Effective and eventual immutability

Table of Contents

Main website: https://www.e2immu.org.

1. Introduction

This document is work in progress. In some places, it is nothing but a skeleton. As of October 2021, it is two versions behind the *Road to immutability*.

2. Installing e2immu

2.1. Obtaining the analyser

For now, the analyser's binaries have not been uploaded to a central jar repository (like MavenCentral) yet. Please clone the following projects from GitHub:

- e2immu-support, a small jar containing the annotations and some support classes
- e2immu, the analyser

The support jar has been compiled with the Java 10+ API; this can easily be stripped down to Java 8 if required. The analyser makes extensive use of the Java 16 API *and* language features.

The IntelliJ plugin is still in its infancy. It is not yet available in IntelliJ's plugin repository. To experiment with it, clone

- e2immu-annotation-store, the annotation store
- e2immu-intellij-plugin, the IntelliJ plugin

2.2. The support jar

The annotations and support classes can be compiled with any JDK providing the Java 10+ API. Execute:

```
./gradlew publishToMavenLocal
```

to make the jar, with reference `org.e2immu:e2immu-support:0.2.0`, locally available.

2.3. The analyser

Please ensure you have at least a Java 16 JDK. Gradle 7.0 (lower versions do not play nice with Java 16) is provided via the Gradle wrapper.

To make the jars available, publish them to your local Maven repository:

```
./gradlew publishToMavenLocal
```

This pushes the following jars to your local Maven repository:

- `org.e2immu:analyser:0.1.2`, the analyser
- `org.e2immu:analyser-cli:0.1.2`, a small library extending the analyser with a command line
- `org.e2immu:analyser-store-uploader:0.1.2`, a small library to upload annotations to the annotation store
- `org.e2immu:gradle-plugin:0.1.2`, the Gradle plugin which you'll need to run the analyser in a practical setting

Of course, version numbers may have changed when you read this.

2.4. The annotation store

The annotation store is a separate project, consisting of two Java classes. Build it with:

```
./gradlew build
```

and start the annotation store with

```
./gradlew run
```

The `build.gradle` file provides support to change the port, in case 8281 is already occupied. For example,

```
./gradlew run -De2immu-port=9999
```

will start the annotation store listening to port 9999. Please make sure this change of port is reflected in the configuration of the IDE plugin, which also needs to connect to the annotation store.

2.5. The IntelliJ plugin

The plugin is compiled with Java 8 language features only.

⚠ At this point (April 2021), the analyser still crashes on the sources of the plugin!

3. Using e2immu

3.1. Basic use

Starting your first project, basic use of the *e2immu* analyser looks like

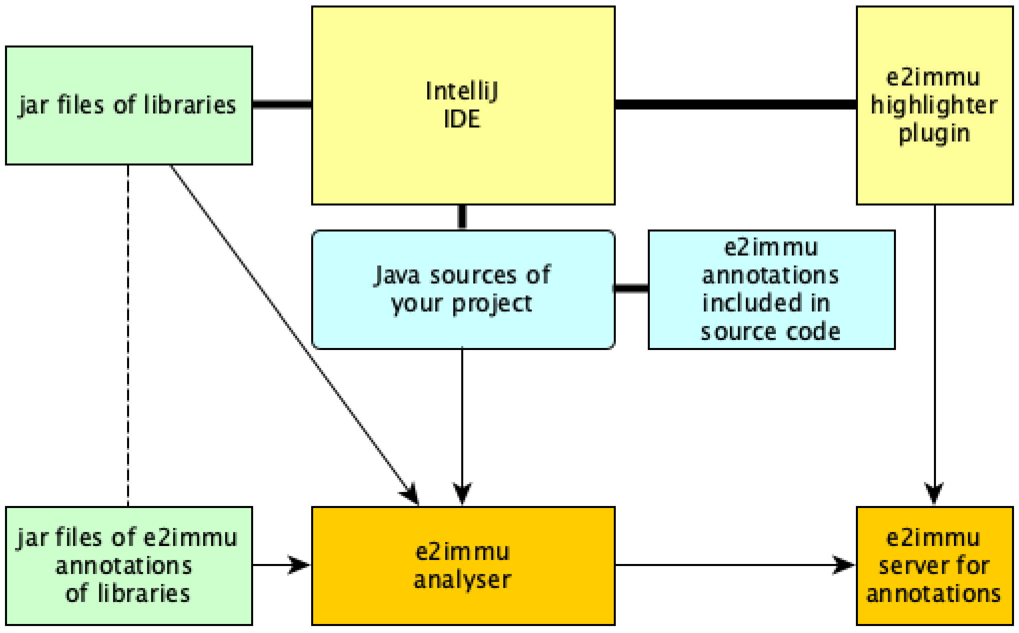
jar files of libraries
IntelliJ IDE
e2immu highlighter plugin
Java sources of your project
e2immu annotations included in source code
jar files of e2immu annotations of libraries
e2immu analyser
e2immu server for annotations

Figure 1. Basic use of e2immu

Having installed *e2immu* 's IntelliJ IDEA plugin, and having started a local annotation server, you can edit your project, occasionally run the analyser, and make use of pre-annotated libraries.

The analyser produces computed annotations, errors and warnings, which you can of course read from the command line. It also pushes these errors, warnings, and computed annotations to the annotation server, which is continuously consulted by the highlighter plugin. So while you're working on your project, each time you run the analyser, your editor is updated. We're suggesting that you confirm critical annotations in the source code. They will get the annotation type `VERIFY` which is the default for source code read by the analyser, they allow you to 'stabilize' your code with respect to class types like `@Container` or {e2immutable}.

This set-up will get you pretty far, as long as

- your project consists of a single set of source files
- all the libraries you are using have been pre-annotated.

3.2. Full flow

The *e2immu* analyser is set up to read, in order of decreasing priority,

1. the source code of your project
2. annotated API sources, as a replacement for class files with XML annotations
3. class files and associated XML annotation files from jars, for libraries used in your project

If your project is or becomes a library for other projects to use, the computed annotations have to be made available to the users of the library:

1. for fast turn-around development on your own or in small teams, you can use the analyser run on the library. This is depicted in Fast turnaround use of e2immu. to upload computed annotations to the annotation store, and instruct the analyser on the project consuming the library to consult this store. For this purpose, the annotation store has the ability to store annotations in user-defined *projects*; the analyser can read from any such *projects*.
2. the standard procedure is for the computed annotations to be included in the `jar` file of the project: the analyser can directly write `annotations.xml` files in the resources of your project, one for each package. This action can take place after the compilation phase and before the packaging phase in you build tool. *e2immu* provides a plugin for Gradle for now. This flow is depicted in Add annotation.xml files to your jar.

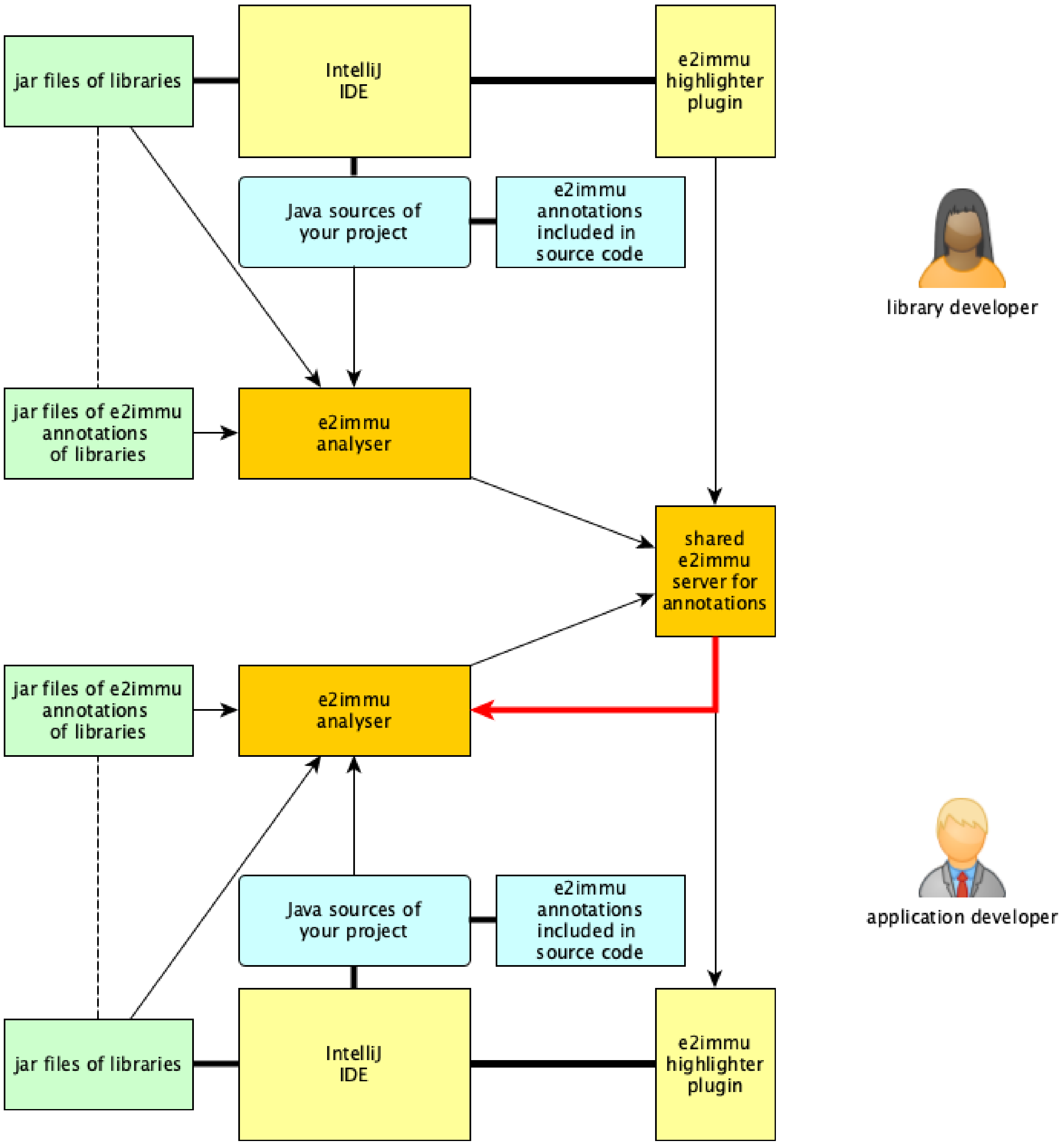
jar files of libraries
IntelliJ IDE
e2immu highlighter plugin
Java sources of your project
e2immu annotations included in source code
library developer
jar files of e2immu annotations of libraries
e2immu analyser
shared e2immu server for annotations
jar files of e2immu annotations of libraries
e2immu analyser
application developer
Java sources of your project
e2immu annotations included in source code
jar files of libraries
IntelliJ IDE
e2immu highlighter plugin

Figure 2. Fast turnaround use of e2immu

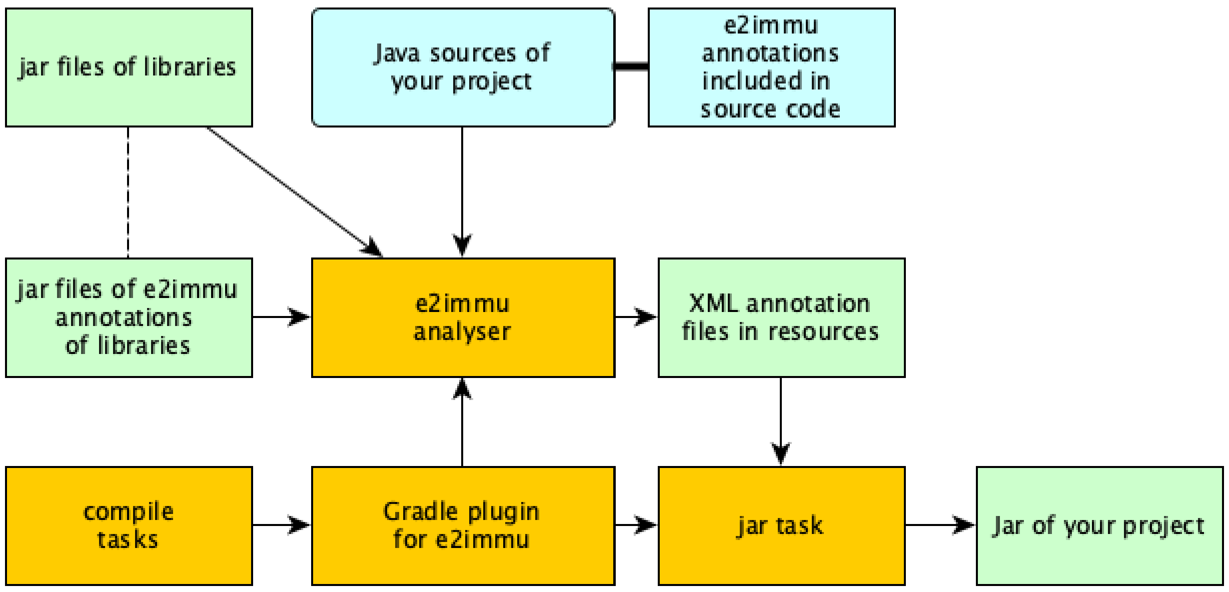
jar files of libraries
Java sources of your project
e2immu annotations included in source code
jar files of e2immu annotations of libraries
e2immu analyser
XML annotation files in resources
compile tasks
Gradle plugin for e2immu
jar task
Jar of your project

Figure 3. Add annotation.xml files to your jar

If you are annotating external libraries with *e2immu* annotations, there are two options

1. you can use the *external annotations* feature of IntelliJ IDEA to create annotations files. These files are best grouped into a new jar file, on per library, which is to be included in the dependencies of your project.
2. you can use annotated API sources, a kind of Java source file which contains all the declarative aspects of the types you'll be using. These files are quick to create, provide a nice overview, and can be used in combination with the underlying JAR so that you only have to copy those declarations that you want to annotate. Their main advantage is clarity: all types, fields, and methods *relevant to you* are close together, with their annotations

Annotated API sources can be generated by the analyser from jars and XML annotations, presenting only those types, methods and fields that your project is using.

Updated annotation files can be generated by the analyser from the combination of annotated API sources and existing annotation files.

3.3. The analyser's command line

The input to the analyser is largely controlled by the following primary locations

- `--source=<dir>`: the directories where `.java` sources are to be found. They can be `:` or `,` separated; the argument can also be repeated. When nothing is specified, the analyser assumes `src/main/java`.
- `--classpath=<cp>`: the classpath. This classpath should include the `.class` files corresponding to the `.java` files presented to the analyser. The format is as parsed by the JDK classpath: colon separated, with wildcards for multipe jar files in the same directory, containing jar files, `.class` files, or directories. Multiple `--classpath` options may be present; all are concatenated. When nothing is specified, `build/classes/java/main:build/resources/main` is assumed.
- `--jre=<dir>`: location of the JRE if a different one from the analyser's is to be taken

- `--restrict-source=<packages>`: restrict the input to the following packages. The parameter can be comma separated, with wildcards as detailed in the note below.

> The Maven or Gradle plugin typically takes care of correct values for source input and classpath.

Then there are typical options like

- `--quiet` (short `-q`): do not write warnings, errors, etc to the standard output. They are still uploaded when the `--upload` option is activated.
- `--debug=<logtargets>`: log targets to activate for debug output. Their names can be found in the class `org.e2immu.analyser.util.Logger.LogTarget`.
- `--ignore-errors`: do not end the analyser in an error state when errors have been raised by the analyser.

The following options are available to control the output to the annotation server:

- `--upload` (short: `-u`): upload annotations to an annotation server
- `--upload-url=<url>`, change the default URL which is `http://localhost:8281`
- `--upload-project=<project>`, change the default project which is `default`
- `--upload-packages=<packages>`: a comma-separated list of package names for which annotations are to be uploaded. The default is to upload all annotations of all types encountered during the parsing process.

The following options are available to control the output of Annotation XML files written:

- `--write-annotation-xml` (short: `-w`): create annotation files to be included in the resources, and hence the jar of the project.
- `--write-annotation-xml-packages=<packages>`: a comma-separated list of package names for which annotation.xml files are to be written. The default is to write them for all the packages of `.java` files parsed
- `--write-annotation-xml-dir=<directory>`: alternative location to write the Xml files. The value defaults to the resources directory of the project.
- `--write-annotated-api` (short: `-a`)
- `--write-annotated-api-packages=<packages>`: a comma-separated list of package names for which annotated API files are to be written. The default is to write them for all the packages of `.java` files parsed.
- `--write-annotated-api-dir=<directory>`: alternative location to write the annotated API files. The default is the main directory of the project

> When describing packages, a dot at the end of a package name may be used to indicate the inclusion of all sub-packages. The wildcard `java.` includes `java.lang`, `java.io`, etc.

3.4. The Gradle plugin

The easiest way to use the analyser is via the Gradle plugin.

Example of `build.gradle` *file*

```
plugins {
    id 'java'
    id 'org.e2immu.analyser'
}

...

repositories {
    ...
}

dependencies {
   ...
}

e2immu {
    skipProject = false
    sourcePackages = 'org.e2immu.'
    jmods = 'java.base.jmod,java.se.jmod'
    jre = '/Library/Java/JavaVirtualMachines/openjdk-11.0.2.jdk/Contents/Home/'
    writeAnnotatedAPIPackages = 'org.e2immu.'
    writeAnnotationXMLPackages = 'org.e2immu.'
}
```

The list of properties configurable differs slightly from the one of the command line. Gradle takes care of source and class path.

4. Annotated APIs

4.1. Purpose

The modification status of a class often depends on *foreign* method calls: calls on objects defined outside your own code base. In some situations, their source code is available, but that need not be the case. So to build up a fair picture, one could say that the *e2immu* analyser would have to parse all types and methods "from the ground up", and decide on their modification status in an incremental way. This procedure would be slow and hugely impractical.

On top of the incremental problem, APIs often come in the form of an interface without implementation. Expressing the modification status by hand (the terminology we use is *contracting*, or writing the contract) is the only way forward.

Thirdly, manually annotating APIs can help you *override* the implied modification status, or any other feature supported by the analyser. Here are two simple examples why that can come in handy:

```
interface List<E> extends Collection<E> {
    ...
    @Modified
    boolean add(@NotNull E e);
}
```

Here, we acknowledge that `add` modifies the list, but we prohibit passing `null` as an element. Secondly, we may decide to ignore modifications to the output stream `System.out`, by writing

```
class System {

    @IgnoreModifications
    @NotNull
    static final PrintStream out;

    @NotNull
    @IgnoreModifications
    static final PrintStream err;

    ...
}
```

so that we can still claim that the following method is not modifying:

```
@NotModified
static int square(int x) {
    int result = x*x;
    System.out.println("The square of "+x+" is "+result);
    return result;
}
```

It should be clear from these examples, and from the terminology used throughout the rest of the documentation, that we express the modification status, and other aspects of the code, by means of Java annotations.

In this section, we describe a practical way of annotating foreign APIs; we call it simply *Annotated API* files. The next section deals with a more compact form, *annotation XML* files, which are less readable but (maybe) simpler and faster to load.

4.2. Preparing Annotated API files

Should we then start to systematically annotate all the libraries that our project uses? That is one way of approaching the problem; however, because we have a source code analyser at hand, we can easily detect exactly which types, methods, and fields are used by our code. The analyser's command line interpreter (CLI) provides options for generating a template Annotated API file for a whole library, for selected packages, or for exactly those types, methods and fields required.

The following Gradle task, taken from the `e2immu/annotation-store` project, contains the code to produce a single `IoVertxCore.java` file:

Part of build-api.gradle in e2immu/annotation-store

```
plugins {
    id 'java'
}

task runIoVertxAAPI(type: JavaExec) {
    group = "Execution"
    description = "Prepare an AnnotatedAPI file for io.vertx.core"

    classpath = sourceSets.main.runtimeClasspath
    main = 'org.e2immu.analyser.cli.Main'

    Set<File> reducedClassPath = sourceSets.main.runtimeClasspath.toList()
    reducedClassPath += sourceSets.test.runtimeClasspath
    reducedClassPath.removeIf({ f -> f.path.contains("build/classes")
        || f.path.contains("build/resources") })

    args('--classpath=' + reducedClassPath.join(":") + ":jmods/java.base.jmod",
            '-a',
            "--write-annotated-api-packages=io.vertx.core",
            "--source=none",
            "-d=CONFIGURATION,BYTECODE_INSPECTOR")
}
```

The task can be run with the command `./gradlew -b build-api.gradle runIoVertxAAPI`.

4.3. The Annotated API file format

Annotated API files are standard Java files, they will be inspected by a standard Java parser, so all normal syntax rules need to be followed. They deviate in the following way:

- The primary types become sub-types of a primary type named after the package.
- To ensure that there is no clash with preloaded primary types, they have a dollar `$` suffix.
- A string constant, `PACKAGE_NAME`, specifies the package to which 'dollar types' are transferred.
- All types become classes, all methods return a default value. Actually, none of the decorations to a type, method, or field matter, as long as the analyser can identify the structure uniquely.

This excerpt from the annotated API file for `java.util` used by the tests in the analyser, shows what this looks like:

Start of the JavaUtil.java annotated API file

```
public class JavaUtil {
    public static final String PACKAGE_NAME = "java.util";

    static class Enumeration$ {
        boolean hasMoreElements() { return false; }
        E nextElement() { return null; }
        Iterator<E> asIterator() { return null; }
    }

    static class Map$ {
        static class Entry {
            K getKey() { return null; }
            V getValue() { return null; }
    ...
```

Once all relevant types, methods and fields can be written, they can be annotated, as in:

A second part of the JavaUtil.java annotated API file

```
public class JavaUtil {
    public static final String PACKAGE_NAME = "java.util";

    ...

    @Container
    // this is not in line with the JDK, but we will block null keys!
    static class Collection$<E>  {

        boolean add$Postcondition(E e) { return contains(e); }
        @Modified
        boolean add(@Dependent1 @NotNull E e) { return true; }

        @Independent
        boolean addAll(@Dependent1 @NotNull1 java.util.Collection<? extends E>
collection) {
            return true;
        }

        static boolean clear$Clear$Size(int i) { return i == 0; }
        @Modified
        void clear() { }

        static boolean contains$Value$Size(int i, Object o, boolean retVal) {
            return i != 0 && retVal;
        }
        @NotModified
        boolean contains(@NotNull Object object) { return true; }

        ...
    }
}
```

Also on display here are Companion methods, static methods describing either how the state of a `Collection` instance changes after a modifying method call, or certain edge cases can be resolved using this state information.

4.4. Annotation types

All *e2immu* annotations have a parameter of the enum type `AnnotationType`, which takes 4 different values:

VERIFY

this is the default value inserted when parsing Java code. This corresponds to the standard use of *e2immu* annotations: normally the analyser will compute them for you, but you may want to assert their presence.

VERIFY_ABSENT

mostly for debugging: insert in the Java code by hand to make sure the analyser does not end up computing this assertion for you.

COMPUTED

added to annotations inserted by the analyser

CONTRACT

added to annotations inserted when parsing annotation XMLs or annotated APIs. This type indicates that a value has not been computed, but stipulated by the user.

The list of available annotations can be found here. In Annotated API files, CONTRACT is the default type, and needs not be specified.

4.5. The Collections framework

4.5.1. Not null

We strongly object to the use of `null` in sets and maps — note that this can be valid point of view even if one embraces `null` as a valuable concept denoting *absence of a value*. It is our opinion that one should not store *absence of a value* in a set, nor should one use *absence of a value* as the key in a map. The same does not hold for arrays, and should therefore not be so important for lists. But then, consistency prevails. We are happy with `null` in arrays, but not in standard collections.

Consequently, our `java.util` annotated APIs are littered with `@NotNull` and {nn1} annotations. Note that we have not equipped *e2immu* with a `@NotNull` annotation on types. When a collection comes in as a parameter, and the collection should be present, `@NotNull` is the obvious choice:

```
public int combinedSize(@NotNull Set<T> set) {
    return someValue + set.size();
}
```

The implementation of {nn1} is rather patchy at the moment, focusing on arrays, functional interfaces, and iteration over collections, as in:

```
public void method(@NotNull1 Set<T> set) {
    for(T t: set) {
        if("x".equals(t.toString)) { // forces t to be @NotNull, set to be @NotNull1
            ...
        }
    }
}
```

4.5.2. Iterating over maps

Interestingly, `Map.Entry` has a `setValue()` method which allows the developer to change the value of a mapping during iteration. As a consequence, we annotate the mutable type as

```
@Container
@Independent1
interface Entry<K, V> {
    @NotNull
    K getKey();

    @NotNull
    V getValue();

    @Modified
    V setValue(V v);
}
```

The method `Map.entrySet()` returns a set of entries, which is a *view* on the map, indicating that the set is backed by the map, and changes to the set imply changes to the map. We must therefore annotate

```
@Container // and implicitly @Dependent
interface Map$<K, V> {
   ...

   // implicitly @Dependent
   @NotNull1
   Set<Map.Entry<K, V>> entrySet();
}
```

To use `entrySet()` in a for-each construct, we must annotate the interface `Iterable`, which is extended by `Collection` and therefore also by `Set`:

```
@Container // and implicitly @Dependent
interface Iterable$<T> {
    @NotModified
    void forEach(@NotNull @Independent1 Consumer<? super T> action);

    // implicitly @Dependent, `Iterator` has `remove()`
    @NotNull
    Iterator<T> iterator();

    @NotNull
    @Independent1
    Spliterator<T> spliterator();
}
```

Finally, we note that `Iterator` has a `remove()` operation, which makes every iterator dependent on its iterable source:

```
@Container
@Independent1
interface Iterator$<T> {
    @Modified
    default void forEachRemaining(@NotNull @Independent1 Consumer<? super T> action) {
    }

    @Modified
    boolean hasNext();

    @Modified // implicitly @Independent1
    T next();

    @Modified
    void remove();
}
```

With this background, we can analyse the dependencies between `entry` and `map` in the common construct

```
for(Entry<K, V> entry: map.entrySet()) {
    ...
}
```

which is equivalent to the more elaborate construct

```
Iterator<Entry<K, V>> iterator = map.entrySet().iterator();
while(iterator.hasNext()) {
    Entry<K, V> entry = iterator.next();
    K key = entry.getKey();
    ...
}
```

Both methods `entrySet()` and `iterator()` are {dependent}, so we start off with local variable `iterator` linked to `map`. Modifications to the iterator (e.g., by calling the `remove()` method), will modify the map.

The `entry` is obtained by calling the method `next()` on the iterator, which is marked {independent1}. This means that the method result is linked to the hidden content of the iterator, which consists of `Entry` objects. These `Entry` objects, however, are not part of the hidden content of `Map`: only objects of types `K` and `V` have that property. Because the `Entry` objects have been obtained from `Map` in a {dependent} way, and `Map` is mutable, we equate changes to an `Entry` object to changes to the map.

In a picture, the situation looks like:

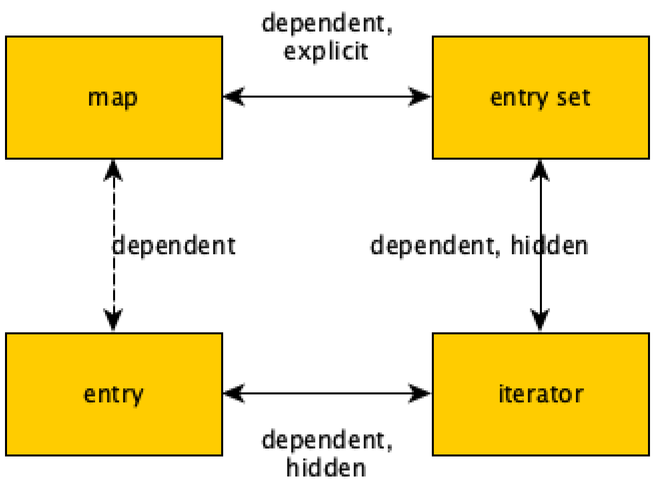

Figure 4. Current situation, entry to map

> The links are: {independent1} from `entry` to `iterator`, {dependent} from `iterator` to the entry set, and {dependent} from the entry set to the `map`. Because `entry` is hidden in `iterator`, and `iterator` is dependent on map `map`, and `entry` is not of a hidden type in `map`, and it is mutable, we link `entry` to `map` in a {dependent} way.

Suppose we tell the analyser that the user can never use the `setValue` method of `Entry`. Then, `Entry` becomes {e2container}. Because it is now immutable, `entry` can only be content linked to `map`, i.e., at the {independent1} level. This is depicted by:

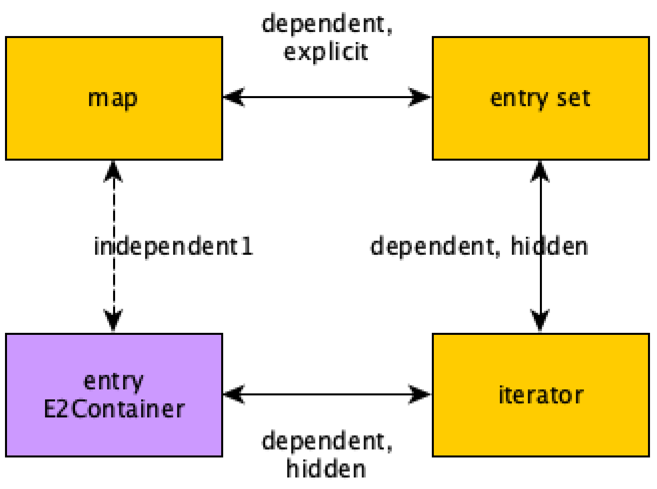

Figure 5. No setValue() method, entry to map

Alternatively, we could forbid the modification of entry sets, e.g., by contracting `entrySet()` to return an {e2container} object. (Calling a modifying method on such an object will cause *e2immu* to raise an error.) Then, `iterator`, still linked to the entry set, remains linked to the mutable `map`: while the *set* is immutable, the non-hidden `Entry` objects are not. As a consequence, `Entry` objects remain linked to the map, because the previous rule still applies.

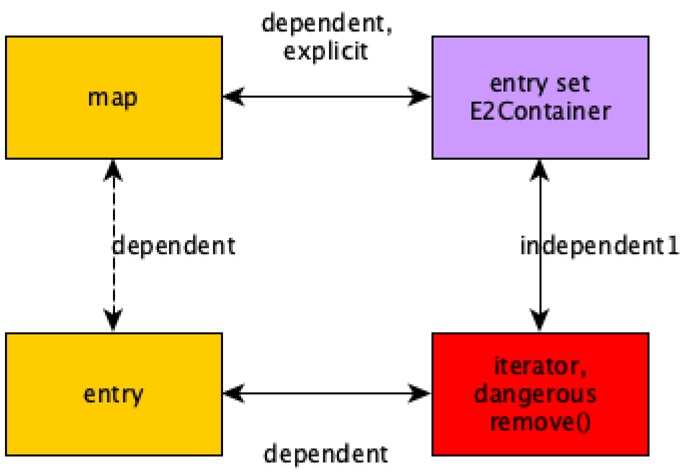

Figure 6. entrySet() immutable, entry to map

It would be safer to forbid the use of the `remove()` method in the iterator at the same time. However, this does not change the linking situation:

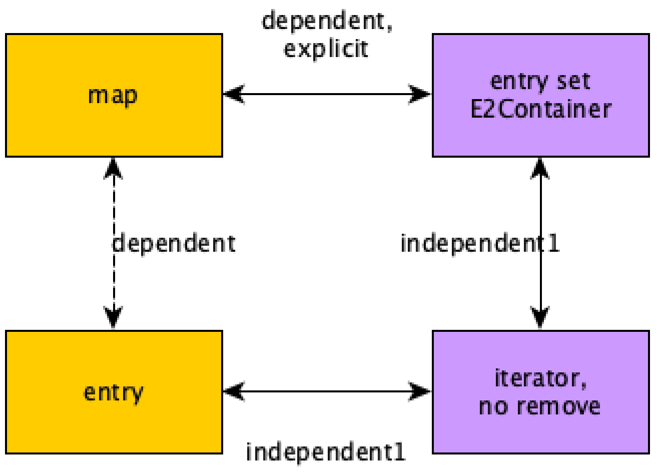

Figure 7. entrySet() immutable, and no remove() method, entry to map

Finally, we remark that only changing the `remove()` method does not change the linking situation between `Entry` and `Map`.

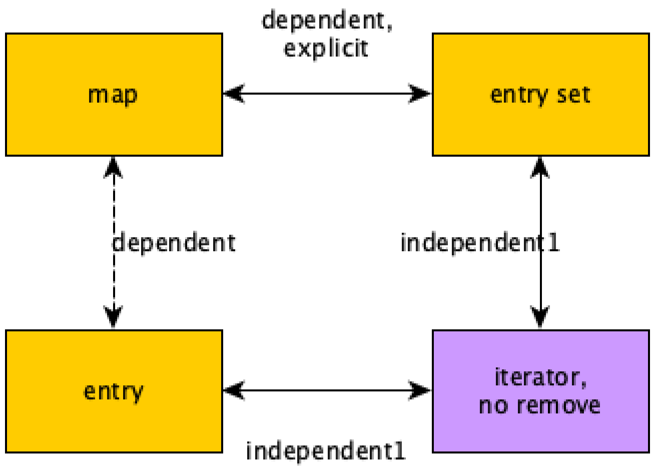

Figure 8. No remove() method, entry to map

5. Annotation XML files

TODO

6. Visualising immutability

TODO

7. The annotation store as a demo project

This section assumes you have installed the analyser and its supporting jars, as described in Installing e2immu.

Next to playing a role in the communication between the IntelliJ plugin and the analyser, the annotation store serves as a demo project for the analyser. In its `build.gradle` file, the relevant lines to run the analyser are:

```
plugins {
    ...
    id 'org.e2immu.analyser'
}
...
e2immu {
    debug = "OUTPUT" //INSPECT,BYTECODE_INSPECTOR,ANALYSER,DELAYED"
    jmods = 'java.base.jmod,java.logging.jmod'
    sourcePackages = "org.e2immu.kvstore"
    readAnnotatedAPIPackages = "org.e2immu.kvstoreaapi"
    writeAnnotationXML = true
    writeAnnotationXMLPackages = "org.e2immu."
    upload = true
}
```

Because the analyser, as a Gradle plugin, is only available in your local Maven repository, the following lines need to be present in `settings.gradle`:

```
pluginManagement {
    repositories {
        mavenLocal()
        ...
    }
    resolutionStrategy {
        eachPlugin {
            if (requested.id.namespace == 'org.e2immu') {
                useModule('org.e2immu:gradle-plugin:0.1.2')
            }
        }
    }
}
```

Run the analyser:

```
./gradlew e2immu-analyser
```

It should "fail" with 2 errors and 4 warnings. If you have not started an annotation store (yet), you should also see an `IOException` warning you that uploading to the annotation store failed.

The `debug` options to `e2immu` listed above activate only the `OUTPUT` debug logger, which writes out the sources enriched with all annotations computed by the analyser. Run:

```
./gradlew e2immu-analyser --debug
```

to find them, obviously among a lot of other debug output.

Next to the `build.gradle` build file, there is a second one, `build-api.gradle`. It provides two tasks, `runIoVertxInspected` and `runIoVertxUsage`, which run the analyser via its command line interpreter rather than the Gradle plugin. Their primary goal is to produce templates for annotated API files. The former contains the option `--write-annotated-api=INSPECTED`, the latter the option `--write-annotated-api=USAGE`.

Executing:

```
./gradlew -b build-api.gradle runIoVertxUsage
```

runs the analyser without annotated API sources, which produces a lot of warnings, but also writes template files in the folder:

```
build/annotatedAPIs/org/e2immu/kvstoreaapi/
```

Reference

8. Concepts

8.1. Modification

Modification of parameters is seen as any modification to the entire object graph of the parameter.

Modification of a field is seen as a modification to the accessible content of the field, i.e., as a modification which happens explicitly inside the type of the field.

A method is modifying when it makes a modification to any of the fields of the type of the method. Modifications to a field of a different type within the same primary type TODO.

A modification to a (necessarily non-private) foreign static field is called a *static side effect*. Type which has methods which make static side effects, is marked with `@StaticSideEffects` . Static side effect marking can be disabled by annotating a static, non-private field with `@IgnoreModifications` .

8.2. Hidden content

We divide the object graph of the fields into an *accessible* part, and a *hidden part*. The *depth* of the object graph plays the primary role in separating both parts: most implementations do not access *all* types that are theoretically reachable from the fields. Secondly, we note that every class `C` that can be sub-classed, i.e., which has not been marked `final`, and every interface `I`, can have a subclass or implementation `S` which has fields that are outside its reach. So when faced with the formal type `C` or `I`, a concrete object of type `S` may be present. This object carries a part hidden to the methods and field accessors of `C` or `I`.

We speak of the *hidden content* of the types when we refer to the hidden part of its fields' object graph. Hidden contents are omni-present in Java.

> The core concept of practical immutability is that modifications to the hidden content of a type are irrelevant to that type.

In *e2immu* , the hidden content of a type `T` is computed on a type-by-type basis: it consists of a set of parameterized types `H1`, …, `Hn`. A type `H` belongs to the hidden content of `T` when it is part of the object graph of the fields, and it is not *explicit* in `T`.

> The underlying concept of being explicit is that the type cannot be replaced by an unbound type parameter, or `java.lang.Object` (JLO), to achieve identical semantics inside `T`.

Formally, a type `E` is explicit in `T` when either

1. it is instantiated, e.g., `⋯ = new E(⋯)` occurs in one of the statements of `T`; the exception to the

rule is JLO; note that the presence of a lambda implies the creation of in instance of the functional interface type;

2. a method is called on it, e.g., when `e` is of type `E`, then `e.someMethod(…)` occurs in the statements of `T`; the exceptions to the rule are JLO's methods;
3. one of its fields is accessed, e.g., when `e` is of type `E`, then `e.someField` occurs in the statements of `T`;
4. one of its extended types is explicit in `T`; this rule does not apply to JLO itself;
5. the type occurs as the type of an object forced to comply with some type restriction:
    a. as an argument in a method or constructor call, unless the formal parameter of that method or constructor is JLO, or is an unbound type parameter;
    b. as the formal expression type `E` of a `return` statement in a method `m` of `T` returning a type different from JLO and an unbound type parameter;
    c. as the iterable's formal return type `E` in a `for(X x: e)` construct;
    d. as the selector's formal type in a `switch(e)` statement;
    e. as the formal expression type `E` of a cast argument `x = (X)e`
    f. as the formal expression type `E` of an `e instanceof X` or `e instanceof X x` expression
6. it occurs as an explicit type in a statement of `T`:
    a. as the exception type in a `catch(E e)` clause;
    b. as the type of a cast, e.g., `x = (E)e`;
    c. as the type of an `instanceof` expression, with or without pattern, as in `x instanceof E e`.

If the type `S` extends the type `C`, then types which are transparent in `C` may not be transparent in `S` anymore. The type `S` can also introduce new transparent types. However, a type cannot be transparent in `S` and explicit in `C`: all explicit types of `C` are also explicit in `S`.

If a type `N` is a nested class of an enclosing type `E`, then all explicit types of `N` are also explicit in `E`. A type `E` and its inner class `I` (nested, not static) share the same set of transparent and explicit types.

For practical reasons, `java.lang.Object`, `java.lang.String` and `java.lang.Class` are always explicit.

8.3. Containers

A type is a container when only private methods or private constructors modify their parameters. To the outside world, all parameters must be `@NotModified`.

When a modifying method is called on a parameter, but this modification is semantically deemed to be irrelevant, then `@IgnoreModifications` can be added to the parameter. This is implicitly the case when the parameter is of one of the abstract types in `java.util.function`, such as `Function`, `Consumer`, etc. The modifications caused by their concrete implementations are typically outside the scope of the type. The typical example of this situation is the `forEach` method of the `java.util.Collection` classes: iterating over the collection should be non-modifying, and modifications to the hidden content are semantically irrelevant to the collection.

8.4. Linking and independence

Two variables are linked when a modification in one may imply a modification in the other. We differentiate between modifications to the accessible content (normal *linking*), and modifications to the hidden content (*hidden content linking*).

Linking typically occurs when the object graphs of the variables (partially) overlaps.

A method is *dependent* when one of the fields of the type links to the return value of the method. A constructor or method parameter is *dependent* when it links to one of the fields of the type.

A method or parameter is independent when it is not dependent. However, given the possibility of hidden content linking, we differentiate between different *levels* of independence. When the hidden content of the field being linked to the return value or parameter is mutable or level 1 immutable, we speak of level 1 independence, and mark with {independent1}. When the hidden content is level 2 immutable, we speak of level 2 independence, and write `@Independent1(level=2)`. When no modifications are possible at all, the hidden content must be recursively immutable, and we speak of independence, marked as `@Independent` .

The computation of linking follows the rules:

1. after an assignment `a = b`, the variable `a` is linked to the variable `b`

The computation of hidden content linking is similar but not quite identical:

1. after an assignment `a = b`, the variable `a` is hidden content linked to the variable `b`

8.4.1. Factory methods

A *factory method* is a static method in type `T` returning a (newly created) element of this type. In factory methods, we compute linking and content linking between the parameters and the return value, ignoring the fields.

8.5. Immutability

8.5.1. Level 1 immutable

A field is called *effectively final* when it receives a value during the construction phase of the object. There are two situations: either it is decorated with the modifier `final`, or it is set in a constructor, or a method only (indirectly) accessible from the constructors. We mark effectively final fields with `@Final` ; fields without this property are called *variable* and marked {variable}.

A type is *level 1 immutable* when all its fields are effectively final. No distinction is made between static and instance fields.

A type is eventually level 1 immutable when after executing a marked method, the fields become effectively final. This transition is best understood as the removal of a number of methods, marked either `@Mark` or `@Only` with parameter `before`, which make the field variable. The mark is annotated in the `@Final` annotation, as `@Final(after="mark")`.

8.5.2. Level n immutable

A type is level *n*, *n>1*, immutable when

1. it is level 1 immutable, i.e., all its fields are effectively final
2. all fields are not modified
3. all fields are either private, or at least of level 2 immutable type
4. no parameters of non-private constructors and non-private methods are dependent of the fields, and no return values of non-private methods are dependent on the fields.
5. the hidden content of the type is mutable or level 1 immutable, when *n=2*, and level *n-1* immutable otherwise.

No distinction is made between static and instance fields.

A type is eventually level *n* immutable when after executing a marked method, the fields become effectively final and or not modified.

8.5.3. Recursive immutability

A type is *recursively immutable* when it is level 2 immutable, with rule 5 modified to state that its hidden content is also recursively immutable.

A recursively immutable type is *deeply* immutable, i.e., no part of its fields' object graph can be modified.

A direct consequence of the definition is that any level 2 immutable type which holds recursively immutable fields only, is again recursively immutable. The following types are recursively immutable, and form the basis of the recursion which allows you to compose more of them:

- `java.lang.Object`
- the primitive types, such as `int`, `long`, …
- `java.lang.Integer`, `java.lang.Long`, … the boxed versions of the primitives `int`, `long` …
- `java.lang.String`
- `java.lang.Class`, whose object graph covers a large amount of other types such as

8.5.4. Independence of a type

A recursively immutable type is fully independent, at all times. As such, it is never explicitly marked `@Independent` .

A level 2 immutable type allows for level 1 independence; a level *n* immutable type allows for level *n-1* independence. Again, no explicit annotation is necessary. Obviously, a parameter or return value need not be linked to the fields, so it can be `@Independent` .

Dependence is only possible when the type is at most level 1 immutable; i.e., its accessible content is assignable or at least modifiable. But dependence is not a necessity.

We define the independence of a type as the minimum of the independence values of its return values and parameters.

8.5.5. Dynamic type annotations

8.6. Eventual immutability

8.7. Miscellaneous

8.7.1. Constants

Java literals are constants. An instance of a type whose effectively final fields have only been assigned literal values, is a constant instance. Typical examples of a constant instances are found in parameterized `enum` fields.

8.7.2. Statement time

Technically important for variable fields (Level 1 immutable).

8.7.3. Singleton classes

8.7.4. Utility classes

A class which is at the same time eventually level 2 immutable, and cannot be instantiated.

The level 2 immutability ensures that the (static) fields are sufficiently immutable. The fact that it cannot be instantiated is verified by

1. the fact that all constructors should be private;
2. there should be at least one private constructor;
3. no method or field can use the constructors to instantiate objects of this type.

8.7.5. Extension classes

An extension class is an eventually final type whose static methods all share the same type of first parameter.

8.7.6. Finalizers

9. Analyser details

9.1. Preconditions

TODO

9.2. Instance state

9.3. Companion methods

TODO

10. Overview of annotations

10.1. List of annotations

For each of the annotations, we answer a couple of standard questions:

Basic
: is this an annotation you definitely should understand?

Immu
: is this annotation part of the immutability concept of the analyzer?

Contract
: will you manually insert this annotation often in interfaces?

Type
: does the annotation occur on types?

Field
: does the annotation occur on (static) fields?

Method
: does the annotation occur on methods and constructors?

Parameter
: does the annotation occur on parameters?

This classification hopefully helps to see the wood for the trees in the long list.

10.1.1. @AllowsInterrupt

Basic ✗	Immu ✗	Contract ✓	Type ✗	Field ✗	Method ✓	Param ✗

Contract-only annotation indicating that this method or constructor increases the statement time (Statement time), in other words, it allows the execution to be interrupted.

Default value of the annotation is true. Methods can be annotated with `@AllowsInterrupt(false)` to explicitly mark that they do not interrupt.

External methods not annotated will not interrupt.

10.1.2. @BeforeMark

Basic ✗	Immu ✓	Contract ✓	Type ✗	Field ✓	Method ✓	Param ✓

Summary

Annotation computed when an eventually immutable type is guaranteed to be in its *before* state, i.e., none of the marked methods have been called yet. As a dynamic type annotation (Dynamic type annotations), it is the opposite of {e1immutable}, {e2immutable}, or its container variants {e1container}, {e2container}, {ercontainer}. They guarantee that an eventually immutable object is in its *after* state, i.e., a marked method has been called, and the object has become immutable.

10.1.3. @Constant

Basic ✗	Immu ✗	Contract ✗	Type ✗	Field ✓	Method ✓	Param ✗

Summary

The analyser emits this annotation when a field has a constant final value, or a method returns a constant value. Its primary purpose is to help debug the analyser. More details in Constants.

Example

In this simple example, an `enum` constant is returned by the `highest` method:

```java
@ImmutableContainer
public enum Enum_3 {
    ONE(1), TWO(2), THREE(3);

    public final int cnt;

    Enum_3(int cnt) {
        this.cnt = cnt;
    }

    @Constant("THREE")
    public static Enum_3 highest() {
        return THREE;
    }
}
```

10.1.4. @Container

Basic ✓	Immu ✓	Contract ✓	Type ✓	Field ✗	Method ✗	Param ✗

Summary

The analyser computes this essential annotation for types which do not modify the parameters of their constructors and non-private methods. See Containers for an in-depth discussion.

Group

The opposite is {mutableModifiesArguments} if the type has {variable} fields, or {e1immutable} if all the type's fields are effectively final.

Example

The following examples present containers: @Constant, @Dependent, @E1Container. Non-containers are in @E1Immutable and @MutableModifiesArguments.

10.1.5. @Dependent

Basic ✓	Immu ✓	Contract ✓	Type ✓	Field ✗	Method ✓	Param ✓

Summary

Annotation used to indicate that the type's fields link to the non-private method's parameter or return value, or the non-private constructor's parameters.

This annotation is the default value in the {dependent}, {independent1}, `@Independent` sequence; hence, it is never explicitly generated by the analyser.

Group

Its opposites are is `@Independent` (no linking is possible) and {independent1} (content linking only).

Example

The assignment of a mutable set to a field typically causes a dependency:

```
class DependentSet<String> {
    private final Set<String> set;

    public DependentSet(@Dependent Set<String> set) {
        this.set = set;
    }

    @Dependent
    public Set<String> getSet() {
        return set;
    }
}
```

A similar example is in @E1Immutable.

10.1.6. @E1Container

Basic ✓	Immu ✓	Contract ✓	Type ✓	Field ✓	Method ✓	Param ✓

Summary

This annotation is a short-hand for the combination of {e1immutable} and `@Container` , as described in Level 1 immutable and Containers.

Group

This annotation sits in between {mutableModifiesArguments}, `@Container` and {e2container}.

Example

In the following example of an eventually level 1 immutable type, the field `j` remains variable until the user of the class calls `setPositiveJ`.

Example of an eventually @E1Container type

```java
@E1Container(after = "j")
class EventuallyE1Immutable_2_M {

    @Modified
    private final Set<Integer> integers = new HashSet<>();

    @Final(after = "j")
    private int j;

    @Modified
    @Only(after = "j")
    public boolean addIfGreater(int i) {
        if (this.j <= 0) throw new UnsupportedOperationException("Not yet set");
        if (i >= this.j) {
            integers.add(i);
            return true;
        }
        return false;
    }

    @NotModified
    public Set<Integer> getIntegers() {
        return integers;
    }

    @NotModified
    public int getJ() {
        return j;
    }

    @Modified
    @Mark("j")
    public void setPositiveJ(int j) {
        if (j <= 0) throw new UnsupportedOperationException();
        if (this.j > 0) throw new UnsupportedOperationException("Already set");

        this.j = j;
    }

    @Modified
    @Only(before = "j")
    public void setNegativeJ(int j) {
        if (j > 0) throw new UnsupportedOperationException();
        if (this.j > 0) throw new UnsupportedOperationException("Already set");
        this.j = j;
    }
}
```

10.1.7. @E1Immutable

Basic ✓	Immu ✓	Contract ✓	Type ✓	Field ✓	Method ✓	Param ✓

Summary

This annotation indicates that a type is level 1 immutable, effectively or eventually, meaning all fields are effectively or eventually final.

Group

This annotation sits in between {mutableModifiesArguments} and {e2immutable}.

Example

The `add` method modifies its parameter `input`; at the same time, the dependence between the constructor's parameter and the field prevents the type from being level 2 immutable:

```
@E1Immutable
class AddToSet {
    private final Set<String> stringsToAdd;

    @Dependent
    public AddToSet(Set<String> set) {
        this.stringsToAdd = set;
    }

    public void add(@Modified @NotNull1 Set<String> input) {
        input.addAll(set);
    }
}
```

10.1.8. @E2Container

Basic ✓	Immu ✓	Contract ✓	Type ✓	Field ✓	Method ✓	Param ✓

Summary

This annotation is a short-hand for the combination of {e2immutable} and `@Container` , as described in Level n immutable and Containers.

Group

Example

```
```

TODO

10.1.9. @E2Immutable

Basic ✓	Immu ✓	Contract ✓	Type ✓	Field ✓	Method ✓	Param ✓

Summary

This annotation indicates that a type is level 2 immutable, effectively or eventually.

It has two additional parameters:

1. *level*, which is a numeric value at least 2, indicating the level of immutability
2. *recursive*, which is true when recursive, or deep, immutability is meant

Example

```
```

TODO

10.1.10. @ImmutableContainer

Basic ✓	Immu ✓	Contract ✓	Type ✓	Field ✓	Method ✓	Param ✓

Summary

This annotation is a short-hand for the combination of effectively recursively immutable (written as `@E2Immutable(recursive=true)`) and `@Container` , as described in Level n immutable and Containers.

Group

it sits at the very end of the `@Container` , {e1container}, {e2container}, `@E2Container(level=3)`, …, {ercontainer} sequence.

Example

```
```

TODO

10.1.11. @ExtensionClass

Basic ✓	Immu ✗	Contract ✗	Type ✓	Field ✗	Method ✗	Param ✗

Summary

An extension class is a level 2 immutable class which uses More details can be found in Extension classes.

Example

```
```

TODO

10.1.12. @Final

Basic ✓	Immu ✓	Contract ✗	Type ✗	Field ✓	Method ✗	Param ✗

Summary

This annotation indicates that a field is effectively or eventually final. Fields that have the Java modifier `final` possess the annotation, but the analyser does not write it out to avoid clutter.

Mode

Use this annotation to contract in the green mode, with the opposite, {variable}, being the default. In the red mode, `@Final` is the default.

Parameters

The `after="mark"` parameter indicates that the field is eventually final, after the marking method.

Details

A field is effectively final when no method, transitively reachable from a non-private non-constructor method, assigns to the field. A field is eventually final if the above definition holds when one excludes all the methods that are pre-marking, i.e., that hold an annotation `@Only(before="mark")` or `@Mark("mark")`.

Example

Please find an example of an eventually final field in the example of @E1Container.

Example for @Variable, @Final

```
@Container
class ExampleManualVariableFinal {

    @Final
    private int i;

    @Variable
    private int j;

    public final int k; ①

    public ExampleManualVariableFinal(int p, int q) {
        setI(p);
        this.k = q;
    }

    @NotModified
    public int getI() {
        return i;
    }

    @Modified ②
    private void setI(int i) {
        this.i = i;
    }

    @NotModified
    public int getJ() {
        return j;
    }

    @Modified
    public void setJ(int j) {
        this.j = j;
    }
}
```

① This field is effectively final, but there is no annotation because of the `final` modifier.

② Note that only the constructor accesses this method.

10.1.13. @Finalizer

```

```

TODO

10.1.14. @Fluent

Basic ✓	Immu ✗	Contract ✓	Type ✗	Field ✗	Method ✓	Param ✗

Summary

This annotation indicates that a method returns `this`, allowing for method chaining.

Mode

There is no opposite for this annotation.

Details

Fluent methods do not return a real value. This is of consequence in the definition of independence for methods, as dependence on `this` is ignored.

Example

```
@Fluent
public Builder setValue(char c) {
    this.c = c;
    return this;
}
```

10.1.15. @Identity

Basic ✓	Immu ✗	Contract ✓	Type ✗	Field ✗	Method ✓	Param ✗

Summary

This annotation indicates that a method returns its first parameter.

Mode

There is no opposite for this annotation.

Details

Apart for all the obvious consequences, this annotation has an explicit effect on the linking of variables: a method marked `@Identity` only links to the first parameter.

Example

```
@Identity
public static <T> requireNonNull(T t) {
    if(t == null) throw new NullpointerException();
    return t;
}
```

10.1.16. @IgnoreModifications

Basic ✗	Immu ✓	Contract ✓	Type ✗	Field ✓	Method ✗	Param ✓

Summary

Helper annotation to mark that modifications on a field or parameter are to be ignored, because they fall outside the scope of the application. This annotation is implicit on parameters of a functional interface type of of `java.util.function` (e.g., `Consumer`, `Function`, etc.).

Mode

There is no opposite for this annotation. It can only be used for contracting, the analyser cannot generate it.

Example

The only current use is on `System.out` and `System.err`. The `print` method family is obviously modifying to these fields, however, we judge it to be outside the scope of the application.

10.1.17. @Independent

Basic ✓	Immu ✓	Contract ✓	Type ✗	Field ✗	Method ✓	Param ✗

Summary

Annotation used to indicate that a method or constructor avoids linking the fields of the type to the return value and parameters. This annotation is only present when there are support data fields. Additionally, on methods, the analyser only computes the annotation when the method is `@NotModified`.

Mode

Use this annotation in the green mode. Its opposite is {dependent}.

- TODO check definition for methods, parameters dependent as well?
- TODO why do we ignore dependence on this?

10.1.18. @Independent1

Basic ✗	Immu ✗	Contract ✗	Type ✗	Field ✗	Method ✓	Param ✓

Summary

As one of the [concept-higher-order-modification] annotations, {independent1} on a parameter, of implicitly immutable type, indicates that this parameter is assigned to one of the fields, or assigned into the object graph of one of the fields. When computed on a method, the return value of the method, again of implicitly immutable type, is known to be part of the object graph of the fields.

Mode

This annotation has no opposite. It implies `@Independent` because it appears on implicitly immutable types only.

Example

This annotation has been contracted in many collection-framework methods, such as

```
Collections.add(@Dependent1 E e);

@Dependent1
E List.get(int index);
```

The most direct example explaining the definition is:

```
public class Dependent1_0<T> {
    @Linked1(to = {"Dependent1_0:t"})
    private final T t;

    public Dependent1_0(@Dependent1 T t) {
        this.t = t;
    }

    @Dependent1
    public T getT() {
        return t;
    }
}
```

10.1.19. @Linked

Basic ✗	Immu ✓	Contract ✗	Type ✗	Field ✓	Method ✗	Param ✗

Summary

Annotation to help debug the dependence system.

Mode

There is no opposite.

10.1.20. @Linked1

Basic ✗	Immu ✓	Contract ✗	Type ✗	Field ✓	Method ✗	Param ✗

Summary

Annotation to help debug the dependence system.

Mode

There is no opposite.

10.1.21. @Mark

Basic ✗	Immu ✓	Contract ✓	Type ✗	Field ✗	Method ✓	Param ✗

TODO

10.1.22. @Modified

Basic ✓	Immu ✓	Contract ✓	Type ✗	Field ✓	Method ✓	Param ✓

Summary

Core annotation which indicates that modifications take place on a field, parameter, or in a method.

Mode

It is the default in the green mode, when `@NotModified` is not visible.

10.1.23. @MutableModifiesArguments

Basic ✓	Immu ✓	Contract ✗	Type ✓	Field ✗	Method ✗	Param ✗

Summary

This annotation appears on types which are not a container and not level 1 immutable: at least one method will modify its parameters, and at least one field will be variable. Definitions are in Containers and Level 1 immutable.

Mode

It is the default in the green mode when none of `@Container` , {e1immutable}, {e1container}, {e2immutable}, {e2container} is present. Use it for contracting in the red mode.

Example

Types with non-private fields cannot be level 1 immutable. Here we combine that with a parameter modifying method:

```
@MutableModifiesArguments
class Mutate {
    @Variable
    public int count;

    public void add(@Modified List<String> list) {
        for(int i=0; i<count; i++) {
            list.add("item "+i);
        }
    }
}
```

10.1.24. @NotModified

Basic ✓	Immu ✓	Contract ✓	Type ✗	Field ✓	Method ✓	Param ✓

Summary

Core annotation which indicates that no modifications take place on a field, parameter, or in a method.

Mode

It is the default in the red mode, when its opposite `@Modified` is not present.

Example

```
```

TODO

10.1.25. @NotNull

Basic ✓	Immu ✗	Contract ✓	Type ✗	Field ✓	Method ✓	Param ✓

Summary

Core annotation to indicate that a field, parameter, or result of a method can never be `null`.

Mode

Use this annotation for contracting in the green mode. It is the opposite of `@Nullable` .

10.1.26. @NotNull1

Basic ✗	Immu ✗	Contract ✓	Type ✗	Field ✓	Method ✓	Param ✓

Summary

the content of an object is `@NotNull` , meaning that all the accessible fields of the object are `@NotNull` .

10.1.27. @Nullable

Basic ✓	Immu ✗	Contract ✓	Type ✗	Field ✓	Method ✓	Param ✓

Summary

This annotation indicates that the field, parameter, or result of a method can be `null`.

Mode

This is the default in the green mode, when `@NotNull` is not present. Use it to contract in the red mode.

10.1.28. @Only

Basic ✗	Immu ✓	Contract ✓	Type ✗	Field ✗	Method ✓	Param ✗

Summary

Essential annotation for methods in eventually immutable types.

Mode

There is no opposite.

Example

The following example shows a useful `@Only(before="…")` method. Please find an example with a useful `@Only(after="…")` method in @TestMark.

```

```

10.1.29. @Singleton

Basic ✓	Immu ✗	Contract ✗	Type ✓	Field ✗	Method ✗	Param ✗

Summary

This annotation indicates that the class is a singleton: only one instance can exist.

Mode

There is no opposite for this annotation.

Example

There are many ways to ensure that a type has only one instance. This is the simplest example:

```
@Singleton
public class OnlyOne {
  public static final INSTANCE = new OnlyOne();

  public final int value;

  private OnlyOne() {
      value = new Random().nextInt(10);
  }
}
```

10.1.30. @StaticSideEffects

Basic ✓	Immu ✓	Contract ✗	Type ✓	Field ✗	Method ✗	Param ✗

Summary

This annotation indicates that at least one method in the type calls a modifying method on a foreign static field.

Example

The expression `System.out.println("x")` calls the modifying method `println` on the foreign static field `out` of the type `System`. When this field is not decorated with `@IgnoreModifications` in the Annotated APIs, the type containing the method calling `println` will be annotated with `@StaticSideEffects` .

10.1.31. @TestMark

Basic ✗	Immu ✓	Contract ✓	Type ✗	Field ✗	Method ✓	Param ✗

Summary

Part of the eventual system, this annotation is computed for methods which return the state of the object with respect to eventuality: *after* is `true`, while *before* is `false`.

Parameters

a parameter `before` exists to reverse the values: when `before` is true, the method returns `true` when the state is *before* and `false` when the state is *after*.

Mode

There is no opposite for this annotation.

Example

The `@TestMark` annotation in the following example returns `true` when `t != null`, i.e., *after* the marked method `setT` has been called:

```
@E2Immutable(after = "t")
public class EventuallyE2Immutable_2<T> {

    private T t;

    @Mark("t")
    public void setT(T t) {
        if (t == null) throw new NullPointerException();
        if (this.t != null) throw new UnsupportedOperationException();
        this.t = t;
    }

    @Only(after = "t")
    public T getT() {
        if (t == null) throw new UnsupportedOperationException();
        return t;
    }

    @TestMark("t")
    public boolean isSet() {
        return t != null;
    }
}
```

10.1.32. @UtilityClass

Basic ✓	Immu ✗	Contract ✗	Type ✓	Field ✗	Method ✗	Param ✗

Summary

This annotation indicates that the type is a utility class: its static side is eventually level 2 immutable, and it cannot be instantiated. As a consequence, should only have static methods.

Mode

There is no opposite for this annotation.

Details

The level 2 immutability ensures that the (static) fields are sufficiently immutable. The fact that it cannot be instantiated is verified by

1. the fact that all constructors should be private;
2. there should be at least one private constructor;
3. no method or field can use the constructors instantiate objects of this type.

Example

The following utility class is copied from the analyser:

```
@UtilityClass
public class IntUtil {

    private IntUtil() {
    }

    // copied from Guava, DoubleMath class
    public static boolean isMathematicalInteger(double x) {
        return !Double.isNaN(x) && !Double.isInfinite(x) && x == Math.rint(x);
    }
}
```

10.1.33. @Variable

Basic ✓	Immu ✓	Contract ✗	Type ✗	Field ✓	Method ✗	Param ✗

Summary

This annotation indicates that a field is not effectively or eventually final, i.e., it is assigned to in methods accessible from non-private non-constructor methods in the type.

Mode

This annotation is the default in the green mode. It is the opposite of `@Final` .

Example

Any non-eventual type with setters will have fields marked {variable}:

```
@Container
class HoldsOneInteger {

  @Variable
  private int i;

  public void set(int i) {
    this.i = i;
  }

  public int get() {
    return i;
  }
}
```

10.2. Relations between annotations

In this section we summarize the relations between annotations.

10.2.1. On types

Note that:

- {e2container} is a shorthand for the combination of {e2immutable} and `@Container` ;
- {e1container} is a shorthand for the combination of {e1immutable} and `@Container` ;
- {e2immutable} requires {e1immutable};
- a type is {mutableModifiesArguments} if and only if it is not {e1immutable} and not `@Container` ;
- when a type is {mutableModifiesArguments}, there will be at least one field marked {variable}, and at least one parameter marked `@Modified` ;
- by definition, types without fields are {e2immutable};
- all primitive types are implicitly {e2container}; the analyser will not mark them.

10.2.2. On fields

Note that {variable} fields can be `@NotNull` ! This obviously requires a not-null initialiser to be present; all other assignments must be not-null as well. The opposite, a `@Final` field that is `@Nullable` , can only occur when the effectively final value is the `null` constant.

The following opposites are easily seen:

- a field is either `@Final` or {variable}
- a field is either `@NotModified` or `@Modified`
- a field is either `@NotNull` (or {nn1}, or {nn2}), or `@Nullable` (see also Nullability).

Note that:

- {variable} implies `@Modified` , whether modifying methods exist for the field or not;
- `@Final` `@Modified` : part of {e1immutable};
- `@Final` `@NotModified`: part of {e2immutable}; sufficient for implicitly immutable types; for other types, the visibility and dependence rules kick in.

From the field's owning type, following the definitions, we obtain:

- if a type is effectively {e2immutable}, all its fields are `@Final` `@NotModified`;
- if a type is effectively {e1immutable}, all its fields are `@Final` ;
- if a type is {mutableModifiesArguments}, at least one of its field is {variable}.

Further, note that:

- fields of a primitive type are always `@NotNull` and `@NotModified`, but neither are marked.

10.2.3. On constructors

Non-trivial constructors have the `@Modified` property. When there is support data, a constructor is either `@Independent` (green) or {dependent} (red). A constructor without annotations therefore implies either that the type is not {e1immutable}, or that the constructor is not assigning to support data fields.

10.2.4. On methods

Opposites:

- a method is either `@NotModified` (green) or `@Modified` (red)
- a method is either `@Independent` (green) or {dependent} (red). This property is only relevant when there is support data, and the method is `@NotModified`
- a method is either `@NotNull` (or {nn1}, or {nn2}) (green), or `@Nullable` (red)

Furthermore,

- if a type is effectively {e2immutable} (green), all its methods are `@NotModified` (green).

Note that:

- quite trivially, `void` methods have no annotations relating to a return element
- methods returning a primitive type are `@NotNull` , but this is not marked

10.2.5. On parameters

Opposites:

- a parameter is either `@NotModified` (green) or `@Modified` (red)
- a parameter is either `@NotNull` (or {nn1}, or {nn2}) (green), or `@Nullable` (red)

Implications:

- if a type is `@Container` , the parameters of all non-private methods and constructors are `@NotModified` (green);
- a parameter of primitive type, unbound parameter type, functional type, or {e2immutable} type, is always `@NotModified` (green);

Note that:

- if a type is {mutableModifiesArguments} (red), at least one of its parameters is `@Modified` (red), which will be marked;
- quite trivially, parameters of a primitive type are always `@NotNull` and `@NotModified`, but we will not mark parameters of a primitive type.

10.2.6. Nullability

By convention,

- {nn1} implies `@NotNull`
- {nn2} implies `@NotNull` , {nn1}
- etc.

This way of working makes most sense in an immutable setting.

10.2.7. Eventually and effectively immutable

Field types and method return types can be eventually or effectively immutable when their formal type is not level 1 or level 2 immutable, but the dynamic or computed type is. In the latter case, static analysis shows that all assignments to the field, or all return statements, result in an immutable object. In the former case, object flow computation proves that the mark has been passed for this object to have become immutable.

When a type is level 1 or level 2 eventually immutable, and the object flow computation proves that all assignments or return statements yield an object which is in a state *before* the mark, the analyser will emit `@BeforeMark` .

Fields take the annotation of the eventual state, with the qualification of `after="…"`:

property	not present	eventually	effectively
finality of field	{variable}	`@Final(after="mark")`	`@Final`
modification of field	`@Modified`	`@NotModified(after="mark")`	`@NotModified`

11. List of errors and warnings

11.1. Opinionated

The errors described in this section relate to bad practices that are not tolerated by the analyser:

ASSIGNMENT_TO_FIELD_OUTSIDE_TYPE

Assigning to a field outside the type of that field, is not allowed. Replace the assignment with a setter method. Try to do assignments as close as possible to object creation.

METHOD_SHOULD_BE_MARKED_STATIC

Methods which do not refer to the instance, should be marked `static`.

NON_PRIVATE_FIELD_NOT_FINAL

Non-private fields must be effectively final (marked `@Final`). Who knows what can happen to the field if you allow that?

PARAMETER_SHOULD_NOT_BE_ASSIGNED_TO

Parameters should not be assigned to. The implementation of the analyser assumes that parameters cannot be assigned to, so this is probably the one error you really do not want to see.

11.2. Evaluation

ASSERT_EVALUATES_TO_CONSTANT_FALSE

The condition in the `assert` statement is always false.

ASSERT_EVALUATES_TO_CONSTANT_TRUE

The condition in the `assert` statement is always true.

CONDITION_EVALUATES_TO_CONSTANT_ENN

The null or not-null check in the `if` or `switch` statement evaluates to constant. This message is computed via a `@NotNull` on a field.

CONDITION_EVALUATES_TO_CONSTANT

The condition in the `if` or `switch` statement evaluates to a constant.

INLINE_CONDITION_EVALUATES_TO_CONSTANT

The condition in the inline conditional operator `… ? … : …` evaluates to a constant.

PART_OF_EXPRESSION_EVALUATES_TO_CONSTANT

Part of a short-circuit boolean expression (involving `&&` or `||`) evaluates to a constant.

11.3. Empty, unused

Errors of this type are typically trivial to clean up, so why not do it immediately?

EMPTY_LOOP

Empty loop: the loop will run over an `Iterable` which the analyser believes is empty.

IGNORING_RESULT_OF_METHOD_CALL

Ignoring result of method call. That’s fine when the method is modifying, but the call is pretty useless when the method is `@NotModified`.

UNNECESSARY_METHOD_CALL

Unnecessary method call, like, e.g., calling `toString()` on a `String`.

UNREACHABLE_STATEMENT

Unreachable statement, often because of an error from the Evaluation category.

UNUSED_LOCAL_VARIABLE

Unused local variable.

UNUSED_LOOP_VARIABLE

Unused loop variable.

UNUSED_PARAMETER

Unused parameter. Not raised when the method is overriding another method.

USELESS_ASSIGNMENT

Useless assignment.

TRIVIAL_CASES_IN_SWITCH

Trivial cases in `switch`.

PRIVATE_FIELD_NOT_READ

Private field not read outside constructors. If this is intentional, turn it into a local variable.

11.4. Verifying annotations

The following errors relate to the annotations you added to your source code, to verify that a type, method or field has a given property.

ANNOTATION_ABSENT

Annotation missing. You wrote the annotation in the source code, but it is absent from the computation of the analyser.

ANNOTATION_UNEXPECTEDLY_PRESENT

You explicitly write that the annotation should be absent, using `absent=true`, still, the analyser computes it.

CONTRADICTING_ANNOTATIONS

Contradicting annotations

WRONG_ANNOTATION_PARAMETER

Wrong annotation parameter: the annotation is both in the source code, and computed by the analyser. However, the associated values are differing.

WORSE_THAN_OVERRIDDEN_METHOD_PARAMETER

Property value worse than overridden method’s parameter

WORSE_THAN_OVERRIDDEN_METHOD

Property value worse than overridden method

11.5. Immutability

CALLING_MODIFYING_METHOD_ON_E2IMMU

Calling a modifying method on level 2 immutable type is not allowed. This error is typically raised when the type is only dynamically computed to be level 2 immutable, such as in the case of the immutable version of a collection.

DUPLICATE_MARK_CONDITION

Duplicate mark precondition

EVENTUAL_AFTER_REQUIRED

Calling a method requiring `@Only(after)` on an object in state `@Only(before)`.

EVENTUAL_BEFORE_REQUIRED

Calling a method requiring `@Only(before)` on an object in state `@Only(after)`.

INCOMPATIBLE_IMMUTABILITY_CONTRACT_AFTER_NOT_EE1

Incompatible immutability contract: Contracted to be @E2Immutable after the mark, formal type is not (eventually) @E1Immutable. Variants exist for `@Only(before="…")`, and level 2 immutable.

INCOMPATIBLE_PRECONDITION

Incompatible preconditions

WRONG_PRECONDITION

Wrong precondition

PRECONDITION_ABSENT

Precondition missing

ONLY_WRONG_MARK_LABEL

@Only annotation, wrong mark label

MODIFICATION_NOT_ALLOWED

Illegal modification suspected

11.6. Odds and ends

CIRCULAR_TYPE_DEPENDENCY

Methods that call each other circularly, make it difficult for the analyser to compute modifications correctly.

DIVISION_BY_ZERO

The analyser suspects division by zero here.

FINALIZER_METHOD_CALLED_ON_FIELD_NOT_IN_FINALIZER

A `@Finalizer` method can only be called on a field, when in another `@Finalizer` method. Please refer to Finalizers.

FINALIZER_METHOD_CALLED_ON_PARAMETER=

A `@Finalizer` method cannot be called on a parameter. Please refer to Finalizers.

NULL_POINTER_EXCEPTION

The analyser suspects that this will always raise a null-pointer exception.

POTENTIAL_NULL_POINTER_EXCEPTION

The analyser suspects, and only warns, for a potential null-pointer exception.

TYPES_WITH_FINALIZER_ONLY_EFFECTIVELY_FINAL

Fields of types with a `@Finalizer` method can only be assigned to an effectively final (`@Final`) field. Please refer to Finalizers.

Technical

12. Implementation

The principles of *e2immu* guide the implementation of the analyser:

1. Try to use containers exclusively, in other words: never modify your parameters.
2. Use as many immutable objects as possible; either directly, via builders, or via eventually final constructs.
3. Never backtrack on decisions: delay making a decision as long as you cannot be sure, but once a decision is made, it is rock-solid.

The container requirement has been broken in exactly one situation: that of the `ExpressionContext`. Because we intend to use the implementation as one of the more serious test cases of *e2immu* , one such situation is acceptable. It nicely contrasts with `EvaluationContext`, which does follow the rules, and which relies on an `EvaluationResult` to be applied.

12.1. Main flow

The main orchestrating class of *e2immu* is the `Parser` class, which initiates the different phases of *e2immu* :

1. configuration and input preparation;
2. inspection, transforming source code into objects at the type, method and field level;
3. resolution, tying these objects together, while at the same time parsing statements and expressions;
4. analysis, providing semantic information;

5. output, showing and storing the results of the analysis.

The key classes for configuration and input preparation are, easily enough, `Configuration`, `Input`, `Resources`, `ClassPath`.

12.2. Inspection

The *e2immu* project currently depends on two libraries to help transform the Java source code and byte code into objects: JavaParser for the source, and ASM for the byte code. Currently, the byte code inspector does not inspect statements.

Byte code inspection is done on-demand, because the system libraries provided with the Java Runtime are too extensive to convert *a priori*. The main class responsible is the `ByteCodeInspector`.

Inspection using JavaParser is primarily carried out by the `TypeInspector` and the `MethodInspector`.

Both inspection types fill up a `TypeMap` (which exists as a builder and a final, immutable implemention constructed after the resolution phase), which keeps track of all types known to *e2immu* . A `TypeContext` object provides support for translating locally known simple type names to fully qualified ones.

The end result of inspection are the types

1. `TypeInfo`, with inspection details in `TypeInspection`;
2. `MethodInfo`, with inspection details in `MethodInspection`;
3. `FieldInfo`, with inspection details in `FieldInspection`;
4. `ParameterInfo`, with inspection details in `ParameterInspection`.

12.3. Resolution

The resolution phase has two primary goals. Firstly, it parses method bodies into statements (implementations of the interface `Statement`), starting with `Block`, and expressions (implementations of `Expression`). Central to the inspection of expressions is the `ExpressionContext`. Secondly, the resolution phase resolves all remaining type and method determination issues:

- which method call is meant in the face of overrides and overloads;
- it keeps track of the place of a type in the hierarchy, available for reference to the analyser;
- it determines the types of objects created with the diamond operator `<>`, of lambda's without explicit parameter types.

Even though JavaParser has its own symbol-solving module, *e2immu* uses its own implementation. The most complicated of the resolution classes is `ParseMethodCallExpr`, which determines the exact method call. A lot of logic dealing with the type hierarchy is in `ParameterizedType`, the class responsible for representing types decorated with concrete values for type parameters.

The `Resolver` is the main class here, it adds `TypeResolution` to `TypeInfo`, and `MethodResolution` to `MethodInfo`.

The output of the resolution phase is a list of types, sorted in order of dependency. This sorting step is necessary for analysis, to the extent that circular dependencies between types make analysis a lot harder. The class `SortedType` transfers information from the `Resolver` into the `PrimaryTypeAnalyser`.

12.4. Analysers

In *e2immu* , a number of different analysers cooperate to determine the semantic properties of the code:

- the *type analyser*, instantiated once for each (nested) type;
- the *method analyser*, instantiated for each method or constructor, even some hidden ones;
- the *parameter analyser*, instantiated for each parameter of each method and constructor;
- the *field analyser*, instantiated for each field;
- the *statement analyser*, instantiated for each statement of each method and constructor.

There are obvious and strong dependencies between these analysers, e.g., the modification aspect of a field depends on determining whether there are statements that modify it. At the same time, there are dependencies between different aspects of the code, e.g., between methods calling other methods, methods initialising fields, etc. There is no easy way to determine an exact execution order for this multitude of analyser instances.

To this end, the analyser implements an iterative approach, that gives each analyser one opportunity per iteration to determine its values and properties based on previously established facts.

When a value or property cannot yet be determined, typically because a dependency has not yet been met, this value or property is **delayed**. The delay system keeps track of the causes of the delay, to allow for detecting circular dependencies. The latter can then be broken, to allow the iterative system to continue. As soon as a value has been determined, it is written out for others to refer to, and cannot be changed anymore.

The analysers themselves consist of a number of components (or tasks), which are executed as part of the iteration. This execution takes place in a fixed order; it starts at the first, not-yet-executed (in the first iteration) or non-delayed (in the subsequent ones) component, and visits all others in the chain. The main loop stops iterating when all components have been resolved (are in the *done* state, rather than the *delayed* state).

The order of the main analyser groups is

1. methods. For each method, the analyser processes the
   a. parameters;
   b. statements, from first to last, recursively descending as explained later;
   c. method itself;
2. fields;
3. types.

Inside the primary type (the .java file) the methods are processed alphabetically, as are the fields and nested types. This orchestration is the responsibility of the `PrimaryTypeAnalyser`, which is also used, recursively, to analyse types defined inside statements. It employs the `AnalyserComponents` class to execute the components.

Annotated API files (or classes) are analysed by the `AnnotatedAPIAnalyser`, which is a shallow version of the primary type analyser, the type analyser (`ComputingTypeAnalyser`) and the field analyser (`FieldAnalyser`). Methods without code are analysed by the `ShallowMethodAnalyser`, rather than the `ComputingMethodAnalyser` and `ComputingParameterAnalyser`.

In the special situation of *sealed* classes, we need a variant on the shallow analysers which aggregate data from the computed analysers in the child types. These are the `AggregatingTypeAnalyser`, `AggregatingMethodAnalyser`, and `AggregatingParameterAnalyser`.

12.5. Statements and expressions

Due to the hierarchical nature of statements, statement analysers and statement analysis objects are also structured hierarchically. The statements of a method are internally numbered from 0 onward. Because of the hierarchical nature, a dotted system is used: each sub-block introduces a dot, a sub-block number, and a dot again. So `0.1.2` indicates the 3rd statement in the 2nd block of the first statement of the method. The second block can be a "catch"-block, or the "else" block in an "if-else" statement.

The general `Statement` interface has an implementation per type of statement specific to the Java language. Many of them contain expressions, represented by the `Expression` interface. Again, the typical range of expression implementations exist, mostly corresponding to expressions existing in the language. Specific to this implementation are `DelayedExpression`, `DelayedVariableExpression`, and `PropertyWrapper`.

Expressions can be evaluated in the context in which they appear, the `EvaluationContext`. The result of this evaluation is an `EvaluationResult` object, which is subsequently processed by the statement analyser. When dependencies inside the expression have not been resolved yet (e.g., the return value of a method is still unknown, or a variable doesn't have a value yet), the end result is an expression which contains delayed components. Each expression answers the `isDelayed` method, and can return a `CausesOfDelay` object to identify exactly what the reasons for the delay are.

Some evaluation leads to simplification of the expression, which is sometimes a reason to emit a warning to the developer. When a complex expression evaluates to a constant, for example, it is likely that they should know about this.

12.6. Analysis objects

Each analyser has a corresponding analysis object, which contains the results of the analysis. After analysis, the analyser is dropped, the analysis object remains. Each analysis component consists of two implementations: the builder, which holds the values while the analyser is alive, and the effectively immutable implementation which survives the analyser. There is currently one exception: the `StatementAnalyser` only has a `StatementAnalysis` data companion, which holds eventually immutable objects. The `MethodAnalysis` interface has a `MethodAnalysisImpl` immutable implementation, and a `MethodAnalysisImpl.Builder` mutable builder.

An important aspect of the builders is that writing information is constrained: while causes of delay can be overwritten in each iteration, once a value has been determined for a property or some piece of information the analyser has to store, it cannot be changed anymore. To this end, we employ a variation of eventually final helper classes, such as `EventuallyFinal` and `VariableFirstThen`, with `setVariable` and `setFinal` write methods.

The `StatementAnalysis` data object holds a number of sub-objects, where data is stored per topic:

- `NavigationData` holds the data structure that points to the next statement, and the first statements of sub-blocks. This data structure can be modified by statement replacements.
- `FlowData` holds the conditions under which this statement will be executed, and what the effect is on the flow of execution. E.g., a `throws` statement will cause either a guaranteed, or a conditional escape from the execution flow.
- `StateData` holds state of the variables. After a conditional escape, for example, the state will be the negation of the condition that caused the escape.
- `MethodLevelData` holds the data necessary for the method analyser; typically, only the method level data of the last statement of the method will be inspected.
- `ConditionManager` is a support object that holds condition, state, and precondition in one object.

12.7. Properties

The analysers compute semantic information, some of which can be expressed as numeric values for properties, many of which apply to most of the analysers. These properties can then be visualised, either by coloring, highlighting in an IDE, or by adding annotations.

A property (implemented by the enumeration `Property`) has a numeric value when it is not delayed. Otherwise, it takes a `CausesOfDelay` value, which enumerates the reasons why no value was computed for this property. The property-value map `Properties` maps `Property` to `DV`. The latter stands for *delayable value*.

There are four types of properties:

- value properties
- context properties
- external properties
- internal marker properties

12.7.1. Value properties

Value properties are associated with a expression which has been evaluated to a non-delayed result. When a variable takes a value, the variable’s value properties are computed directly from the value. There are five value properties:

- `NOT_NULL_EXPRESSION`: the not-null aspect of an evaluated expression, seen outside its context. Its associated annotations are `@Nullable` , `@NotNull` , and {nn1}.

- `IMMUTABLE`: the immutability aspect of an evaluated expression, tied to the dynamic type, seen outside its context. Associated annotations are {mutableModifiesArguments}, {e1immutable}, {e2container}, …
- `INDEPENDENT`: the "independent" aspect of the evaluated expression, tied to the dynamic type. Annotations are {dependent}, {independent1}, `@Independent` .
- `CONTAINER`: the "container" aspect, by default tied to the dynamic type, but potentially modified (from false to true) by a contract. The annotation needed to contract an abstract parameter to be of container type is `@Container` .
- `IDENTITY`: the fact that the evaluated expression is exactly the value of the first parameter of the method. Corresponds to `@Identity` on the method.

12.7.2. Context properties

In contrast, context properties are unique to a variable, and are held independently of the value that the variable takes. Context properties accumulate information about the variable from one statement to the next. There are three context properties:

- `CONTEXT_NOT_NULL`: the not-null aspect of the variable in the given context. E.g., when the variable appeared in the scope of a method, it must be not-null. When the variable appeared as the argument of an `addAll` method, it must be content-not-null. CNN takes the Condition Manager into account: in the context of `if(x != null)`, the statement `x.method()` does not force the CNN of `x` to the not null value.
- `CONTEXT_IMMUTABLE`: can be higher than the formal immutability value
- `CONTEXT_MODIFIED`, independent of value, value delays; however, values **change** depending on modification due to companions! As soon as a variable representing a field, or `this`, has this property set to true, the method in which this occurs is marked modifying, which is visualised by the `@Modified` property.

12.7.3. External properties

When a field is assigned to a parameter in a constructor, the value of the field and that of the parameter are bound to each other. To break a very predictable circular dependency, parameters receive a value right from the first iteration. Because value properties are bound to this value, their eventual values may differ from the ones necessarily chosen in the very first iteration. The external properties receive the final values of the value properties for parameters, and for all other values for which a circular dependency had to be broken. A second example is the `this` variable, which can only get the correct immutability value once the type analyser has established it, but modification computation cannot continue until `this` has a value, and the type analyser needs the modification computations. There are two external properties:

- `EXTERNAL_NOT_NULL`: of relevance when CNN demands a high value, but the field cannot allow for one. So ENN < CNN, and this will result in a potential null-pointer warning
- `EXTERNAL_IMMUTABLE`: of relevance when CIMM demands a higher value than the formal type’s value, but the dynamic value cannot go as high. Results in a modification warning.

By convention, external properties on parameters have the value `NOT_INVOLVED` when the parameter is not linked to a field.

The property `MODIFIED_OUTSIDE_METHOD` is similar to an external property. It originates in the field analyser. TODO rename?

12.8. Variables

A statement analyser, and associated statement analysis builder object, is present for each statement. The builder holds information about the state of all the variables known to the statement.

Parameters are known from the first statement onwards; fields are only introduced in the statement that refers to them. All subsequent statements will also know the statement.

Each variable in the statement analyser has values for three "levels"

- the *initial* value, or value of the previous statement (I)
- the *evaluated* value (E)
- the *merged* value (M), as the summary of all nested statements

Obviously, not all statements allow for nested statements; they will not have an M level. Information about a variable is stored in a `VariableInfoContainer`, which holds `VariableInfo` instances for each of the three levels. A `VariableInfo` object holds:

- the variable and its name
- a value
- linked variables
- at which statement times it was read
- by which statements it was written
- by which statement it was last read.
- a property-value map.

When a field is detected during the evaluation phase, it cannot yet have a value at the initial level. A delayed value expression is returned for this field. In the next iteration, the field analyser may provide an initial value for the field.

Variable fields and loop variables have "local copies" which exist starting from the 2nd iteration.

A variable which points to a generic value (of type `Instance`) is evaluated to a `VariableExpression`, rather than the value itself. When variable `b` is assigned to variable `a`, `a` can have this `VariableExpression` as a value. No further redirections are possible.

Variables are identified by their fully qualified name. Local copies can be identified by the `$` sign and the suffix, either specifying the latest assignment and read statement ids, or the statement time.

12.9. Clustering

Variables link to other variables at different levels:

- static assignments (`a = b`)
- dynamic assignments, e.g., `a = Objects.requireNonNull(b)`, which become assignments after evaluation
- linking at the accessible content level (dependent objects), `s = t.subList(0, 3)`
- linking at the hidden content level, `a.add(b)`

The context properties are assigned to clusters of variables: after executing `a=b; b.doSomething()`, for example, we must set the context not null value of `a` to not-null as well.

The static assignment level is different from the three other levels in that it is available from the first iteration onward. Local copies of variable fields and variables assigned to in a loop are only introduced in the 2nd iteration; however, they simply expand the cluster and cannot cause different context property values.

The only context property which is computed across the first three levels (i.e., static and dynamic assignment, and linking at the accessible content level) is *context modification*: after executing `s = t.subList(0, 3); s.add(x);` both `s` and `t` are likely to be modified.

The result of linking is stored in the `linkedVariables` field of `VariableInfo`; it is of type `LinkedVariables`.

12.10. Typical execution order

Parameters start with values, nullable, mutable in case of self-references; then they rely on ENN, ExtImm.

`this` also starts with a value, relies on ENN, ExtImm

statical assignment linking → CNN, CIMM - delays only on dependent methods

CondMgr works on delayed values, to assist in correct CNN values

field value ← last statement + initialiser values + CNN, CIMM

in statements, delayed variable value is replaced by field value + IMM + NNE

Rest of linking follows as soon as there are values → CM → type immutable

Breaking a circular computation CNN → field → ENN → NNP → CNN deactivates this local CNN.

ENN, ExtImm can always augment on fields when value was chosen after breaking circular computation

12.11. Output system

The output system of *e2immu* is very lightweight, yet sufficiently flexible to be parameterizable between extremely compact and nicely readable. Output elements (implementations of `OutputElement`, such as `Text`, `Symbol`, `Space`, and `Guide`) are collected in an `OutputBuilder`, and finally emitted by the `Formatter` which is parameterized by `FormatterOptions`. The `OutputBuilder` implements the `Collector` interface from the JDK streaming package, which allows for simple conversion and collection of lists and streams of statements, expressions, types, etc. into `OutputElement` and then `OutputBuilder` instances.

At the moment the implementation is not tied to the inspection system, so there is no way to link back to line numbers and symbol positions of the original source. It is also not possible to maintain the existing formatting.

Errors and warnings are stored in an enumeration called `Message.Label`, and are decorated with a `Location` and some textual information. They are collected in a `Messages` object. Resource files allow for translation of the messages in other languages.

The `Location` object holds an `Identifier`, which can be of the `PositionalIdentifier` variety that refers to the line number and position in the original source code, provided by the JavaParser.

13. Supporting tools

13.1. Gradle plugin

The Gradle plugin greatly facilitates the use of the analyser by integrating it your project's build process. We based its initial implementation on the one from SonarQube.

Example of `build.gradle` file

```
plugins {
    id 'java'
    id 'org.e2immu.analyser'
}

...

repositories {
    ...
}

dependencies {
   ...
}

e2immu {
    skipProject = false
    sourcePackages = 'org.e2immu.'
    jmods = 'java.base.jmod,java.se.jmod'
    jre = '/Library/Java/JavaVirtualMachines/openjdk-11.0.2.jdk/Contents/Home/'
    writeAnnotatedAPIPackages = 'org.e2immu.'
    writeAnnotationXMLPackages = 'org.e2immu.'
}
```

The list of properties configurable differs slightly from the one of the command line. Gradle takes care of source and class path.

13.2. Key-value store

The key-value store is a simple, two-class implementation of an independent key-value store acting as a bridge between the *e2immu* analyser and the IntelliJ IDEA plugin. It is mostly agnostic to the purpose in the project: it stores key-value pairs in sub-stores called projects.

Unless directed differently by the `e2immu-port` parameter, the store starts listening on port 8281 for HTTP communication. The protocol summary is:

```
# get an annotation name for an element
# curl http://localhost:8281/v1/get/project-name/element-description
#
# set the annotation name for an element
# curl http://localhost:8281/v1/set/project-name/element-description/annotation-name
#
# get annotation names for a whole list of elements
# curl -X POST @elements.json http://localhost:8281/v1/get/project-name
#
# set the annotation names for a whole map of elements
# curl -X PUT @elementsandannotations.json http://localhost:8281/v1/set/project-name
#
# get all key-value pairs for a project
# curl http://localhost:8281/v1/list/project-name
#
# list all projects
# curl http://localhost:8281/v1/list
```

Projects that do not exist will be created on-demand. The bulk *get* operation may receive more elements than that it asked for: depending on the effects of recent *set* operations, the store may include recently updated keys that were asked for recently as well.

Start the store by running:

```
~/g/e/annotation-store (master)> gradle run

> Task :annotation-store:run
Aug 01, 2020 9:19:55 AM org.e2immu.kvstore.Store
INFO: Started kv server on port 8281; read-within-millis 5000
<============---> 75% EXECUTING [1m 39s]
> :annotation-store:run
```

In another terminal, experiment with the `curl` statements:

```
~> curl http://localhost:8281/v1/get/default/hello
{"hello":""}⏎
~> curl http://localhost:8281/v1/set/default/hello/there
{"updated":1,"ignored":0,"removed":0}⏎
~> curl http://localhost:8281/v1/get/default/hello
{"hello":"there"}⏎
~> curl http://localhost:8281/v1/set/default/its/me
{"updated":1,"ignored":0,"removed":0}⏎
~> curl http://localhost:8281/v1/list/default
{"its":"me","hello":"there"}⏎
~> curl http://localhost:8281/v1/list
{"projects":["default"]}⏎
```

Removing a key-value pair only works via the PUT method:

```
~> curl http://localhost:8281/v1/set/default/its/
<html><body><h1>Resource not found</h1></body></html>
~> cat update.json
{"its":"","hello":"here"}
~> curl -X PUT -d @update.json  http://localhost:8281/v1/set/default
{"updated":1,"ignored":0,"removed":1}
~> curl http://localhost:8281/v1/list/default
{"hello":"here"}
```

13.3. IntelliJ IDEA plugin

Without a plugin for an IDE, the analyser would be hard to use, and the main purpose of the project, namely, unobtrusively assisting in better coding, would remain very far off. The current plugin for IntelliJ IDEA is absolutely minimal. We based our initial implementation on the Return statement highlighter plugin by Edoardo Luppi.

13.3.1. Visual objectives

For the proof of concept, we aim to color elements of the source code in such a way that there is visual information explaining why a type is not {e2immutable} or `@Container` . In the green mode, we highlight the immutable elements, which useful when they are sparse. In the red mode, we warn for mutable ones in an otherwise pretty immutable environment:

Following the Relations between annotations for types, we color:

annotation on type	green mode	red mode
{e2container} (incl. primitives)	green	black
{e2immutable}	green	black
{e1container}	brown	brown
{e1immutable}	brown	brown
`@Container`	blue	blue
{mutableModifiesArguments}	black	red
`@BeforeMark`	purple	purple

For fields, we note that {variable}- `@Final` and `@NotModified`- `@Modified` can technically occur in each combination:

- {variable} `@Modified` : impossible for unmodifiable types
- {variable} `@NotModified`
- `@Final` `@Modified` : part of {e1immutable}, impossible for unmodifiable types
- `@Final` `@NotModified`: part of {e2immutable}

We therefore color the annotation hierarchy for fields as:

combination	annotation on field	green mode	red mode
{variable} `@Modified`	{variable}	black	red
{variable} `@NotModified`			
`@Final` `@Modified`	`@Modified`	brown	brown
`@Final` `@NotModified`, unmodifiable types	`@Final`	green	black
`@Final` `@NotModified`, modifiable types only	`@NotModified`	green	black
{supportData} (when field `@NotModified` and owning type {e1immutable})	{supportData}	green italics	black italics

The {supportData} annotation is relevant to understand why a type is not level 2 immutable. In other situations, it simply clutters. The analyser will only emit it when the type is already {e1immutable} or {e1container}, and the field is already `@NotModified`.

The plugin transfers dynamic type annotations involving immutability (such as {e2container}) from the field to the type. As a consequence, the left-hand side `Set` type will color green in:

```
private final Set<String> strings = Set.of("abc", "def");
```

according to the first color scheme.

Methods declarations mix dependency with modification. Independence is not necessary when there are no support types, and, given that we only start showing support data types when all fields are `@NotModified`, which implies that all methods are `@NotModified`, it makes sense to emit the dependency annotations only in the `@NotModified` situation:

annotation on method	green mode	red mode
`@Independent` (implying `@NotModified`)	green	black
{dependent} (implying `@NotModified`)	brown	brown
`@NotModified` (no support data)	green	black
`@Modified`	black	red

The interesting aspect to constructors is whether they are independent or not. To be consistent with the system for methods, the analyser will only emit the annotation when the type is showing support data.

annotation on constructor	green mode	red mode
`@Independent` (when support data)	green	black

annotation on constructor	green mode	red mode
{dependent} (when support data)	brown	brown
no support data	black	black

The situation of parameters is binary. The analyser colors:

annotation on parameter	green mode	red mode
`@NotModified`	green	black
`@Modified`	black	red

13.3.2. Information flow

Exactly which information does the analyser store in the key-value store? The keys are type names, method names, parameter names, or field names in a format that both analyser and plugin understand:

- for a type, we use the fully qualified name, with sub-types separated by dots;
- for a method, we use the *distinguishing name*, which is a slightly custom format that looks like

  type’s fully qualified name '.' method name '(' parameter type ',' parameter type … ')'

  where the parameter type is 'T#2' or 'M#0' when it is the third type parameter of the method’s type, or the first type parameter of the method, respectively. If the parameter type is not a type parameter, the fully qualified name suffices;
- for a parameter, we append the '#' sign followed by the index to the method’s distinguishing name, starting from 0;
- for a field, we use the fully qualified name of the type, a ':', and the name of the field.

On top of that comes the combination of a type and a field or method for the dynamic type annotations of fields and methods: the composite key is the field’s or method’s key followed by a space, and the type’s key.

The values consist of a single annotation type name in lowercase, like *e2immutable* or *notmodified*.

13.3.3. Implementation

The `JavaAnnotator` class links the plugin to the abstract syntax tree (or PSI in IntelliJ-speak). It is instrumental to decide which textual elements are to be highlighted.

The plugin framework creates the annotator on the basis of the `java.xml` configuration file. It is statically connected to the `JavaConfig` singleton instance which holds the configuration of the plugin.

It is also statically connected to the `AnnotationStore` singleton which is responsible for the assignment of elements to annotations. Elements will be described in a standardized format which will extend fully qualified type names. Annotations will be described as the simple names of the e2immu annotations.

The annotation store connects to an external server whose address is modifiable in the plugin configuration. By default, it connects to a local instance on http://localhost:8281. For now, this is a key-value store which keeps track of request and update times.

The annotation store keeps a cache where elements have a certain TTL. As soon as an element is not in the cache, the plugin requests the annotation value from the external server.

Where next?

There are many areas for improvement and extension:

- much better information transfer between analyser, user, and source code
- towards a full-option static code analyser
- towards more automation for creating annotated API files
- more plugins, for other build tools and other IDEs
- …

14. Copyright and License