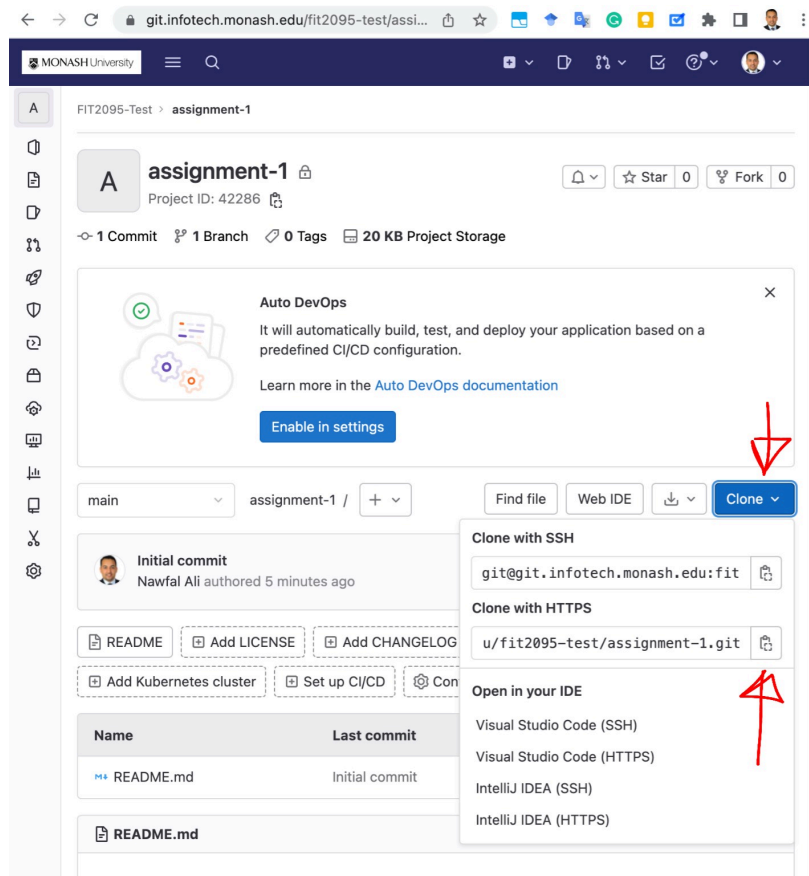
For example, the application using the token or the purpose of the token. Do not give sensitive information for the name of the token, as it will be visible to all project members.

Expiration date

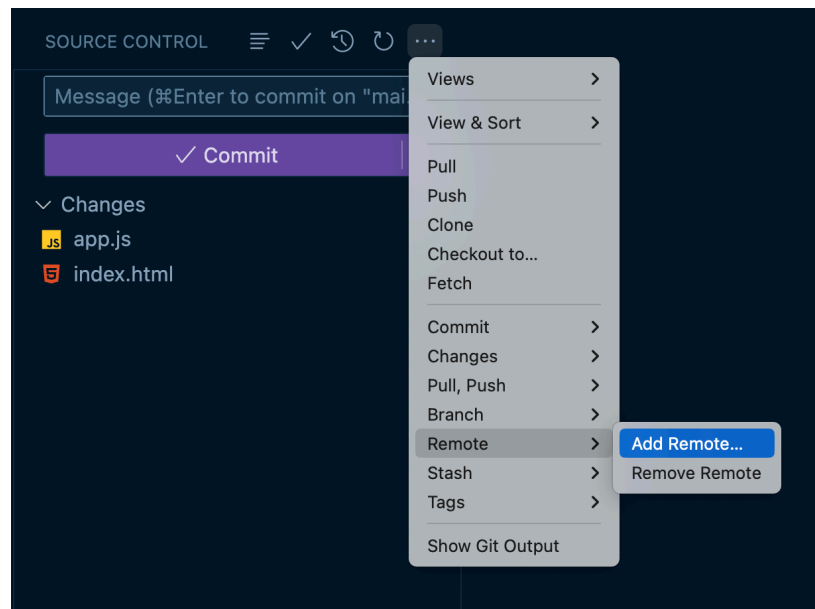
- Now, let's push our project to the remote repo.
- From the VSCode window, select the version control tool and click on Initialize Repository:



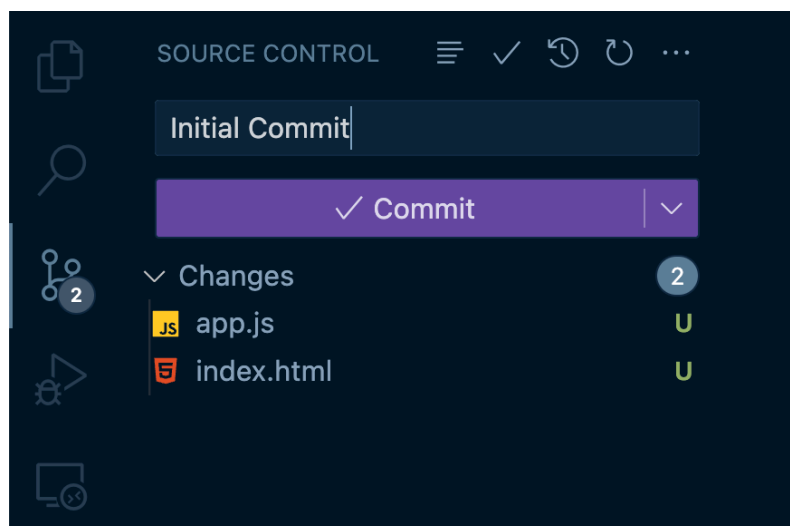
- The next step is to add (i.e. link) the remote URL to the local repo.
- First, copy the remote URL by clicking the Clone Button and then Copy the HTTPS URL



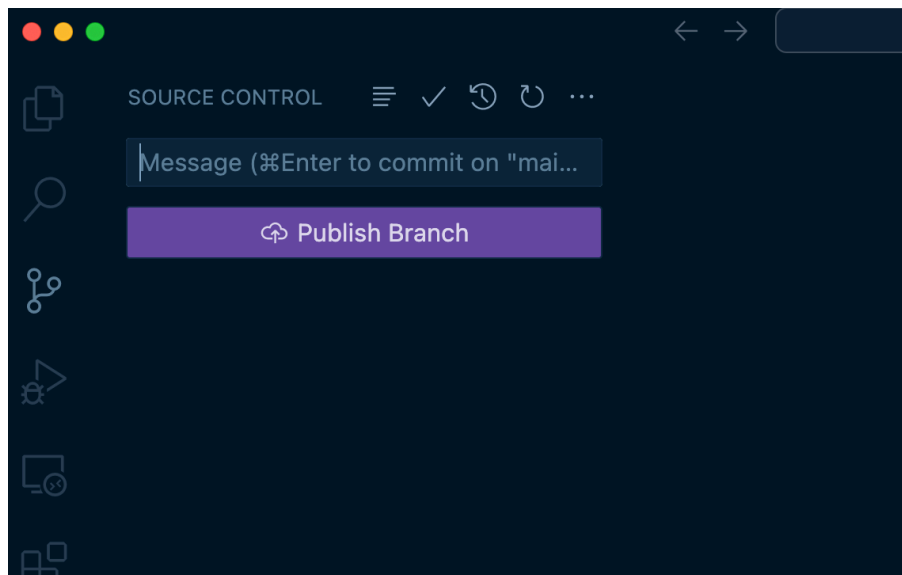
- then, from the VSCode window, click on the three dots-->Remote-->Add Remote and paste the remote URL



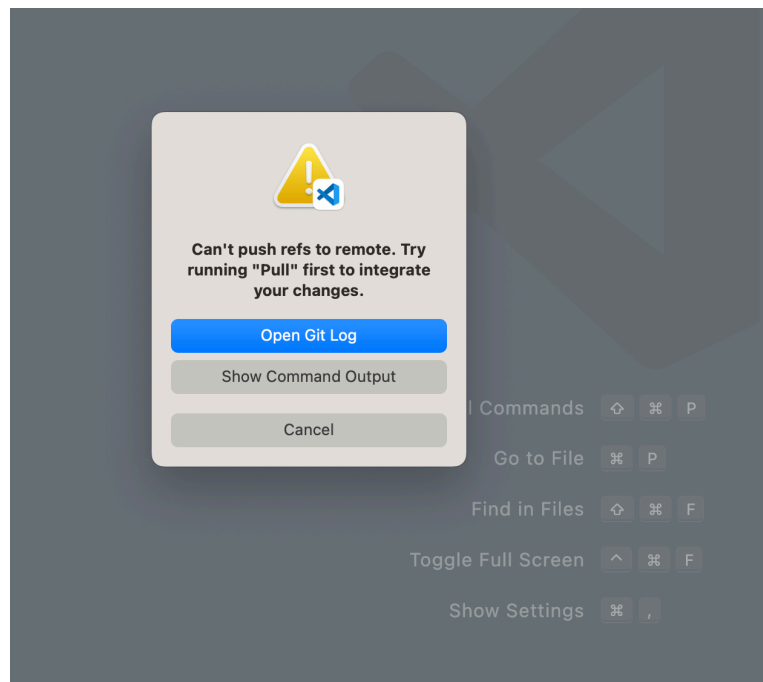
- It needs the following:
 - the remote name branch name, which is '**main**'
 - your user name (Monash Username)
 - and the access token that you generated earlier
- Now, it's time to commit our work to the remote repo.
 - Write a short descriptive message (a must) and hit the Commit button. This action commits your changes to the local repo.



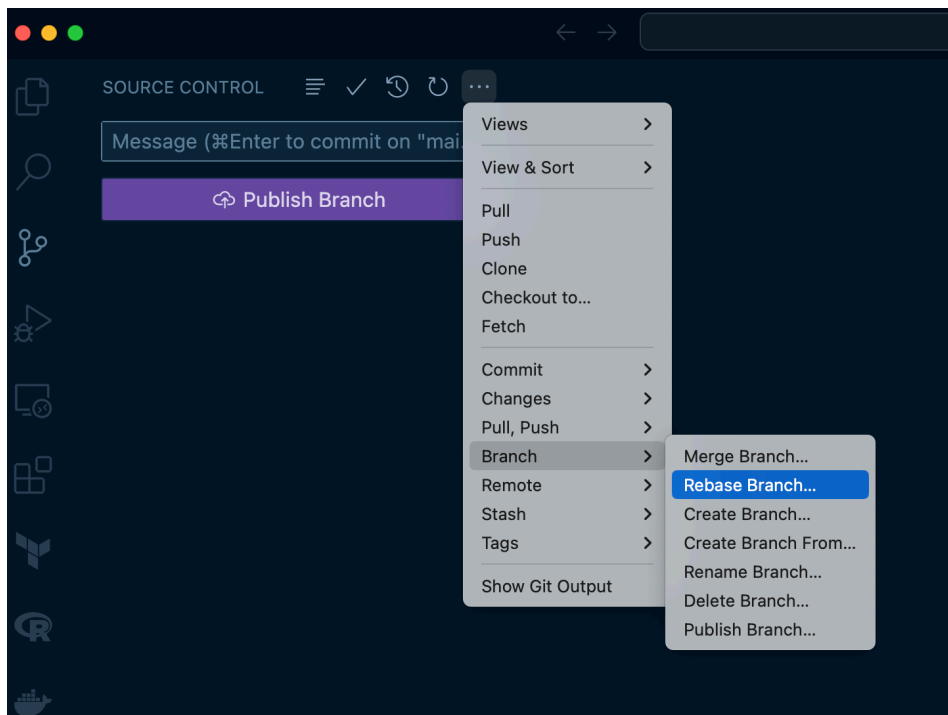
- Now, click on 'Publish Branch'.



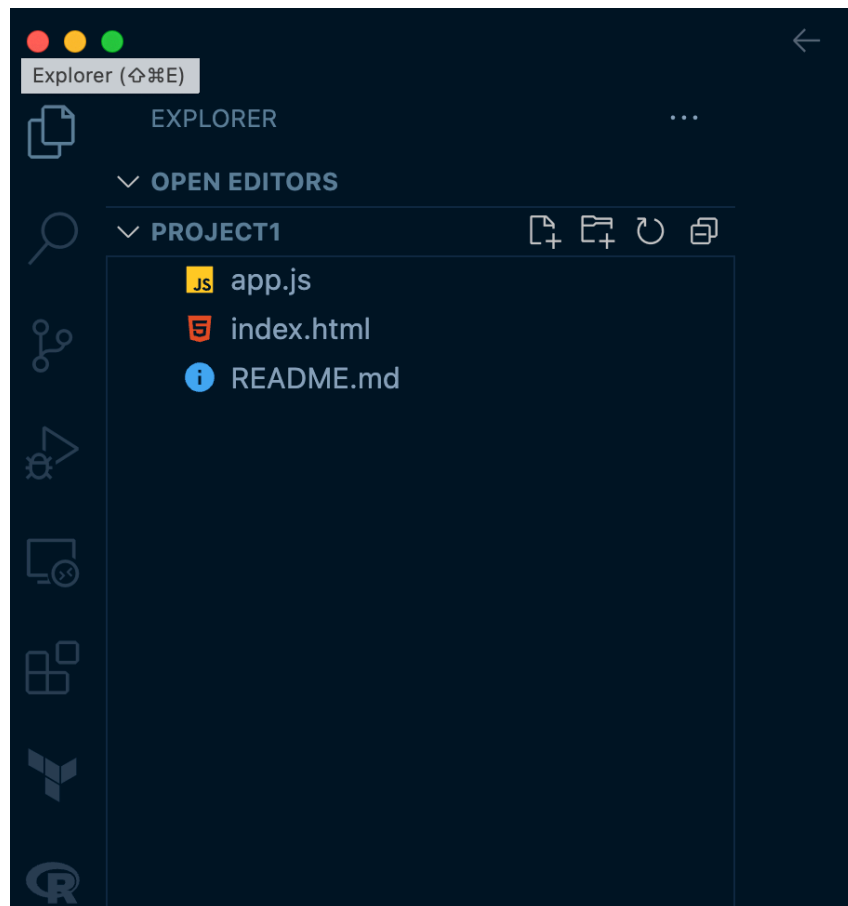
- If you get the error 'Can't push refs to remote, which occurs because the remote repo does have a README.MD file, and this file is not available in the local repo.



- Therefore, we need to bring the remote file down to the local repo using the rebase command.
 - From the three dots-->Branch-->select Rebase Branch



- Select the only branch 'main'; now you should have the remote README.MD file locally.



- Go back to the source control and click on Publish Branch.
- Refresh your GitLab page, and you should see the three files.



Initial Commit
Nawfal Ali authored 6 minutes ago

2fea5ce4

- README
- + Add LICENSE
- + Add CHANGELOG
- + Add CONTRIBUTING
- + Add Kubernetes cluster
- + Set up CI/CD
- Configure Integrations

Name	Last commit	Last update
README.md	Initial commit	42 minutes ago
app.js	Initial Commit	1 minute ago
index.html	Initial Commit	1 minute ago

README.md


Special Consideration



Extension requests sent by [email will not be processed.](#)

[Click HERE for Short and Long Special Considerations](#)

Code Documentation

 Watch the How-To recording at the bottom of this page.

This unit will use **JSDoc**, an API documentation generator for JavaScript.

Installation and Usage

JSDoc supports stable versions of Node.js 8.15.0 and later. You can install JSDoc globally or in your project's `node_modules` folder.

To install globally:

```
npm install -g jsdoc
```

Getting Started with JSDoc 3

Getting started

JSDoc 3 is an API documentation generator for JavaScript, similar to Javadoc or phpDocumentor. You add documentation comments directly to your source code, right alongside the code itself. The JSDoc tool will scan your source code and generate an HTML documentation website for you.

Adding documentation comments to your code

JSDoc's purpose is to document the API of your JavaScript application or library. It is assumed that you will want to document things like modules, namespaces, classes, methods, method parameters, and so on.

JSDoc comments should generally be placed immediately before the code being documented. Each comment must start with a `/**` sequence in order to be recognized by the JSDoc parser. Comments beginning with `/*`, `/**`, or more than 3 stars will be ignored. This is a feature to allow you to suppress parsing of comment blocks.

The simplest documentation is just a description

```
/** This is a description of the foo function. */  
function foo() {  
}
```

Adding a description is simple—just type the description you want in the documentation comment.

Special "JSDoc tags" can be used to give more information. For example, if the function is a constructor for a class, you can indicate this by adding a `@constructor` tag.

Use a JSDoc tag to describe your code

```
/**  
 * Represents a book.  
 * @constructor  
 */  
function Book(title, author) {  
}
```

More tags can be used to add more information. See the [home page](#) for a complete list of tags that are recognized by JSDoc 3.

Adding more information with tags

```
/**  
 * Represents a book.  
 * @constructor  
 * @param {string} title - The title of the book.  
 * @param {string} author - The author of the book.  
 */  
function Book(title, author) {  
}
```

Generating a website

Once your code is commented, you can use the JSDoc 3 tool to generate an HTML website from your source files.

By default, JSDoc uses the built-in "default" template to turn the documentation into HTML. You can edit this template to suit your own needs or create an entirely new template if that is what you prefer.

Running the documentation generator on the command line

```
jsdoc book.js
```

This command will create a directory named `out/` in the current working directory. Within that directory, you will find the generated HTML pages.

JSDoc Demonstration

References:

- <https://github.com/jsdoc/jsdoc>
- <https://jsdoc.app/about-getting-started.html>
- <https://www.typescriptlang.org/docs/handbook/jsdoc-supported-types.html>

Code Style

In this unit, you must Google JavaScript Style for your source code. Find below excerpts from Google's guide.

Package names

Package names are all `lowerCamelCase`. For example, `my.exampleCode.deepSpace`, but not `my.examplecode.deepspace` or `my.example_code.deep_space`.

Class names

Class, interface, record, and typedef names are written in `UpperCamelCase`.

Type names are typically nouns or noun phrases. For example, `Request`, `ImmutableList`, or `VisibilityMode`. Additionally, interface names may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

Method names

Method names are written in `lowerCamelCase`.

Method names are typically verbs or verb phrases. For example, `sendMessage`. Getter and setter methods for properties should be named `getFoo` (or optionally `isFoo` or `hasFoo` for booleans), or `setFoo(value)` for setters.

Enum names

Enum names are written in `UpperCamelCase`, similar to classes, and should generally be singular nouns. Individual items within the enum are named in `CONSTANT_CASE`.

Constant names

Constant names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores.

Non-constant field names

Non-constant field names (static or otherwise) are written in `lowerCamelCase`.

These names are typically nouns or noun phrases. For example, `computedValues`.

Parameter names

Parameter names are written in `lowerCamelCase`. Note that this applies even if the parameter expects a constructor.

One-character parameter names should not be used in public methods.

Local variable names

Local variable names are written in `lowerCamelCase`, except for module-local (top-level) constants, as described above.

Example

```
// Constants
const NUMBER = 5;
/** @const */ exports.NAMES = ImmutableList.of('Ed', 'Ann');
/** @enum */ exports.SomeEnum = { ENUM_CONSTANT: 'value' };

// Not constants
let letVariable = 'non-const';
class MyClass { constructor() { /** @const {string} */ this.nonStatic = 'non-static'; } };
/** @type {string} */ MyClass.staticButMutable = 'not @const, can be reassigned';
const /** Set<string> */ mutableCollection = new Set();
const /** ImmutableSet<SomeMutableType> */ mutableElements = ImmutableSet.of(mutable);
const Foo = goog.require('my.Foo'); // mirrors imported name
const logger = log.getLogger('loggers.are.not.immutable');
```

Reference

[Click HERE to access Google JavaScript Style Guide.](#)