# Contributing

## Filing Issues

Whether you find a bug, typo or an API call that could be clarified, please [file an issue](https://github.com/realm/realm-java/issues) on our GitHub repository.

When filing an issue, please provide as much of the following information as possible in order to help us fix it:

1. \*\*Goals\*\*

2. \*\*Expected results\*\*

3. \*\*Actual results\*\*

4. \*\*Steps to reproduce\*\*

5. \*\*Code sample that highlights the issue\*\* (link to full Android Studio projects that we can compile ourselves are ideal)

6. \*\*Version of Realm/Android Studio/OS\*\*

If you'd like to send us sensitive sample code to help troubleshoot your issue, you can email <help@realm.io> directly.

## Contributing Enhancements

We love contributions to Realm! If you'd like to contribute code, documentation, or any other improvements, please [file a Pull Request](https://github.com/realm/realm-java/pulls) on our GitHub repository. Make sure to accept our [CLA](#CLA)!

### CLA

Realm welcomes all contributions! The only requirement we have is that, like many other projects, we need to have a [Contributor License Agreement](https://en.wikipedia.org/wiki/Contributor\_License\_Agreement) (CLA) in place before we can accept any external code. Our own CLA is a modified version of the Apache Software Foundation’s CLA.

[Please submit your CLA electronically using our Google form](https://docs.google.com/forms/d/1bVp-Wp5nmNFz9Nx-ngTmYBVWVdwTyKj4T0WtfVm0Ozs/viewform?fbzx=4154977190905366979) so we can accept your submissions. The GitHub username you file there will need to match that of your Pull Requests. If you have any questions or cannot file the CLA electronically, you can email <help@realm.io>.

## Repository Guidelines

### Code Style

While we havn't described our code style yet, please just follow the existing style you see in the files you change.

For source code written in C++, we format it using `clang-format`. You can use the [plugin](https://plugins.jetbrains.com/plugin/8396-clangformatij): mark the entire file and right-click to execute `clang-format` before committing any changes. Of course, if you don't use Android Studio to edit C++ code, run `clang-format` on the command-line.

### Nullability by Annotataion

To improve code quality and usability in Kotlin, nullability of parameters and return types must be annotated with JSR305 annotations.

If a parameter is nullable, you must add `@Nullable` annotation to the parameter. On the other hand, if a parameter is non-null, you don't need to add `@Nonnull` annotation since all parameters are treated as `@Nonnull` by default.

For return types, there is no default nullability. If a method can return `null` as a return value, you must add `@Nullable` annotation to the return type. Currently, `Nonnull` annotation is not mandatory if the method never return `null`.

When you add a new package, you must add `package-info.java` and add `@javax.annotation.ParametersAreNonnullByDefault` to the package. Please note that you can't add multiple `package-info.java` in the same package but different location (for example, main and androidTest). When you add a package to both main and androidTest, you only need to add `package-info.java` to main.

### Unit Tests

All PR's must be accompanied by related unit tests. All bug fixes must have a unit test proving that the bug is fixed.

You can use `./realm/gradlew connectedCheck createDebugCoverageReport` to generate a coverage report to check for

missing unit test coverage. The aim is 100% code coverage.

When writing unit tests, use the following guide lines:

1) Unit tests must be written using JUnit4.

2) All tests for a class should be grouped in a class called `<className>Tests`, unless the functionality is cross-

cutting like [`RxJavaTests`](https://github.com/realm/realm-java/blob/master/realm/realm-library/src/androidTest/java/io/realm/RxJavaTests.java)

or [`RealmAsyncQueryTests`](https://github.com/realm/realm-java/blob/master/realm/realm-library/src/androidTest/java/io/realm/RealmAsyncQueryTests.java).

3) Test methods should use camelCase and underscore `\_` between logical sections to increase method name readability.

Methods should ideally start with the name of the method being tested. Patterns like: `<methodName>\_<description>`,

`<methodName>\_<param>\_<description>` or `<description>` are encouraged.

4) All unit tests creating Realms must do so using the [`TestRealmConfigurationFactory`](https://github.com/realm/realm-java/blob/master/realm/realm-library/src/androidTest/java/io/realm/rule/TestRealmConfigurationFactory.java)

or [`RunInLooperThread`](https://github.com/realm/realm-java/blob/master/realm/realm-library/src/androidTest/java/io/realm/rule/RunInLooperThread.java)

test rules. This ensures that all Realms are properly closed and deleted between each test.

5) Use the `@RunInLooperThread` rule for any test that depends on Realms notification system.

6) Input-parameters should be boundary tested. Especially `Null/NotNull`, but also the state of Realm objects like

unmanaged objects, deleted objects, objects from other threads.

7) Unit tests are not required to only have 1 test. It is acceptable to combine multiple tests into one unit test, but

if it fails, it should be clear why it failed. E.g. you can group related tests with the same setup like negative

tests. If you do so, make sure to separate each "subtest" with a comment stating what you test.

8) Use only `@Test(expected = xxx.class)` if the test case contains one line. If the test contains multiple

lines and it is the last line that is tested, use the `ExceptedException` rule instead. In all other cases, use

the following pattern:

try {

somethingThatThrowsIllegalArgument();

} catch (IllegalArgumentException ignored) {

}

9) Use comments to make the intent of the unit test easily understandable at a glance. A simple one line comment is

often easier to read `thanALongCamelCasedSentenceThatAttemptsToDescribeWhatHappens`. Describe the test steps inside

the method, if it's not glaringly obvious.

This is an example of how a unit test class could look like:

@RunWith(AndroidJUnit4.class)

public class RealmTests {

@Rule

public final TestRealmConfigurationFactory configFactory = new TestRealmConfigurationFactory();

@Rule

public final RunInLooperThread looperThread = new RunInLooperThread();

private Realm realm;

@Before

public void setUp() {

RealmConfiguration config = configFactory.createConfiguration();

realm = Realm.getInstance(config);

}

@After

public void tearDown() {

if (realm != null) {

realm.close();

}

}

@Test(expected = IllegalStateException.class)

public void createObject\_outsideTransaction() {

realm.createObject(Foo.class);

}

@Test

public void createObject\_illegalInput {

// Class not part of the schema

try {

realm.createObject(Foo.class);

} catch (IllegalArgumentException ignored) {

}

// Null class

try {

realm.createObject(null);

} catch (IllegalArgumentException ignored) {

}

}

@Test

@RunTestInLooperThread

public void addChangeListener\_notifiedOnLocalCommit() {

realm.addChangeListener(new RealmChangeListener() {

@Override

public void onChange() {

assert(1, realm.allObjects(Foo.class).size());

looperThread.testComplete();

}

});

realm.beginTransaction();

realm.createObject(Foo.class);

realm.commitTransaction();

}

}

### Javadoc

All public classes and methods must have Javadoc describing their purpose.

```java

/\*\*

\* Checks if given field is equal to the provided value.

\*

\* <pre>

\* {@code

\* // A multi-line code sample should be formatted like this.

\* // Please wrap the code element in a <pre> tag.

\* }

\* </pre>

\*

\* @param fieldName the field to compare.

\* @param fieldValue the value to compare with.

\* @param caseSensitive if {@code true}, substring matching is case sensitive. Setting this to {@code false} works for English locale characters only.

\* @param caseSensitive if true, substring matching is case sensitive. Setting this to false only works for English

\* locale characters.

\* @return the query object.

\* @throws java.lang.IllegalArgumentException if one or more arguments do not match class or field type.

\* @throws IllegalArgumentException if field name doesn't exists, it doesn't contain a list of links or the type

\* of the object represented by the DynamicRealmObject doesn't match.

\* @deprecated Please use {@link #average(String)} instead.

\* @see #endGroup()

\*/

public RealmQuery<E> equalTo(String fieldName, String fieldValue, boolean caseSensitive) {

// ...

}

```

\* Method descriptions begin with a verb phrase, e.g. "Checks" instead of "Check".

\* Capitalize the first letter of the method and @deprecated descriptions. Everything else starts with lower case.

\* Empty line between method description and the rest.

\* End all descriptions with a period `.` (except @see).

\* Reference other Realm classes using `{@link ...}`.

\* Wrap Java values in `{@code ...}`.

\* @throws description must start with "if".

\* Never list generic exceptions like `RuntimeException`, `Exception` or `Error`. Always reference the specific error.

\* Line-length maximum is 120 chars. Parameter descriptions that go above this, should be split into multiple lines and indented. Otherwise do not use indentation (contrary to Oracle guidelines).

Above is based on the official guidelines from Oracle regarding Javadoc: http://www.oracle.com/technetwork/articles/java/index-137868.html

### Branch Strategy

We have two branches for shared development: `master` and `releases`. We make releases from each.

`master`:

\* The `master` branch is where major/minor versions are released from.

\* It is for new features and/or breaking changes.

`releases`:

\* The releases branch is where patch versions are released from.

\* It is mainly for bug fixes.

\* Every commit is automatically merged to `master`.

\* Minor changes (e.g. to documentation, tests, and the build system) may not affect end users but should still be merged to `releases` to avoid diverging too far from `master` and to reduce the likelihood of merge conflicts.