The Road to Immutability

Table of Contents

Dedication . . . 2
1. Introduction . . . 2
2. Assumptions . . . 3
3. The purpose of annotations . . . 4
4. Level 1 immutability . . . 4
5. Modification . . . 8
6. Containers . . . 11
7. Linking, independence . . . 15
8. Accessible and hidden content . . . 18
9. Level 2 immutability . . . 22
    9.1. Definition and examples . . . 22
    9.2. Inheritance . . . 29
    9.3. Generics . . . 30
    9.4. Abstract methods . . . 31
    9.5. Static side effects . . . 37
    9.6. Value-based classes . . . 40
    9.7. Dynamic type annotations . . . 42
10. Eventual immutability . . . 42
    10.1. Builders . . . 42
    10.2. Definition . . . 44
    10.3. Propagation . . . 47
    10.4. Before the mark . . . 47
    10.5. Extensions of annotations . . . 48
    10.6. Frameworks and contracts . . . 48
11. Modification, part 2 . . . 49
    11.1. Cyclic references . . . 50
    11.2. How to compute linking . . . 51
    11.3. Locally implemented abstract methods . . . 52
12. Hidden content . . . 55
    12.1. Visitors . . . 55
    12.2. Propagating modifications . . . 58
    12.3. Hidden content linking . . . 60
    12.4. Iterator, Iterable, loops . . . 61
    12.5. Eventual immutability . . . 63
13. Higher-order immutability . . . 63
    13.1. Definition . . . 63

13.2. Independence of types . . . 63
13.3. Field access restrictions . . . 67
14. Support classes . . . 69
14.1. FlipSwitch . . . 69
14.2. SetOnce . . . 70
14.3. EventuallyFinal . . . 72
14.4. Freezable . . . 73
14.5. SetOnceMap . . . 73
14.6. Lazy . . . 74
14.7. FirstThen . . . 76
14.8. Support classes in the analyser . . . 78
15. Other annotations . . . 79
15.1. Nullable, not null . . . 79
15.2. Identity and fluent methods . . . 80
15.3. Finalizers . . . 81
15.4. Utility classes . . . 86
15.5. Extension classes . . . 86
15.6. Singleton classes . . . 87
16. Preconditions and instance state . . . 88
17. Copyright and License . . . 90

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

Main website: https://www.e2immu.org. Third major iteration, beginning of October 2021.

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 to academic literature* in this version of the document 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 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. Some annotations, like `@Linked` and `@Constant`, serve no other purpose than debugging.

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 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 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 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 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 be at least `@E1Immutable` .

As above with effective finality, the term *effective* is present to make a distinction between formal immutability, and immutability that the 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:

Example 5, 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 IllegalStateException("Already set");
        this.t = t;
    }

    @Only(after="t")
    public void get() {
        if(this.t == null) throw new IllegalStateException("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 6, 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 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:

Example 7, modifying and non-modifying methods

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

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

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

@E1Immutable ①
class CountedInfo {
    @Final
    @Modified
    private final Counter counter = new Counter();

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

① The next section will show that the annotation will actually be `@E1Container` , representing the combination of `@E1Immutable` and `@Container` .

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` defined earlier.

Proceeding, let us also look at (a part of) the `Collection` interface, where we've restricted the annotations to `@NotModified` and `@Modified` . An abstract method without `@Modified` is assumed to be non-modifying, i.e., `@NotModified` is implicitly present. (The reason for this choice is explained later, in Abstract methods.) While in normal classes the analyser computes the annotations, in interfaces the user stipulates or *contracts* behaviour by annotating:

Example 8, 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);

    boolean contains(Object object);

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

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

    boolean isEmpty();

    @Modified
    boolean remove(Object object);

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

    int size();

    Stream<E> stream();

    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 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 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) in `java.util.function`; they are `@IgnoreModifications` by convention.

In order to keep the narrative going, we defer a discussion of modification in the context of parameters of abstract types to the sections Abstract methods and Hidden content. 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:

Example 9, 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 nested types to build immutable objects. Examples of these will follow later, after the definition of level 2 immutability.

In the following example, the first class is computed to be a container, the second is a container according to the contract, and the third is a class which cannot be a container:

Example 10, two containers and a type that is not a container

```
@Container
class ErrorMessage {
    @Variable
    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 implicitly
    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.

We conclude this section by noting that arrays are essentially level 1 immutable containers: a chunk of memory is held in an effectively final field, and array access reads and writes from this memory object. Indeed, consider the following semi-realistic implementation of an `Integer` array based on a `ByteBuffer`:

Example 11, an array is a level 1 immutable container

```
@E1Container
interface Array<T> {
    int length();

    T get(int index);

    @Modified
    void set(int index, T t);
}

@E1Container
static class IntArray implements Array<Integer> {
    private final ByteBuffer byteBuffer;
    private final int size;

    public IntArray(int size) {
        this.size = size;
        byteBuffer = ByteBuffer.wrap(new byte[size * Integer.BYTES]);
    }

    @Override
    public int length() {
        return size;
    }

    @Override
    public Integer get(int index) {
        return byteBuffer.getInt(index * Integer.BYTES);
    }

    @Override
    @Modified
    public void set(int index, Integer i) {
        byteBuffer.putInt(index * Integer.BYTES, i);
    }
}

@Test
public void test() {
    IntArray ia = new IntArray(5);
    for (int i = 0; i < 5; i++) ia.set(i, i + 1);
    assertEquals(3, ia.get(2));
}
```

7. Linking, independence

Let us now elaborate on how we will compute modifications, in a path towards level 2

immutability. Consider the following example:

Example 12, a field assigned to a 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 13, a field linked to a 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 14, alternative `add` method for `LinkExample1`

```
@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 15, linking using a method call

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

① The modifying method call `set` will modify `sub`, and `list` as well!

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:

> **Definition**: Two objects are independent of each other when no modification to the first can imply a modification to the second.
>
> Conversely, two objects are linked when a modification to the first may imply a modification to the second.

Linked objects typically share a common sub-object: the object returned by `subList`, for example, is "backed" by the original list, in other words, it maintains a reference to the original list.

We will discuss linking in more detail in How to compute linking. 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).

Linking and independence is important when it occurs from fields to parameters and return values of methods:

> **Definition**: A method or constructor parameter is **independent** when it is independent of the fields of the type. A method is **independent** when its return value is independent of the fields of the type.
>
> The independence is marked with `@Independent` on the method for the return value, and on the relevant parameters otherwise.

When a constructor parameter is not independent, any modification made to the object presented to this parameter as an argument may have an influence on the object graph of the fields of the constructor's type. But do all these modifications matter to the type?

8. Accessible and hidden content

We will try to make our case using two examples. First, consider `Counter` and `Counters`:

Example 16, Counter, Counters

```
interface Counter {
  void increment();
  int getValue();
  String getName();
}

class Counters {
  private final Map<String, Counter> counters;

  public Counters(Collection<Counter> counterCollection) {
    this.counters = counterCollection.stream().collect
      (Collectors.toUnmodifiableMap(Counter::getName, c -> c));
  }

  public Counter getCounter(String name) {
    return counters.get(name);
  }

  public int getValue(String name) {
    return getCounter(name).getValue();
  }

  public void increment(String name) {
    getCounter(name).increment();
  }

  public void incrementAll() {
    counters.values().forEach(Counter::increment);
  }
}
```

The constructor `Counters` copies every counter in the `counterCollection` into a new, unmodifiable map. Clearly, external modifications to the collection itself (i.e., adding, removing a new `Counter` element) made after creation of the `Counters` object, will have no effect on the object graph of the field `counters`:

```
List<Counter> list = new ArrayList<>();
Collections.addAll(list, new CounterImpl("sunny days"), new CounterImpl("rainy days")
);
Counters counters = new Counters(list);
Counter sunnyDays = list.remove(0);
assert "sunny days".equals(sunnyDays.getName());
assert sunnyDays == counters.getCounter("sunny days");
```

However, consider the following statements executed after creating a `Counters` object:

Example 17, after creating a Counters object

```
int rainyDays = counters.getValue("rainy days");
Counter c = counters.get("rainy days");
c.increment();
assert c.getValue() == rainyDays + 1;
assert counters.getValue("rainy days") == rainyDays + 1;
```

An external modification (`c.increment()`) to an object presented to the constructor as part of the collection has an effect on the object graph of the fields, to the extent that an identical, non-modifying method call returns a different value!

We must conclude that the parameter of the constructor `counterCollection` is linked to the field `counters`, even if modifications at the collection level have no effect.

Now we put the `Counters` example in contrast with the `Levels` example, where the modifying method `increment()` has been removed from `Counter` to obtain `Level`:

Example 18, Level, Levels

```
interface Level {
  int getValue();
  String getName();
}

class Levels {
  private final Map<String, Level> levels;

  public Levels(Collection<Level> levelCollection) {
    this.levels = levelCollection.stream().collect
      (Collectors.toUnmodifiableMap(Level::getName, c -> c));
  }

  public Level getLevel(String name) {
    return levels.get(name);
  }

  public int getValue(String name) {
    return getLevel(name).getValue();
  }
}
```

As a consequence of the absence of `increment()` in `Level`, we had to remove `increment()` and `incrementAll()` from `Levels` as well. In fact, whether the `Level` instances are modifiable or not, does not seem to matter anymore to `Levels`.

We propose to split the object graph of a field into two parts: its accessible part, and its hidden part.

> **Definition:** A type `A`, part of the object graph of the fields of type `T`, is **accessible** inside the type `T` when any of its methods or fields is accessed. The methods of `java.lang.Object` are excluded from this definition.
>
> A type that is part of the object graph of the fields, but is not accessible, is **hidden** (when it is an unbound type parameter) or **transparent** (when it is not).

A type which is transparent can be replaced by an unbound type parameter, which is why we will use the term *hidden* from now on.

When a type `C` extends from a parent type `P`, we see an instance of `C` as being composed of two parts: the methods and fields of `P`, augmented by the methods and fields of `C`. Whilst the part of the parent, `P`, can be accessible, the part of the child `C` may remain hidden. Similarly, when `T` implements the interface `I`, but the interface is used as the formal type, then the methods and fields of `I` are accessible, but the ones augmented by the implementation `T` remain hidden. In the example of `Level`, implementation or extensions may be modifiable (such as `Counter`), but when presented with `Level` only, there are no modifications to be made. Inside `Levels`, no such extensions are accessible.

Note that we must make this distinction, because every interface is meant to be implemented, and every (non-`final`) type can be extended in Java. These extensions could be completely outside the control of the current implementation (even though we can use the analyser to constrain them).

Armed with this definition, we split the combined object graph of the fields of a type into the accessible content, and the hidden content:

> **Definition:** The **accessible content** of a type are those objects of the object graph of the fields that are of accessible type.
>
> The **hidden content** of a type are those objects of the object graph of the fields that are of hidden or transparent type.

In the first example of this section, `LinkExample1`, objects of the type `X` form the hidden content of `LinkExample1`, while the `Set` instance is the accessible content. In `Counters`, `Map`, `String` and `Counter` are accessible, but whatever augments to `Counter` by implementing it remains hidden. Exactly the same applies to `Levels`: `Map`, `String` and `Level` are accessible, but whatever augments `Level` by implementing it remains hidden.

One of the central tenets of immutability will be that

> A type is not responsible for modifications to its hidden content.

We end this section by defining what independence means with respect to the accessible and hidden content of the fields. The definition of independence given in the previous section is absolute, in the sense that it covers the whole object graphs of the objects being linked, or not.

When a parameter is linked to a field, we could try to find out if the modifications affect the accessible content, given that we state that modifications to the hidden content are outside the scope of the type anyway. In other words, we could distinguish between different forms of non-independence, or, in better English, dependence.

> **Definition:** a parameter or method return value is **dependent** on the fields if and only if it is linked to the accessible content of the type.

In other words, a parameter or method return value is dependent when a modification on the argument or returned value has the possibility to cause a modification in the accessible part of the fields.

Linking between parameters or return value and fields which does not involve the accessible part of the fields, will be not be called dependence, but a lighter form of independence. We will elaborate more in Hidden content. In the following sections, we will often use the term 'independent' when we mean 'not-dependent', i.e., when we only require one of the lighter forms of independence.

In terms of annotations, dependence will be the default state for objects of types where dependence

is possible. We will not annotate it. Instead, `@Independent` on parameters and methods will be used for absolute independence. When a type is deeply immutable, `@Independent` is the default state, and therefore it will be omitted. We use `@Independent1` for the lighter forms of independence.

Now, all pieces of the puzzle are available to introduce immutability of types.

9. Level 2 immutability

9.1. Definition and examples

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. The first one prevents the type from making changes to its own fields:

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

Our friend the `Pair` satisfies this first rule:

Example 1, the class `Pair`, revisited

```
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 even possible to modify their content from inside `Pair`, since there are no modifying methods one can call on unbound types. The types `K` and `V` are hidden in `Pair`; it does not have any accessible content.

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:

Example 2: the types `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 the fields' content, preventing the type from becoming level 2 immutable.

From the restriction of rule 1, that all its fields should remain unmodified, it follows that, excluding external changes, every method call on a level 2 immutable container object with the same arguments will render the same result. We note that this statement cannot be bypassed by using *static* state, i.e., state specific to the type rather than the object. The definitions make no distinction between static and instance fields.

To obtain a useful definition of immutability, one which is not too strict yet follows our intuitive requirements, we should allow modifiable fields, if they are properly shielded from the modifications they intrinsically allow. We will introduce two additional rules to constrain the modifications of this modifiable data. Together with the first rule, and building on level 1 immutability, we define:

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 type themselves.

Rule 3: No parameters of non-private methods or non-private constructors, no return values of non-private methods, are dependent on (the accessible part of) the fields.

Rule 2 is there to ensure that the modifiable fields of the object cannot be modified externally by means of direct field access to the non-private fields. Rule 3 ensures that the modifiable fields of the object cannot be modified externally by obtaining or sharing references to the fields via a parameter or return value.

Types which are level 2 immutable will be marked `@E2Immutable` . When they are containers too, which should be the huge majority, we write `@E2Container` , stressing the container property.

Note that:

- We state that all primitive types are level 2 immutable, as is `java.lang.Object`. Whilst this is fairly obvious in the case of primitives, level 2 immutability for `Object` requires us to either ignore the methods related to synchronization, or to assume that its implementation (for it is not an abstract type) has no fields.
- A consequence of rule 1 is that all methods in a level 2 immutable type must be `@NotModified`.
- A field whose type is an unbound type parameter, can locally be considered to be of level 2 immutable type, and therefore need not be private. This is because the type parameter could be substituted by `java.lang.Object`, which we have just declared to be level 2 immutable. More details can be found in the section on Generics.
- Constructor parameters whose formal type is an unbound type parameter, are of hidden type inside the type of the constructor. As a consequence, rule 3 does not apply to them. This will be expanded on in Hidden content.
- The section on Inheritance will show how the immutability property relates to implementing interfaces, and sub-classing. This is important because the definition is recursive, with `java.lang.Object` the level 2 immutable base of the recursion. All other types must extend from it.
- The section on Abstract methods will detail how level 2 immutability is computed for abstract types (interfaces, abstract classes).
- 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.

Let us go to examples immediately.

Example 19, explaining level 2 immutability: with array, version 1, not good

```
@E1Container
class ArrayContainer1<T> {
    @NotModified
    private final T[] data;

    public ArrayContainer1(T[] ts) {
        this.data = ts;
    }

    @NotModified
    @Independent1
    public Stream<T> stream() {
        return Arrays.stream(data);
    }
}
```

After creation, external changes to the source array `ts` are effectively modifications to the field `data`.

This construct fails rule 3, as the parameter `ts` is dependent. The field is a modifiable data structure, and must be shielded from external modifications.

Note the use of the lighter form of independence, `@Independent1` , on the return value of `stream()`, to indicate that modifications to the hidden content are possible on objects obtained from the stream.

Example 20, explaining level 2 immutability: with array, version 2, not good

```
@E1Container
class ArrayContainer2<T> {
    @NotModified
    public final T[] data;

    public ArrayContainer2(@Independent1 T[] ts) {
        this.data = new T[ts.length];
        System.arraycopy(ts, 0, data, 0, ts.length);
    }

    @NotModified
    @Independent1
    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 21, explaining level 2 immutability: with array, version 3, safe

```
@E2Container
class ArrayContainer3<T> {
    @NotModified
    private final T[] data; ①

    public ArrayContainer3(@Independent1 T[] ts) {
        this.data = new T[ts.length]; ②
        System.arraycopy(ts, 0, data, 0, ts.length);
    }

    @NotModified
    @Independent1
    public Stream<T> stream() {
        return Arrays.stream(data);
    }
}
```

① The array is private, and therefore protected from external modification via the direct access route.

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 modifiable structure, rather than someone else's. Here is the same group of examples, now with JDK Collections:

Example 22, explaining level 2 immutability: with collection, version 1, not good

```
@E1Container
class SetBasedContainer1<T> {
    @NotModified
    private final Set<T> data;

    @Dependent
    public SetBasedContainer1(Set<T> ts) {
        this.data = ts; ①
    }

    @NotModified
    @Independent1
    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 23, explaining level 2 immutability: with collection, version 2, not good

```
@E1Container
class SetBasedContainer2<T> {
    @NotModified
    public final Set<T> data; ①

    public SetBasedContainer2(@Independent1 Set<T> ts) {
        this.data = new HashSet<>(ts);
    }

    @NotModified
    @Independent1
    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 24, explaining level 2 immutability: with collection, version 3, safe

```
@E2Container
class SetBasedContainer3<T> {
    @NotModified
    private final Set<T> data; ①

    public SetBasedContainer3(@Independent1 Set<T> ts) {
        this.data = new HashSet<>(ts); ②
    }

    @NotModified
    @Independent1
    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. The next one is level 2 immutable as well:

Example 25, explaining level 2 immutability: with collection, version 4, safe

```
@E2Container
class SetBasedContainer4<T> {

    @E2Container
    public final Set<T> data; ①

    public SetBasedContainer4(@Independent1 Set<T> ts) {
        this.data = Set.copyOf(ts); ②
    }

    @NotModified
    @Independent1
    public Stream<T> stream() {
        return data.stream();
    }
}
```

① the data is public, but the `Set` is `@E2Immutable` itself, because its content is the result of `Set.copyOf`, which is an implementation that blocks any modification.

② Independence guaranteed.

The section on Dynamic type annotations will explain how the `@E2Container` annotation travels to the field `data`.

The independence rule, rule 3, is there to ensure that the type does not expose its modifiable data

through parameters and return types:

Example 26, explaining level 2 immutability: with collection, version 5, not good

```
@E1Container
class SetBasedContainer5<T> {
    @NotModified
    private final Set<T> data; ①

    public SetBasedContainer5(@Independent1 Set<T> ts) {
        this.data = new HashSet<>(ts); ②
    }

    @NotModified
    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. The presence of the getter is equivalent to adding the modifiers `public final` to the field.

Note that by decomposing rules 0 and 1, 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 final example shows a type which violates this rule 1, because a modifying method has been added:

Example 27, explaining level 2 immutability: with collection, version 6, not good

```
@E1Container
class SetBasedContainer6<T> {
    @Modified
    public final Set<T> set = new HashSet<>();

    @Modified
    public void add(@Independent1 T t) { set.add(t); }

    @NotModified
    @Independent1
    public Stream<T> stream() { return set.stream(); }
}
```

9.2. 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, in terms of inheritance, is that the analyser prohibits changing the modification status of methods from non-modifying to modifying in a derived type. This means, for example, that the analyser will block a modifying `equals()` or `toString()` method, in any class. Similarly, no implementation of `java.util.Collection.size()` will be allowed to be modifying.

The guiding principle here is that of *consistency of 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. A method overriding one marked `@Modified` does not have to be modifying, but it is not allowed to be explicitly marked `@NotModified`:

Example 28, illegal modification status of methods

```
abstract class MyString implements Collection<String> {
    private String string = "";

    @Override
    public int size() {
        string = string + "!"; ①
        return string.length();
    }

    @Override
    @NotModified ②
    public abstract boolean add(String s);
}
```

① Not allowed! Any implementation of `Collection.size()` must be non-modifying.

② Not allowed! You cannot explicitly (contractually) change `Collection.add()` from `@Modified` to `@NotModified` in a sub-type.

Following the same principles, we observe that types deriving from a `@Container` super-type need not be a container themselves. So while we may state that `Collection` is a container, it is perfectly possible to implement a collection which has public methods which modify their parameters, *as long as the methods inherited from* `Collection` *do not modify their parameters*. In other words, you can add new parameter-modifying methods, but you cannot change the modification status of `size`!

Note that sealed types (since JDK 17) reject the 'you can always extend' assumptions of Java types. In this case, all sub-types are known, and visible. The single practical consequence is that if the parent type is abstract, its annotations need not be contracted: they can be computed because all implementations are available to the analyser.

9.3. Generics

Type parameters are either *unbound*, in which case they can represent any type, or they explicitly extend a given type. Because the unbound case is simply a way of saying that the type parameter extends `java.lang.Object`, we can say that all type parameters extend a certain type, say `T extends`

`E`.

The analyser simply treats the parameterized type `T` as if it were the type `E`. In the case of an unbound parameter type, only the public methods of `java.lang.Object` are accessible. By definition, the type belongs to the hidden content, as defined in Accessible and hidden content.

The analyser recognises types that can be replaced by an unbound parameter type, when they are used *transparently*, and therefore belong to the hidden content: no methods are called on it, save the ones from `java.lang.Object`; none of its fields are accessed, and it is not used as an argument to parameters where anything more specific than `java.lang.Object` is required. It will issue a warning, and internally treat the type as an unbound parameter type, and hence `@E2Container` , even if the type is obviously modifiable.

The following trivial example should clarify:

Example 29, a type used transparently in a class

```
@E2Container
public class OddPair {

    private final Set<String> set;
    private final StringBuilder sb;

    public OddPair(Set<String> set, StringBuilder sb) {
        this.set = set;
        this.sb = sb;
    }

    public Set<String> getSet() { return set; }
    public StringBuilder getSb() { return sb; }
}
```

Nowhere in `OddPair` do we make actual use of the fact that `set` is of type `Set`, or `sb` is of type `StringBuilder`. The analyser encourages you to replace `Set` by some unbound parameter type, say `K`, and `StringBuilder` by some other, say `V`. The result is, of course, the type `Pair` as defined earlier.

Making a concrete choices for a type parameter may have an effect on the immutability level, as will be explained in Hidden content. Some examples are easy to see: any level 1 immutable type whose fields consists only of types of unbound type parameter, will become deeply immutable when the unbound type parameters are substituted for deeply immutable types. Any level 2 immutable type whose hidden content consists only of types of unbound type parameter, will become deeply immutable when the unbound type parameters are substituted for deeply immutable types. The `Pair` mentioned before is a case in point, and an example for both rules: `Pair<Integer, Long>` is deeply immutable.

9.4. Abstract methods

Because `java.lang.Object` is a level 2 immutable container, trivial extensions are, too:

Example 30, trivial extensions of `java.lang.Object`

```
@E2Container
interface Marker { }

@E2Container
class EmptyClass { }

@E2Container
class ImplementsMarker implements Marker { }

@E2Container
class ExtendsEmptyClass extends ImplementsMarker { }
```

Things only become interesting when methods enter the picture. Annotation-wise, we stipulate that

Unless otherwise explicitly annotated, we will assume that abstract methods, be they in interfaces or abstract classes, are `@NotModified`.

Furthermore, we will also impose special variants of the rules for level 2 immutability of an abstract type `T`, to be obeyed by the abstract methods:

Variant of rule 1: Abstract methods must be non-modifying.

Variant of rule 3: Abstract methods returning values must be independent, i.e., the object they return must be not be dependent on the fields. They cannot expose the fields via parameters: parameters of non-primitive, non-level 2 immutable type must not be dependent.

The consequence of these choices is that implementations and extensions of abstract and non-abstract types will have the opportunity to have the same immutability properties. This allows us, e.g., to think of every implementation of `java.util.Set` as a level 1 immutable container, if we limit to the public methods of `Set`. Similarly, we can treat any implementation of `Comparable`, defined as:

Example 31, `java.lang.Comparable` *annotated*

```
@E2Container
interface Comparable<T> {

    // @NotModified implicitly present
    int compareTo(@NotModified T other);
}
```

as a level 2 immutable type when the only method we can access is `compareTo`.

As for as the modification status of the *parameters* of abstract methods is concerned, we start off with `@Modified` rather than with `@NotModified`:

Unless otherwise explicitly annotated, or their types are level 2 immutable, we will assume that the parameters of abstract methods, be they in interfaces or abstract classes, are `@Modified` . Overriding the method, the contract can change from `@Modified` to `@NotModified`, but not from `@NotModified` to `@Modified` .

While it is possible to compute the immutability and container status of interface types, using the rules presented above, it often makes more practical sense to use the annotations as contracts: they may save a lot of annotation work on the abstract methods in the interface. We repeat that no implementation of a level 2 immutable interface is guaranteed to be level 2 immutable itself; nor does this guarantee hold for the container property unless no new non-private methods have been added.

We continue this section with some examples which will form the backbone of the examples in Hidden content.

If semantically used correctly, types implementing the `HasSize` interface expose a single numeric aspect of their content:

Example 32, the `HasSize` interface

```
@E2Container // computed (or contracted)
interface HasSize {

    // implicitly present: @NotModified
    int size();

    @NotModified // computed, not an abstract method!
    default boolean isEmpty() {
        return size() == 0;
    }
}
```

We extend to:

Example 33, still level 2 immutable: `NonEmptyImmutableList`

```
@E2Container // computed, contracted
interface NonEmptyImmutableList<T> extends HasSize {

    // implicitly present: @NotModified
    @Independent1 ①
    T first();

    // implicitly present: @NotModified
    void visit(@Independent1 Consumer<T> consumer); ② ③

    @NotModified ④
    @Override
    default boolean isEmpty() {
        return false;
    }
}
```

① Whilst formally, `T` can never be dependent because it must belong to the hidden content of the interface, contracting the `@Independent1` annotation here will force all concrete implementations to have an non-dependent `first` method. If the concrete choice for `T` is modifiable, the independence rule must be satisfied.

② The parameter `consumer` would normally be `@Modified` , which would break the `@Container` property that we wish for `NonEmptyImmutableList`. However, as detailed and explained in Hidden content, the abstract types in `java.util.function` receive an implicit `@IgnoreModifications` annotation.

③ The hidden content of the type is exposed to the outside world via the `accept` method in the consumer, similarly to being exposed via the return value of the `first` method.

④ Computed, because it is not an abstract method.

The `Consumer` interface is defined and annotated as:

Example 34, the java.util.function.Consumer interface, annotated

```
@FunctionalInterface
interface Consumer<T> {

    @Modified
    void accept(T t); // @Modified on t implicit
}
```

Implementations of the `accept` method are allowed to be modifying (even though in `NonEmptyImmutableList.visit` we decide to ignore this modification!). They are also allowed to modify their parameter, as we will demonstrate shortly.

Let's downgrade from `@E2Container` to `@E1Container` by adding a modifying method:

Example 35, not level 2 immutable anymore: `NonEmptyList`

```
@E1Container
interface NonEmptyList<T> extends NonEmptyImmutableList<T> {

    @Modified
    void setFirst(@NotModified @Independent1 T t);
}
```

The method `setFirst` goes against the default annotations twice: because it is modifying, and because it promises to keep its parameter unmodified. The `@Independent1` annotation states that arguments to `setFirst` will end up in the hidden content of the `NonEmptyList`. Implementations can even lose level 1 immutability:

Example 36, mutable implementation of `NonEmptyList`

```
@Container
static class One<T> implements NonEmptyList<T> {

    @Variable
    private T t;

    @NotModified
    @Override
    public T first() {
        return t;
    }

    @Modified
    @Override
    public void setFirst(T t) {
        this.t = t;
    }

    @NotModified
    @Override
    public int size() {
        return 1;
    }

    @NotModified
    @Override
    public void visit(Consumer<T> consumer) {
        consumer.accept(t);
    }
}
```

Here is a (slightly more convoluted) implementation that remains `@E1Container` :

Example 37, level 1 immutable implementation of `NonEmptyList`

```
@E1Container
static class OneWithOne<T> implements NonEmptyList<T> {
    private final One<T> one = new One<>();

    @NotModified
    @Override
    public T first() {
        return one.first();
    }

    @Modified
    @Override
    public void setFirst(T t) {
        one.setFirst(t);
    }

    @NotModified
    @Override
    public int size() {
        return 1;
    }

    @NotModified
    @Override
    public void visit(Consumer<T> consumer) {
        consumer.accept(first());
    }
}
```

Obviously, a `@E2Container` implementation is not possible: the immutability status of an extension (`OneWithOne`, `One`) cannot be better than that of the type it is extending from (`NonEmptyList`).

We end the section by showing how concrete implementations of the `accept` method in `Consumer` can make modifications. First, modifications to the parameter:

Example 38, modification to the parameter of `Consumer.accept`

```
One<StringBuilder> one = new One<>();
one.setFirst(new StringBuilder());
one.visit(sb -> sb.append("!"));
```

The last statement is maybe more easily seen as:

Example 39, modification to the parameter of `Consumer.accept`*, written out*

```
one.visit(new Consumer<StringBuilder> {

   @Override
   public void accept(StringBuilder sb) {
       sb.append("!");
   }
});
```

Second, modifications to the fields of the type:

Example 40, the method `Consumer.accept` *modifying a field*

```
@E1Container
class ReceiveStrings implements Consumer<String> {

    @Modified
    public final List<String> list = new ArrayList<>();

    @Modified
    @Override
    public void accept(String string) {
        list.add(string);
    }
}
```

9.5. 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:

Example 41, modifications on static fields are modifications

```
@E1Container
public class CountAccess<K> {

    @NotModified
    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:

Example 42, modification on static field, explicitly ignored

```
@E2Container
public class CountAccess<K> {

    @NotModified
    private final K k;

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

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

    @NotModified ①
    public K getK() {
        counter.getAndIncrement(); ①
        return k;
    }

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

① The effects of the modifying method `getAndIncrement` are ignored.

Note that when the modification takes place inside the constructor, it is still not ignored, because for static fields, static code blocks act as the constructor:

Example 43, modification of static field can occur inside constructor

```
@E1Container
public class HasUniqueIdentifier<K> {

    public final K k;
    public final int identifier;

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

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

Only modifications in a static code block are ignored:

Example 44, static code blocks are the constructors of static fields

```
public class CountAccess<K> {
    ...
    private static final AtomicInteger counter;

    static {
        counter = new AtomicInteger();
        counter.getAndIncrement(); ①
    }
    ...
}
```

① Modification, part of the construction process.

Nevertheless, we introduce the following rule which does distinguish between modifications on static and instance types:

> 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 is still consistent 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

Example 45, static side effects annotation

```
@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 giving access to the singleton.

9.6. 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 a requirement to be a value-based class. However, we expect many level 2 immutable types will become value-classes. Revisiting the example from the previous section, we can construct a counter-example:

Example 46, 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.

9.7. 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':

Example 47, revisiting `SetBasedContainer6`

```
@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.

10. Eventual immutability

In this section we explore types which follow a two-phase life cycle: the start off as mutable, then somehow become immutable.

10.1. Builders

We start with the well-established *builder* paradigm.

Example 48, static nested builder type

```
@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 {

    @E2Container
    public final List<Point> points;

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

    @E1Container(builds=Polygon.class)
    static class Builder {

        @Modified
        private final List<Point> points = new ArrayList<>();

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

        @NotModified
        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*:

Example 49, use of `FirstThen` *to make a type 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);
    }
}
```

10.2. Definition

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

Example 50, state change in a boolean field

```
@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 IllegalStateException();
        set.add(t);
    }

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

    @Only(after="frozen")
    public Stream<T> stream() {
        if(!frozen) throw new IllegalStateException();
        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:

Example 51, state change going from empty to non-empty

```
@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 IllegalStateException();
        if(data.isEmpty()) throw new IllegalArgumentException();
        set.addAll(data);
    }

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

Example 52, utility method for `EventuallyFinal`

```
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.

10.3. 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.

10.4. 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,

Example 53, `@BeforeMark` *annotation*

```
@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.

10.5. 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.

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

10.6. 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.

Example 54, 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, mid 2021, contracted eventual immutability has not been implemented yet in the analyser.

11. Modification, part 2

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

11.1. 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:

Example 55, 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 very well. It imposes more annotation work on the programmer, however, because the interface's methods need contracts.

11.2. How to compute linking

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 `@E2Immutable` type, there cannot be any linking; `v` does not link to `a` nor `b`.
2. If `b` is of `@E2Immutable` type, `v` cannot link to `b`.
3. When `method` has the annotation `@Independent` (or the lighter version, `@Independent1`), `v` cannot link to `a`.

Recall that primitives, `java.lang.Object`, `java.lang.String`, and unbound parameter types, are `@E2Immutable` .

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

1. When `b` is of `@E2Immutable` type, `v` cannot link to `b`.
2. If `A` is `@E2Immutable` , then `v` cannot link to `b`, because all its constructor parameters are independent.
3. When `b` has been marked `@Independent` or `@Independent1` , `v` cannot link to `b`.

> Rule 4: in a modifying method call `a.method(b)`, `a` potentially links to `b`

This situation is similar to that of the constructor (rule 3), with `a` taking the role of `v`.

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. a pattern variable `p` in an instance-of statement `a instanceof P p` links to `a`
4. Binary operators return primitives or `java.lang.String`, which prevents linking: in `v = a + b`, `v` does not link to `a` nor `b`.

Note: in a method call `v = a.method(b, c, d)`, links between `b`, `c`, and `d` are possible. They are covered by the `@Modified` annotation: when a parameter is `@NotModified`, no modifications at all are possible, not even indirectly. We do not compute individual linking, because we advocate the use of containers: all parameters should be `@NotModified`.

11.3. Locally implemented abstract methods

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: the one where the SAMs have a local implementation. The general case, where objects of abstract types come in via a

parameter, will be addressed in Hidden content. Consider the following example:

Example 56, concrete implementation of suppliers

```
@E1Container
class ApplyLocalFunctions {

    @Container
    static class Counter {
        private int counter;

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

    @Modified ①
    private final Counter myCounter = new Counter();

    @Modified ②
    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();
    }
}
```

① Modified in `getAndIncrement` and `explicitGetAndIncrement`

② Modified because its modifying method (`get`) is called in `myIncrementer`

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 can discuss:

Example 57, 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 cause 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.

12. Hidden content

In this section, we consider modifications to the hidden content of a type, and explain when they are of importance. We give a formal definition of deeply or recursively immutable types, introduce the annotation `@ERContainer` , as well as a sliding scale between level 2 and recursively immutable. The annotation `@Independent1` , which we have touched upon earlier earlier, will be formally introduced.

12.1. Visitors

Let's start by going back to `NonEmptyImmutableList`, first defined in Abstract methods:

Example 58, revisiting `NonEmptyImmutableList`

```
@E2Container
interface NonEmptyImmutableList<T> extends HasSize {

    // implicitly present: @NotModified
    @Independent1
    T first();

    // implicitly present: @NotModified
    void visit(@Independent1 Consumer<T> consumer); // @IgnoreModifications

    @NotModified
    @Override
    default boolean isEmpty() {
        return false;
    }
}
```

We start the discussion with the following level 2 immutable implementation of this interface:

Example 59, level 2 immutable implementation of `NonEmptyImmutableList`

```
@E2Container
class ImmutableOne<T> implements NonEmptyImmutableList<T> {
    private final T t;

    public ImmutableOne(@Independent1 T t) {
        this.t = t;
    }

    @Override
    public int size() {
        return 1;
    }

    @Override
    public T first() {
        return t;
    }

    @Override
    public void visit(Consumer<T> consumer) {
        consumer.accept(t);
    }
}
```

We need the `visit` method (1) to be non-modifying, and (2) not to modify its parameter `consumer`. However, following the normal definitions of modification, the following two statements hold:

1. Because `accept` is `@Modified` , we should mark the parameter `consumer` as `@Modified` .
2. Because `t`, the parameter of `accept`, is `@Modified` , we should mark `visit` as `@Modified` .

The result of the first statement would violate the `@Container` property on `ImmutableOne`, and we'd be very reluctant to do that: according to the intuitive definition in Containers, `ImmutableOne` is a type that holds data, but does not change it. This statement still holds in the presence of a `visit` method, which is nothing but a way of exposing the object in a way similar to the method `first`. The second one would make `visit` modifying, which again goes against our intuition: looping over elements is, in itself, not modifying.

Luckily, there are two observations that come to the rescue.

First, we believe it is correct to assume that concrete implementations of `Consumer` can be semantically unrelated to `ImmutableOne`. As a consequence, we could say that the only modifications that concern us in this `visit` method are the modifications to `accept` 's parameter `t`. Other modifications, for example those to the fields of the type in which the implementation is present, may be considered to be outside our scope. However, if we replace `Consumer` with `Set` and `accept` with `add`, we encounter a modification that we really do not want to ignore, in an otherwise equal setting. Therefore, it does not look like we can reason away potential modifications by `accept`. We will have to revert to a contracted `@IgnoreModifications` annotation on the parameter `consumer`, if we want to avoid `ImmutableOne` losing the `@Container` property.

The second modification, however, is one we will ignore in the `ImmutableOne` type, and *defer* or *propagate* to the place where a concrete implementation of the consumer is presented. We can ignore it here, because `t` is part of the hidden content of the type; what happens to its content happens outside the zone of control of `ImmutableOne`. The fact that it is passed as an argument to a method of `consumer` is reflected by the `@Independent1` annotation. It will take care of the propagation of modifications from the concrete implementation into the hidden content.

This results in the following annotations for `visit` in `ImmutableOne`:

Example 60, the `visit` method in `ImmutableOne`, fully annotated

```
@NotModified
public void visit(@IgnoreModifications @Independent1 Consumer<T> consumer) {
    consumer.accept(t);
}
```

Note that we assume that we will need `@IgnoreModifications` for almost every use of a functional interface from `java.util.function` occurring as a parameter. These types are for generic use; one should never use them to represent some specific data type where modifications are of concern to the current type. Therefore, we make this annotation implicit in exactly this context.

A parameter of formal functional interface type will be marked `@IgnoreModifications` implicitly.

Looking at the more general case of a `forEach` implementation iterating over a list or array, we therefore end up with:

Example 61, a generic `forEach` implementation

```
@NotModified
public void forEach(@Independent1 Consumer<T> consumer) {
    for(T t: list) consumer.accept(t);
}
```

Modifications to the parameter, made by the concrete implementation, are propagated into the hidden content of `list`, as shown in the next section. The `@Independent1` annotation appears because hidden content in `list` is exposed to the `consumer` parameter. This annotation does not appear for the accessible content of the level 2 immutable type. Parameters of modifiable type are already shielded from external modification by the `@Independent` annotation, which is "trivial" for level 2 immutable types.

12.2. Propagating modifications

Let us apply the `visit` method of `NonEmptyImmutableList` to `StringBuilder`:

Example 62, propagating the modification of `visit`

```
static void print(@NotModified NonEmptyImmutableList<StringBuilder> list) {
    one.visit(System.out::println); ①
}

static void addNewLine(@Modified NonEmptyImmutableList<StringBuilder> list) {
    one.visit(sb -> sb.append("\n")); ②
}
```

① Non-modifying method implies no modification on the hidden content of `list`.

② Parameter-modifying lambda propagates a modification to `list` 's hidden content.

It is the second method, `addNewLine`, that is of importance here. Thanks to the {dependent1} annotation, we know of a modification to `list`, even if `list` is of level 2 immutable type! It may help to see the for-loop written out, if we temporarily assume that we have added an implementation of `Iterable` to `NonEmptyImmutableList`, functionally identical to `visit`:

Example 63, alternative implementation of `addNewLine`

```
static void addNewLine(@Modified NonEmptyImmutableList<StringBuilder> list) {
    for(StringBuilder sb: list) {
      sb.append("\n"));
    }
}
```

We really need the link between `sb` and `list` for the modification on `sb` to propagate to `list`. Without this propagation, we would not be able to implement the full definition of modification of parameters, as stipulated in Modification, in this relatively straightforward, and probably frequently occurring situation.

Moving from `NonEmptyImmutableList` to `NonEmptyList`, defined here, which has a modifying method, allows us to contrast two different modifications:

Example 64, contrasting the modification on the parameter `sb` to that on `list`

```
static void addNewLine(@Modified NonEmptyList<StringBuilder> list) {
    list.visit(sb -> sb.append("\n")); ①
}

static void replace(@Modified NonEmptyList<StringBuilder> list) {
    list.setFirst(new StringBuilder("?")); ②
}
```

① Modification to the hidden content of `list`

② Modification to the modifiable content of `list`

Without storing additional information (e.g., using an as yet undefined annotation like `@Modified1` on `list` in `addNewLine`), however, we cannot make the distinction between a modification to the string builders inside `list`, or a modification to `list` itself. In other words, applying the two methods further on, we cannot

Example 65, using `print` and `addNewLine`

```
static String useAddNewLine(@NotModified StringBuilder input) { ①
    NonEmptyList<StringBuilder> list = new One<>();
    list.setFirst(input);
    addNewLine(list);
    return list.getFirst().toString();
}

static String useReplace(@NotModified StringBuilder input) {
    NonEmptyList<StringBuilder> list = new One<>();
    list.setFirst(input);
    replace(list); ②
    return list.getFirst().toString();
}
```

① Should be `@Modified` , however, in the 3rd statement we cannot know that the modification is to `input` rather than to `list`

② This action discards `input` from `list` without modifying it.

> The example shows that the introduction of {dependent1} only gets us so far: from the concrete, modifying implementation, to the parameter (or field). We do not plan to keep track of the distinction between modification of hidden content vs modification of modifiable content to a further extent.

Finally, we mention again the modification to a field from a concrete lambda:

Example 66, modification of a field outside the scope

```
List<String> strings = ...
@Modified
void addToStrings(@NotModified NonEmptyList<StringBuilder> list) {
  list.visit(sb -> strings.add(sb.toString()));
}
```

12.3. Hidden content linking

Going back to `ImmutableOne`, we see that the constructor links the parameter `t` to the instance's field by means of assignment. Let us call this binding of parameters of hidden content to the field *content linking*, and mark it using {dependent1}, *content dependence*:

Example 67, constructor of `ImmutableOne`

```
private final T t;

public ImmutableOne(@Independent1 T t) {
    this.t = t;
}
```

Returning a part of the hidden content of the type, or exposing it as argument, both warrants a {dependent1} annotation:

Example 68, more methods of `ImmutableOne`

```
@Independent1
@Override
public T first() {
    return t;
}

@Override
public void visit(@Independent1 Consumer<T> consumer) {
    consumer.accept(t);
}
```

Observe that content dependence implies 'normal' independence, as described in Linking, independence and How to compute linking, exactly because we are dealing with parameters of level 2 immutable type.

Another place where the hidden content linking can be seen, is the *for-each* statement:

Example 69, for-each loop and hidden content linking

```
ImmutableList<StringBuilder> list = ...;
List<StringBuilder> builders = ...;
for(StringBuilder sb: list) {
    builders.add(sb);
}
```

Because the `Collection` API contain an `add` method annotated as:

Example 70, `add` in `Collection` annotated

```
@Modified
boolean add(@NotNull @Independent1 E e);
```

indicating that after calling `add`, the argument will become part of the hidden content of the collection, we conclude that the local loop variable `sb` gets content linked to the `builders` list. Similarly, this loop variable contains hidden content from the `list` object.

We reuse the annotation {dependent1} to indicate that the hidden content of two objects are linked. Let us look at a possible implementation of `Collection.addAll`:

Example 71, a possible implementation of `addAll` in `Collection`

```
@Modified
boolean addAll(@NotNull1 @Independent1 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 hidden content linked to the hidden content of the instance.

We are now properly armed to see how a for-each loop can be implemented using an iterator whose hidden content links to that of a level 1 immutable container.

12.4. Iterator, Iterable, loops

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

Example 72, the `Iterator` type, without `remove` method

```
@E1Container
interface Iterator<T> {

    @Modified
    @Independent1
    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 hidden content held by the iterator.

The interface `Iterable` is a supplier of iterators:

Example 73, the `Iterable` type

```
@E2Container
interface Iterable<T> {

    @Independent1
    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`. Second, the iterator itself is independent of the fields of the implementing type, but has the ability to return its hidden content.

The loop, on a variable `list` of type implementing `Iterable<T>`, is expressed as `for(T t: list) { ⋯ }`, and can be interpreted as

Example 74, implementation of for-each using an `Iterator`

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

The iterator `it` content-links to `list`; via the `next` method, it content-links the hidden content of the `list` to `t`.

12.5. Eventual immutability

How does the whole story of eventual level 1 or level 2 immutability mix with hidden content? At some point, once a necessary precondition has been met, the hidden content will be well-defined, and modifying methods become unavailable. Before that, fields that will eventually contain the hidden content may still be `null`, or may be re-assigned. This should not have any effect, however, on the computation of hidden content linking, `@Independent1` annotations, and the propagation of modifications, since the actual types do not change. The two concepts are sufficiently perpendicular to each other, and can easily co-exist.

13. Higher-order immutability

13.1. Definition

Without much ado, we extend the definition of immutability, by adding a single rule:

Definition: level n immutability, n > 2:

(**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 type.

Rule 3: No parameters of non-private methods or non-private constructors, no return values of non-private methods, are dependent on (the accessible part of) the fields.

Rule 4: The hidden content of the type is at least level n-1 immutable.

In a level 2 immutable type, the hidden content can be mutable or level 1 immutable. In a level 3 immutable type, the hidden content has to be level 2 immutable. Annotation-wise, we use `@E2Container(level = 4)`, `@ERContainer` in combination with the `@Container` property, and `@E2Immutable(level = 3)`, `@E2Immutable(recursive = true)` in its absence.

Why this definition? Firstly, it is consistent with what we would think of as recursively or deeply immutable types: their hidden content is either absent or recursively immutable itself; think n equals infinity. In reality, of course, types are build from the "ground up" starting from recursively immutable types and the array construct. Observe that a level 2 immutable type whose hidden content consists of recursively immutable types only, becomes recursively immutable itself.

Secondly, it allows us to quantify the level of independence as well.

13.2. Independence of types

A concrete implementation of an iterator is often a nested type, static or not (inner class), of the iterable type:

Example 75, implementation of an `Iterator`

```
@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
    @Independent1
    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`, in other words, the `IteratorImpl` object must not expose the `ImmutableArray` 's fields to the outside world. It cannot be level 2 immutable, because it needs to hold the state of the iterator. However, it should protect the fields owned by its enclosing type, up to the same standard as required for level 2 immutability.

We propose to add a definition for the independence of a type, similar to the "shielding off" part of the definition of level 2 immutability:

Definition: an **external modification** is a modification, carried out outside the type,

1. on a field, directly accessed from the object, or
2. on an argument or return value, executed after the constructor or method call on the object.

Clearly, such external modifications are not possible when the type of the argument, return value, or field is level 2 immutable. External modifications are only possible when the constructor, method or field is non-private.

Armed with this definition, we can define the independence of types:

Definitions:

A type is **dependent** when external modifications impact the accessible content of the type.

A type is **level 1 independent**, annotated `@Independent1` , when external modifications cannot impact the accessible content of the type. The hidden content of the type is mutable or modifiable.

A type is **level n independent, n>1**, annotated `@Independent1(level = n)`, when external modifications cannot impact the accessible content of the type. The hidden content of the type is level n immutable.

A type is **independent**, annotated `@Independent` , when external modifications cannot impact the content of the type.

Consider the static variant of `IteratorImpl`, which makes it more obvious that `IteratorImpl` maintains a reference to the element array of its enclosing type:

Example 76, implementation of an `Iterator` as a static nested type

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

    @Container
    @Independent1
    static class IteratorImpl implements Iterator<T> {
        @Modified
        private int i;

        private final T[] elements;

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

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

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

The type `T` is part of the hidden content, the `T[]` and the counter `i` are part of the accessible content. No external modification can impact the array or the counter; indeed, only `T` and a `boolean` are exposed. The latter is recursively immutable, so does not allow modifications. The former allows modifications on the hidden content, whence the `@Independent1` annotation for `IteratorImpl`.

Recursively immutable types are independent as a type, but a type does not even have to be level 1 immutable to be independent. In fact, any type communicating via recursively immutable types to the outside world is independent:

Example 77, simple getter and setter, independent

```
@Independent
@Container
class GetterSetter {
    private int i;

    public int getI() {
        return i;
    }

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

The following table summarizes the relationship between immutability and independence:

	Mutable, Level 1 imm (modifiable)	**Level 2 immutable**	**Level 3 immutable**	**Level n immutable**	**Recursively immutable**
Dependent	✔ `Set`	✘	✘	✘	✘
Independent1	✔ `Iterator<T>`	✔ `Optional<T>`, `Set.of(T)`	✘	✘	✘
Independent2	✔ `Iterator<Optional<T>>`	✘	✔ `Set.of(Optional<T>)`	✘	✘
Independent n-1	✔	✘	✘	✔	✘
Independent	✔ `Writer`, `Iterator<String>`	✘	✘	✘	✔ `int`, `String`, `Class`

There is no need to define different levels of hidden content linking of parameters and fields, beyond "independent or not". The important aspect there is whether linking to the hidden content is possible or not. Either it is not, or it is structurally impossible. The latter is only possible when the type is recursively immutable, which corresponds to the type being independent.

13.3. Field access restrictions

Let us end this section with a note on the *non-private* requirement for field and method access. The definitions of immutability and independence insist on the properties holding for all non-private fields, methods and constructors.

First, consider nested types. Any nested type (a class defined either statically or nested inside another class, an interface defined inside another type) has access to the private methods of the primary type and other nested types inside the primary type. We first need to investigate whether

this additional access plays havoc with the immutability and independence rules.

Because all nested types of a primary type are fully known at analysis time, as they must reside in the same `.java` file, it is possible to ensure that a field, accessible beyond its own class even though it is private to the nested type, remains `@NotModified`. Consider:

Example 78, immutability of a nested type

```
public class NestedTypeExample {

    @E1Container ①
    static class HoldsStringBuilder {

        @Modified ②
        private final StringBuilder sb = new StringBuilder();

        public HoldsStringBuilder(String s) {
            add(s).add(s);
        }

        private HoldsStringBuilder add(String s) { ③
            sb.append(s);
            return this;
        }

        @Override
        public String toString() {
            return sb.toString();
        }
    }

    public static String break1(String s) {
        HoldsStringBuilder hsb = new HoldsStringBuilder(s);
        hsb.add("modify!");
        return hsb.toString();
    }

    public static String break2(String s) {
        HoldsStringBuilder hsb = new HoldsStringBuilder(s);
        hsb.sb.append("modify field");
        return hsb.toString();
    }

    public static StringBuilder break3(String s) { ④
        HoldsStringBuilder hsb = new HoldsStringBuilder(s);
        hsb.sb.append("modify field");
        return hsb.sb;
    }
}
```

① Would have been `@E2Container` , were it not for the `break` methods

② Because of `break2`

③ Not only part of construction, because of `break1`

④ Introduces a dependence of `sb` on a method return value

The solution here, clearly, is to extend the rules to all non-private methods and constructors of the primary type and all its nested types.

The second question to answer is whether we can or should relax the requirement of private access, e.g., for a restriction of 'private and same package', or even 'non-public'. Remember that the `protected` access modifier allows access to classes that inherit from the type, *and* to members of the same package.

First, consider allowing 'package-private'. If we were to assume that all types in the same package are fully visible to the analyser at the time of analysis, we could consider extending the rules to analyse all types in the package at the same time, as we did for nested types inside a primary type. However, firstly, it is perfectly possible, even if it is bad practice, to spread a package over multiple jars. This denies the analyser complete visibility over the types in a package. Secondly, the complications that arise computationally are too much for efficient analysis.

So there's no point in considering `protected` access. Even if inheritance where the only criterion used to define this access level, we would not allow it, because the child class can be invisible to the analyser at the time of analysis of the parent.

When annotating APIs (see *e2immu* manual), we do use the public vs non-public criterion instead of the non-private vs private one, mostly as a matter of convenience. We assume (hope?) that library designers and implementers shield off internal types sufficiently, and rely on the project implementer to stick to their package prefix.

14. 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.

14.1. 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:

Example 79, most of `org.e2immu.support.FlipSwitch`

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

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

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

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

    private boolean copy$Precondition() { return !t; } ①
    @Mark("t") ②
    @Modified
    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.

② The `@Mark` is present, even if it is executed conditionally.

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

14.2. SetOnce

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

Example 80, parts of `org.e2immu.support.SetOnce`

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

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

    @Mark("t")
    @Modified
    public void set(@NotNull @Independent1 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
    @Independent1
    @NotModified
    public T get() {
        if (t == null) {
            throw new IllegalStateException("Not yet set");
        }
        return t;
    }

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

    @Independent1 ①
    @NotModified
    public T getOrDefault(T defaultValue) {
        if (isSet()) return get();
        return defaultValue;
    }
}
```

① Even if it is only linked to the hidden content conditionally.

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.

14.3. 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:

Example 81, `org.e2immu.support.EventuallyFinal`

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

    @Independent1
    public T get() {
        return value;
    }

    @Mark("isFinal")
    public void setFinal(@Independent1 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(@Independent1 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;
    }
}
```

Note the occurrence of a negated `@TestMark` annotation: `isVariable` returns the negation of the normal `iFinal` mark test.

14.4. 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.

Example 82, `org.e2immu.support.Freezable`

```
@ERContainer(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 recursively immutable: derived classes can only go *down* the immutability scale, not up!

14.5. SetOnceMap

We discuss one example that makes use of (derives from) `Freezable`: a freezable map where no objects can be overwritten:

Example 83, part of `org.e2immu.support.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(@Independent1 @NotNull K k, @Independent1 @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);
    }

    @Independent1
    @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) { ②
        return map.containsKey(k);
    }

    ...
}
```

① The analyser will warn for 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.

② Implicitly, the parameter `K k` is `@Independent` , because the method is `@NotModified`, and it is not of abstract type.

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

14.6. Lazy

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

Example 84, `org.e2immu.support.Lazy`

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

    @NotNull1
    @Independent1(after = "t")
    private Supplier<T> supplier;

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

    public Lazy(@NotNull1 @Independent1 Supplier<T> supplier) { ①
        this.supplier = supplier;
    }

    @Independent1
    @NotNull
    @Mark("t") ②
    public T get() {
        if (t != null) return t;
        t = Objects.requireNonNull(supplier.get()); ③
        supplier = null; ④
        return t;
    }

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

① The annotation has travelled from the field to the parameter; therefore the parameter has `@Independent1` .

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

③ Here `t`, part of the hidden content, links to `supplier`, as explained in Hidden content linking. The statement also causes the `@NotNull1` annotation, as defined in Nullable, not null and Identity and fluent methods.

④ After the transition from mutable to effectively immutable, the field `supplier` moves out of the picture.

After calling the marker method `get()`, `t` cannot be assigned anymore, and it becomes `@Final` . The constructor parameter `supplier` is `@Independent1` , as its hidden content (the result of `get()`) links to that of `Lazy`, namely the field `t`.

But why is `supplier` as a field not linked to the constructor parameter? Clearly, `supplier` is part of the accessible content of `Lazy`, as its `get()` method gets called. The criterion is: a modification on one may cause a modification on the other. Modifications can only be made by calling the `get()` method, as there are no other methods, and no fields. Consequently, the constructor should link to the field,

and `supplier` cannot be `@Independent1` .

The answer lies in the eventual nature of `Lazy`: *before* the first call to `get`, the `supplier` field is of relevance to the type, and `t` is not. *After* the call to `get()`, the converse is true, because `supplier` has been emptied. We should extend rule 2 of effective immutability by slightly augmenting rule 2:

> **Rule 2**: All fields are either private, level 2 immutable type, or equal to null.

A null field cannot be modified, and cannot be but `@Independent` , so no changes are necessary to rules 1 and 3. One can argue that they do not belong to the accessible content, nor to the hidden content, since they cannot be accessed, and are content-less: rule 4 should not be affected. In combination with effective finality, this allows the eventually "blanking out" of modifiable fields in immutable types.

14.7. FirstThen

A variant on `SetOnce` is `FirstThen`, an eventually level 2 immutable container which starts off with one value, and transitions to another:

Example 85, `org.e2immu.support.FirstThen`

```
@E2Container(after = "mark")
public class FirstThen<S, T> {
    private volatile S first;
    private volatile T then;

    public FirstThen(@NotNull @Independent1 S first) {
        this.first = Objects.requireNonNull(first);
    }

    @TestMark(value = "first", before = true)
    @NotModified
    public boolean isFirst() {
        return first != null;
    }

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

    @Mark("mark")
    public void set(@Independent1 @NotNull T then) {
        Objects.requireNonNull(then);
        synchronized (this) {
            if (first == null) throw new IllegalStateException("Already set");
            this.then = then;
            first = null;
```

```
        }
    }

    @Only(before = "mark")
    @Independent1
    @NotModified
    @NotNull
    public S getFirst() {
        if (first == null)
            throw new IllegalStateException("Then has been set"); ①
        S s = first;
        if (s == null) throw new NullPointerException();
        return s;
    }

    @Only(after = "mark")
    @Independent1
    @NotModified
    @NotNull
    public T get() {
        if (first != null) throw new IllegalStateException("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.

14.8. 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`:

Example 86, explaining `org.e2immu.analyser.model.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`:

Example 87, explaining `org.e2immu.analyser.model.TypeInfo`

```
@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.

15. 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.

15.1. 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.

15.1.1. 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:

Example 88, `@NotNull` annotations on `Collection`

```
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 would be useful when working with entry sets:

Example 89, `@NotNull` annotations on `Map`

```
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. There is currently no plan to implement beyond `@NotNull1` , however.

15.2. 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:

Example 90, methods marked `@Identity` *and* `@Fluent`

```
@E1Container(builds=List.class)
class Builder {

    @NotNull
    @NotModified
    @Identity
    private static <T> T requireNonNull(@NotNull T t) {
        if(t == null) throw new IllegalArgumentException();
        return t;
    }

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

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

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

    @NotModified
    @ERContainer
    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();
}
```

15.3. 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, 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.

Note that the `@Finalizer` annotation is always contracted; it cannot be computed.

Let us start from the following example, using EventuallyFinal:

Example 91, 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
    @BeforeMark
    public EventuallyFinal<String> getData() {
        return data;
    }
}
```

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

Example 92, calling the finalizer method

```
@ERContainer
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:

Example 93, illegal call

```
@ERContainer
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.

15.3.1. 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.

Example 94, conceptual example of a 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
    @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.

Example 95, 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
    @ERContainer 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.

15.4. 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.

15.5. 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:

Example 96, an extension class

```
@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.

15.6. 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:

Example 97, 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:

Example 98, 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 IllegalStateException();
        created = true;
        this.k = k;
    }

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

16. 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:

Example 99, 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:

Example 100, 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,

Example 101, 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

Example 102, 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:

Example 103, adding an element to an empty list

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

the value of `list` becomes

Example 104, 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:

Example 105, `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.

17. Copyright and License