Effective and eventual immutability

Table of Contents

Using e2immu

1. Using e2immu

1.1. Basic use

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

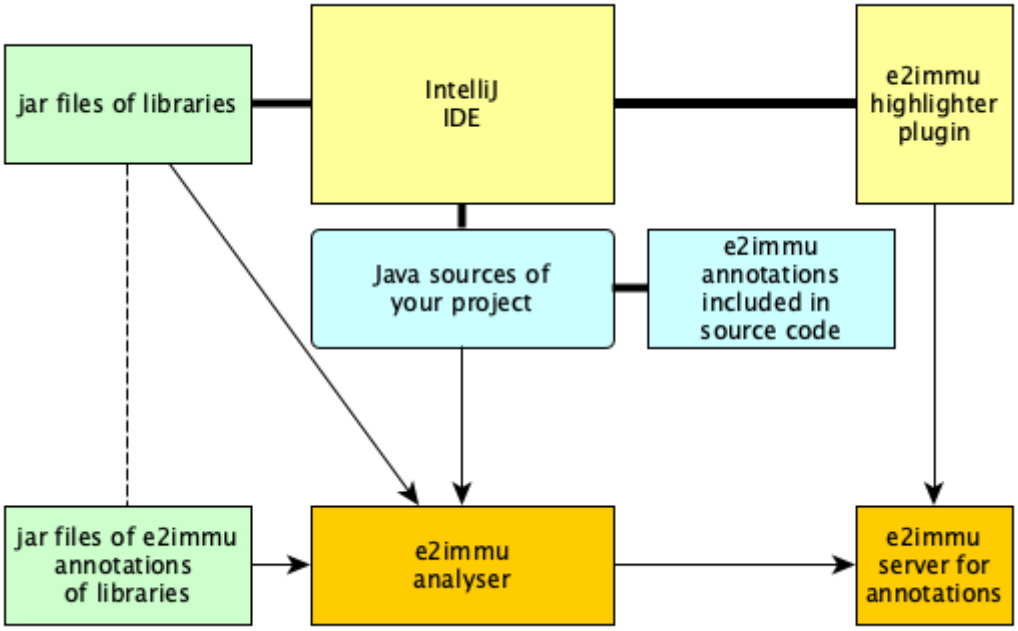

Figure 1. Basic use of e2immu

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.

1.2. 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.

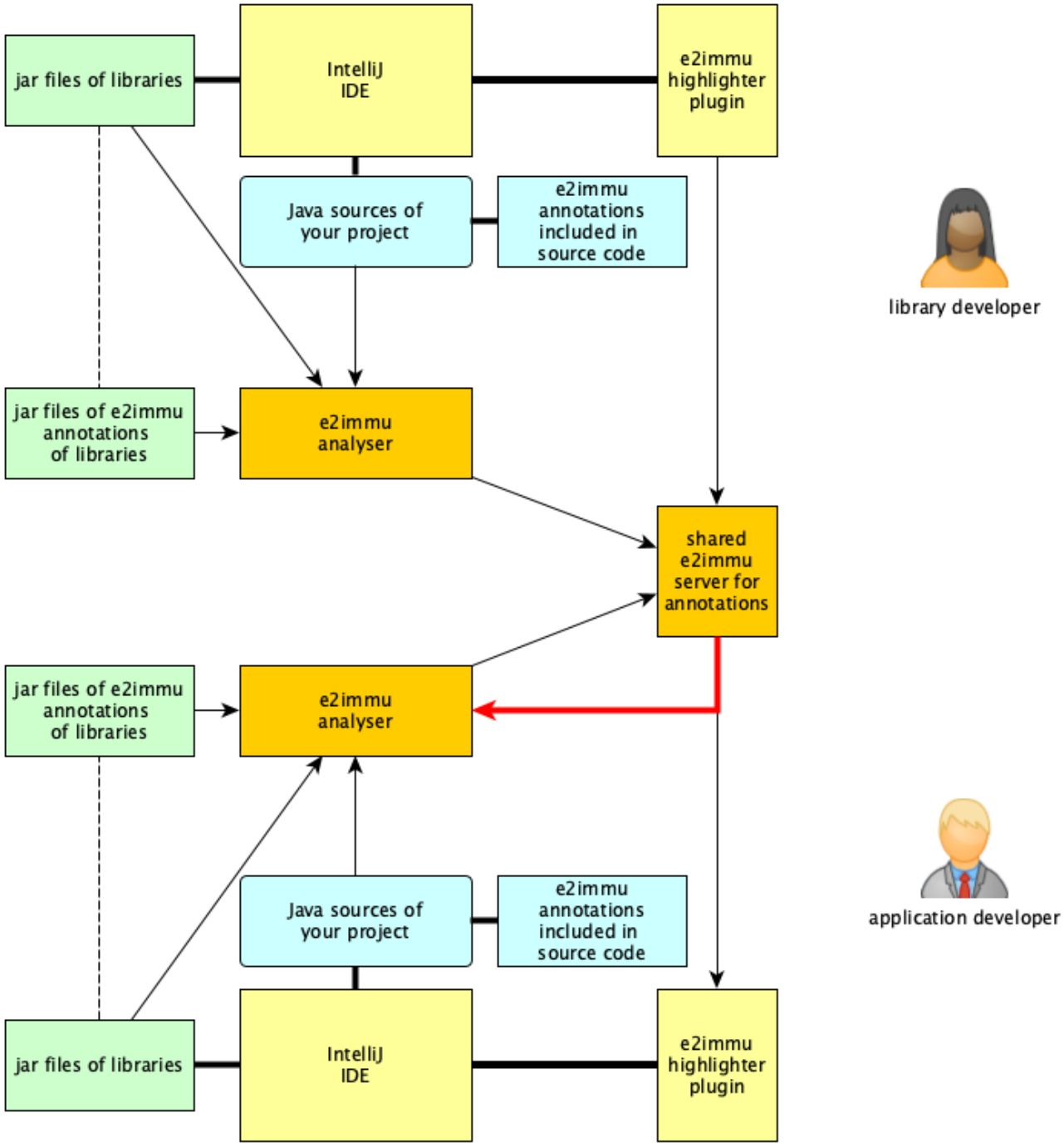

Figure 2. Fast turnaround use of e2immu

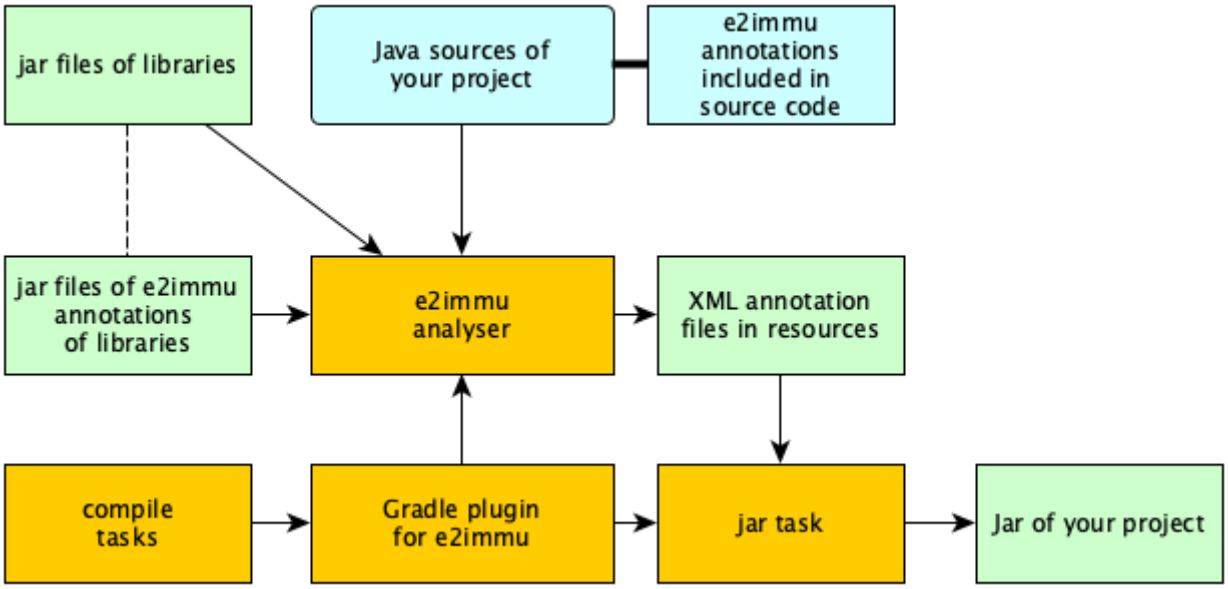

Figure 3. Add annotation.xml files to your jar

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.

1.3. The e2immu analyser 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
- `--ignore-errors`: do not end the analyser in an error state when errors have been raised by the analyser. They are still uploaded when the `--upload` option is activated.

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 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. 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.

1.4. Using 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.

2. Annotated APIs

2.1. Preparing an AnnotatedAPI file

Need to speed up.

The analyser's command line interpreter (CLI) provides options for generating a template AnnotatedAPI file from a library.

The following example, taken from the `e2immu/annotation-store` project, contains the Gradle build 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`.

2.2. The file format

AnnotatedAPI files are standard Java files, they will be inspected by a standard Java parser, so all standard 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.

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; }
    ...
```

2.3. 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.

3. Visualising immutability

3.1. Inheritance rules of annotations

As a general rule, we would like to impose that any class that implements an interface, inherits all the annotations of that interface. For example, if `java.util.List` is a container, then we want any implementation of that interface to be a container too.

Frequently used JDK classes such as `java.lang.String` are marked `final` themselves, so that they

cannot be subclassed. But in general, it seems a bad idea to weaken contracts. On the other hand, we can mark `java.lang.Object` as `@E2Immutable` , while obviously not all its derived classes can be of that type.

But the marks for eventually final and immutable classes push us to add some rules. Consider the `Verticle` interface and parts of its default implementation `AbstractVerticle` (please refer the section on Vertx integration TODO for an in-depth treatment):

Excerpts and annotations of Verticle.java and AbstractVerticle.java

```
@EventuallyFinal(after = "init", framework = true)
interface Verticle {

    @Mark("init")
    void init(@NullNotAllowed Vertx vertx, @NullNotAllowed Context context);

    @NotAllowed(before = "init")
    @NotModified
    Vertx getVertx();

    @NotAllowed(before = "init")
    void start(Promise<Void> startPromise) throws Exception;

    @NotAllowed(before = "init")
    void stop(Promise<Void> startPromise) throws Exception;
}

public abstract class AbstractVerticle implements Verticle {
    protected Vertx vertx;
    protected Context context;

    @Override
    public Vertx getVertx() {
        return vertx;
    }

    @Override
    public void init(Vertx vertx, Context context) {
        this.vertx = vertx;
        this.context = context;
    }
    ...
}
```

Our primary motivation here is to make sure that `vertx` and `context` are never assigned to in code derived from `AbstractVerticle`: the code analyser will ensure they become effectively final. On top of that, we can try to monitor calls the `getVertx` method before the verticle has been registered, and calls to `init` from outside the framework.

Now, inside a class that derives from `AbstractVerticle`, a similar situation arises:

MyVerticle.java

```
@EventuallyFinal(after = "start", framework = true)
public class MyVerticle extends AbstractVerticle {
    private HttpServer server;
    private JsonObject configuration;

    @Override
    @Mark("start")
    public void start(Promise<Void> startPromise) {
        server = vertx.createHttpServer(); ①
        ConfigRetriever retriever = ConfigRetriever.create(vertx, options); ①
        retriever.getConfig(ar -> {
            if (ar.failed()) {
                startPromise.fail("Cannot read config");
            } else {
                configuration = ar.result();
                startPromise.complete(); ②
            }
        });
    }

    private void handleRequest(RoutingContext routingContext) {
        textResult(routingContext, "a=" + configuration.a);
    }
}
```

① The `vertx` field has been initialised, is effectively final, and is non-null.

② Logical exit point of the method: only after having executed this handler, which assigns to `configuration`, the rest of the program can continue.

The two fields in this subclass will become effectively final, but, crucially, later than the `vertx` field. The `handleRequest` method will be only be called after the `start` method logically ends.

Should we treat the effective finality `AbstractVerticle` and `MyVerticle` as independent? Or do we want to impose `@Final` on any concrete implementation of `AbstractVerticle`? My intuition would say *yes*, we should do that, because of the nature of the class imposed on `MyVerticle` by the `Verticle` interface: it is going to be a class which has a start/stop lifecycle, instantiated by a framework, where code runs in the context of the framework and services started in that framework. Instances of this class should not be passed on; public methods should not be available for 'outsiders'. In most situations, the start/stop nature is binary, and while it may have a protracted starting phase, it will not become a start/phase 1/phase 2/stop lifecycle. But there are too many other examples where this would seem impractical. Therefore, we'll stick to

Summary of inheritance rules

As a general rule, a class inherits all annotations from directly implemented interfaces, but not from a possible parent class and its interfaces implemented. An interface that extends another interface does not inherit the annotations from its ancestor.

However, methods inherit the annotations of methods they override.

Marks for @Final and @E2Immutable annotations are restricted to the methods listed in the type that has the annotation, and the fields linked to them.

Reference

4. Concepts

4.1. Modification

4.2. Containers

4.3. Linking and independence

4.4. Immutability

4.4.1. Level 1 immutable

Terminology used:

Variable field
: marked `@Variable`

Effectively final field
: marked `@E1Immutable` or `@E1Container` .

Eventually final field
: marked `@E1Immutable` or `@E1Container` , with `after=xxx` parameter

A type is level 1 immutable when all its fields are effectively final. 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.

4.4.2. Implicitly immutable types

Terminology:

Implicitly immutable content
: all the fields of a type which are of implicitly immutable type.

4.4.3. Level 2 immutable

A type is level 2 immutable when

1. it is level 1 immutable, i.e., all its fields are effectively final
2. all fields are not modified
3. if a field is not of implicitly immutable type, it must be either private, or level 2 immutable
4. all constructors and non-private methods are independent of the fields

A type is eventually level 2 immutable when after executing a marked method, the fields become effectively final and or not modified. 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 or modify the fields.

4.4.4. Dynamic type annotations

4.5. Eventual immutability

4.6. Higher-order modifications

4.7. Miscellaneous

4.7.1. 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.

4.7.2. Statement time

Technically important for variable fields (Level 1 immutable).

4.7.3. Singleton classes

4.7.4. 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.

4.7.5. Extension classes

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

5. Overview of annotations

5.1. Annotation modes

Depending on where you find yourself in the continuum between object-oriented programming and functional programming, or, put differently, almost never following the container and level 2 immutable rules, or mostly adhering to them, you may wish to be either notified for following the rules, or warned by the analyser for breaking the rules. It is in this spirit that we envisage working in

Green mode

we assume that containers and level 2 immutable objects are sparse, and want to highlight when you follow the rules.

Red mode

when the majority of your types are level 2 immutable, and non-containers are rare, it may be more interesting to be warned when a type is *not* following the rules.

Because annotations often occur in an opposing pair, the choice of a mode can help to reduce information overload. For example, in red mode, @MutableModifiesArguments will be prominently displayed, and @E2Container will be "invisible". Similarly, @Modified is highlighted and @NotModified will be plain.

In green mode, we want @Container and @E2Container to be visible, while "normal" types are unmarked in black. The @NotModified will be highlighted, while @Modified remains plain.

5.2. Inheritance of annotations

In general, the analyser allows a method to do better than the method it overrides. It will raise an error when the overriding method is `@Modified` , where the original is `@NotModified`. TODO

5.3. 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.

5.3.1. @AllowsInterrupt

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

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

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

External methods not annotated will not interrupt.

5.3.2. @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` . 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.

Mode

not immediately relevant

5.3.3. @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.

Mode

This annotation has no opposite.

Example

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

```
@E2Container
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;
    }
}
```

5.3.4. @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 methods. See Containers for an in-depth discussion.

Mode

Use this annotation in green mode. 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.

5.3.5. @Dependent

Basic ✔	Immu ✔	Contract ✔	Type ✔	Field ✖	Method ✔	Param ✔

Summary

Annotation used to indicate that the type's fields link to the method's parameter or return value, or the constructor's parameters. This annotation is only present for fields that are not of implicitly immutable type. Additionally, on methods, the analyser only computes the annotation when the method is `@NotModified`.

Mode

Use this annotation in the red mode. Its opposite is `@Independent` .

Example

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

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

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

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

A similar example is in @E1Immutable.

5.3.6. @Dependent1

Basic ✖	Immu ✖	Contract ✖	Type ✖	Field ✖	Method ✔	Param ✔

Summary

As one of the Higher-order modifications annotations, `@Dependent1` 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;
    }
}
```

5.3.7. @Dependent2

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

Summary

This annotation is one of the Higher-order modifications annotations. It is only computed for independent parameters or methods. When computed on a parameter, it indicates that part of the implicitly immutable content of the argument will be assigned to the fields of the method's type. When computed on a method, it signifies that part of the implicitly immutable content of the return value is assigned to the fields of the method's type.

This annotation is central to iteration over the implicitly immutable content of a type.

Mode

This annotation has no opposite. By definition, implies `@Independent` .

Example

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

```
boolean Collections.addAll(@Dependent2 Collection<? extends E> coll);

@Dependent2
Stream<E> Collections.stream();
```

5.3.8. @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.

Mode

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

```
@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;
    }
}
```

5.3.9. @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.

Mode

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);
    }
}
```

5.3.10. @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 2 immutable and Containers.

Mode

This annotation is the default in the red mode.

Example

```
```

5.3.11. @E2Immutable

Basic ✔	Immu ✔	Contract ✔	Type ✔	Field ✔	Method ✔	Param ✔

Summary

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

Mode

This annotation is the default in the red mode.

Details

Level 2 immutability adds extra restrictions on top of level 1 immutability:

1. all fields must be not modified;
2. all fields of support data types must be either private, or level 2 immutable themselves;
3. all non-private methods and constructors must be marked `@Independent` , i.e.,
    a. in the case of constructors, the parameters must not link to the fields of support data types;
    b. in the case of methods, neither the return value nor the parameters must link to the fields of support data types. A consequence of requirement of not modified fields, is that non-private methods cannot be modifying.

5.3.12. @ExtensionClass

Basic ✔	Immu ✘	Contract ✘	Type ✔	Field ✘	Method ✘	Param ✘

5.3.13. @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.

5.3.14. @Finalizer

5.3.15. @Fluent

Basic ✔	Immu ✖	Contract ✔	Type ✖	Field ✖	Method ✔	Param ✖

Summary

This annotation indicates that a method returns `this`.

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.

5.3.16. @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.

5.3.17. @IgnoreModifications

Basic ✖	Immu ✔	Contract ✔	Type ✖	Field ✔	Method ✖	Param ✖

Summary

Helper annotation to mark that modifications on a field are to be ignored, because they fall outside the scope of the application.

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.

5.3.18. @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?

5.3.19. @Linked

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

Summary

Annotation to help debug the dependence system.

Mode

There is no opposite.

5.3.20. @Linked1

5.3.21. @Mark

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

5.3.22. @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.

5.3.23. @Modified1

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

Summary

This annotation is part of the higher-order modifications.

Mode

the opposite annotation is `@NotModified1`. In green mode, this one is the default; in red mode, `@NotModified1` is the default.

Example

Applying a c

5.3.24. @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);
        }
    }
}
```

5.3.25. @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

5.3.26. @NotModified1

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

Summary

This annotation is part of the higher-order modifications. Contracted to a parameter of an abstract type, it indicates that the abstract method cannot be implemented in a modifying way. Computed on parameters of any type with implicitly immutable content, it signifies that

Mode

It exists only in the green mode; there is no opposite. It can only be used for contracting, the analyser cannot generate it.

This annotation is a dynamic type annotation on functional types in fields, methods and parameters. The analyser can compute it in certain circumstances; in other cases, the user can show intent by requesting this property.

Note that because suppliers have no parameters, only modifications to the closure apply. Functional interfaces are always normally `@NotModified`: there are no modifying methods on them apart from the abstract method.

Example

We first show an example of a `@NotModified1` contract:

```
```

Here, the analyser computes the annotation:

```
```

5.3.27. @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` .

5.3.28. @NotNull1

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

5.3.29. @NotNull2

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

5.3.30. @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.

5.3.31. @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.

```
```

5.3.32. @PropagateModification

Basic ✖	Immu ✖	Contract ✖	Type ✖	Field ✔	Method ✖	Param ✔

Summary

This annotation is part of the higher-order modifications. The analyser adds this annotation when an abstract method without modification information is called on a parameter. This abstract method can be modifying or not, and in general it cannot be known which is the case. The annotation then informs the analyser that modifications need computing at caller-time.

The annotation is also possible on fields, in case the parameter becomes the effectively final value of a field.

Mode

There is no opposite.

Example

A typical implementation of `forEach` is a nice example:

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t); ①
    ...
}
@Container
public interface Set<T> {
    default void forEach(@PropagateModification Consumer<T> consumer) {
      for(T t: this) consumer.accept(t);
    }
    ...
}
```

① No modification information present on `accept`.

5.3.33. @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);
  }
}
```

5.3.34. @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;
    }
}
```

5.3.35. @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);
    }
}
```

5.3.36. @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;
  }
}
```

5.4. Relations between annotations

In this section we summarize the relations between annotations.

5.4.1. 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.

5.4.2. 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 `@NotNull1` , or `@NotNull2`), 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.

5.4.3. 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.

5.4.4. 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 `@NotNull1` , or `@NotNull2`) (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

5.4.5. On parameters

Opposites:

- a parameter is either `@NotModified` (green) or `@Modified` (red)
- a parameter is either `@NotNull` (or `@NotNull1` , or `@NotNull2`) (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.

5.4.6. Nullability

By convention,

- `@NotNull1` implies `@NotNull`
- `@NotNull2` implies `@NotNull` , `@NotNull1`
- etc.

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

5.4.7. 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`

Technical

6. The analyser

6.1. Inspection

Special naming conventions for creating Java classes that hold annotations and companion methods:

Types

A sub-type immediately below the primary type in the file can have a dollar at the end. This indicates that the sub-type becomes a primary type, with the name identical to the sub-type name without the dollar, and the package detailed by a static final field called `PACKAGE_NAME` in the primary type.

Methods

the rare occasion of adding an annotation to `getClass`, which is final in `java.lang.Object`, requires a dollar at the end which the inspector will remove.

Constructors

when the type has a $ at the end, the constructor has to follow. The inspector removes the dollar silently.

Note that the name of the primary type holding the sub-types is completely irrelevant.

6.2. Resolution

6.3. Analyser implementation

6.3.1. Code structure

`PrimaryTypeAnalyser`:

- starts from a `SortedType`, which contains one `PrimaryType` and a list of `WithInspectionAndAnalysis` (methods, fields, sub-types)
- creates and calls `TypeAnalyser`, `MethodAnalyser`, `FieldAnalyser`, and assigns `TypeAnalysis`, `MethodAnalysis`, `FieldAnalysis` to `TypeInfo.typeAnalysis`, `MethodInfo.methodAnalysis`, `FieldInfo.fieldAnalysis`.
- the `TypeAnalyser` creates, fills, and returns the `TypeAnalysis` object
- the `FieldAnalyser` creates, fills, and returns the `FieldAnalysis` object
- the `MethodAnalyser` creates, fills, and returns the `MethodAnalysis` object, but it also creates:
  - `ParameterAnalyser` objects, which fill, create and return `ParameterAnalysis` objects
  - `BlockAnalyser` objects, which fill, create and return `StatementAnalysis` objects. Each `StatementAnalysis` object corresponds recursively to a `Statement` in the `methodBody` block.

`TypeResolution` contains:

- the list of circular dependencies
- the set of implicitly immutable data types for this type

The `TypeAnalysis` object holds:

- a map of constant object flows
- a map of approved preconditions, ready for `@Mark` and `@Only`
- annotations and properties via the parent `Analysis` object

`FieldResolution` contains:

- a flag marking if the type is implicitly immutable

The `FieldAnalysis` object holds:

- the object flow of the field
- an effectively final value, which is an instance of `FinalFieldValue` unless we know better; it is not set when the field is `@Variable`
- a list of variables linked to the field (other fields, parameters)
- a set of internal object flows
- an error map for marking illegal assignments
- an error flag
- annotations and properties via the parent `Analysis` object

`MethodResolution` contains:

- a set of methods of the same type that this method (transitively) calls
- a flag noting whether it is part of construction
- a flag whether it creates an object of itself
- a flag indicating that it only calls static methods

The `MethodAnalysis` object holds:

- a single return value, if available
- summary information about `This` (mostly modification, we can maybe reduce this to single boolean?)
- a list of `StatementAnalysis` objects holding return value information. NOTE we can also store them in the method analyser, there may not be a need to hold on to them.
- a map of variables linked to fields and parameters
- a set of variables linked to the return value(s) of the method
- the joint precondition
- the precondition part for `@Mark` and `@Only`
- information about possible `@Mark` and `@Only` annotations (can maybe moved directly into the

annotations)

- a set of internal object flows
- the object flow of the method
- a fair number of error flags

The `StatementAnalysis` object holds:

- information about the statement: `Statement` reference, parent, next, list of blocks, indices and index
- a flag indicating if the statement (or block represented by the initial statement of the block) ecapes
- a flag indicating that the next statement is never reached
- an error flag
- a precondition represented by this statement
- the state of the method after evaluating this statement
- the precondition of the method after evaluating this statement
- if the statement has an expression part, the value of the expression
- replacement information
- a flag indicating if the method flow reaches the statement
- all variables referenced so far, and their properties (fully incremental)
- dependency information so far

Evaluation of expressions requires an `EvaluationContext` object which moves from statement to statement, applying resulting changes into the `StatementAnalysis` object after each evaluation. Some of these changes then trickle down to the method analyser which updates the `MethodAnalysis` object.

6.3.2. Circular dependencies

The analyser approaches primary types independently, albeit in a carefully computed order of dependency between them. When it detects a circular dependency between two or more primary types, it issues a warning to indicate that a different, less powerful modification detection algorithm kicks in. We consider circular dependencies bad programming practice; generally, interfaces can be introduced to remedy this. The manual modification annotations on the interface method effectively substitute for the assumptions that the less powerful modification algorithm makes.

Inside a primary type, the analyser deals with circular dependencies between the sub-types, the methods and the fields by running multiple iterations. The main reason it has to do this is that all fields are visible to all sub-types, even if they are marked `private`.

The processing list determines the order in which the analyser processes fields, methods and sub-types.

6.3.3. Nested classes

`@Final` : across all methods in the primary type

Parent or enclosing type when non-static must be have the property as well: `@E1Immutable` , `@E2Immutable` , `@Container` , `@Independent` .

What to do with abstract superclasses? They cause a problem because of the abstract methods, which can have any modification status. TODO think and implement.

Eventual? TODO think and implement.

6.3.4. Modification of a field

The code executes the following steps:

1. Wait until `@Final` or `@Variable` has been established. If `@Variable` , then the field becomes `@Modified` .
2. If the field is of a functional interface type, the field is `@NotModified` unless we can establish that there is an initializer or unambiguous constructor assignment with an explicit declaration (method reference, lambda, anonymous class implementation).
3. As a short-cut, determine that the field is `@NotModified` if its type is level 2 immutable. Whilst not technically necessary, this short-cut may resolve situations more quickly.
4. Wait until the field summaries in methods have been set. This typically takes exactly one iteration, because a method which reads a field is later in the processing list.
5. Wait until modification information is available for those methods which read the field. Importantly, we consider the methods (and SAM declarations of fields) of all types in the primary type.
6. Determine modification based on the modification information in the field summaries.

6.3.5. Modification of a method

The code executes the following steps:

1. If the method's field summaries contains an assignment to any field, inside the primary type, then the method is `@Modified` .
2. Wait until linking information (and hence modification information on fields) becomes available.
3. If any of the field summaries contains a marker for a modified field, then the method becomes `@Modified` . The analyser provides these marks when, amongst others, it sees a modifying method call on the field, or the field is an argument to a modifying parameter.
4. Next, check the modification status of `this` in `thisSummary`, when the analyser has observed a local method call. The method is `@Modified` when the analyser has observed a modification to any of the `this` objects (`super`, ...).
5. Then, check the marker for circular method calls or undeclared functional interfaces. In this situation, the modification status of the method depends on the presence of other modifying

methods, non-private fields, on dependent methods.

6. Finally, check the marker to copy the modification status from another method. The analyser issues this marker when the method passes on a functional interface argument to the other method.

6.3.6. Condition and state

A method can have restrictions on the parameter and field values called *preconditions*. In general, these restrictions end up as a boolean expression in `MethodAnalysis.precondition`. The exception to this rule are the not-null and size restrictions on parameters, which become properties during evaluation, and are written out as separate annotations.

Preconditions that participate in `@Mark` and `@Only` are stored in `MethodAnalysis.preconditionForEventual`. They are computed from normal preconditions.

As the analyser progresses through the blocks and statements, it keeps track of:

- the current *condition*, which is the boolean conjunction of all conditions in ever deeper `if` statements, (negated in the `else` block);
- the current *state*, which is the boolean conjunction of all restrictions on variables.

In the condition, top-level disjunctions indicate independent statements, while in the state, top-level conjunctions indicate independent statements:

```
void method1(String a, String b) {
    if(a == null || b == null) {
        // condition and state are: a == null || b == null;
        throw new NullPointerException();
    }
    // state is: a != null && b != null; empty condition
    ...
}
```

In Java, because of short-circuiting, this is functionally identical to the 'independent' form:

```
void method2(String a, String b) {
    if(a == null) {
        // condition and state are: a == null
        throw new NullPointerException();
    }
    // state is: a != null; empty condition
    if(b == null) {
        // condition is: b == null; state is: a != null && b == null
        throw new NullPointerException();
    }
    // state is: a != null && b != null; empty condition
    ...
}
```

Condition and state travel deeper inside the blocks:

```
void method3(String a, String b) {
    if(a == null) {
        // condition and state are: a == null
        if(b == null) {
            // condition and state are: a == null && b == null
            throw new UnsupportedOperationException();
        }
        // condition is: a == null; state is: a == null && b != null
        return;
    }
    // state is: a != null; empty condition
    ...
}
```

Both in `method1` and `method2`, the escape via a runtime exception introduces `@NotNull` annotations on the parameters. The analyser employs dedicated logic to ensure that in a second pass, it does not flag the condition in the `if` statement as a constant value. In `method3`, the conjunction in the condition after two successive `if` statements does not allow for individual not-null restrictions. The result is a precondition, annotated as `@Precondition("(not (null == a) or not (null == b))")`. Note that, perhaps counter-intuitively, if we were to replace the `return` statement with a `throws` statement, it would have `a == null` as condition, and not `a == null && b != null`.

The rules for adding and removing to condition and state are:

1. start the method with the state equal to the preconditions, if applicable;
2. when entering a conditional block, start a new `ConditionManager` with the statement's expression added to the current condition and state;
3. when a conditional block does not return, add the boolean complement to the state;
4. in case of assignments or modifying methods, clear the state (partially);
5. the state of parameters and effectively final fields travels up from inside blocks, but only if

these blocks are unconditional. This is most notably the case for a `synchronized` block.

6.3.7. Computation of @Mark, @Only

The presence of eventual properties (level 1 immutable, level 2 immutable, content not null, ...) follows from the computation of the `@Only` and `@Mark` annotations. Here, we document how the analyser computes them.

The analyser associates eventuality with a precondition on a field (or technically, on one or more fields); it labels the precondition with a mark string. Methods that guard against the precondition are *before* the mark, methods that guard against the boolean complement are *after* the mark. We define a guard here as the throwing of a run-time exception when the field's value does not satisfy the condition.

Each method holds information about such a precondition in `SetOnce<Eventual> eventual`. The `MethodAnalyser` computes the information in `computeOnlyMarkPrepWork` and `computeOnlyMarkAnnotate`.

Then, the `TypeAnalyser` combines the information of the methods in `analyseOnlyMarkEventuallyE1Immutable`; the end result of the whole computation resides in `SetOnceMap<String, Value> approvedPreconditions` in `TypeAnalysis`. The keys are the different preconditions that have been approved for eventuality computation, the values are the associated mark strings consisting of the variable names of the fields in the precondition. Note: the current implementation relies on the precondition to be lifted using an assignment rather than a content change; the code resides in the level 1 immutability check. As a consequence, it is currently not possible to use the size of a collection, for example, as a precondition.

The eventual annotation will receive a comma-separated list with all the marks in `approvedPreconditions`.

6.3.8. Statement analyser

Steps:

1. Create local variable (`for(T t: ts)`, assignment statement)
2. evaluate initialisers (classic `for`, `try` with resources, normal assignment statements). The results of the initialisers need to be known to the evaluation context, but cannot be permanent yet.
3. evaluate updates (classic `for`). The results of the updaters need to be known to the evaluation context, but cannot be permanent yet.
4. evaluate main expression (many statements). The results need to be known to the evaluation context.
5. specific `return` statement code, update method-level data;
6. specific `if`, `switch` code to check evaluations to constant
7. primary block, recursive call; merge back by calling `lift`
8. sub-blocks, recursive calls; merge back by calling `lift`
9. determine state after this statement
10. finally, make the results of the evaluation(s) permanent by calling `finalise`

6.3.9. Post- and pre-conditions

A non-modifying method without parameters can define an *aspect* of a type, like `size` or `length`. We define methods complementary to a normal class or interface method, using the naming convention *method name$action$aspect*. Let's call them companion methods for now.

Modification+aspect

for modifying methods, show how the aspect changes from before to after the modification.

Value (+aspect)

compute edge case values, and return the normal value otherwise. Aspect can be used, but does not have to be present.

Precondition

describe the precondition in terms of parameters or field values

Invariant

without aspect present, it has to be a condition on fields which holds all the time.

Invariant+aspect

this expression describes an invariant of the aspect. For example, `size >= 0`, before and after any method.

Postcondition

describe the postcondition of the fields after the modification. (This is the modification without aspect.)

Transfer (+aspect)

assign state to the return element, transferring it from the current object.

Generate

generate any other type of

Erase

we need to think about something that erases state. A `clear()` method on a collection should remove all `contains(x)` clauses.

Implementation issues:

1. Get rid of SIZE and everything surrounding it
2. Parsing the methods, and storing them in the type in a reasonable data structure. We'll work with the evaluated inline-functions; they have a translation mechanism already built in.
3. The system generates preconditions, ensure that they can be tested.
4. From a companion method we have to translate.

6.3.10. Nullity of fields

If the field is effectively final, it receives its values from

1. its initialiser
2. assignments in the constructor and methods that are exclusively part of the construction phase
3. modifying methods cannot change the value, but they can erase constructor information (`instance type X` instead of `new X(⋯)`)

If the field is variable, there can be assignments in public methods not part of the assignment phase.

Either way, nullity comes from both methods where an assignment take place, and methods where restrictions are placed on the nullity. The latter can take place in modifying and non-modifying methods.

The statement analyser of the first occurrence of a field in a method introduces a value for the field *only when both the field's value and its nullity are known*. We need to ensure here that this rule will not be the cause of endless delay loops.

The expected complication is in methods where there is no value yet for the field, but we still expect the nullity of the field variable to be computed. The field analyser must be able to grab this nullity independently of the value:

- for each method, it needs to know if nullity computations have not been halted. Typically, this is done in two stages:

6.3.11. Loop variables overview

Different situations

1. variable defined outside a loop
   a. read in the loop, but not assigned in the loop: normal situation
   b. assigned in the loop
2. variable defined in the loop statement. Either can be assigned inside the loop as well.
   a. `for(String s: strings)`
   b. `for(int i=0; i<n; i++)`

`localVariablesAssignedInThisLoop` needs to be frozen before we know the distinction between situations 1 and 2. As long as the latest assignment is before the loop, the variable in the loop is represented by `var$loop`. Once we're beyond the assignment, it becomes `var$loop$assigmentId`.

We make no distinction between `forEach` and `for`: in both cases, the variable is added to the `localVariablesAssignedInThisLoop` set. While we could force the variable to be `final` in the former case, we'll want different versions in case modifying methods are called on them.

In the `forEach` case, we assign a value, once, during the evaluation phase of the loop statement.

6.3.12. General property computation

We have decided on a very strict *write once* system: once a value has been determined, it cannot be changed anymore. Properties are written in a heavily controlled map of type `VariableProperties`. The value `Level.DELAY` is never written in the map, as it indicates that no value has been determined yet.

In the statement analyser, property values travel from one statement to the next. Therefore, the last statement often contains the relevant information for the other analysers to read.

6.3.13. Not-null property

Variables in the statement analyser can have not-null induced by the context (parameters in method calls, scope, etc). At the same time, fields and parameters can get not-null values from the field and parameter analyser. Combining both in a single, non-incremental value turns out to be impossible. Therefore, we split, for every variable, non-null in

1. `CONTEXT_NOT_NULL`: non-null value induced by context
2. `NOT_NULL_EXPRESSION`: non-null value coming from assignments and statement context
3. `EXTERNAL_NOT_NULL`: from the field analyser

The combination of `CONTEXT_NOT_NULL` and `EXTERNAL_NOT_NULL` is the property called `NOT_NULL_EXPRESSION`, which is read-only unless there is a contract not-null. It is delayed when one of its components is delayed.

While evaluating expressions, we use the property `NOT_NULL_EXPRESSION`.

The method analyser records the return value's not-null in `NOT_NULL_EXPRESSION`, read from the return variable's `NOT_NULL_EXPRESSION`. The field analyser writes in `EXTERNAL_NOT_NULL`, and reads from the variables' `CONTEXT_NOT_NULL` (when no assignment was made), and the assignment's values' `NOT_NULL_EXPRESSION`.

The parameter analyser maintains the two aspects: context from the statement analyser, and external from the field analyser.

The statement analyser uses the properties `CONTEXT_NOT_NULL_DELAY` and `CONTEXT_NOT_NULL_DELAY_RESOLVED` to control a delay on `CONTEXT_NOT_NULL`, when a method's not-null properties, where the variable occurs as argument, have not been analysed yet.

It is important to record the flow of the properties, and especially the mechanisms to break delays when no information is present.

In the statement analyser

Let us consider the following cases:

1. the variable occurs in the `ChangeData` of `apply` of the main expression: any delays are resolved to `NULLABLE`, when no context not-null delay is present
2. the variable does not occur in the `ChangeData`, but is still involved (say the local copy of a variable field `s$0`)

In the field analyser

In the parameter analyser

Once a context not-null is known in the statement analyser's last statement, the parameter's value can be set. Important: this is one-way traffic! The first statement does NOT copy this value into the 'INITIAL'. Once a value can come from a field, the parameter analyser will copy it.

6.3.14. Modification property

The property indicating modification of a variable has been split in the same way as the not-null property:

1. variables in statements use `CONTEXT_MODIFIED` and `MODIFIED_OUTSIDE_METHOD`, summarized in `MODIFIED_VARIABLE`.
2. the method's modification status is in `MODIFIED_METHOD`
3. a field's modification is in `MODIFIED_OUTSIDE_METHOD`
4. modification of `this` is recorded in `METHOD_CALLED`

The statement analyser uses the properties `METHOD_DELAY` and `METHOD_DELAY_RESOLVED` to control a delay on `CONTEXT_MODIFIED`, when a method's modification properties, where the variable occurs as argument or scope, have not been analysed yet.

Again let us record the flow of properties and delay-breaking mechanisms:

In the statement analyser

1. should normally be broken in `MethodLevelData`, compute modification.

In the parameter analyser

Entirely the same as for the not-null. === Statement analysis

There is one `StatementAnalysis` object per statement, so a method consists of a chain of statements. All variables that exist in statement *i* will exist in statement *i+1* as well. A method's last statement is the place to find the final value of non-local variables.

Statement analysis is carried out until all statement analysers reach status `DONE`. A statement analyser *i+1* cannot reach done unless *i* is already has reached `DONE`. Four main aspects control delays in the statement analyser:

1. the status of the analyser of the previous statement
2. the value of variables
3. the linked variable set of variables
4. the three expressions in the condition manager: condition, state, pre-condition

6.4. Variables

We recognize the following types of variables:

- fields, where it is important whether the field is effectively final, variable, or as yet undecided

in the first iteration(s)

- parameters, which are non-assignable
- local variables, among which copies of variable fields and versions of variables used in loops but defined outside them
- `this`, `super`
- dependent variables, like the cells in an array; mostly treated as local variables
- one return variable for each method, which is assigned a value in a `return` statement

Some variables never leave the statement analyser(s). However, there are important information flows:

The method analyser reads

1. the fact whether fields have been assigned to help decide on the modification of the method;
2. the value of the return variable of the last statement to decide on the method's return value;
3. the modification properties of the `this` and `super` variables to help decide on the modification of the method.

The field analyser reads

1. almost all information about the fields occurring in the last statement

The parameter analyser reads

1. properties of the parameter variables in the last statement
2. properties and from the field analyser in case a parameter has been assigned to a field or linked to a field

The statement analyser copies back information

1. from the field analyser, about fields occurring in the statement
2. from the parameter analyser

It is important to note that there is very limited 'circular' information flow.

Fields are created in the statement analyser only when they occur. Given the write-once nature of the design, their value remains undecided until the field analyser has decided whether the variable is effectively final or not. In the latter case, the actual value of a field can change from one statement to the next (or even within statements); this requires additional logic.

Modification and linking is a one-way flow from

1. main computation is in the statement analyser
2. from the statement analyser to the field analyser: the computation of `CONTEXT_MODIFIED`, closely coupled with the computation of the linked variables of a field variable. All variables start in the first statement with an empty set of linked variables, and a `Level.FALSE` value for the context modification, unless the user explicitly contracted the modification status of the parameter with

an annotation.

3. from the field analyser to parameter analyser: in the form of the `MODIFIED_IN_METHOD` property, which ends up combined with `CONTEXT_MODIFIED` into the parameter's final `MODIFIED_VARIABLE` property determining its `@Modified` / `@NotModified` annotation.

External not-null (ENN) travels from the parameter analyser back to the statement analysers, after values of fields have been established. Because a statement reaches `DONE` status as soon as value and linked variables have been established, an explicit delay system exists to make sure that statemens which involve fields with a delayed ENN are revisited once.

6.4.1. Variable fields

Variable fields interact with synchronisation. A time increase is equivalent to an opportunity for another thread to update the content of variable fields.

The analyser can determine if a method increases time when called. Calls to non-private methods always increase time because the user can override them. Certain statements increase time (`try-catch`, `synchronized`). The method resolver determines if a method increases time or not.

The statement analyser keeps track of time. Determining time obviously depends on the methods being called, but because the method resolver determines if a method increases time, this happens in a deterministic way in the first iteration of the analyser.

6.4.2. Different variable states

Each statement analyser keeps up to three different states for each variable:

Initial state

the initial state is either the state as inherited from the previous statement, or the variable's initial state. The latter situation occurs for every variable introduced in the first statement of each method.

Evaluation state

the state after evaluation of the main expression of the statement. Obviously, not all statements have such an expression, or the variable may not occur at all in the expression, in which case the evaluation state can be skipped.

Merge state

if the statement has sub-blocks, a summary state exists to allow progressing to the next statement.

Information always travels from initial, potentially via evaluation, to merge; never in a different order! The analyser computes the evaluation of statement *i* before starting the statement analyser of the sub-blocks, and computes the merge of statement *i* after all statement analysers in all sub-blocks have been activated.

We use suffixes `-C`, `-E` and `:M` respectively, which means that the following comparisons hold alphanumerically:

```
0-C < 0-E < 0.0.0-C < 0.0.0-E < 0.0.0:M < 0.0.1-C < ... < 0:M < 1-C < ...
```

6.4.3. Data stored for a variable

The `VariableInfo` objects hold the following data:

Variable

the variable being stored. Always present from the start.

Assignment ID

A marker for when this particular `VariableInfo` object is created. It contains the statement index, and the level in the container, separated by a colon. This value is present from the creation of the `VariableInfo` object, even though it can vary across `VariableInfo` objects for the same variable.

Value

starts off with `NO_VALUE` (not assigned), and becomes immutable when it receives a value.

Linked variables

start off with `null`, and becomes immutable when it receives a value. Bar one exception, the linked variables come with the assignment of a value, potentially in a later iteration. The linked variables of a variable field change when the analyser creates a new local copy for reading, i.e., outside an assignment.

Properties

a properties map, with `Level.DELAY` being the default value.

Statement time

a constant `NOT_A_VARIABLE_FIELD` for all non-variable fields. For fields, starts off with `VARIABLE_FIELD_DELAYED` (not assigned) until we know if the field is variable or not. Then, for variable fields, it stores the statement time of the latest assignment.

Object flow

generally also set during an assignment. The value `NO_FLOW` is used when no value as been set.

All these fields are immutable once they have been set, except the `properties` map, which is incremental. The `READ` and `ASSIGNED` properties have the following special behaviour: ... TODO

6.4.4. Tests: Basics_0

```
private final String effectivelyFinal = "abc";
public String getEffectivelyFinal() {
    return effectivelyFinal;
}
```

Analysers:

- Statement analyser, iteration 0, statement 0, return statement. The field `effectivelyFinal` does not yet exist in the list of variables; the change data yields a `FieldReference` pointing to a value of `NO_VALUE`, and a marker that the variable is read in this statement.
- Method analyser: TODO
- Field analyser, iteration 0: it is immediately clear that the field is explicitly final, and that the initialiser is level 2 immutable. Therefore, an effectively final value can be assigned: `"abc"`. Everything can be concluded.
- Statement analyser, iteration 1: at initialisation, the variable is created, and the effectively final value is filled. The main evaluation can therefore return a value. The return variable can also be set.
- Method analyser: can conclude all except for the approved pre-conditions, which will remain unresolved?

6.4.5. Tests: Basics_1

```
public final Set<String> f1;

public Basics_1(Set<String> p0, Set<String> p1, String p2) {
    Set<String> s1 = p0;
    this.f1 = s1;
}

public Set<String> getF1() { return f1; }
```

Iteration 0, statement analyser for the constructor:

- statement 0: at initialisation, `this` and the three parameters are created as variables, in the `INITIAL` phase. The local variable `s1` is created, and assigned to the first parameter. Both exist in `EVAL` phase, one with an assignment id, the other with a read id.
- statement 1: a field reference is created, and assigned the value `p0`. The field is linked to this parameter.

Iteration 0, statement analyser for the getter: * statement 0: the field does not yet exist in the list of variables. It is marked as read during evaluation, and will be created during the initialisation step of the next iteration. The return variable has been created at initialisation, but for now has no value.

Iteration 0, field analyser:

- the field is marked as effectively final.
- because there is no /

Iteration 1, statement analyser for getter: * the field `f1` has the value `instance type Set<String>`, not linked locally. * the return variable points to the field `f1`.

IMPORTANT

We could also return the value of `f1` in the return statement, but at the moment we don't think it matters that much.

Iteration 1, field analyser: Modification is determined, as is nullability.

Iteration 2: parameter analyser: the `@NotNull` of the field travels to the parameter.

6.4.6. Tests: Modification_0

```
public final Set<String> set1 = new HashSet<>();

public void add(@NotNull String v) {
    set1.add(v);
}
```

Iteration 0, statement analyser in `add`, statement 0: * the expression evaluates to `NO_VALUE` as `set1` is not a known variable

Iteration 0, field analyser: * the field is explicitly final. We don't have to wait for modification since the field is public.

Iteration 1, statement analyser in `add`: Value, linked variables, and `@Modified` determined.

6.4.7. Breaking delays

Break that only works partially, in iteration 1+ assignment from analyser to variable:

```
 if (initialValue.expressionIsDelayed && notYetAssignedToWillBeAssignedToLater) {
    String objectId = index + "-" + fieldReference.fieldInfo.fullyQualifiedName();
    Expression initial = NewObject.initialValueOfField(objectId,
        primitives, fieldReference.parameterizedType());
    initialValue = new ExpressionAndDelay(initial, false);
}
```

Infinite when break not present, with break the analyser works fine:

```
public class FirstThen_0<S> {

    private S first;

    public FirstThen_0(@NotNull S first) {
        this.first = Objects.requireNonNull(first);
    }

    public void set() {
        if (first == null) throw new UnsupportedOperationException("Already set");
        first = null;
    }
}
```

Infinite when break not present, too early with break:

```
public class Singleton_7 {

    private static boolean created;

    public Singleton_7(int k) {
        if (created) throw new UnsupportedOperationException();
        created = false;
    }
}
```

Infinite either way:

```
public class Project_2 {

    static class Container {
        final String value;

        public Container(String value) {
            this.value = value;
        }
    }

    private final Map<String, Container> kvStore = new ConcurrentHashMap<>();

    public String set(String key, String value) {
        Container prev = kvStore.get(key);
        if (prev == null) {
            new Container(value); // removing container also solves the problem
        }
        return prev.value; // cause of the problem (change to key or constant solves
the issue)
    }
}
```

7. Supporting tools

7.1. 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.

7.2. 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"}
```

7.3. 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.

7.3.1. 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
`@Dependent` (when support data)	brown	brown

annotation on constructor	green mode	red mode
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

7.3.2. 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*.

7.3.3. 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?

8. Copyright and License