# Contributing to Buck

We encourage the reporting of issues and bugs, along with pull requests to help make the Buck codebase better. The following are some information and guidelines to help you contribute to Buck.

## Tour of the Codebase

This is a high-level overview of how the Buck repository is organized.

### `docs/`

Buck's public documentation. The docs on https://github.com/facebook/buck are generated from the files in this directory.

### `config/`

This is where configuration files live (currently, this includes only logging configurations).

### `src/`

This is where Buck's Java source code lives.

### `test/`

This is where tests, both unit and integration tests, should go. If an integration test needs to create a sample project to run on, follow the example of creating a testdata directory in the folder where the integration test code lives and putting each sample project in its own subdirectory of testdata. Use com.facebook.buck.testutil.integration.ProjectWorkspace to drive your integration test.

### `testdata/`

This directory is deprecated. See the description of the test/ directory about how to organize test data for new integration tests.

### `third-party/`

This is where third-party dependencies live. These dependencies are organized by programming language.

### `webserver/`

The static content served by Buck's webserver. (See .buckconfig for details.)

## Development Workflow

### Building Buck

As explained in [Downloading and Installing Buck](https://buck.build/setup/install.html), you can build Buck by running ant in the directory where you checked out Buck from GitHub. If you modify Buck's source code, running ant again should be sufficient to rebuild it. If you are concerned that Buck may have gotten in some sort of bad state, run ant clean && ant to do a clean build.

### Running Tests

Buck's tests use JUnit and are under the test subdirectory. Here are some examples for running them with Buck:

- Running all tests: use `buck test`

- Running all tests under specific directory: use `buck test //test/com/facebook/buck/file/...`

- Test a particular Java class: use `buck test --test-selectors NameOfTest`. Wildcards are also possible, e.g. `buck test --test-selectors 'Cxx.\*Test'`.

- Faster way of running a specific class: use `buck test --test-selectors RemoteFileTest //test/com/facebook/buck/file:file`

- Keeping tests' files: to avoid auto-deleting temporary directories generated for JUnit tests, use `BUCK\_TEST\_KEEP\_TEMPORARY\_PATHS=1 buck (test options...)`

To find out more about testing options refer to an article about [test](https://buck.build/command/test.html) command.

### Using the IntelliJ IDE

Buck contains the project files so that it can be opened in IntelliJ. Developing Buck in IntelliJ works perfectly fine; however, it does not automatically overwrite the existing `.class` files as Eclipse does. (You could likely add some sort of build step to IntelliJ to make this possible, but we have not.) Therefore, if you elect to develop Buck using IntelliJ, you may want to create an alias for buck that runs ant before running Buck to reduce the friction of developing with IntelliJ.

### Code Style

We use [google-java-format](https://github.com/google/google-java-format) to format Java files.

You can either install the IntelliJ plugin or use a pre commit hook to automatically re-format files.

To add a pre commit hook:

- Build `google-java-format` jar following instructions from [github page](https://github.com/google/google-java-format).

- Copy `config/git-hooks/pre-commit.template` to `.git/hooks/pre-commit`

- Make the script executable: `chmod +x .git/hooks/pre-commit`

- Replace `<<PATH\_TO\_JAR>>` in `.git/hooks/pre-commit` with the location of google-java-format jar.

### Warnings

- Code should compile without warnings. To enforce this, we use a combination of `-Xlint` and `-Werror` flags to `javac` in addition to a number of other static checks provided by PMD.

- Warnings should either be fixed or annotated with `@SuppressWarnings("type-of-warning")`, as appropriate.

## Logging

Buck logs debugging information to `buck-out/log/buck-0.log`. (Older logs are rotated to `buck-out/log/buck-1.log` etc.)

To add more logs to your code, import `com.facebook.buck.core.util.log.Logger`, create a static instance, and add your logs:

```java

import com.facebook.buck.core.util.log.Logger;

public class MyClass {

private static final Logger LOG = Logger.get(MyClass.class);

public doStuff(MyData data) {

// Supports String.format() formatters.

LOG.debug("Doing stuff: %s", data);

try {

doSomethingThatThrows();

} catch (IOException e) {

// Logs a stack trace with a message.

LOG.error(e, "Couldn't do stuff!");

throw e;

}

}

}

```

Buck's Logger exposes five log levels:

| `com.facebook.buck.core.util.log.Logger` | `java.util.logging.Level` equivalent|

| ------------------------------ | ------------------------------------|

| `error()` | `SEVERE` |

| `warn()` | `WARNING` |

| `info()` | `INFO` |

| `debug()` | `FINE` |

| `verbose()` | `FINER` |

By default, Buck only logs to disk messages at `debug()` level and higher. Feel free to pepper your code with `verbose()` logs and even check them in—they won't have any impact on performance and won't clutter up the logs.

If you want to change the global logging level, edit `config/logging.properties.st` (in the buck repo), `.bucklogging.properties` (in the root of the repo in which you're running buck), or `.bucklogging.local.properties` (same) and specify a `java.util.logging.Level` equivalent to the level you want to log (see the table above):

```

.level=FINER

```

You can also control the level of individual loggers, identified by package or class name:

```

com.facebook.buck.stuff.MyClass.level=FINER

```

Extensive logging can help you get to the bottom of build issues, especially in circumstances where additional information is hard to get, for example in continuous integration builds.

If you notice the logs are getting too big to retain for long periods of time, or for all the builds, you can use the CompressingFileHandler to compress the logs, by configuring it in `.bucklogging.properties`, for example:

```

# Enable the console logging handler and the file handler to

# write rotating log files under buck-out/log/buck-\*.log in the

# project(s) being used.

handlers=com.facebook.buck.cli.bootstrapper.ConsoleHandler,com.facebook.buck.cli.bootstrapper.CompressingFileHandler

# Log to buck-out/log/buck-\*log.

com.facebook.buck.log.CompressingFileHandler.pattern=buck-out/log/buck-%g.log.gz

# Write to disk all log messages not otherwise filtered by the top-level ".level"

# property.

com.facebook.buck.log.CompressingFileHandler.level=ALL

# Ignore the environment and always write UTF-8 to files.

com.facebook.buck.log.CompressingFileHandler.encoding=UTF-8

# Replace the default fugly multiline log formatter with a custom one.

com.facebook.buck.log.CompressingFileHandler.formatter=com.facebook.buck.cli.bootstrapper.LogFormatter

# Rotate up to this many log files, then start deleting the oldest one.

com.facebook.buck.log.CompressingFileHandler.count=25

```

## Immutable Value Types

Buck makes heavy use of immutable value types (Java objects which only hold final members and contain no business logic). These ease development and testing, as stateful APIs can be rewritten as stateless APIs which accept and return all the value types upon which they need to act.

Traditional Java value types cause two problems:

1. Each value type requires a lot of manually-written boilerplate code, including `equals()`, `toString()`, and `hashCode()`.

1. Value type constructors take large numbers of arguments, causing any change to the value type to necessitate changes in a large number of clients and tests.

To fix both of these issues, we use the `Immutables.org` library automatically generate source code for immutable value types (including implementations for `equals()`, `toString()`, and `hashCode()`) as well as `Builder`s to put together instances of a value type piece by piece.

To use immutable value types in a Buck `java\_library`, use the `java\_immutables\_library` function in your `BUCK` file. For example:

```

java\_immutables\_library(

name = 'value',

srcs = glob(['\*.java'])

deps = [

'//third-party/java/guava:guava',

],

visibility = ['PUBLIC'],

)

```

Then, declare an interface or abstract class named Abstract\* containing the type and data accessors. Make sure to annotate it with `@org.immutables.value.Value.Immutable` and `@com.facebook.buck.util.immutables.BuckStyleImmutable`.

```java

import com.facebook.buck.util.immutables.BuckStyleImmutable;

import org.immutables.value.Value;

import java.util.List;

import java.util.Optional;

@Value.Immutable

@BuckStyleImmutable

interface AbstractType {

String getName();

List<Long> getPhoneNumbers();

Optional<String> getDescription();

}

```

Note that the `AbstractType` type is package private; the generated class will be named `Type` and will be public.

This will generate two classes:

- `Type`: A concrete, public, final implementation of `AbstractType` with private final members.

- `Type.Builder`: A `Builder` which generates instances of `Type`.

For example, to generate an instance of and check its members in a unit test:

```java

Type t = Type.builder()

.setName("Jenny")

.addPhoneNumbers(8675309L)

.build();

assertEquals("Jenny", t.getName());

assertEquals(ImmutableList.of(8675309L), t.getPhoneNumbers());

assertFalse(t.getDescription().isPresent());

```

For more documentation, see the reference at https://immutables.github.io.