# Contributing

## JUnit Contributor License Agreement

- You will only Submit Contributions where You have authored 100% of the content.

- You will only Submit Contributions to which You have the necessary rights. This means

that if You are employed You have received the necessary permissions from Your employer

to make the Contributions.

- Whatever content You Contribute will be provided under the Project License(s).

### Project Licenses

- All modules use [Eclipse Public License v2.0](LICENSE.md).

## Commit Messages

As a general rule, the style and formatting of commit messages should follow the guidelines in

[How to Write a Git Commit Message](https://chris.beams.io/posts/git-commit/).

In addition, any commit that is related to an existing issue must reference the issue.

For example, if a commit in a pull request addresses issue \#999, it must contain the

following at the bottom of the commit message.

```

Issue: #999

```

## Pull Requests

Our [Definition of Done](https://github.com/junit-team/junit5/wiki/Definition-of-Done)

offers some guidelines on what we expect from a pull request.

Feel free to open a pull request that does not fulfill all criteria, e.g. to discuss

a certain change before polishing it, but please be aware that we will only merge it

in case the DoD is met.

Please add the following lines to your pull request description:

```markdown

---

I hereby agree to the terms of the JUnit Contributor License Agreement.

```

## Coding Conventions

### Naming Conventions

Whenever an acronym is included as part of a type name or method name, keep the first

letter of the acronym uppercase and use lowercase for the rest of the acronym. Otherwise,

it becomes \_impossible\_ to perform camel-cased searches in IDEs, and it becomes

potentially very difficult for mere humans to read or reason about the element without

reading documentation (if documentation even exists).

Consider for example a use case needing to support an HTTP URL. Calling the method

`getHTTPURL()` is absolutely horrible in terms of usability; whereas, `getHttpUrl()` is

great in terms of usability. The same applies for types `HTTPURLProvider` vs

`HttpUrlProvider`, etc.

Whenever an acronym is included as part of a field name or parameter name:

- If the acronym comes at the start of the field or parameter name, use lowercase for the

entire acronym -- for example, `String url;`.

- Otherwise, keep the first letter of the acronym uppercase and use lowercase for the

rest of the acronym -- for example, `String defaultUrl;`.

### Formatting

#### Code

Code formatting is enforced using the [Spotless](https://github.com/diffplug/spotless)

Gradle plugin. You can use `gradle spotlessApply` to format new code and add missing

license headers to source files. Formatter and import order settings for Eclipse are

available in the repository under

[src/eclipse/junit-eclipse-formatter-settings.xml](src/eclipse/junit-eclipse-formatter-settings.xml)

and [src/eclipse/junit-eclipse.importorder](src/eclipse/junit-eclipse.importorder),

respectively. For IntelliJ IDEA there's a

[plugin](https://plugins.jetbrains.com/plugin/6546) you can use in conjunction with the

Eclipse settings.

It is forbidden to use \_wildcard imports\_ (e.g., `import static org.junit.jupiter.api.Assertions.\*;`)

in Java code.

#### Documentation

Text in `\*.adoc` and `\*.md` files should be wrapped at 90 characters whenever technically

possible.

In multi-line bullet point entries, subsequent lines should be indented.

### Javadoc

- Javadoc comments should be wrapped after 80 characters whenever possible.

- This first paragraph must be a single, concise sentence that ends with a period (".").

- Place `<p>` on the same line as the first line in a new paragraph and precede `<p>` with a blank line.

- Insert a blank line before at-clauses/tags.

- Favor `{@code foo}` over `<code>foo</code>`.

- Favor literals (e.g., `{@literal @}`) over HTML entities.

- New classes and methods should have `@since ...` annotation.

- Use `@since 5.0` instead of `@since 5.0.0`.

- Do not use `@author` tags. Instead, contributors are listed on [GitHub](https://github.com/junit-team/junit5/graphs/contributors).

- Do not use verbs in third person form (e.g. use "Discover tests..." instead of "Discovers tests...")

in the first sentence describing a method.

#### Examples

See [`ExtensionContext`](junit-jupiter-api/src/main/java/org/junit/jupiter/api/extension/ExtensionContext.java) and

[`ParameterContext`](junit-jupiter-api/src/main/java/org/junit/jupiter/api/extension/ParameterContext.java) for example Javadoc.

### Tests

#### Naming

- All test classes must end with a `Tests` suffix.

- Example test classes that should not be picked up by the build must end with a `TestCase` suffix.

#### Assertions

- Use `org.junit.jupiter.api.Assertions` wherever possible.

- Use AssertJ when richer assertions are needed.

- Do not use `org.junit.Assert` or `junit.framework.Assert`.

#### Mocking

- Use either [Mockito](https://github.com/mockito/mockito) or hand-written test doubles.

### Logging

- In general, logging should be used sparingly.

- All logging must be performed via the internal `Logger` façade provided via the JUnit [LoggerFactory](https://junit.org/junit5/docs/current/api/org/junit/platform/commons/logging/LoggerFactory.html).

- Levels defined in JUnit's [Logger](https://junit.org/junit5/docs/current/api/org/junit/platform/commons/logging/Logger.html) façade.

- \_error\_ (JUL: `SEVERE`, Log4J: `ERROR`): extra information (in addition to an Exception) about errors that will halt execution

- \_warn\_ (JUL: `WARNING`, Log4J: `WARN`): potential usage or configuration errors that should not halt execution

- \_info\_ (JUL: `INFO`, Log4J: `INFO`): information the users might want to know but not by default

- \_config\_ (JUL: `CONFIG`, Log4J: `CONFIG`): information related to configuration of the system (Example: `ServiceLoaderTestEngineRegistry` logs IDs of discovered engines)

- \_debug\_ (JUL: `FINE`, Log4J: `DEBUG`)

- \_trace\_ (JUL: `FINER`, Log4J: `TRACE`)

### Deprecation

Publicly available interfaces, classes and methods have a defined lifecycle

which is described in detail in the [User Guide](https://junit.org/junit5/docs/current/user-guide/#api-evolution).

This process is using the `@API` annotation from [API Guardian](https://github.com/apiguardian-team/apiguardian).

It also describes the deprecation process followed for API items.

To deprecate an item:

- Update the `@API.status` to `DEPRECATED`.

- Update `@API.since`. Please note `since` describes the version when the

status was changed and not the introduction of the element.

- Add the `@Deprecated` Java annotation on the item.

- Add the `@deprecated` JavaDoc tag to describe the deprecation, and refer to

an eventual replacement.

- If the item is used in existing code, add `@SuppressWarnings("deprecation")`

to make the build pass.