The Road to Immutability

Table of Contents

Effective and eventual immutability with *e2immu*, a static code analyser for Java.

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

Dedication

This work is dedicated to those who have had, or are still having, a difficult pandemic.

1. Introduction

This document aims to be a logical walk through of the concepts of the *e2immu* project. It does not intend to be complete, and is not structured for reference.

The overarching aim of the *e2immu* project is to improve every day programming by making code more readable, more robust, and more future-proof. More concretely, the project focuses on adding various forms of immutability protections to your Java code base, by making the immutable nature of the types more visible.

Why Java? As a widely used object-oriented programming language, it has evolved over the years, and it has been increasingly equipped with functional programming machinery. It is therefore possible to write Java code in different styles, from overly object oriented to almost fully functional. Combine this with lots of legacy code, both in house and in libraries, and many large software projects will end up mixing styles a lot. This adds to the complexity of understanding and maintaining the code base.

Why immutability? An important aspect of understanding the code of large software projects is to try to assess the *object lifecycle* of the data it manages: when and how are these objects modified? In object-oriented programming, full of public getters and setters, objects can be modified all the time. In many a functional set-up, objects are immutable but new immutable versions pop up all the time. Java allows for the whole scala from object-oriented to functional, and the whole ecosystem reflects this choice.

An easy way to envisage the life cycle of an object is to assume that it consists of a building phase, followed by an immutable phase. We set out to show that there are different forms of immutability, from very strict deep immutability to weak guarantees of non-modification, that can be made visible in the code. We believe that code complexity can be greatly reduced when the software engineer is permanently aware of the modification state of objects.

The *e2immu* project consists of a set of definitions, a static code analyser to compute and enforce rules and definitions, and IDE support to visualize the results without cluttering. Using *e2immu* in your project will help to maintain higher coding standards, the ultimate beneficiary being code that will survive longer.

A *lack of references* in this version of the work is explained by the fact that this is my first foray into the world of static code analysers, and theory of software engineering and programming languages. Academically coming from the theory of machine learning, I spent a decade and a half writing software and managing teams of software engineers. This work builds on that practical experience alone. I did not consult or research the literature, and I realise I may be duplicating quite a lot here. I only, explicitly, want to mention JetBrain's brilliant IntelliJ IDEA, which acts as my gold standard.

2. Assumptions

We discuss the Java language, version 8 and higher. We have already indicated that we believe that Java offers too much freedom to programmers. In this section, we impose some limits that are not critical to the substance of the discussion, but facilitate reasoning. Think of them as low-hanging fruit programming guidelines:

- Exceptions do not belong to the normal programming flow; they are meant to raise situations that the program does not want to deal with.
- Parameters of a method cannot be assigned; we act as if they always have the `final` modifier. The simple way around is to create a new local variable, and assign the parameter to it.
- We make no distinction between the various non-private access modifiers (package-private, protected, public). Either a field, method or type is private, or it is not.
- Static fields can only be used for non-constant purposes in very limited circumstances; one example is a variable to check enforce a singleton. The whole topic of using statics to access thread-local variables is outside the scope of *e2immu* at the moment.
- Methods must be static if they do not access non-static fields, and do not implement or overload some interface or class method.
- Synchronization is orthogonal to the data of the program; whilst it may have an influence on *when* certain code runs, it should not be used to influence the semantics of the code.

The *e2immu* code analyser warns for many other doubtful practices, as detailed in the user manual.

3. The purpose of annotations

In this document we will add many annotations to the code fragments shown. We are acutely aware annotations clutter the code and can make it less readable. Some IDEs, however, like

JetBrains' IntelliJ IDEA, have extensive support to make working with annotations visually pleasing.

The *e2immu* code analyser computes almost all the annotations that we add to the code fragments in this document. The complementary IDE plugin uses them to color code types, methods and fields. Except when the annotations act as a contract, in interfaces, they do not have to be present in your code.

Explicitly adding the annotations to classes can be helpful during software development, however. Say you intend for a class to be immutable, then you can add the corresponding annotation to the type. Each time the code analyser runs, and the computation finds the type is not immutable, it will raise an error.

Explicit annotations also act as a safe-guard against the changing of semantics by overriding methods. Making the method `final`, or the type `final`, merely *prohibits* overriding, which is typically too strong a mechanism.

The final situation where explicit annotations in the code are important, is for the development of the analyser. We add them to the code as a means of verification: the analyser will check if it generates the same annotation at that location. A number of annotations serve no other purpose than debugging: `@Linked`, `@Independent`, `@Constant`, ...

4. Level 1 immutability

Let us start with a definition:

> **Definition**: We say a field is **effectively final** when it either has the modifier `final`, or it is not assigned to in methods that can be transitively called from non-private (non-constructor) methods.

The code analyser annotates with `@Final` in the latter case; there is no point in cluttering with an annotation when the modifier is already there. It annotates fields that are not effectively final with `@Variable` .

This definition allows effectively final fields to be assigned in methods accessible only from the constructor:

Example 1, effectively final, but not with the `final` modifier

```
class EffectivelyFinal1 {
    @Final
    private Random random;

    public EffectivelyFinal1() {
        initialize(3L);
    }

    private void initialize(long seed) {
        random = new Random(seed);
    }

    // no methods access initialize()

    public int nextInt() {
        return random.nextInt();
    }
}
```

Obviously, if the same method is also accessible after construction, the field becomes variable:

Example 2, the method setting the field is accessible after construction

```
class EffectivelyFinal2 {
    @Variable
    private Random random;

    public EffectivelyFinal2() {
        reset();
    }

    public void reset() {
        initialize(3L);
    }

    private void initialize(long seed) {
        random = new Random(seed);
    }

    public int nextInt() {
        return random.nextInt();
    }
}
```

Note that it is perfectly possible to rewrite the first example in such a way that the `final` modifier can be used. From the point of view of the code analyser, this does not matter. The wider definition will allow for more situations to be recognized for what they really are.

When an object consists solely of primitives, or deeply immutable objects such as `java.lang.String`, having all fields effectively final is sufficient to generate an object that is again deeply immutable.

Example 3, an object consisting of primitives and a string.

```
class DeeplyImmutable1 {
    public final int x;
    public final int y;
    public final String message;

    public DeeplyImmutable1(int x, int y, String message) {
        this.message = message;
        this.x = x;
        this.y = y;
    }
}
```

Example 4, another way of being effectively final

```
class DeeplyImmutable2 {
    @Final
    private int x;
    @Final
    private int y;
    @Final
    private String message;

    public DeeplyImmutable2(int x, int y, String message) {
        this.message = message;
        this.x = x;
        this.y = y;
    }

    public String getMessage() {
        return message;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

Examples 3 and 4 are functionally equivalent: there is no way of changing the values of the fields once they have been set. In the real world there may be a reason why someone requires the getters. Or, you may be given code as in Example 2, but you are not allowed to change it. Whatever the reason, the code analyser should recognize effective finality.

Note that we will not make a distinction between any of the different non-private access modes in Java. Only the private modifier gives sufficient guarantees that no reassignment to the fields is possible.

We now have observed that for the purpose of defining immutability, having all your fields effectively final can be sufficient in certain circumstances. We use this as the basis for the first level of immutability:

> **Definition**: We call a type **effectively level 1 immutable** when all its fields are effectively final.

The code analyser annotates level 1 immutable types with `@E1Immutable` . Types that are not `@E1Immutable` because they have at least one `@Variable` field, are annotated either `@MutableModifiesArguments` or `@Container` , depending on properties of the methods' parameters to be explained later.

Note that as of more recent versions of Java, the `record` type enforces explicitly final fields, along with additional support for equality and visibility. Any `record` will at least be `@E1Immutable` .

As above with effective finality, the term *effective* is present to make a distinction between formal immutability, and immutability that the code analyser computes. It will also serve to distinguish from *eventual* immutability, where (in this case) the finality will be achieved only after the code reaches a certain state. More on this later, but here is a first example of an eventually level 1 immutable type:

Simplified version of `SetOnce`

```
@E1Immutable(after="t")
class SetOnce<T> {
    private T t;

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

    @Only(after="t")
    public void get() {
        if(this.t == null) throw new UnsupportedOperationException("Not yet set");
        return this.t;
    }
}
```

Once a value has been set, the field `t` cannot be assigned anymore.

We have just observed that if one restricts to primitives and types like `java.lang.String`, level 1

immutability is sufficient to guarantee deep immutability. It is not feasible, and we do not wish to, work only with deeply immutable objects. Moreover, it is easy to see that level 1 immutability is not enough to guarantee what we intuitively may think immutability stands for:

Example 5, level 1 immutability does not guarantee intuitive immutability

```
@E1Immutable
class StringsInArray {
    private final String[] data;
    public StringsInArray(String[] strings) {
        this.data = strings;
    }
    public String getFirst() {
        return data[0];
    }
}

...
String[] strings = { "a", "b" };
StringsInArray sia = new StringsInArray(strings);
Assert.assertEquals("a", sia.getFirst());
strings[0] = "c"; ①
Assert.assertEquals("c", sia.getFirst()); ②
```

① External modification of the array.

② As a consequence, the data structure has been modified.

To continue, we must first understand the notion of modification.

5. Modification

> **Definition**: a **method is modifying** if it causes an assignment in the object graph of the fields of the object it is applied to.

We use the term 'object graph' to denote the fields of the object, the fields of these fields, etc., to arbitrary depth.

Consequently, a method is not modifying if it only reads from the object graph of the fields. The code analyser uses the annotations `@NotModified` and `@Modified` . They are exclusive, and the analyser will compute one or the other for every method of the type. All non-trivial constructors are modifying, so we avoid clutter by not annotating them.

It follows from the definition that directly assigning to the fields also causes the `@Modified` mark for methods. As a consequence, setters are `@Modified` , while getters are `@NotModified`. Consider:

```
class Counter {
    @Variable
    private int counter;

    @NotModified
    public int getCounter() {
        return counter;
    }

    @Modified
    public int increment() {
        counter += 1;
        return counter;
    }
}

class CountedInfo {
    @Final
    @Modified
    private final Counter counter = new Counter();

    @Modified
    public void printInfo(String info) {
        System.out.println("Message " + counter.increment() + ": "+info);
    }
}
```

We also see in the example that the `printInfo` method is `@Modified` . This is because it calls a modifying method on one of the fields: `increment`.

Moving from methods to parameters and fields, keeping the same two annotations,

> **Definition**: The analyser marks a **parameter** as **modified** when the parameter's method applies an assignment or modifying methods on the object that enters the method via the parameter. This definition holds with respect to the parameter's entire object graph.

We will apply a similar reasoning to a field:

> **Definition**: The analyser marks a **field** as **modified** when at least one of the type's methods, transitively reachable from a non-private non-constructor method, applies at least one assignment to or modifying method on this field.

Let us start by agreeing that the methods of `Object` and `String` are all `@NotModified`. This is pretty obvious in the case of `toString`, `hashCode`, `getClass`. It is less obvious for the `wait` and other synchronization-related methods, but remember that as discussed in the Assumptions, we exclude synchronization support from this discussion.

Note also that we cannot add modifying methods to the type `DeeplyImmutable1` in Example 3.

Proceeding, let us also look at (a part of) the `Collection` interface, where we've restricted the annotations to `@NotModified` and `@Modified` . While in normal classes the analyser computes the annotations, in interfaces the user stipulates or *contracts* behaviour by annotating:

Modification aspects of the Collection interface

```
public interface Collection<E> extends Iterable<E> {
    @Modified
    boolean add(E e);

    @Modified
    boolean addAll(@NotModified Collection<? extends E> collection);

    @NotModified
    boolean contains(Object object);

    @NotModified
    boolean containsAll(@NotModified Collection<?> c);

    @NotModified
    void forEach(Consumer<? super E> action);

    @NotModified
    boolean isEmpty();

    @Modified
    boolean remove(Object object);

    @Modified
    boolean removeAll(@NotModified Collection<?> c);

    @NotModified
    int size();

    @NotModified
    Stream<E> stream();

    @NotModified
    Object[] toArray();
}
```

Adding an object to a collection (set, list) will cause some assignment somewhere inside the data structure. Returning the size of the collection should not.

Under supervision of the static code analyser, you will not be able to create an implementation of this interface which violates the modification rules. This is intentional: no implementation should modify the data structure when `size` is called.

Adding all elements of a collection to the object (in `addAll`) should not modify the input collection, whence the `@NotModified`. Other types in the parameters have not been annotated with `@NotModified`:

- `Object` because it is deeply immutable;
- `E` because it is of an unbound generic type, it has the same methods available as `Object`. No code statically visible to implementations of `Collection` can make modifications to `E`;
- `Consumer` because it is a functional interface (an interface with a single abstract method); they are `@NotModified` by definition.

In order to keep the narrative going, we defer a discussion of modification in the context of parameters of functional interface types to the section Higher-level modifications. Here, we continue with the first use case of modification: containers.

6. Containers

Loosely speaking, a container is a type to which you can safely pass on your objects, it will not modify them. This is the formal rule:

> **Definition**: a type is a **container** when no non-private method or constructor modifies its parameters.

Whatever else the container does, storing the parameters in fields or not, it will not change your objects. You obviously remain free to change them elsewhere; then the container will hold on to the changed object, not some copy.

Containers are complementary to immutable objects, and we will find that many immutable objects are containers, while some containers are the precursors to immutable types. There are two archetypes for containers: collections and builders.

The code analyser will annotate a type that is both level 1 immutable, and a container, with `@E1Container` . This occurs frequently enough to justify a separate annotation. The simple but useful utility type `Pair` trivially satisfies both requirements:

A pair of objects

```
@E1Container
public class Pair<K,V> {
    public final K k;
    public final V v;
    public Pair(K k, V v) {
        this.k = k;
        this.v = v;
    }
    public K getK() {
        return k;
    }
    public V getV() {
        return v;
    }
}
```

While it is clearly level 1 immutable, it will remain to be seen if it satisfies all criteria for intuitive immutability. However, it is easily recognized as a container: a type you use and trust to hold objects.

Containers occur frequently as static sub-types to build immutable objects. Examples of these will follow later, after the definition of level 2 immutability.

Let us conclude this section with an example consisting of three types: the first a class computed to be a container, the second a container according to the contract, and the third a class which cannot be a container:

```
@Container
class ErrorMessage {
    private String message;

    public ErrorMessage(String message) {
        this.message = message;
    }

    @NotModified
    public String getMessage() {
        return message;
    }

    @Modified
    public void setMessage(String message) {
        this.message = message;
    }
}

@Container
interface ErrorRegistry {
    @NotModified
    List<ErrorMessage> getErrors();

    @Modified
    void addError(@NotModified ErrorMessage errorMessage); ①
}

class BinaryExpression extends Expression {
    public final Expression lhs;
    public final Expression rhs;

    // ...

    public void evaluate(@Modified ErrorRegistry errorRegistry) {
        // ...
        if(lhs instanceof NullConstant || rhs instanceof NullConstant) {
            errorRegistry.addError(new ErrorMessage(...)); ②
        }
        // ...
    }
}
```

① Implementations of `ErrorRegistry` will not be allowed to use the `setMessage` setter in `addError`.

② Here a modifying method call takes place.

The `BinaryExpression` class is not a container, because it uses one of the parameters of a public method, `errorRegistry` of `evaluate`, as a writable container.

7. Linking and independence

Let us now elaborate on how we will compute modifications, in a path towards level 2 immutability. Consider the following example:

Example 6, field linked to constructor parameter

```
class LinkExample1<X> {
    private final Set<X> set;

    public LinkExample1(Set<X> xs) {
        this.set = xs;
    }

    public void add(X x) {
        set.add(x);
    }
}
```

After construction, an instance of `LinkExample1` contains a reference to the set that was passed on as an argument to its constructor. We say the field `set` links to the parameter `xs` of the constructor. In this example, this is an expensive way of saying that there is an assignment from one to the other. However, linking can become more complicated.

The *e2immu* analyser will add modification annotations to `LinkExample1` as follows:

Example 7, field linked to constructor parameter, with annotations

```
class LinkExample1<X> {
    @Modified
    private final Set<X> set;

    public LinkExample1(@Modified Set<X> xs) {
        this.set = xs;
    }

    @Modified
    public void add(X x) {
        set.add(x);
    }
}
```

The parameter `x` of `LinkExample1.add` is `@NotModified` because the first parameter of `Set.add` is `@NotModified`. The `add` method modifies the field, which causes the annotation first on the method, then on the field, and finally on the parameter of the constructor. Because of the latter, `LinkExample1` cannot be marked `@Container` .

Linking looks at the underlying object, and not at the variable. Consider the following alternative `add` method:

Example 8, alternative add method

```
@Modified
public void add(X x) {
    Set<X> theSet = this.set;
    X theX = x;
    theSet.add(theX);
}
```

Nothing has changed, obviously. Finally, as an example of how linking can become more complicated than following assignments, consider a typical *view* on a collection:

Example 9, linking using a method call

```
List<X> list = createSomeLargeList();
List<X> sub = list.subList(1, 5);
sub.set(0, x); // will modify sub, and list!
```

On the other side of the spectrum, linking does not work on objects that cannot be modified, like primitives or deeply immutable objects such as the primitives, or `java.lang.String`.

Let us summarize by:

> Intuitively, linking two variables means that modifying the content of one variable implies that the content of the linked variable may be modified too.

We will discuss linking formally in Linking, formally. For now, assume that a field links to another field, or to a parameter, if there is a possibility that both variables represent (part of) the same object (their object graphs overlap).

The opposite of linking is independence. While the code analyser will not express all the linking that goes on, it will annotate (in)dependence on the entry and exit points of the type: a method's return value and parameters, and a constructor's parameters.

> **Definition**: A method or constructor is **independent** when neither the parameters, nor the returned value (in case there is one, not `void`, not `this`) link to any of the fields of the type. The independence (or dependence) is marked with `@Independent` (or `@Dependent`) on the method for the return value, and on the relevant parameters otherwise.

It follows immediately that:

- empty constructors of top-level types and static sub-types (but not necessarily inner classes, sub-types that are not static) are always independent;
- non-modifying methods that return primitives or deeply immutable objects are independent, since no assignments to fields take place, and the returned objects cannot be modified.

Examples follow soon, once immutability has been defined in more detail. Note that we will mark a constructor as `@Independent` when all its parameters are `@Independent` , and `@Dependent` when at least one parameter is `@Dependent` .

8. Level 2 immutability

8.1. Definition

First, what do we want intuitively? A useful form of immutability, less strong than deeply immutable, but better than level 1 immutability for many situations. We propose the following description:

> After construction, an immutable type holds a number of objects; the type will not change their content, nor will it exchange these objects for other objects, or allow others to do so. The type is not responsible for what others do to the content of the objects it was given.

Technically, level 2 immutability is much harder to define than level 1 immutability. We identify three rules, on top of the obvious level 1 immutability requirement. One of these must be observed at all times:

> **Definition**: the **first rule of level 2 immutability** is that all fields must be `@NotModified`.

Our friend the `Pair` satisfies this first rule:

```
public class Pair<K,V> {
    public final K k;
    public final V v;
    public Pair(K k, V v) {
        this.k = k;
        this.v = v;
    }
}
```

Note that since `K` and `V` are unbound generic types, it is not possible to modify their content from inside `Pair`, since there are no modifying methods one can call on unbound types.

How does it fit the intuitive rule for immutability? The type `Pair` holds two objects. The type does not change their content, nor will it exchange these two objects for others, or allow others to do so. It is clear the users of `Pair` may be able to change the content of the objects they put in the `Pair`. Summarizing: `Pair` fits the intuitive definition nicely.

Here is an example which shows the necessity of the first rule more explicitly:

Point and Line

```
@Container
class Point {
    @Variable
    private double x;

    @Variable
    private double y;

    @NotModified
    public double getX() {
        return x;
    }

    @Modified
    public void setX(double x) {
        this.x = x;
    }

    @NotModified
    public double getY() {
        return y;
    }

    @Modified
    public void setY(double y) {
        this.y = y;
    }
}

@E1Container
class Line {
    @Final
    @Modified
    private Point point1;

    @Final
    @Modified
    private Point point2;

    public Line(Point point1, Point point2) {
        this.point1 = point1;
        this.point2 = point2;
    }

    @NotModified
    public Point middle() {
        return new Point((point1.getX() + point2.getX())/2.0,
             (point1.getY()+point2.getY())/2.0);
    }

    @Modified
```

```
    public void translateHorizontally(double x) {
        point1.setX(point1.getX() + x); ①
        point2.setX(point2.getX() + x);
    }
}
```

① Modifying operation on `point1`.

The fields `point1` and `point2` are effectively final. Without the translation method, the fields would be `@NotModified` as well. The translation method modifies their content, rendering the type not level 2 immutable.

Assuming a type's goal is to store a number of objects, it is easy to see that a level 1 immutable type cannot hold additional, modifiable state. It follows that every method call on the container object with the same arguments will render the same result. (Note that this cannot be bypassed by using *static* state, i.e., state specific to the type rather than the object. Our definitions make no distinction between static and instance fields.)

In order to hold an arbitrary (or even modestly large) amount of objects, a type has to have *support data*: think an array, a tree structure, buckets for a hash table, etc. The rest of the definition of level 2 immutability is essentially about expressing the immutability of this support data. After construction, a level 2 immutable type will still hold a fixed number of objects, and the type will not change their content, nor exchange them for other objects, nor allow others to exchange them.

We will introduce two additional rules to the definition of level 2 immutability. They will only be of relevance for *some* fields; requiring them for all fields results in too strong a definition, and we have seen higher up that there are situations where the first rule is sufficient. Rather than specifying which fields need to follow the additional rules, it is easier to say which fields are exempt from them:

1. fields that are of level 2 immutable type themselves;
2. fields whose type can be replaced by an unbound type parameter.

Note that for all modification purposes we can replace unbound type parameters with `java.lang.Object`, which is level 2 immutable. We will call those fields *implicitly immutable*, and the total of the values of fields of implicitly immutable type the *implicitly immutable content* of the type. Their primary characteristic is that their content is inaccessible from within the type

Constructor parameters of unbound type parameter, or method return types of unbound type parameter are always independent, in the same way that level 2 immutable types are: inside the class, there's no way of modifying them.

We now add rules 2 and 3 to the definition, and obtain:

Definition: level 2 immutability:

(**Rule 0**: The type is level 1 immutable: all fields are effectively final)

Rule 1: All fields are `@NotModified`.

Rule 2: All fields are either private, or of level 2 immutable or implicitly immutable type.

Rule 3: All constructors and non-private methods are independent of the fields.

Rule 2 is there to ensure that the content of the object cannot be modified by means of access to the non-private fields. Rule 3 ensures that the content of the object cannot be modified externally.

The first rule can be reached *eventually* if there is one or more methods that effect a transition from the mutable to the immutable state. This typically means that all methods that assign or modify fields become off-limits after calling this marker method. Eventuality for rules 2 and 3 seems too far-fetched. We address the topic of eventual immutability fully in the section Eventual immutability.

The section Higher-level modifications will discuss modification and independence of types with abstract methods, such as functional interface types.

Let us go to examples immediately.

Example with array, v1

```
class ArrayContainer1<T> {
    private final T[] data;
    public ArrayContainer1(T[] ts) {
        this.data = ts;
    }
    public Stream<T> stream() {
        return Arrays.stream(data);
    }
}
```

After creation, changes to the source array `ts` are effectively changes to the data array `data`. This construct fails rule 3, independence. Here the array of type `T[]` is the support data that holds `T`, which also appears in the return type of the `stream` method, held by `Stream`.

Example with array, v2, still not OK

```
class ArrayContainer2<T> {
    public final T[] data;
    public ArrayContainer2(T[] ts) {
        this.data = new T[ts.length];
        System.arraycopy(ts, 0, data, 0, ts.length);
    }
    public Stream<T> stream() {
        return Arrays.stream(data);
    }
}
```

Users of this type can modify the content of the array using direct field access! This construct fails rule 2, which applies for the same reasons as in the previous example.

Example with array, v3, safe

```
class ArrayContainer3<T> {
    private final T[] data; ①
    public ArrayContainer3(T[] ts) {
        this.data = new T[ts.length]; ②
        System.arraycopy(ts, 0, data, 0, ts.length);
    }
    public Stream<T> stream() {
        return Arrays.stream(data);
    }
}
```

① The array is private, and therefore protected from external modification.

② The array has been copied, and therefore is independent of the one passed in the parameter.

The independence rule enforces the type to have its own structure rather than someone else's. Here's the same group of examples, now with JDK Collections:

Example with collection, v1

```
class SetBasedContainer1<T> {
    private final Set<T> data;
    public SetBasedContainer1(Set<T> ts) {
        this.data = ts; ①
    }
    public Stream<T> stream() {
        return data.stream();
    }
}
```

① After creation, changes to the source set are effectively changes to the data.

The lack of independence of the constructor violates rule 3 in the first example.

Example with collection, v2, still not OK

```
class SetBasedContainer2<T> {
    public final Set<T> data; ①
    public SetBasedContainer2(Set<T> ts) {
        this.data = new HashSet<>(ts);
    }
    public Stream<T> stream() {
        return data.stream();
    }
}
```

① Users of this type can modify the content of the set after creation!

Here, the `data` field is public, which allows for external modification.

Example with set, v3, safe

```
class SetBasedContainer3<T> {
    private final Set<T> data; ①
    public SetBasedContainer3(Set<T> ts) {
        this.data = new HashSet<>(ts); ②
    }
    public Stream<T> stream() {
        return data.stream();
    }
}
```

① The set is private, and therefore protected from external modification.

② The set has been copied, and therefore is independent of the one passed in the parameter.

Finally, we have a level 2 immutable type.

Example with set, v4, safe

```
class SetBasedContainer4<T> {
    public final ImmutableSet<T> data; ①
    public SetBasedContainer4(Set<T> ts) {
        this.data = Set.copyOf(ts); ②
    }
    public Stream<T> stream() {
        return data.stream();
    }
}
```

① the data is public, but the `ImmutableSet` is `@E2Immutable` itself.

② Independence guaranteed.

The independence rule 3 is there to ensure that the type does not expose its support data through parameters and return types:

Example with set, v5, unsafe

```
class SetBasedContainer5<T> {
    private final Set<T> data; ①
    public SetBasedContainer5(Set<T> ts) {
        this.data = new HashSet<>(ts); ②
    }
    public Set<T> getSet() {
        return data; ③
    }
}
```

① No exposure via the field

② No exposure via the parameter of the constructor

③ ... but exposure via the getter. We could as well have made the field `public final`.

Note that by decomposing all definitions, we observe that requiring all fields to be `@Final` and `@NotModified` is equivalent to requiring that all non-private fields have the `final` modifier, and that methods that are not part of the construction phase, are `@NotModified`.

The following type is `@Container` , the field is `@Final` , but it is not `@NotModified`:

```
class Example2 {
    @Final
    @Modified
    public final Set<T> set = new HashSet<>();

    @Modified
    public void add(T t) { set.add(t); }

    @NotModified
    public Stream<T> stream() { return set.stream(); }
}
```

8.2. Dynamic type annotations

When it is clear a method returns an immutable set, but the formal type is `java.util.Set`, the `@E2Immutable` annotation can 'travel':

```
@E2Container
class SetBasedContainer6<T> {
    @E2Container
    public final Set<T> data;

    public SetBasedContainer4(Set<T> ts) {
        this.data = Set.copyOf(ts);
    }

    @E2Container
    public Set<T> getSet() {
        return data;
    }
}
```

Whilst `Set` in general is not `@E2Immutable` , the `data` field itself is.

The computations that the analyser needs to track dynamic type annotations, are similar to those it needs to compute eventual immutability. We introduce them in the next chapter.

8.3. Inheritance

Deriving from a class that is level 2 immutable, is the most normal situation: since `java.lang.Object` is a level 2 immutable container, every class will do so. Clearly, the property is not inherited. Most importantly, the analyser prohibits changing the modification status of methods: once a method is non-modifying, it cannot become modifying in a derived class. This means, for example, that the analyser will block a modifying `equals()` method! Note that this rule applies to implementations of methods of all super-types: no implementation of `java.util.Collection.size()` will be allowed to be modifying.

The guiding principle here is that of *consistency* or *expectation*: software developers are expecting that `equals` is non-modifying. They know that a setter will make an assignment, but they'll expect a getter to simply return a value. No getter should ever be modifying.

The other direction is more interesting, while equally simple to explain: deriving from a parent class cannot increase the immutability level. Explicitly changing a modifying method into a non-modifying one is not allowed by the analyser.

We will allow interfaces to be used to set expectations: when `java.util.Set` is defined to be a `@Container` , it makes sense to demand that every implementation of `Set` is a `@Container` as well. This rule will extend to `@E1Immutable` , `@E2Immutable` , and `@Independent` .

8.4. Static side effects

Up to now, we have made no distinction between static fields and instance fields: modifications are modifications. Inside a primary type, we will stick to this rule. In the following example, each call to `getK` increments a counter, which is a modifying operation because the type owns the counter:

Modifications on static fields are modifications

```
@E1Container
public class CountAccess<K> {
    private final K k;

    @Modified
    private static final AtomicInteger counter = new AtomicInteger();

    public CountAccess(K k) {
        this.k = k;
    }

    @Modified
    public K getK() {
        counter.getAndIncrement();
        return k;
    }

    @NotModified
    public static int countAccessToK() {
        return counter.get();
    }
}
```

We can explicitly ignore modifications with the `@IgnoreModifications` annotation, which may make sense from a semantic point of view:

Modification on static field explicitly ignored

```
@E2Container
public class CountAccess<K> {
    private final K k;

    @IgnoreModifications
    private static final AtomicInteger counter = new AtomicInteger();

    public CountAccess(K k) {
        this.k = k;
    }

    @NotModified
    public K getK() {
        counter.getAndIncrement();
        return k;
    }

    @NotModified
    public static int countAccessToK() {
        return counter.get();
    }
}
```

The next section, on Value-based classes, briefly explores these semantic differences.

Note that when the modification takes place inside the constructor, it is "ignored", because constructors are meant to be modifying:

Modification takes place inside constructor

```
@E2Container
public class HasUniqueIdentifier<K> {
    public final K k;
    public final int identifier;

    @NotModified
    private static final AtomicInteger generator = new AtomicInteger();

    public HasUniqueIdentifier(K k) {
        this.k = k;
        identifier = generator.getAndIncrement();
    }
}
```

When static modifying methods are called, on a field not belonging to the primary type or any of the parent types, or directly on a type expression which does not refer to any of the types in the primary type or parent types, we will make an exception to this rule, and classify the modification as a *static side effect*. This leads to consistency with the rules of level 2 immutable types, which only

look at the fields and assume that when methods do not modify the fields, they are actually non-modifying.

Without an `@IgnoreModifications` annotation on the field `System.out` (which we would typically add), printing to the console results in

```
@StaticSideEffects
@NotModified
public K getK() {
    System.out.println("Getting "+k);
    return k;
}
```

We leave it up to the programmer or designer to determine whether static calls deserve a `@StaticSideEffects` warning, or not. In almost all instances, we prefer a singleton instance (see Singleton classes) over a class with modifying static methods. In singletons the normal modification rules apply, unless `@IgnoreModifications` decorates the static field which allows access to the singleton.

8.5. Value-based classes

Quoting from the JDK 8 documentation, value-based classes are

1. final and immutable (though may contain references to mutable objects);
2. have implementations of equals, hashCode, and toString which are computed solely from the instance's state and not from its identity or the state of any other object or variable;
3. make no use of identity-sensitive operations such as reference equality (==) between instances, identity hash code of instances, or synchronization on an instances's intrinsic lock;
4. are considered equal solely based on equals(), not based on reference equality (==);
5. do not have accessible constructors, but are instead instantiated through factory methods which make no commitment as to the identity of returned instances;
6. are freely substitutable when equal, meaning that interchanging any two instances x and y that are equal according to equals() in any computation or method invocation should produce no visible change in behavior.

Item 1 requires level 1 immutability (all fields are `@Final`) but does not specify any of the restrictions we require for level 2 immutability. Item 2 implies that should `equals`, `hashCode` or `toString` make a modification to the object, its state changes, which would then change the object with respect to other objects. We could conclude that these three methods cannot be modifying.

Loosely speaking, objects of a value-based class can be identified by the values of their fields. Level 2 immutability (or deeper) is not requirement to be a value-based class. However, most level 2 immutable types will become value-classes. Revisiting the example from the previous section, we can construct a counter-example:

Level 2 immutable type which is not value-based

```
@E2Container
public class HasUniqueIdentifier<K> {
    public final K k;
    public final int identifier;

    @NotModified
    private static final AtomicInteger generator = new AtomicInteger();

    public HasUniqueIdentifier(K k) {
        this.k = k;
        identifier = generator.getAndIncrement();
    }

    @Override
    public boolean equals(Object other) {
        if(this == other) return true;
        if(other instanceof HasUniqueIdentifier<?> hasUniqueIdentifier) {
            return identifier == hasUniqueIdentifier.identifier;
        }
        return false;
    }
}
```

The `equals` method violates item 2 of the value-class definition, maybe not to the letter but at least in its spirit: the field `k` is arguably the most important field, and its value is not taken into account when computing equality.

9. Eventual immutability

There are many constraints on real-world use cases where immutability is not immediate. The type then follows a life cycle described as create, use, drop.

Challenges:

- external frameworks insist on a way of working perpendicular to that of your project
- some graph-like data structures simply cannot be made immutable using conventional tools
- once you go eventually immutable, how is the propagation organized?

When possible, use a builder.

9.1. Builders

```
@E2Container
class Point {
    public final double x;
    public final double y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}

@E2Container
class Polygon {

    public final List<Point> points;

    private Polygon(List<Point> points) { ①
        this.points = points;
    }

    @Container(builds=Polygon.class)
    static class Builder {
        private List<Point> points = new ArrayList<>();

        public void addPoint(Point point) {
            points.add(point);
        }

        public Polygon build() {
            return new Polygon(List.copyOf(points));
        }
    }
}
```

① The private constructor combined with the construction of an immutable copy in the `build` method guarantees level 2 immutability.

If your code can live with two different types (`Polygon.Builder`, `Polygon`) to represent polygons in their different stages (mutable, immutable), the builder paradigm is great. If, on the other hand, you want to hold polygons in a type that spans both stages of the polygon lifecycle, it becomes difficult to do this with an eye on immutability. One solution is the use of an interface that is implemented both by the builder and the immutable type.

The FirstThen type can also assist in this situation: it holds an initial object (the *first*) until a state change occurs, and it is forced to hold a second object (the *then*). Once it is in the final state, it cannot change anymore. It is *eventually immutable*:

```
class PolygonManager {
    // initially, the polygon is in builder phase
    public final FirstThen<Polygon.Builder, Polygon> polygon =
        new FirstThen<>(new Polygon.Builder());

    // ...

    public void construct() {
        // in builder phase ...
        polygon.getFirst().add(point);
        // transition
        polygon.set(polygon.getFirst().build());
        // from here on, polygon is immutable!
    }

    public Point firstPoint() {
        return polygon.get().points.get(0);
    }
}
```

9.2. Definition

We propose a system of eventual immutability based on a single transition of state inside an object.

First example of frozen class

```
@E2Container(after="frozen")
class SimpleImmutableSet1<T> {
    private final Set<T> set = new HashSet<>();
    private boolean frozen;

    @Only(before="frozen")
    public boolean add(T t) {
        if(frozen) throw new UnsupportedOperationException();
        set.add(t);
    }

    @Mark("frozen")
    public void freeze() {
        if(frozen) throw new UnsupportedOperationException();
        frozen = true;
    }

    @Only(after="frozen")
    public Stream<T> stream() {
        if(!frozen) throw new UnsupportedOperationException();
        return set.stream();
    }

    @TestMark("frozen")
    public boolean isFrozen() { ①
        return frozen;
    }

    public int size() { ①
        return set.size();
    }
}
```

① These methods can be called any time.

The analyser has no problem detecting the presence of preconditions, and observing that one method changes its own precondition. The rules, however, are sufficiently general to support arbitrary preconditions, as shown in the following variant. This example does not require an additional field, but relies on the empty/not-empty state change:

Second example of frozen class

```
@E2Container(after="set")
class SimpleImmutableSet2<T> {
    private final Set<T> set = new HashSet<>();

    @Mark("set")
    public void initialize(Set<T> data) {
        if(!set.isEmpty()) throw new UnsupportedOperationException();
        if(data.isEmpty()) throw new IllegalArgumentException();
        set.addAll(data);
    }

    @Only(after="set")
    public Stream<T> stream() {
        if(set.isEmpty()) throw new UnsupportedOperationException();
        return set.stream();
    }

    public int size() {
        return set.size();
    }

    @TestMark("set")
    public boolean hasBeenInitialised() {
        return !set.isEmpty();
    }
}
```

Let us summarize the annotations:

- The `@Mark` annotation marks methods that change the state from *before* to *after*.
- The `@Only` annotation identifies methods that, because of their precondition, can only be executed without raising an exception before (when complemented with a `before="⋯"` parameter) or after (with a `after="⋯"` parameter) the transition.
- The analyser computes the `@TestMark` annotation on methods which return the state as a boolean. There is a parameter to indicate that instead of returning `true` when the object is *after*, the method actually returns `true` on *before*.
- Finally, the eventuality of the type shows in the `after="⋯"` parameter of `@E1Immutable` , `@E2Immutable` or their container versions.

In each of these annotations, the actual value of the ⋯ in the `after=` or `before=` parameters is the name of the field.

In case there are multiple fields involved, their names are represented in a comma-separated fashion.

The `@Mark` and `@Only` annotations can also be assigned to parameters, in the event that marked methods are called on a parameter of eventually immutable type. Consider the following utility

method for EventuallyFinal, frequently used in the analyser:

Code fragment from EventuallyFinalExtension

```
public static <T> void setFinalAllowEquals(@Mark("isFinal") EventuallyFinal<T>
eventuallyFinal, T t) {
    if (eventuallyFinal.isVariable() || !Objects.equals(eventuallyFinal.get(), t)) {
        eventuallyFinal.setFinal(t);
    }
}
```

Here, the `setFinal` method's `@Mark` annotation travels to the parameter, where it is applied to the argument each time the static method is applied.

9.3. Propagation

The support types detailed in Support classes can be used as building blocks to make ever more complex eventually immutable classes. Effectively final fields of eventually immutable type will at some point hold objects that are in their final or `after` state, in which case they act as level 2 immutable fields.

The analyser itself consists of many eventually immutable classes; we show some examples in Support classes in the analyser.

For everyday use of eventual immutability, this is probably the most important consequence of all definitions up to now.

9.4. Before the mark

A method can return an eventually immutable object, guaranteed to be in its initial or `before` state. This can be annotated with `@BeforeMark` . Employing `SimpleImmutableSet1` from the example above,

```
@BeforeMark
public SimpleImmutableSet1 create() {
    return new SimpleImmutableSet1();
}
```

Similarly, the analyser can compute a parameter to be `@BeforeMark` , when in the method, at least one before-mark methods is called on the parameter.

Finally, a field can even be `@BeforeMark` , when it is created or arrives in the type as `@BeforeMark` , and stays in this state. This situation must occur in a type with a `@Finalizer` , as explained in Finalizers.

9.5. Extensions of annotations

When a type is eventually level 1 immutable, should the field(s) of the state transition be `@Variable` or `@Final` ? Similarly, when a type is eventually level 2 immutable, should the analyser mark the

initially mutable or assignable fields `@Modified` or `@NotModified`?

Basically, we propose to mark with the end state, qualifying with the parameter `after`:

property	not present	eventually	effectively
finality of field	`@Variable`	`@Final(after="mark")`	`@Final`
non-modification of field	`@Modified`	`@NotModified(after="mark")`	`@NotModified`

Since in an IDE it is not too easy to have multiple visual markers, it seems best to use the same visuals as the end state.

When a type is effectively level 1 immutable (not eventually), all fields are effectively final. The analyser wants to emphasise the rules needed to obtain (eventual) level 2 immutability, by clearly indicating which fields break the level 2 immutability rules. In the case of eventual level 2 immutability,

- modifications to the support data cease after a given mark
- the analyser disallows modifications to the other fields.

Eventual finality simply adds a `@Final(after="mark")` annotation to each of these situations.

9.6. Frameworks and contracts

A fair number of Java frameworks introduce dependency injection and initializer methods. This concept is, in many cases, compatible with the idea of eventual immutability: once dependency injection has taken place, and an initializing method has been called, the framework stops intervening in the value of the fields.

It is therefore not difficult to imagine, and implement in the analyser, a *before* state (initialization still ongoing) and an *after* state (initialization done) associated with the particular framework. The example below shows how this could be done for the `Verticle` interface of the vertx.io framework.

Excerpts and annotations of Verticle.java and AbstractVerticle.java

```
@E1Container(after = "init")
interface Verticle {

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

    @Only(after = "init")
    Vertx getVertx();

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

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

public abstract class AbstractVerticle implements Verticle {
    @Final(after="init")
    protected Vertx vertx;

    @Final(after="init")
    protected Context context;

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

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

Currently, early 2021, contracted eventual immutability has not been implemented yet in the analyser.

10. Modification, part2

This section goes deeper into modification, linking and independence. We start with cyclic references.

10.1. Cyclic references

We need to study the situation of seemingly non-modifying methods with modifying parameters. Up to now, a method is only modifying when it assigns to a field, calls a modifying method on one of the fields, or directly calls a modifying method on `this`. However, there could be indirect modifications, as in:

Indirect modifications

```
@E2Container
public class CyclicReferences {

    @MutableModifiesArguments
    static class C1 {

        @Variable
        private int i;

        @Modified
        public int incrementAndGet() {
            return ++i;
        }

        @Modified ①
        public int useC2(@Modified C2 c2) {
            return i + c2.incrementAndGetWithI();
        }

    }

    @E1Immutable
    static class C2 {

        private final int j;

        @Modified
        private final C1 c1;

        public C2(int j, @Modified C1 c1) {
            this.c1 = c1;
            this.j = j;
        }

        @Modified
        public int incrementAndGetWithI() {
            return c1.incrementAndGet() + j;
        }
    }
}
```

① `useC2` does not directly modify `i`, but `incrementAndGetWithI` does so indirectly.

This observation forces us to tighten the definition of a non-modifying method: on top of the definition given above, we have to ensure that none of the modifying methods called on a parameter which is `@Modified` , call one of 'our' modifying methods. These rules are mostly, but not easily, enforceable when all code is visible.

An additional interface can help to remove the circular dependency between the types. This has the advantage of simplicity, both for the programmer and the analyser, which at this point doesn't handle circular dependencies too well. It imposes more annotation work on the programmer, however, because the interface's methods need contracting.

10.2. Linking, formally

To compute linking, the analyser tries to track actual objects, with the aim of knowing if a field links to another field or a parameter. It computes a dependency graph of variables depending on other variables, with the following four basic rules:

Rule 1: in an assignment `v = w`, variable `v` links to variable `w`.

Rule 2: in an assignment `v = a.method(b)`,`v` potentially links to `a` and `b`.

Note that saying `v` links to `a` is the same as saying that the return value of `method` links to some field inside `A`, the type of `a`. This is especially clear when `a == this`.

We discern a number of special cases:

1. When `v` is of primitive or `@E2Immutable` type, there cannot be any linking; `v` does not link to `a` nor `b`.
2. If `b` is of primitive or `@E2Immutable` type, `v` cannot link to `b`.
3. When `method` has the annotation `@Independent` , `v` cannot link to `a`.
4. If `a` is of `@Independent` type (which includes all `@E2Immutable` types), all its methods are independent; therefore, `v` cannot link to `a`.

It is important to note that the analyser only computes independence for non-modifying methods, so that all methods of implicitly immutable return type are automatically independent.

Rule 3: in an assignment `v = new A(b)`, `v` potentially links to `b`.

1. When the constructor `A` is independent, `v` cannot link to `b`.
2. When `b` is of primitive or `@E2Immutable` type, `v` cannot link to `b`.
3. If `A` is `@E2Immutable` , then `v` cannot link to `b` nor `c`, because all constructors are independent.

Most of the other linking computations are consequences of the basic rules above. For example,

1. in an assignment `v = condition ? a : b`, `v` links to both `a` and `b`.
2. type casting does not prevent linking: in `v = (Type)w`, `v` links to `w`
3. Binary operators return primitives or `java.lang.String`, which prevents linking: in `v = a + b`, `v` does not link to `a` nor `b`.

> Rule 4: in an array access `v = a[index]`, `v` links to `a`.

Note: links between `b`, `c`, and `d` are possible, but are covered by the `@Modified` annotation: when a parameter is `@NotModified`, no modifications at all are possible, not even indirectly.

10.3. Abstract methods and functional interfaces

Abstract methods are present in interfaces, and abstract classes. Their very definition is that no implementation is present at the place of definition: only the ins (parameters) and outs (return type) are pre-defined.

Functional interfaces are interfaces with a single abstract method; any other methods in the interface are required to have a `default` implementation. The following table lists some frequently used ones:

Name	single abstract method (SAM)
`Consumer<T>`	`void accept(T t);`
`Function<T,R>`	`R apply(T t);`
`BiFunction<T, U, R>`	`R apply(T t, U u);`
`Supplier<R>`	`R get();`
`Predicate<T>`	`boolean test(T t);`

It is important not to forget that *any* interface defining a single abstract method can be seen as a functional interface. While the examples above all employ generics (more specifically, unbound type parameters), generics are not a requirement for functional interfaces. The Java language offers syntactic sugar for functional programming, but the types remain abstract Java types.

We will not make any distinction between a functional interface and an abstract type. If one were forced to make one, the *intention to hold data* would be the dividing line between a functional interface, which conveys no such intention, and an abstract type, which does.

In this section we want to discuss a limited application of functional interfaces: that where the SAMs have a local implementation. The general case, where abstract types come in via a parameter, will be taken up in Higher-level modifications. Consider the following example:

Concrete implementation of suppliers

```
class ApplyLocalFunctions {

    @Container
    static class Counter {
        private int counter;

        @Modified
        public int increment() {
            return ++counter;
        }
    }

    @Modified
    private final Counter myCounter = new Counter();

    @Modified
    private final Supplier<Integer> getAndIncrement = myCounter::increment;

    @Modified
    private final Supplier<Integer> explicitGetAndIncrement = new Supplier<Integer>()
{
        @Override @Modified
        public Integer get() {
            return myCounter.increment();
        }
    };

    @Modified
    public int myIncrementer() {
        return getAndIncrement.get();
    }

    @Modified
    public int myExplicitIncrementer() {
        return explicitGetAndIncrement.get();
    }
}
```

The fields `getAndIncrement` and `explicitGetAndIncrement` hold instances of anonymous *inner classes* of `ApplyLocalFunctions`: these inner classes hold data, they have access to the `myCounter` field. Their concrete implementations of `get` each modify `myCounter`. A straightforward application of the rules of modification of fields makes `getAndIncrement` and `explicitGetAndIncrement` `@Modified` : in `myIncrementer`, a modifying method is applied to `getAndIncrement`, and in `myExplicitIncrementer`, a modifying method is applied to `explicitGetAndIncrement`.

Given that `ApplyLocalFunctions` is clearly `@E1Container` , and the inner classes hold no other data, the inner classes are `@E1Container` as well.

Now, if we move away from suppliers, but use consumers, we obtain:

Concrete implementation of consumers

```
class ApplyLocalFunctions2 {

    @Container
    static class Counter {
        private int counter;

        @NotModified
        public int getCounter() {
            return counter;
        }

        @Modified
        public int increment() {
            return ++counter;
        }
    }

    @NotModified
    private final Counter myCounter = new Counter();

    @E2Immutable ①
    private static final Consumer<Counter> incrementer = Counter::increment;

    @E2Immutable
    private static final Consumer<Counter> explicitIncrementer = new Consumer<Counter
>() {
        @Override
        @NotModified
        public void accept(@Modified Counter counter) { ②
            counter.increment();
        }
    };

    @E2Container ③
    private static final Consumer<Counter> printer = counter ->
        System.out.println("Have " + counter.getCounter());

    @E2Container
    private static final Consumer<Counter> explicitPrinter = new Consumer<Counter>() {
        @Override
        @NotModified
        public void accept(@NotModified Counter counter) { ④
            System.out.println("Have " + counter.getCounter());
        }
    };

    private void apply(@Container(contract = true) Consumer<Counter> consumer) { ⑤
```

```
        consumer.accept(myCounter);
    }

    public void useApply() {
        apply(printer); // should be fine
        apply(explicitPrinter);
        apply(incrementer); // should cause an ERROR ⑥
        apply(explicitIncrementer); // should case an ERROR
    }
}
```

① The anonymous type is static, has no fields, so is at least `@E2Immutable` . It is not a container. This is clearly visible in the explicit variant…

② Here we see why `incrementer` is not a container: the method modifies its parameters.

③ Now, we have a container, because in the anonymous type does not modify its parameters.

④ Explicitly visible here in `explicitPrinter`.

⑤ If we insist that all parameters are containers, …

⑥ We can use the annotations to detect errors. Here, `incrementer` is not a container.

Using the `@Container` annotation in a dynamic way allows us to control which abstract types can use the method: when only containers are allowed, then the abstract types must not have implementations which change their parameters.

11. Higher-level modifications

From a type's point of view, fields are either of an explicit type, they are accessible, or they are of implicitly immutable type, in other words, replaceable by `Object` and inaccessible. Our focus up to now has been on modification of the explicit types: we have argued that they are the only ones that matter for practical immutability. Now we will try to characterise modifications that are beyond the scope of the type.

Please note that characterising *all* modifications is a hopelessly complex, and unnecessary, task. We will, however, need to deal with what can be seen as the first level of higher level modifications, if we want to be able to characterise the modifications going on in extremely important constructs such as iterators.

11.1. Propagating modifications

The basis of this section is an example container class called `Circular`:

First methods of Circular

```
@Container
class Circular<T> {

    private T x;
    private T y;
    private boolean next;

    public Circular() {
    }

    @Independent
    public Circular(Circular<T> c) {
        x = c.x;
        y = c.y;
        next = c.next;
    }

    @Modified
    public void add(T t) {
        if (next) {
            this.y = t;
        } else {
            this.x = t;
        }
        next = !next;
    }

    @NotModified
    @E2Container
    public Stream<T> stream() {
        return Stream.of(x, y);
    }
}
```

We also make use of the `Consumer` functional interface:

Consumer

```
@FunctionalInterface
interface Consumer<T> {
    void accept(T t);
}
```

Note that method and parameter remain unmarked in terms of modification. Using the `Consumer` we introduce a `forEach` method which iterates over the two elements:

forEach in Circular

```
@NotModified
public void forEach(@PropagateModification Consumer<T> consumer) {
    consumer.accept(x);
    consumer.accept(y);
}
```

From the point of view of `Circular`, no modifications occur because `accept` operates on fields of the implicitly immutable type `T`. No modifying methods are called on the parameter `consumer`, therefore it is not modified, and `Circular` can remain a container.

How do we propagate the modification? In this example we will use `StringBuilder` as an archetypal modifiable type.

Propagating the modification of forEach

```
static void print(@NotModified @NotModified1 Circular<StringBuilder> c) {
    c.forEach(System.out::println); ①
}

static void addNewLine(@NotModified @Modified1 Circular<StringBuilder> c) {
    c.forEach(sb -> sb.append("\n")); ②
}

static void replace(@Modified @NotModified1 Circular<StringBuilder> c) {
    c.forEach(sb -> c.add(new StringBuilder("x" + sb))); ③
}
```

① non-modifying method implies no modification on the implicitly immutable content of `c`

② parameter-modifying lambda propagates modification to `c's implicitly immutable content

③ object-modifying lambda changing `c` but not its content

We have introduced two new annotations, `@Modified1` and `@NotModified1`. Their meaning is intuitively clear from the example: the method `print` does not modify the implicitly immutable content of `Circular`, whereas the `addNewLine` method will do so. Conversely, `replace` modifies the `Circular` parameter directly, ignoring its content.

It is clear that we make a distinction between modifying the `Circular` instance, and modifying its content, the objects of type `StringBuilder`. Does this contradict the initial definition of modification of a parameter? Yes, but it allows us to be more specific.

How can we use this additional firepower? Consider the following two examples:

Using print and addNewLine

```
static String usePrint(@NotModified StringBuilder sb1,
                       @NotModified StringBuilder sb2,
                       @NotModified StringBuilder sb3) {
    Circular<StringBuilder> circular = new Circular<>();
    circular.add(sb1); ①
    circular.add(sb2);
    circular.add(sb3);
    print(circular);
    return circular.stream().collect(Collectors.joining());
}

static String useAddNewLine(@Modified StringBuilder sb1,
                            @Modified StringBuilder sb2,
                            @Modified StringBuilder sb3) {
    Circular<StringBuilder> circular = new Circular<>();
    circular.add(sb1);
    circular.add(sb2);
    circular.add(sb3);
    addNewLine(circular); ②
    return circular.stream().collect(Collectors.joining());
}
```

① `circular` now holds `sb1`

② `@Modified1` implies that the elements held by circular are modified (but not circular itself)

Proper propagation of the modifications relies on knowing that `circular` holds the parameters `sb1`, `sb2` and `sb3`. (We will assume it is too complicated to assess whether `sb1` is still held by `circular` or not.) This will be accomplished by computing 'content links', which give rise to 'content (in)dependence', all in a way very similar to ordinary linking and (in)dependence.

11.2. Content linking

Going back to `Circular`, we see that the `add` method binds the parameter `t` to the instance by means of assignment. Let us call this binding of parameters of implicitly immutable types *content linking*, and mark it using `@Dependent1` , *content dependence*:

Extra annotation on add

```
@Modified
public void add(@Dependent1 T t) {
    if (next) {
        this.y = t;
    } else {
        this.x = t;
    }
    next = !next;
}
```

Note that content dependence implies normal independence, exactly because we are dealing with parameters of implicitly immutable type. Thanks to this annotation, the statement `circular.add(sb1)` can content link `sb1` to circular. When propagating the modification of `addNewLine's parameter, all variables content linked to the argument get marked.

A second way, next to assignment, of adding to content links is Java's for-each loop:

For-each loop and content linking

```
Collection<StringBuilder> builders = ...;
for(StringBuilder sb: builders) { circular.add(sb); }
```

The local loop variable `sb` gets content linked to `circular`. Crucially, however, it is not difficult to see that `sb` is also content linked to `builders`! The `Collection` API will contain an `add` method annotated as:

```
@Modified
boolean add(@NotNull @Dependent1 E e) { return true; }
```

indicating that after calling `add`, the argument will become part of the implicitly immutable content of the collection. We need yet another annotation, `@Dependent2` , to indicate that the implicitly immutable content of two objects are linked. Looking at a possible implementation of `addAll`:

addAll

```
@Modified
boolean addAll(@NotNull1 @Dependent2 Collection<? extends E> collection) {
    boolean modified = false;
    for (E e : c) if (add(e)) modified = true;
    return modified;
}
```

The call to `add` content links `e` to `this`. Because `e` is also content linked to `c`, the parameter `collection` holds implicitly immutable content linked to the implicitly immutable content of the instance.

Again, note that `@Dependent2` implies independence, because it deals with the implicitly immutable content.

We're now properly armed to see how a for-each loop can be defined as an iterator whose implicitly immutable content links to that of a container.

11.3. Iterator, Iterable, loops

Let us start with the simplest definition of an iterator, without `remove` method:

Iterator, without remove

```
interface Iterator<T> {

    @Modified
    @Dependent1
    T next();

    @Modified
    boolean hasNext();
}
```

Either the `next` method, or the `hasNext` method, must make a change to the iterator, because it has to keep track of the next element. As such, we make both `@Modified` . Following the discussion in the previous section, `next` is `@Dependent1` , because it returns part of the implicitly immutable content held by the iterator.

The interface `Iterable` is a supplier of iterators:

Iterable

```
interface Iterable<T> {

    @NotModified
    @Dependent2
    Iterator<T> iterator();
}
```

First, creating an iterator should never be a modifying operation on a type. Typically, as we explore in the next section, it implies creating a sub-type, static or not, of the type implementing `Iterable`. Secondly, the iterator itself is independent of the fields of the implementing type, but has the ability to return its implicitly immutable content.

The loop, on a variable `list` of type implementing `Iterable<T>`,

```
for(T t: list) { ... }
```

can be interpreted as

```
Iterator<T> iterator = list.iterator();
while(it.hasNext()) {
    T t = it.next();
    ...
}
```

The iterator is `@Dependent2` . Via the `next` method, it content-links the implicitly immutable content of the `list` to `t`.

11.4. Independence of types

A concrete implementation of an iterator is a sub-type, static or not, of the iterable type:

```
@E2Container
public class ImmutableArray<T> implements Iterable<T> {

    @NotNull1
    private final T[] elements;

    @SuppressWarnings("unchecked")
    public ImmutableArray(List<T> input) {
        this.elements = (T[]) input.toArray();
    }

    @Override
    @Independent
    public Iterator<T> iterator() {
        return new IteratorImpl();
    }

    @Container
    @Independent
    class IteratorImpl implements Iterator<T> {
        private int i;

        @Override
        public boolean hasNext() {
            return i < elements.length;
        }

        @Override
        @NotNull
        public T next() {
            return elements[i++];
        }
    }
}
```

For `ImmutableArray` to be level 2 immutable, the `iterator()` method must be independent of the field `elements`. How do we know this? The implementation type `IteratorImpl` cannot be level 2 immutable, because it needs to hold the state of the iterator. However, it should protect the fields of its enclosing type. We propose to add a definition for the independence of a type, very similar to the one enforced for level 2 immutability:

Definition: A type is **independent** when it follows these three rules:

Rule 1: All constructor parameters linked to fields, and therefore all fields linked to constructor parameters, must be `@NotModified`;

Rule 2: All fields linked to constructor parameters must be either private or level 2 immutable;

Rule 3: All return values of methods must be independent of the fields linked to constructor parameters.

The static variant of `IteratorImpl` makes rules 1 and 2 more obvious:

Static iterator implementation

```
@E2Container
public class ImmutableArray<T> implements Iterable<T> {
    ...

    @Container
    @Independent
    static class IteratorImpl implements Iterator<T> {
        private int i;
        private final T[] elements;

        private IteratorImpl(T[] elements) {
            this.elements = elements;
        }

        @Override
        public boolean hasNext() {
            return i < elements.length;
        }

        @Override
        @NotNull
        public T next() {
            return elements[i++];
        }
    }
}
```

11.5. More on implicitly immutable types

Looking at the Lazy example, which immutability properties should we give the field `Supplier<T> supplier`? The type `T` is implicitly immutable in `Lazy`, as it can be replaced by `java.lang.Object`.

We propose to add abstract types, where *only* the abstract method is called, to the set of implicitly

immutable types of the enclosing type. The reasoning is that whilst you can access the type, you still cannot know the effect of this action.

12. Support classes

The `e2immu-support-1.0.0.jar` library (in whichever version it comes) essentially contains the annotations of the analyser, and a small selection of support types. They are the eventually immutable building blocks that you can use in your project, irrespective of whether you want analyser support or not.

We discuss a selection of the building blocks here.

12.1. FlipSwitch

Simpler than `FlipSwitch` is not possible for an eventually immutable type: it consists solely of a single boolean, which is at the same time the data and the guard:

Code fragment: most of FlipSwitch

```
@E2Container(after = "t")
public class FlipSwitch {

    @Final(after = "t")
    private volatile boolean t;

    private boolean set$Precondition() { return !t; } ①
    @Mark("t")
    public void set() {
        if (t) throw new IllegalStateException("Already set");
        t = true;
    }

    @NotModified
    @TestMark("t")
    public boolean isSet() {
        return t;
    }

    private boolean copy$Precondition() { return !t; } ①
    @Mark("t") // but conditionally
    public void copy(FlipSwitch other) {
        if (other.isSet()) set();
    }
}
```

① This companion method is present in the code to validate the computation of the precondition. See Preconditions and instance state for more details.

Note that methods which only conditionally change the immutability status of a type, such as `copy`,

also get a `@Mark` annotation. The analyser is not omniscient, and plays safe.

The obvious use case for this helper class is to indicate whether a certain job has been done, or not.

12.2. SetOnce

One step up from `FlipSwitch` is `SetOnce`: a place-holder for one object which can be filled exactly once:

Code fragment: parts of SetOnce

```
@E2Container(after = "t")
public class SetOnce<T> {

    @Final(after = "t")
    private volatile T t;

    @Mark("t")
    public void set(@NotNull T t) {
        if (t == null) throw new NullPointerException("Null not allowed");
        if (this.t != null) {
            throw new IllegalStateException("Already set: have " + this.t + ", try to
set " + t);
        }
        this.t = t;
    }

    @Only(after = "t")
    @NotNull
    @NotModified
    public T get() {
        if (t == null) {
            throw new IllegalStateException("Not yet set");
        }
        return t;
    }

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

    @NotModified
    public T getOrElse(T alternative) {
        if (isSet()) return get();
        return alternative;
    }
}
```

The analyser relies heavily on this type, with additional support to allow setting multiple times, with exactly the same value. This can be ascertained with a helper method, which, as noted in the previous section, also gets the `@Mark` annotation.

12.3. EventuallyFinal

Slightly more flexible than `SetOnce` is `EventuallyFinal`: the type allows you to keep writing objects using the `setVariable` method, until you write using `setFinal`. Then, the state changes and the type becomes level 2 immutable:

Code fragment: EventuallyFinal

```
@E2Container(after = "isFinal")
public class EventuallyFinal<T> {
    private T value;
    private boolean isFinal;

    public T get() {
        return value;
    }

    @Mark("isFinal")
    public void setFinal(T value) {
        if (this.isFinal) {
            throw new IllegalStateException("Trying to overwrite a final value");
        }
        this.isFinal = true;
        this.value = value;
    }

    @Only(before = "isFinal")
    public void setVariable(T value) {
        if (this.isFinal) throw new IllegalStateException("Value is already final");
        this.value = value;
    }

    @TestMark("isFinal")
    public boolean isFinal() {
        return isFinal;
    }

    @TestMark(value = "isFinal", before = true)
    public boolean isVariable() {
        return !isFinal;
    }
}
```

Here's also an example of a negated `@TestMark` annotation: `isVariable` return the negation of the normal `iFinal` mark test.

12.4. Freezable

The previous support class, `EventuallyFinal`, forms the template for a more general approach to eventual immutability: allow free modifications, until the type is *frozen* and no modifications can be allowed anymore.

Code fragment: Freezable

```
@E2Container(after = "frozen")
public abstract class Freezable {

    @Final(after = "frozen")
    private volatile boolean frozen;

    @Mark("frozen")
    public void freeze() {
        ensureNotFrozen();
        frozen = true;
    }

    @TestMark("frozen")
    public boolean isFrozen() {
        return frozen;
    }

    private boolean ensureNotFrozen$Precondition() { return !frozen; } ①
    public void ensureNotFrozen() {
        if (frozen) throw new IllegalStateException("Already frozen!");
    }

    private boolean ensureFrozen$Precondition() { return frozen; } ①
    public void ensureFrozen() {
        if (!frozen) throw new IllegalStateException("Not yet frozen!");
    }
}
```

① This companion method is present in the code to validate the computation of the precondition. See Preconditions and instance state for more details.

Note that as discussed in Inheritance, it is important for `Freezable`, as an abstract class, to be level 2 immutable: derived classes can only go *down* the immutability scale, not up!

12.5. SetOnceMap

We'll show one example that depends on `Freezable`: a freezable map where no objects can be overwritten:

Code fragment: part of SetOnceMap

```
@E2Container(after = "frozen")
public class SetOnceMap<K, V> extends Freezable {

    private final Map<K, V> map = new HashMap<>();

    @Only(before = "frozen")
    public void put(@NotNull K k, @NotNull V v) {
        Objects.requireNonNull(k);
        Objects.requireNonNull(v);
        ensureNotFrozen();
        if (isSet(k)) {
            throw new IllegalStateException("Already decided on " + k + ": have " +
                get(k) + ", want to write " + v);
        }
        map.put(k, v);
    }

    @NotNull
    @NotModified
    public V get(K k) {
        if (!isSet(k)) throw new IllegalStateException("Not yet decided on " + k);
        return Objects.requireNonNull(map.get(k)); ①
    }

    public boolean isSet(K k) {
        return map.containsKey(k);
    }

    ...
}
```

① The analyser will warn a potential null pointer exception here, not (yet) making the connection between `isSet` and `containsKey`. This connection can be implemented using the techniques described in Preconditions and instance state.

The code analyser makes frequent use of this type, often with an additional guard that allows repeatedly putting the same value to a key.

12.6. Lazy

`Lazy` implements a lazily-initialized immutable field, of unbound generic type `T`. Properly implemented, it is an eventually level 2 immutable type.

Code fragment: Lazy

```
@E2Container(after = "t")
public class Lazy<T> {

    @NotNull1 @PropagateModification @Dependent1
    private final Supplier<T> supplier;

    @Final(after = "t")
    private volatile T t;

    public Lazy(@NotNull1 @PropagateModification @Dependent1 Supplier<T> supplier) {
①
        this.supplier = supplier;
    }

    @NotNull
    @Mark("t") ②
    public T get() {
        if (t != null) return t;
        t = Objects.requireNonNull(supplier.get()); ③
        return t;
    }

    @NotModified
    public boolean hasBeenEvaluated() {
        return t != null;
    }
}
```

① The `@PropagateModification` annotation has travelled from the field to the parameter; so has `@Dependent1` .

② The `@Mark` annotation is conditional; the transition is triggered by nullity of `t`

③ Here `t` content links to `supplier`, as explained in Content linking. The statement also causes the `@NotNull1` annotation, as defined in Nullable, not null.

After calling the marker method `get()`, `t` cannot be assigned anymore, and it becomes `@Final` . Because it is of an unbound generic type, it is `@NotModified`, as is the field `supplier`. Level 2 immutability rules 2 and 3 do not apply for either fields.

12.7. FirstThen

```
package org.e2immu.analyser.util;

import org.e2immu.annotation.*;
import java.util.Objects;

@E2Container(after = "mark")
```

```
public class FirstThen<S, T> {
    private volatile S first;
    private volatile T then;

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

    @NotModified
    public boolean isFirst() {
        return first != null;
    }

    @NotModified
    public boolean isSet() {
        return first == null;
    }

    @Mark("mark")
    public void set(@NotNull T then) {
        Objects.requireNonNull(then);
        synchronized (this) {
            if (first == null) throw new UnsupportedOperationException("Already set");
            this.then = then;
            first = null;
        }
    }

    @NotNull @NotModified @Only(before = "mark")
    public S getFirst() {
        if (first == null)
            throw new UnsupportedOperationException("Then has been set"); ①
        S s = first;
        if (s == null) throw new NullPointerException();
        return s;
    }

    @NotNull @NotModified @Only(after = "mark")
    public T get() {
        if (first != null) throw new UnsupportedOperationException("Not yet set"); ②
        T t = then;
        if (t == null) throw new NullPointerException();
        return t;
    }

    @Override ③
    public boolean equals(@Nullable Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        FirstThen<?, ?> firstThen = (FirstThen<?, ?>) o;
        return Objects.equals(first, firstThen.first) &&
```

```
                Objects.equals(then, firstThen.then);
    }

    @Override ③
    public int hashCode() {
        return Objects.hash(first, then);
    }
}
```

① This is a bit convoluted. The precondition is on the field `first`, and the current implementation of the precondition analyser requires an explicit check on the field. Because this field is not final, we cannot assume that it is still null after the initial check; therefore, we assign it to a local variable, and do another null check to guarantee that the result that we return is `@NotNull`.

② Largely in line with the previous comment: we stick to the precondition on `first`, and have to check `then` to guarantee that the result is `@NotNull`.

③ The `equals` and `hashCode` methods inherit the `@NotModified` annotation from `java.lang.Object`.

Note that if we were to annotate the methods as contracts, rather than relying on the analyser to detect them, we could have a slightly more efficient implementation.

12.8. Support classes in the analyser

Practice what you preach, and all that. The *e2immu* analyser relies heavily on support classes such as `SetOnce`, and on the builder pattern described in the previous section. Almost all public types are containers. Because we intend to use the analyser's code as a showcase for this project, one important class (`ExpressionContext`) was intentionally kept as a non-container.

A good example of our aim for eventual immutability is `TypeInfo`, the primary container holding a type. Initially, a type is nothing but a reference, with a fully qualified name. Source code or byte code inspection augments it with information about its methods and fields. Whilst during inspection information is writable, after inspection this information becomes immutable. We use the builder pattern for `TypeInspection`, using `TypeInspectionImpl.Builder` first and `TypeInspectionImpl` later. The inspection information is stored using `SetOnce`:

Parts of TypeInfo

```
public class TypeInfo {
    public final String fullyQualifiedName;
    public final SetOnce<TypeInspection> typeInspection = new SetOnce<>();
    ...
}
```

Once inspection is over, the code analyser takes over. Results are temporarily stored in `TypeAnalysisImpl.Builder`, then copied into the immutable `TypeAnalysisImpl` class. Both classes implement the `TypeAnalysis` interface to shield off the build phase. Once the immutable type is ready, it is stored in `TypeInfo`:

Parts of TypeInfo, v2

```
@E2Container(after="typeAnalysis,typeInspection")
public class TypeInfo {
    public final String fullyQualifiedName;

    public final SetOnce<TypeInspection> typeInspection = new SetOnce<>();
    public final SetOnce<TypeAnalysis> typeAnalysis = new SetOnce<>();

    ...
}
```

In this way, if we keep playing by the book recursively downward, `TypeInfo` will become an eventually level 2 immutable type. Software engineers writing applications which use the *e2immu* analyser as a library, can feel secure that once the analysis phase is over, all the inspected and analysed information remains stable.

13. Other annotations

The *e2immu* project defines a whole host of annotations complementary to the ones required for immutability. We discuss them briefly, and refer to the user manual for an in-depth analysis.

13.1. Nullable, not null

Nullability is a standard static code analyser topic, which we approach from a computational side: the analyser infers where possible, the user adds annotations to abstract methods. The complement of not-null (marked `@NotNull`) is nullable (marked `@Nullable`).

- A method marked `@NotNull` will never return a null result. This is very standard.
- Calling a parameter marked `@NotNull` will result in a null pointer exception at some point during the object life-cycle.
- A `@NotNull` or `@Nullable` annotation on a field is a consequence of not-null computations on the assignments to the field.

To be able to compute the not-null of parameters, we must specify some sort of flow or direction to break chicken-and-egg situations. We compute in the following order:

1. context not-null of parameters.
2. field not-null
3. external not-null of parameters linked to fields

First, we examine the parameter's usage in the method. Its occurrence in a not-null context directly influences the not-null of the parameter.

13.1.1. Higher order not-null

We use the annotation `@NotNull1` to indicate that none of the object's fields can be null. This concept is useful when working with collections.

Consider the following `@NotNull` variants on the `List` API:

```
boolean add(@NotNull E e);
boolean addAll(@NotNull1 Collection<? extends E> collection);
@NotNull1 static <E> List<E> copyOf(@NotNull1 Collection<? extends E> collection);
@NotNull1 Iterator<E> iterator();
```

They effectively block the use of null elements in the collection. As a consequence, looping over the elements will not give potential null pointer warnings.

This is purely an opinion: we'd rather not use null as elements of a collection. You are free to annotate differently!

Higher orders are possible as well. A second level is useful when working with entry sets:

```
V put(@NotNull K key, @NotNull V value);
@NotNull static <K, V> Map<K, V> copyOf(@NotNull Map<? extends K, ? extends V> map);
@NotNull2 Set<Map.Entry<K, V>> entrySet();
```

Note how the map copy is only `@NotNull` , while the entry set is not null, the entries in this set are not null, and the keys and values are neither.

We do not plan to go beyond `@NotNull2` , however.

13.2. Identity and fluent methods

The analyser marks methods which returns their first parameter with `@Identity`, and methods which return `this` with `@Fluent`. The former are convenient to introduce preconditions, the latter occur frequently when chaining methods in builders. Here is an integrated example:

```
@Container(builds=Set.class)
class Builder {
    @NotNull @NotModified @Identity
    private static <T> T requireNonNull(@NotNull T t) {
        if(t == null) throw new UnsupportedOperationException();
        return t;
    }

    private final List<String> list = new ArrayList<>();

    @Fluent
    public Builder add(@NotNull String s) {
        list.add(requireNonNull(s));
        return this;
    }

    @Fluent
    public Builder add(int i) {
        list.add(Integer.toString(i));
        return this;
    }

    @E2Container
    public List<String> build() {
        return List.copyOf(list);
    }

    public static final Set<String> one23 = new Builder().add(1).add(2).add(3).add("
go").build();
}
```

13.3. Finalizers

Up to now, we have focused on the distinction between the building phase of an object's life-cycle, and its subsequent immutable phase. We have ignored the destruction of objects: critically important for some applications, but often completely ignored by Java programmers because of the silent background presence of the garbage collector. In this section we introduce an annotation, `@Finalizer` , with the goal of being able to mark that calling a certain method means that the object has reached the end of its life-cycle:

> Once a method marked `@Finalizer` has been called, no other methods may be subsequently applied.

Why is this useful? The most obvious use-case for immutability is the meaning of the `build()` method in a builder: can you call it once, or is the builder somehow incremental?

How can the analyser enforce the sequence of method calling on an object? The simplest way is by

some severe restrictions:

> The following need to be true at all times when using types with finalizer methods:
>
> 1. Any field of a type with finalizers must be effectively final (marked with `@Final`).
> 2. A finalizer method can only be called on a field inside a method which is marked as a finalizer as well.
> 3. A finalizer method can never be called on a parameter or any variable linked to it, with linking as defined throughout this document (see Linking and independence).

Interestingly, these restrictions are such that they help you control the life-cycle of objects with a `@Finalizer` , by not letting them out of sight.

Let us start from the following example, using EventuallyFinal:

Example of a type with a finalizer method

```
class ExampleWithFinalizer {
    @BeforeMark
    private final EventuallyFinal<String> data = new EventuallyFinal<>();

    @Fluent
    public ExampleWithFinalizer set(String string) {
        data.setVariable(string);
        return this;
    }

    @Fluent
    public ExampleWithFinalizer doSomething() {
        System.out.println(data.toString());
        return this;
    }

    @Finalizer(contract = true)
    @BeforeMark
    public EventuallyFinal<String> getData() {
        return data;
    }
}
```

Using `@Fluent` methods to go from construction to finalizer is definitely allowed according to the rules:

```
@E2Container
public static EventuallyFinal<String> fluent() {
    EventuallyFinal<String> d = new ExampleWithFinalizer()
        .set("a").doSomething().set("b").doSomething().getData();
    d.setFinal("x");
    return d;
}
```

Passing on these objects as arguments is permitted, but the recipient should not call the finalizer. Actually, given our strong preference for containers, the recipient should not even modify the object! Consider:

```
@E2Container
public static EventuallyFinal<String> stepWise() {
    ExampleWithFinalizer ex = new ExampleWithFinalizer();
    ex.set("a");
    ex.doSomething();
    ex.set("b");
    doSthElse(ex); ①
    EventuallyFinal<String> d = ex.getData();
    d.setFinal("x");
    return d;
}

private static void doSthElse(@NotModified ExampleWithFinalizer ex) {
    ex.doSomething(); ②
}
```

① here we pass on the object

② forbidden to call the finalizer; other methods allowed.

Rules 1 and 2 allow you to store a finalizer type inside a field, but only when finalization is attached to the destruction of the holding type. Examples follow immediately, in the context of the `@BeforeMark` annotation.

13.3.1. Processors and finishers

It is worth observing that finalizers play well with the `@BeforeMark` annotation. They allow us to introduce the concepts of *processors* and *finishers* for eventually immutable types in their *before* state.

The purpose of a *processor* is to receive an object in the `@BeforeMark` state, hold it, use a lot of temporary data in the meantime, and then release it again, modified but still in the `@BeforeMark` state.

Conceptual example of processor

```
class Processor {
    private int count; ①

    @BeforeMark ②
    private final EventuallyFinal<String> eventuallyFinal;

    public Processor(@BeforeMark EventuallyFinal<String> eventuallyFinal) {
        this.eventuallyFinal = eventuallyFinal;
    }

    public void set(String s) { ③
        eventuallyFinal.setVariable(s);
        count++;
    }

    @Finalizer(contract = true)
    @BeforeMark ④
    public EventuallyFinal<String> done(String last) {
        eventuallyFinal.setVariable(last + "; tried " + count);
        return eventuallyFinal;
    }
}
```

① symbolises the temporary data to be destroyed after processing

② the field is private, not passed on, no `@Mark` method is called on it, and it is exposed only in a `@Finalizer`

③ symbolises the modifications that act as processing

④ the result of processing: an eventually immutable object in the same initial state.

The purpose of a *finisher* is to receive an object in the `@BeforeMark` state, and return it in the final state. In the meantime, it gets modified (finished), while there is other temporary data around. Once the final state is reached, the analyser guarantees that the temporary data is destroyed by severely limiting the scope of the finisher object.

Conceptual example of finisher

```
class Finisher {
    private int count; ①

    @BeforeMark ②
    private final EventuallyFinal<String> eventuallyFinal;

    public Finisher(@BeforeMark EventuallyFinal<String> eventuallyFinal) {
        this.eventuallyFinal = eventuallyFinal;
    }

    @Modified
    public void set(String s) { ③
        eventuallyFinal.setVariable(s);
        count++;
    }

    @Finalizer(contract = true)
    @E2Container ④
    public EventuallyFinal<String> done(String last) {
        eventuallyFinal.setFinal(last + "; tried " + count);
        return eventuallyFinal;
    }
}
```

① symbolises the temporary data to be destroyed.

② only possible because the transition occurs in a `@Finalizer` method

③ symbolises the modifications that act as finishing

④ the result of finishing: an eventually immutable object in its end-state.

13.4. Utility class, extensions, singleton

13.4.1. Utility classes

We use the simple and common definition:

> **Definition**: a **utility class** is a level 2 immutable class which cannot be instantiated.

These definitions imply

1. a utility class has no non-static fields,
2. it has a single, private, unused constructor,
3. and its static fields (if it has any) are sufficiently immutable.

13.4.2. Extension classes

In Java, many classes cannot easily be extended. Implementations of extensions typically use a utility class with the convention that the first parameter of the static method is the object of the extended method call:

```
@ExtensionClass(of=String[].class)
class ExtendStringArray {
    private ExtendStringArray() { throw new UnsupportedOperationException(); }

    public static String weave(@NotModified String[] strings) {
        // generate a new string by weaving the given strings (concat 1st chars, etc.)
    }

    public static int appendEach(@Modified String[] strings, String append) {
        // append the parameter 'append' to each of the strings in the array
    }
}
```

We use the following criteria to designate a class as an extension:

A class is an extension class of a type `E` when

- the class is level 2 immutable;
- all non-private static methods with parameters must have a `@NotNull` 1st parameter of type `E`, the type being extended. There must be at least one such method;
- non-private static methods without parameters must return a value of type `E`, and must also be `@NotNull` .

Static classes can be used to 'extend' closed types, as promoted by the Xtend project. Level 2 immutable classes can also play the role of extension facilitators, with the additional benefit of having some immutable data to be used as a context.

Note that extension classes will often not be `@Container` , since the first parameter will be `@Modified` in many cases.

13.4.3. Singleton classes

A singleton class is a class which has a mechanism to limit the creation of instances to a maximum of one. The term 'singleton' then refers to this unique instance.

The *e2immu* analyser currently recognizes two systems for limiting the number of instances: the creation of an instance in a single static field with a static constructor, and a precondition on a constructor using a private static boolean field.

An example of the first strategy is:

First mechanism recognized to enforce a singleton

```
@Singleton
public class SingletonExample {

    public static final SingletonExample SINGLETON = new SingletonExample(123);

    private final int k;

    private SingletonExample(int k) {
        this.k = k;
    }

    public int multiply(int i) {
        return k * i;
    }
}
```

An example of the second strategy is:

Second mechanism recognized to enforce a singleton

```
@Singleton
public class SingletonWithPrecondition {

    private final int k;
    private static boolean created;

    public SingletonWithPrecondition(int k) {
        if (created) throw new UnsupportedOperationException();
        created = true;
        this.k = k;
    }

    public int multiply(int i) {
        return k * i;
    }
}
```

14. Preconditions and instance state

The *e2immu* analyser needs pretty strong support for determining preconditions on methods to be able to compute eventual immutability. A lot of the mechanics involved can be harnessed in other ways as well, for example, to detect common mistakes in the use of collection classes.

We have implemented a system where the value of a variable can be augmented with *instance state* each time a method operates on the variable. In the case of Java collections and `StringBuilder`, size-based instance state is low-hanging fruit. Let's start with an example:

Creating an empty list

```
List<String> list = new ArrayList<>();
if (list.size() > 0) { // WARNING: evaluates to constant
    ...
}
```

When creating a new `ArrayList` using the empty constructor, we can store in the variable's value that its size is 0. First, let us look at the annotations for the `size` method:

Annotations of List.size

```
void size$Aspect$Size() {}
boolean size$Invariant$Size(int i) { return i >= 0; }
@NotModified
int size() { return 0; }
```

The method has two *companion methods*. The first registers `Size` as a numeric *aspect* linked to the `size` method. The second adds an invariant (an assertion that is always true) in relation to the aspect: the size is never negative.

Looking at the annotations for the empty constructor,

Annotations of empty ArrayList constructor

```
boolean ArrayList$Modification$Size(int post) { return post == 0; }
public ArrayList$() { }
```

we see another companion method, that expresses the effect of the construction in terms of the `Size` aspect. (The dollar sign at the end of the constructor is an artifact of the annotated API system; please refer to the *e2immu* manual.) Internally, we represent the value of `list` after the assignment as

Internal representation of an empty list

```
new ArrayList<>()/*0==this.size()*/
```

The expression in the companion results in the fact that the `Size` aspect post-modification is 0. This then gets added to the evaluation state, which allows the analyser to conclude that the expression in the if-statement is a constant true.

This approach is sufficiently strong to catch a number of common problems when working with collections. After adding one element to the empty list, as in:

Adding an element to an empty list

```
List<String> list = new ArrayList<>();
list.add("a");
```

the value of `list` becomes

Internal representation after adding an element

```
instance type ArrayList<String>/*this.contains("a")&&1==this.size()*/
```

The boolean expression in the comments is added to the evaluation state, so that expressions such as `list.isEmpty()`, defined as:

List.isEmpty and its companion method

```
boolean isEmpty$Value$Size(int i, boolean retVal) { return i == 0; }
@NotModified
boolean isEmpty() { return true; }
```

can be evaluated by the analyser. We refer to the manual for a more in-depth treatment of companion methods and instance state.

15. Copyright and License