# How to Contribute

[Fork](https://help.github.com/articles/fork-a-repo) Vavr, send a [pull request](https://help.github.com/articles/using-pull-requests) and keep your fork in [sync](https://help.github.com/articles/syncing-a-fork/) with the upstream repository.

Vavr has no dependencies other than Java.

## Prerequisites

\* [JDK 1.8.0\_40+](https://openjdk.java.net/install/)

\* [IntelliJ IDEA](https://www.jetbrains.com/idea/download/) with default settings

## Building

\* Executing tests: `./gradlew check` (test reports: [./build/reports/tests/test/index.html](./build/reports/tests/test/index.html), code coverage reports: [./build/reports/jacoco/test/html/index.html](./build/reports/jacoco/test/html/index.html))

\* Executing doclint: `./gradlew javadoc`

\* Creating jars: `./gradlew assemble` (see [./build/libs](./build/libs))

## Coding Conventions

We follow \_Rob Pike's 5 Rules of Programming\_:

> \* \*\*Rule 1. You can't tell where a program is going to spend its time.\*\* Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

> \* \*\*Rule 2. Measure.\*\* Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

> \* \*\*Rule 3. Fancy algorithms are slow when n is small, and n is usually small.\*\* Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.)

> \* \*\*Rule 4. Fancy algorithms are buggier than simple ones, and they're much harder to implement.\*\* Use simple algorithms as well as simple data structures.

> \* \*\*Rule 5. Data dominates.\*\* If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

>

> Pike's rules 1 and 2 restate Tony Hoare's famous maxim "Premature optimization is the root of all evil." Ken Thompson rephrased Pike's rules 3 and 4 as "When in doubt, use brute force.". Rules 3 and 4 are instances of the design philosophy KISS. Rule 5 was previously stated by Fred Brooks in The Mythical Man-Month. Rule 5 is often shortened to "write stupid code that uses smart objects".

\_Source: http://users.ece.utexas.edu/~adnan/pike.html\_

### Javadoc

\* Public API needs javadoc, e.g. public classes and public methods.

\* Non-trivial private methods need javadoc, too.

\* Design decisions are worth a comment.

\* A package, which is part of the public API, contains a `package-info.java`.

\* Unit tests contain no javadoc at all (because they introduce no new API and contain no business logic).

\* Running `./gradlew javadoc` results in no javadoc errors.

\* All classes start with the following copyright notice in order to apply the Apache-2.0 license:

```java

/\* \_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

\* \ \/ / \ \/ / \_\_/ / \ \/ / \

\* \\_\_\_\_\_\_/\_\_\_/\\_\_\_\\_\_\_\_\_\_/\_\_\_/\_\_\_\_\_/\_\_\_/\\_\_\_\\_\_\_\_\_\_/\_\_\_/\\_\_\_\

\*

\* Copyright 2020 Vavr, http://vavr.io

\*

\* Licensed under the Apache License, Version 2.0 (the "License");

\* you may not use this file except in compliance with the License.

\* You may obtain a copy of the License at

\*

\* http://www.apache.org/licenses/LICENSE-2.0

\*

\* Unless required by applicable law or agreed to in writing, software

\* distributed under the License is distributed on an "AS IS" BASIS,

\* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

\* See the License for the specific language governing permissions and

\* limitations under the License.

\*/

```

### Packages

\* There is only one first-level package: `io.vavr`.

\* The maximum package depth is two.

\* Package names are denoted in the singular.

\* Packages are sliced by domain (no util or tool packages).

\* Package private classes are used in order to hide non-public API.

\* Inner classes are preferred over package private classes in case of one-to-one dependencies.

### File structure

We organize our classes and interfaces in the following way:

\* The Javadoc of the type contains an overview of the new (i.e. not overridden) API declared in the actual type.

\* The type consists of three sections:

1. static API

2. non-static API

3. adjusted return types

\* The methods of each of these sections are alphabetically ordered.

```java

/\*\*

\* Description of this class.

\*

\* <ul>

\* <li>{@link #containsKey(Object)}}</li>

\* <li>{@link ...}</li>

\* </ul>

\*/

public interface Map<K, V> extends Traversable<Tuple2<K, V>> {

// -- static API

static <K, V> Tuple2<K, V> entry(K key, V value) { ... }

...

// -- non-static API

@Override

default boolean contains(Tuple2<K, V> element) { ... }

boolean containsKey(K key);

...

// -- Adjusted return types

@Override

Map<K, V> distinct();

...

}

```

\* We do not include `@author` javadoc tags because they are redundant. Look up the git file history instead, e.g. on GitHub.

### Unit tests

\* Public API is tested.

\* High-level functionality is tested in first place.

\* Corner cases are tested.

\* Trivial methods are not \_directly\_ tested, e.g. getters, setters.

\* The test method name documents the test, i.e. 'shouldFooWhenBarGivenBaz'

\* In most cases it makes sense to run one assertion per @Test.