The Road to Immutability

Bart Naudts

Version 5, 2026-07-20

***Effective and eventual immutability with** maddi**, a static code analyzer for** Java.*

Table of Contents

Chapter 1. Introduction

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

The overarching aim of the *maddi* project is to improve everyday programming by making code more readable, more robust, and more future-proof. Concretely, the project focuses on adding various forms of immutability protections to your Java code base, by making the (im)mutable 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 scale 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 *maddi* project consists of a set of definitions, a static code analyzer to compute and enforce rules and definitions, and IDE support to visualize the results without cluttering. Using *maddi* 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 analyzers, and theory of software engineering and programming languages. Academically coming from the theory of machine learning, I spent two decades 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 realize I may be duplicating quite a lot here.

1.1. 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 to; 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 (the collective noun for class, interface, record, …) is private, or it is not.
- Synchronization is orthogonal to the data flow 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 *maddi* code analyzer warns for many other doubtful practices, as detailed in the user manual.

1.2. The purpose of annotations

In this document we will add many annotations to the code fragments shown. We are acutely aware annotations may 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 *maddi* code analyzer 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 analyzer 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 analyzer. We add them to the code as a means of verification: the analyzer will check if it generates the same annotation at that location.

Chapter 2. Final fields

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 analyzer annotates with `@Final` in the latter case; there is no point in cluttering with an annotation when the modifier is already there. Fields that are not effectively final are called *variable*, they can optionally be annotated with `@Final(absent=true)` .

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 in the class call initialize()

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

Obviously, if the same method can be called after construction, the field becomes variable:

Example 1, the method setting the field is accessible after construction

```
class EffectivelyFinal2 {
    @Final(absent = true)
    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 analyzer, 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 1, 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 1, 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 analyzer 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, limited circumstances.

The analyzer annotates types which are not immutable, but whose fields are all effectively final, with `@FinalFields` . Types that have at least one variable field are never immutable, and are optionally annotated with `@FinalFields(absent=true)` .

Note that the `record` type enforces explicitly final fields, along with additional support for equality and visibility. Any `record` will therefore be at least `@FinalFields` .

We will in this work make a distinction between a property being *effectively* or *eventually* present. The former indicates a property as computed after construction of the object, which is potentially a little broader than the definition of the language. The latter is used when this property can only be obtained after the code reaches a certain state. More on this later, but here is a first example of a type with eventually final fields:

Example 1, simplified version of `SetOnce`

```
import java.util.Random;

@FinalFields(after="random")
class OneRandom {
    private Random random;

    @Mark("random")
    public void set(Random r) {
        if(r == null) throw new NullPointerException();
        if(this.random != null) throw new IllegalStateException("Already set");
```

```
        this.random = r;
    }

    @Only(after="random")
    public Random get() {
        if(this.random == null) throw new IllegalStateException("Not yet set");
        return this.random;
    }
}
```

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

We have just observed that if one restricts to primitives and types like `java.lang.String`, final fields are 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 final fields alone are not enough to guarantee what we intuitively may think immutability stands for:

Example 1, final fields do not guarantee intuitive immutability

```
@FinalFields
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.

Chapter 3. 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 analyzer uses the annotations `@NotModified` and `@Modified` . They are exclusive, and the analyzer 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.

Why not `@Modifying` and `@NotModifying`? The analyzer will compute modifications of fields and parameters as well. They will either be *modified* or *not modified* by the code. To avoid confusion, we only use one set of annotations.

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 1, 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;
    }
}

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

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

```
}
```

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

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

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

We will apply a similar reasoning to a field:

> **Definition**: The analyzer 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 or modifying method to this field's object graph.

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 have restricted the annotations to `@NotModified` and `@Modified` — others will be introduced later. An abstract method without `@NotModified` is assumed to be modifying, i.e., `@Modified` is implicitly present. The reason for this choice is that from the point of view of the analyzer, modification is the default behavior. The analyzer must *prove* non-modification.

Example 1, modification aspects of the `Collection` interface

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

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

    @NotModified
    boolean contains(Object object);

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

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

```
    @NotModified
    boolean isEmpty();

    boolean remove(Object object);

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

    @NotModified
    int size();

    @NotModified
    Stream<E> stream();

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

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

Under supervision of the analyzer, 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 e.g. `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 unbounded generic type, it has the same methods available as `Object`. No code visible to implementations of `Collection` can make modifications to `E` without explicit down-casting;
- `Consumer` because it is a functional interface (an interface with a single abstract method) in `java.util.function`; they are `@IgnoreModifications` by convention, as explained later.

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 More on hidden content. Here, we continue with the first use case of modification: containers.

Chapter 4. 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 arguments.

Whatever else the container does, storing the arguments in fields or not, it will not change the objects you pass to it. You obviously remain free to change them elsewhere; then the container may hold on to the changed object.

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 simple but useful utility type `Pair` trivially satisfies the container requirements:

Example 1, a `Pair` of objects

```
@Container
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;
    }

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

    @NotModified
    public V getV() {
        return v;
    }
}
```

While its fields are clearly final, 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 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 1, 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
    List<ErrorMessage> getErrors();

    // @Modified implicitly
    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`, or in any other method not mentioned here, if the `errorMessage` has been assigned or added to any

of the fields.

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.

Arrays are essentially containers holding the `@FinalFields` property: 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 1, an array is a container with final fields

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

    @NotModified
    T get(int index);

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

@FinalFields @Container
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
    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));
}
```

It would have been better to show an `ErrorMessage` array, because, contrary to `Integer`, the former is mutable (it has a modifying method `setMessage`). The technical aspect of storing and retrieving the reference to the object, which is not normally available, prevents us from doing this here.

To conclude this section, note that the definition of `@Container` carefully words … *modifies its arguments*. Recall that a parameter is modified when the argument's *entire object graph* is modified, at any point during the object's life-cycle — not merely when the method that receives it contains a modifying statement. It is not because a method does not *visibly* modify its argument, that no modification can exist:

Example 1, a modification to an argument, occurring outside the method that received it

```
class ErrorRegistry {
    @Modified
    private final List<ErrorMessage> messages = new ArrayList<>();

    @Modified
    public void add(@Modified ErrorMessage message) { ①
        messages.add(message);
    }

    @Modified
    public void changeFirst() {
        if(!messages.isEmpty()) {
            messages.get(0).setMessage("changed!"); ②
        }
    }
}
```

① Not a single statement of `add` modifies `message`; it only stores it.

② Yet here, an object which arrived via `message` is modified.

The `add` method stores its argument, which links it to the field `messages`. The `changeFirst` method modifies the object graph of that field, so an object that entered the type through `message` can be modified after `add` has returned. The modification therefore travels, exactly as it does in Linking, dependence: from the method `changeFirst`, to the field `messages`, and finally to the parameter `message` of `add`, which is marked `@Modified` as a result. `ErrorRegistry` is not a container, and objects passed to it are not safe from modification.

It is *linking* that allows the analyzer to make this journey, and it is the subject of the next section. Because modification of parameters is computed this way, the definition above is equivalent to the more practical rule that all parameters of all non-private methods and constructors must be `@NotModified` — which is precisely what the analyzer checks.

Chapter 5. Linking, dependence

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

Example 1, 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 *maddi* analyzer will add modification annotations to `LinkExample1` as follows:

Example 1, 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` (because its type is an unbound type parameter). The `LinkExample1.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 1, 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 1, 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 `java.lang.String`.

Let us summarize by:

> **Definition**: Two objects are independent of each other (or not linked to each other) when no modification to the first can imply a modification to the second, or vice versa.
>
> Conversely, two objects are dependent (or linked) when a modification to the first may imply a modification to the second, or vice versa.

Linked objects must 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.

(in)dependence is defined at the level of objects (instances of some class or interface). Two objects of the same class can be independent of each other. For two objects to be dependent on each other, they must have some common type in their respective type definitions.

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 between fields and parameters, and fields and return values of methods, is important to us:

> **Definition**: A method parameter or constructor parameter is not linked when it is not linked to any of the fields of the type. A method is **not linked** when its return value is not linked to

any of the fields of the type.

When a constructor parameter is linked, 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?

5.1. Accessible and hidden content

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

Example 1, Counter, Counters

```
interface Counter {

  // @Modified implicit
  void increment();

  @NotModified
  int getValue();

  @NotModified
  String getName();
}

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

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

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

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

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 1, 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 1, Level, Levels

```
interface Level {

  @NotModified
  int getValue();

  @NotModified
  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();
  }
}
```

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. Note: if it were not for transparent types, which are clearly accessible but are never accessed, we would not define something "accessible" in terms of "accessed". But we can argue that having transparent types in the code is poor programming practice (to the extent that the analyzer can be configured to raise an error when they are present), and "hidden" is the complement of "accessible".

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`, implementations or extensions may be modifiable (such as `Counter`), but when presented with `Level` only, there are no modifications to be made. Inside `Levels`, where we are limited to `Level`, no such extensions are accessible.

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.

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

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 `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 our definition of immutability will be that

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

Recall that by definition, any modifications to the hidden content must be external to the type.

We end this section by defining what linking means with respect to the accessible and hidden content of the fields. The definition of linking given in the previous section is absolute, in the sense that it covers the whole object graph of the objects being linked.

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 linking:

> **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.
> - **independent** of the fields if and only if it is at most linked to the hidden 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, is called independence. We will elaborate in More on hidden content. In the following sections, we will often use the term 'independent' when we mean 'not-dependent', i.e., when there is no linking or only linking to the hidden part of the object graph of the fields.

In terms of annotations, dependence will be the default state for objects of types where dependence is possible. We will not annotate it most of the time; if we do, we use the annotation `@Independent(absent=true)`. The annotation `@Independent` on parameters and methods will be used for absence of linking. When a type is deeply immutable, `@Independent` is the default state, and therefore it will be omitted. We use `@Independent(hc=true)` to stress the linking to the hidden part.

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

Chapter 6. Immutability

6.1. Definition and examples

First, what do we want intuitively? A useful form of immutability, less strong than deeply immutable, but stronger than final fields 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, immutability is much harder to define than final fields. We identify three rules, on top of the obvious final fields requirement. The first one prevents the type from making changes to its own fields:

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

Our friend the `Pair` satisfies this first rule:

Example 2, 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 3: 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;
    }
}

@Container @FinalFields
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 immutable.

From the restriction of rule 1, that all its fields should remain unmodified, it follows that, excluding external changes, every method call on an 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 final fields, we define:

> **Definition**: A type is **immutable** when
>
> **Rule 0**: All its fields are effectively final.
>
> **Rule 1**: All its fields are `@NotModified`.
>
> **Rule 2**: All its fields are either private, or of 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 immutable will be marked `@Immutable` . When they are containers too, which should be the large majority, we use `@ImmutableContainer` as a shorthand for the combination of the two annotations.

Note that:

- We state that all primitive types are immutable, as is `java.lang.Object`. Whilst this is fairly obvious in the case of primitives, 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 an immutable type must be `@NotModified`.
- A field whose type is an unbound type parameter, can locally be considered to be of 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 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 More on 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 immutable base of the recursion. All other types must extend from it.
- The section on Abstract methods will detail how 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.
- When the type has fields which allow hidden content, or the type is extendable (see [extendability]), the extra parameter `hc=true` will be added to the annotation. The presence of this parameter is for instructive purposes only.

Let us go to examples immediately.

Example 1, explaining immutability: with array, version 1, not good

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

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

    @NotModified
    @Independent(hc=true)
    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 `@Independent(hc=true)` annotation on the return value of `stream()`, to indicate that

modifications to the hidden content are possible on objects obtained from the stream.

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

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

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

    @NotModified
    @Independent(hc=true)
    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 1, explaining immutability: with array, version 3, safe

```
@ImmutableContainer(hc=true)
class ArrayContainer3<T> {
    @NotModified
    private final T[] data; ①

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

    @NotModified
    @Independent(hc=true)
    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 1, explaining immutability: with collection, version 1, not good

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

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

    @NotModified
    @Independent(hc=true)
    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 1, explaining immutability: with collection, version 2, not good

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

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

    @NotModified
    @Independent(hc=true)
    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 1, explaining immutability: with collection, version 3, safe

```
@ImmutableContainer(hc=true)
class SetBasedContainer3<T> {
    @NotModified
    private final Set<T> data; ①
```

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

    @NotModified
    @Independent(hc=true)
    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 an immutable type. The next one is immutable as well:

Example 1, explaining immutability: with collection, version 4, safe

```
@ImmutableContainer(hc=true)
class SetBasedContainer4<T> {

    @ImmutableContainer(hc=true)
    public final Set<T> data; ①

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

    @NotModified
    @Independent(hc=true)
    public Stream<T> stream() {
        return data.stream();
    }
}
```

① the data is public, but the `Set` is `@Immutable` 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 `@Immutable` 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 1, explaining immutability: with collection, version 5, not good

```
@FinalFields @Container
class SetBasedContainer5<T> {
```

```
    @NotModified
    private final Set<T> data; ①

    public SetBasedContainer5(@Independent(hc=true) 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 1, explaining immutability: with collection, version 6, not good

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

    @Modified
    public void add(@Independent(hc=true) T t) { set.add(t); }

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

6.2. Inheritance

Deriving from an immutable class is the most normal situation: since `java.lang.Object` is an immutable container, every class will do so. Clearly, the property is not inherited.

Most importantly, in terms of inheritance, is that the analyzer prohibits changing the modification status of methods from non-modifying to modifying in a derived type. This means, for example, that the analyzer 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 1, illegal modification status of methods

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

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

    @Override
    @Modified ②
    public abstract String getFirst();

    @Override
    @NotModified ③
    public abstract boolean add(String s);
}
```

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

② Not allowed! You cannot explicitly (contractually) change `List.getFirst()` from `@NotModified` to `@Modified` in a subtype.

③ This is allowed: going from `@Modified` to `@NotModified` is possible.

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, and the implementation does not modify the objects linked to the parameters of the `Collection` methods.

Note that sealed types reject the 'you can always extend' assumptions of Java types. In this case, all subtypes 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 analyzer.

6.3. 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 analyzer 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 analyzer recognizes 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 `@ImmutableContainer` , even if the type is obviously modifiable.

The following trivial example should clarify:

Example 1, a type used transparently in a class

```
@ImmutableContainer(hc=true)
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 analyzer 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 choice for a type parameter may have an effect on the immutability status, as will be explained in More on hidden content. Some examples are easy to see: any `@FinalFields` type whose fields consist only of types of unbound type parameter, will become immutable when the unbound type parameters are substituted for immutable types. Any immutable type whose hidden content consists only of types of unbound type parameter, will become deeply immutable (i.e., devoid of hidden content) 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.

6.4. Abstract methods

Because `java.lang.Object` is an immutable container, trivial extensions are, too:

Example 1, trivial extensions of `java.lang.Object`

```
@ImmutableContainer ①
interface Marker { }

@ImmutableContainer
class EmptyClass { }

@ImmutableContainer
class ImplementsMarker implements Marker { }

@ImmutableContainer
class ExtendsEmptyClass extends ImplementsMarker { }
```

① Because interfaces are meant to be extended, adding `hc=true` is completely superfluous.

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 `@Modified` .

Furthermore, we will also impose special variants of the rules for 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 not be dependent, i.e., the object they return must not be dependent on the fields. They cannot expose the fields via parameters: parameters of non-primitive, non-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 treat any implementation of `Comparable`, defined as:

Example 1, `java.lang.Comparable` *annotated*

```
@ImmutableContainer
interface Comparable<T> {

    @NotModified
    int compareTo(@NotModified T other);
}
```

as an immutable type when the only method we can access is `compareTo`.

As far as the modification status of the *parameters* of abstract methods is concerned, we also start off with `@Modified` :

Unless otherwise explicitly annotated, or their types are 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 an immutable interface is guaranteed to be 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 More on hidden content.

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

Example 1, the `HasSize` interface

```
@ImmutableContainer // computed
interface HasSize {

    @NotModified // contracted
    int size();

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

We extend to:

Example 1, still immutable: `NonEmptyImmutableList`

```
@ImmutableContainer // computed
interface NonEmptyImmutableList<T> extends HasSize {

    @NotModified // contracted
    @Independent(hc=true) ①
    T first();

    @NotModified // contracted
    void visit(@Independent(hc=true) 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 `@Independent(hc=true)` annotation here will force all concrete implementations to have a non-dependent `first` method. Even 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 More on 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. It must (and does) agree with the `@NotModified` status of `HasSize.isEmpty()`.

The `Consumer` interface is defined and annotated as:

Example 1, the java.util.function.Consumer interface, annotated

```
@FunctionalInterface
interface Consumer<T> {

    // @Modified implicit
    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 `@ImmutableContainer` to `@FinalFields @Container` by adding a modifying method:

Example 4, not immutable anymore: `NonEmptyList`

```
@FinalFields @Container
interface NonEmptyList<T> extends NonEmptyImmutableList<T> {

    // @Modified implicit
    void setFirst(@Independent(hc=true) T t);
}
```

Note that the method `setFirst` promises to keep its parameter unmodified thanks to the `@Container` annotation on the type. The `@Independent(hc=true)` annotation states that arguments to `setFirst` will end up in the hidden content of the `NonEmptyList`. Implementations can even lose `@FinalFields` :

Example 1, 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 `@FinalFields` and `@Container` :

Example 1, final fields implementation of `NonEmptyList`

```
@FinalFields @Container
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, an `@ImmutableContainer` 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 1, 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 1, 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 1, the method `Consumer.accept` *modifying a field*

```
@FinalFields @Container
class ReceiveStrings implements Consumer<String> {

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

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

6.5. 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:

A primary type `T` is a type defined in its own file, i.e., `T.java`. It is not nested inside some other type.

Example 1, modifications on static fields are modifications

```
@FinalFields @Container
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 contracted `@IgnoreModifications` annotation, which may make sense from a semantic point of view:

Example 1, modification on static field, explicitly ignored

```
@ImmutableContainer(hc=true)
```

```
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 1, modification of static field can occur inside constructor

```
@FinalFields @Container
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 1, 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, any of its enclosing types, or any of its parent types, or directly on a type different from the primary type, any of its enclosing types, or parent types, we classify the modification as a *static side effect*.

This is still consistent with the rules of 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 1, 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.

6.6. Ignoring modifications as manual hidden content

The `@IgnoreModifications` annotation and hidden content (More on hidden content) are two faces of a single idea. Recall why the hidden content of a type never threatens its immutability: by parametricity, an `@Immutable(hc=true)` type holds only the *abstract* interface of its hidden

content — the type parameter `T`, or an unbounded generic — and so it *cannot express* an operation that reaches out of the hidden content into its own accessible state. A client is free to mutate the hidden content (`list.get(i).mutate()`); the container stays `@Immutable(hc=true)` . Confinement is guaranteed by erasure.

An `@IgnoreModifications` field holds a *concrete* type, which *can* express such reaching-out operations. The annotation asserts, by hand, the very property erasure would otherwise have derived: *modifications to this field's object are not part of my immutability contract.* That is the defining predicate of hidden content.

> `@IgnoreModifications` is the *manual* form of hidden content. It reclassifies a field from accessible content (known type, modifications attributable) to hidden content (modifications external to the contract) — hiding by declaration what the type system would otherwise hide by erasure. A type all of whose modifiable content is either erased or `@IgnoreModifications` is `@Immutable(hc=true)` , never hc-free.

The two are interchangeable in the immutability lattice; they differ only in the *provenance* of the "not my concern" judgement — derived (always sound) versus asserted — and that difference is a proof obligation, not an arithmetic one. Which brings us to the guard.

6.6.1. The confinement guard

Because erasure is replaced by an assertion, soundness must be *checked* where it was previously *given*. The check is confinement: an `@IgnoreModifications` field is sound hidden content exactly when a modification reached through it stays inside the *ignored stratum* — the transitive content reachable only through that field — and never escapes into accessible or global state. This lands at two granularities, which are one principle:

- *field granularity (separation)*: the ignored field must be `@Independent` of the type's accessible content — no accessible content is aliased *into* it (else mutating the field mutates accessible state), and no state read *out* of it flows into immutability-relevant accessible content (else the "immutable" surface secretly depends on mutable ignored state);
- *method granularity (containment)*: every modifying call *on* the ignored field must confine its effect to the receiver (and other ignored strata) — touching no accessible field, no aliased parameter, and no global state.

The method-granularity check, with a logger and a stream as the running examples:

call	effect	verdict
`printStream.println(x)`, `store.put(k,v)`	writes the object's own buffer / entries	confined — allowed
`logger.setLevel(WARN)`, `logger.addAppender(a)`	reaches external / global configuration	escapes — flagged
`System.setOut(other)`	replaces a global reference	escapes — flagged

"Changing the mechanics of the stream" is exactly the *escapes-the-stratum* case, and it fails the guard even though `println` on the same object passes.

6.6.2. The stratum boundary, by example

The ignored stratum is closed under two operations that look like modifications but are not escapes:

1. *writing into the stratum* — appending to a log, putting into a metadata store;
2. *referencing accessible content from the stratum* — storing a reference to an (immutable) accessible object inside the ignored object.

The canonical case is an analyzer that reads an immutable syntax structure and records its findings in a per-node property overlay, held on each node behind an `@IgnoreModifications analysis()` store. The analyzer writes that overlay constantly — that is its whole purpose — and the overlay holds references back into the structure (an "assigned to field *f*" finding references the field *f*). Neither escapes:

> Writing into the ignored stratum, and referencing accessible content from it, are confined and permitted. The guard fires only on an *escape*: a modification, reached *through* the stratum, of accessible or global state — for instance `node.analysis().someMutableThing().mutate()` where the mutation lands on accessible content.

This is why bringing the analyzer's own sources into scope does not trip the guard: filling the overlay is a client mutating hidden content, always permitted; storing structural references in it is confinement, not escape. The only way the guard could fire is if the analyzer reached through the overlay and mutated a *committed* accessible field — which would be a genuine violation of the structure's stated immutability, exactly the bug the guard exists to surface. Confinement thus degrades gracefully: an unverifiable assumption at the boundary when the mutating code is out of scope, a checked theorem once the region is closed.

6.6.3. Static side effects are the global-escape arm

Static side effects is not made redundant by this equivalence — it *is* one branch of the confinement guard, met earlier. The two annotations partition the space by *ownership*:

- `@IgnoreModifications` / hidden content governs content the type *holds* through one of its own fields;
- `@StaticSideEffects` governs state the type does *not* hold at all — a reach-out modification of another type's static or global state, which cannot be reclassified as *your* hidden content because it is not your field.

The guard's two landing sites carry different consequences, and this is the whole distinction:

the escape lands on…	consequence
the type's own *accessible* content	**caps immutability** — unsound to ignore (the separation check)
global / static state you do not own	does **not** cap immutability (it inspects only your fields) — but is a real outward effect, flagged `@StaticSideEffects`

So `@StaticSideEffects` covers precisely what the manual-hidden-content reading cannot: an effect that does not corrupt *your* immutability — it is not your field — yet is a genuine outward effect the designer should see. And the two compose: `@IgnoreModifications` on the *target* global field is what downgrades an `@StaticSideEffects` from "un-accounted outward effect" to "disclaimed" — which, from the target owner's perspective, is that field becoming hidden content. This is the `System.out` story above, read from the field's owner: without the `@IgnoreModifications` we would add, `println` is `@StaticSideEffects` ; with it, the effect is sanctioned hidden content.

6.6.4. Recognising an invisible escape: the safe-surface contract

For a source method the analyzer *computes* the `@StaticSideEffects` : it sees the assignment to, or the modifying call on, another type's static field. But the sharpest escape is invisible from source. `System.setOut(other)` replaces the process-wide `System.out`; nothing in the JDK source the analyzer can reach says so, and its own body is empty to us. The same holds for `logger.addAppender(a)` reconfiguring a logging framework.

An invisible escape is recorded the only way it can be — as a *contract* on the library's safe surface, a declaration in the annotated API:

Example 1, a static side effect contracted on the safe surface

```
class System$ {                                  // annotated-API stand-in for
java.lang.System
    @IgnoreModifications static final PrintStream out = null;
    @StaticSideEffects static void setOut(PrintStream out) { }   // effect invisible
from source
}
```

A caller of a `@StaticSideEffects` method *inherits* the static side effect, transitively. This is the mechanism that lets the confinement guard tell "reconfiguring the mechanics of the stream" from "just using it": `sink.reconfigure()` on an `@IgnoreModifications` field, where `reconfigure()` calls `System.setOut`, propagates `@StaticSideEffects` up to `reconfigure()`, and the guard flags the call as leaving the ignored stratum — whereas `sink.println(x)`, calling only the confined `println`, passes. As with `@IgnoreModifications` , this annotation is *contracted* on a library surface (never computed onto the library method itself), and on a source method it is *computed*, never trusted from an annotation.

6.7. 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 instance'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 final fields but does not specify any of the restrictions we require for 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. Immutability is not a requirement to be a value-based class. However, we expect many immutable types will become value-classes. Revisiting the example from the previous section, we can construct a counter-example:

Example 1, immutable type which is not value-based

```
@ImmutableContainer(hc=true)
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.

6.8. Dynamic type annotations

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

Example 1, revisiting `SetBasedContainer4`

```
@ImmutableContainer(hc=true)
class SetBasedContainer4<T> {

    @ImmutableContainer(hc=true)
    public final Set<T> data;

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

    @ImmutableContainer(hc=true)
    public Set<T> getSet() {
        return data;
    }
}
```

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

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

Chapter 7. Eventual immutability

Status. The concepts in this chapter — `@Mark`, `@Only`, `@BeforeMark`, `@TestMark`, and the two-phase life cycle they describe — are part of the design, and were fully computed in an earlier generation of the analyzer. The current engine reads `@Mark`, `@Only`, `@TestMark` and the `after="…"` parameters as *contracts*, stores them as properties, and *computes* them by propagation, for classes and interfaces alike; `@BeforeMark` is not read yet, and the precondition-based detection of the two `SimpleImmutableSet` examples above is not implemented (nor needed for the patterns maddi itself uses). The chapter is retained because the design remains valid and the pattern is central to how the analyzer's own support classes are written (see Support classes in the analyzer). The plan for computing them is in `docs/eventual-immutability.md`.

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

7.1. Builders

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

Example 1, static nested builder type

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

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

@ImmutableContainer
class Polygon {

    @ImmutableContainer
    public final List<Point> points;

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

    @FinalFields(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 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 1, 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);
    }
}
```

7.2. Definition

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

Example 1, state change in a boolean field

```
@ImmutableContainer(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 analyzer 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 1, state change going from empty to non-empty

```
@ImmutableContainer(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 analyzer 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 `@FinalFields` , `@Immutable` or the shorthand `@ImmutableContainer` .

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 analyzer's own code:

Example 1, 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.

7.3. 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 immutable fields.

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

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

7.4. 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 1, `@BeforeMark` annotation

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

Similarly, the analyzer can compute a parameter to be `@BeforeMark` , when in the method, at least one before-mark method 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.

7.5. Extensions of annotations

When a type is eventually `@FinalFields` , should the field(s) of the state transition be marked `@Final` ? Similarly, when a type is eventually immutable, should the analyzer mark the initially mutable or assignable fields `@Modified` or `@NotModified`?

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

|== | property | not present | eventually | effectively | finality of field | no annotation, or `@Final(absent=true)` | `@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 `@FinalFields` (not eventually), all fields are effectively final. The analyzer wants to emphasize the rules needed to obtain (eventual) immutability, by clearly indicating which fields break the immutability rules.

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

7.6. 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 analyzer, 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 1, excerpts and annotations of `Verticle.java` *and* `AbstractVerticle.java`

```
@FinalFields(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, contracted eventual immutability has not been implemented yet in the analyzer.

Chapter 8. Modification, part 2

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

8.1. Cyclic references

We need to study the situation of seemingly non-modifying methods with modified 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 1, indirect modifications

```
@ImmutableContainer
public class CyclicReferences {

    // not @FinalFields, not @Container
    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();
        }

    }

    @FinalFields // not @Container
    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 analyzer, 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.

8.2. How to compute linking

To compute linking, the analyzer 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 `@Immutable` type, there cannot be any linking; `v` does not link to `a` nor `b`.
2. If `b` is of `@Immutable` type, `v` cannot link to `b`.
3. When `method` has the annotation `@Independent` (allowing for hidden content, or not), `v` cannot link to `a`.

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

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

1. When `b` is of `@Immutable` type, `v` cannot link to `b`.

2. If `A` is `@Immutable` , then `v` cannot link to `b`, because all its constructor parameters are independent.
3. When `b` has been marked `@Independent` , `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. The analyzer does not currently compute individual linking between parameters, because we advocate the use of containers: all parameters should be `@NotModified`. However, extra parameters to the `@Independent` annotation allow the user to mark them to ensure correct linking computations.

8.3. 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 normal 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 More on hidden content. Consider the following example:

Example 1, concrete implementation of suppliers

```
@FinalFields @Container
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 `@FinalFields` , and the inner classes hold no other data, the inner classes are `@FinalFields` as well.

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

Example 1, 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();

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

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

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

    @ImmutableContainer
    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 `@Immutable` . 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 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.

Chapter 9. More on hidden content

In this section, we consider modifications to the hidden content of a type, and explain when they are of importance.

9.1. Visitors

Let's go back to `NonEmptyImmutableList`, first defined in Abstract methods:

Example 1, revisiting `NonEmptyImmutableList`

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

    // implicitly present: @NotModified
    @Independent(hc=true)
    T first();

    // implicitly present: @NotModified
    void visit(@Independent(hc=true) Consumer<T> consumer); // implicitly present:
@NotModified

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

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

Example 1, immutable implementation of `NonEmptyImmutableList`

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

    public ImmutableOne(@Independent(hc=true) 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);
    }
}
```

According to the interface contract, we need the `visit` method to be non-modifying, and also 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 the data it has been given. 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 statement 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` are semantically unrelated to `ImmutableOne`. As a consequence, we could say that the only modifications that concern us in this `visit` method, are the potential 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` by `Set` and `accept` by `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.

While we will ignore this second source of modification in the `ImmutableOne` type, we will *defer* or *propagate* it 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 `@Independent` 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 1, the `visit` method in `ImmutableOne`, fully annotated

```
@NotModified
public void visit(@IgnoreModifications @Independent(hc=true) Consumer<T> consumer) {
```

```
    consumer.accept(t);
}
```

Note that we assume that we will need `@IgnoreModifications` for almost every use of a functional interface from the package `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 a formal functional interface type of `java.util.function` 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 1, a generic `forEach` implementation

```
@NotModified
public void forEach(@Independent(hc=true) 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 `@Independent` annotation appears because hidden content in `list` is exposed to the `consumer` parameter. This annotation does not appear for the accessible content of the immutable type.

Recall that parameters of modifiable type can already be shielded from external modification by the `@Independent` annotation.

9.2. Propagating modifications

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

Example 1, 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 `@Modified` annotation,

we know of a modification to `list`. 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 1, alternative implementation of `addNewLine`

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

Note that while `NonEmptyImmutableList` is immutable, its concrete instantiation gives access to a modifying method in its hidden content.

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 1, 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 parameter like `@Modified(hc=true)` 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 compute

Example 1, 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 `@Independent` 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 1, modification of a field outside the scope

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

9.3. 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 `@Independent(hc=true)` , *content dependence*:

Example 1, constructor of `ImmutableOne`

```
private final T t;

public ImmutableOne(@Independent(hc=true) T t) {
    this.t = t;
}
```

Returning a part of the hidden content of the type, or exposing it as argument, both warrant a `@Independent(hc=true)` annotation:

Example 1, more methods of `ImmutableOne`

```
@Independent(hc=true)
@Override
```

```
public T first() {
    return t;
}

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

Observe that content dependence implies absence of dependence, as described in Linking, dependence and How to compute linking, exactly because we are dealing with type parameters of an immutable type.

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

Example 1, for-each loop and hidden content linking

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

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

Example 1, `add` in `Collection` annotated

```
@Modified
boolean add(@NotNull @Independent(hc=true) 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.

Let us look at a possible implementation of `Collection.addAll`:

Example 1, a possible implementation of `addAll` in `Collection`

```
@Modified
boolean addAll(@NotNull(content=true) @Independent(hc=true) 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 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 container.

9.4. Iterator, Iterable, loops

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

Example 1, the `Iterator` type, without `remove` method

```
@Container
interface Iterator<T> {

    @Modified
    @Independent(hc=true)
    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 `@Independent(hc=true)` , because it returns part of the hidden content held by the iterator.

The interface `Iterable` is a supplier of iterators:

Example 1, the `Iterable` type

```
@ImmutableContainer
interface Iterable<T> {

    @Independent(hc=true)
    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 subtype, 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 1, 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`.

9.5. Independence of types

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

Example 1, implementation of an `Iterator`

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

    @NotNull(content=true)
    private final T[] elements;

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

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

    @Container
    @Independent(hc=true)
    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 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 immutable itself, 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 immutability.

We propose to add a definition for the independence of a type, identical to the "shielding off" part of the definition of immutability. Let's first go there in a roundabout way:

> **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 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 **independent**, annotated `@Independent` , when external modifications cannot impact the accessible content of the type. The hidden content of the type is mutable or modifiable.

This definition is entirely equivalent to the definition of immutability without rules 0 and 1, and rules 2 and 3 restricted to those fields that are 'exposed' to the outside world via linking or content linking.

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 1, implementation of an `Iterator` as a static nested type

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

    @Container
    @Independent(hc=true)
    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 immutable, so does not allow modifications. The former allows modifications on the hidden content, whence the `@Independent(hc=true)` annotation for `IteratorImpl`.

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

Example 1, 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 by means of example types:

	Mutable, modifiable	**Immutable with hidden content**	**Immutable without hidden content**
Dependent	✔ `Set`	✖	✖

	Mutable, modifiable	Immutable with hidden content	Immutable without hidden content
Independent with hidden content	✔ `Iterator<T>`	✔ `Optional<T>`, `Set.of(T)`	✖
Independent	✔ `Writer`, `Iterator<String>`	✖	✔ `int`, `String`, `Class`

Chapter 10. The link system

The chapters up to now have used *linking* informally: two variables are linked when modifying the content of one may modify the content of the other. This chapter describes, at a technical but hopefully still readable level, how the analyzer actually computes links. It is the first of two chapters that trade the tutorial style for a look under the hood; the concepts of the earlier chapters — modification, independence, accessible and hidden content — are assumed throughout.

10.1. Links, natures, and summaries

A *link* is a triple: `(from, nature, to)`, where `from` and `to` are variables — in a broad sense that includes fields of variables, array elements, and several synthetic kinds introduced below — and the *nature* expresses **how** the two relate. Where the earlier chapters distinguished only "statically assigned", "dependent" and "independent with hidden content", the engine works with a richer alphabet. The natures that matter most, with the notation used in the analyzer's own output:

symbol	name	meaning, by example
⟷	identity	two spellings of the same variable
← →	assigned from / to	`X x = box.t`: `x ← box.t`
∈ ∋	is element of / contains	`list.get(0)`: `get ∈ list.§xs`
⊆ ⊇	subset / superset	`new ArrayList<>(l)`: `copy.§xs ⊆ l.§xs`
~	shares elements	`dst.addAll(src)`: `dst.§xs ~ src.§xs`
∈ ∋ ⋈	field of / contains field / shares fields	field-level dependence
∈ ∋ ⋈	object-graph containment / overlap	the weakest, "somewhere in the object graph" relations

Reciprocals normally appear in pairs: `x ∈ list.§xs` alongside `list.§xs ∋ x`. A `→` after a variable means it is (or may be) modified at that point: `box.t→`.

The unit of exchange between methods is the *method link summary* (`MethodLinkedVariables`): how a method's return value, parameters and receiver relate, plus which of them it modifies. For example, the summary of a setter `void set(X x)` on a one-field holder reads `[0:box*.t←1:x*] -> -`: parameter `x` is assigned into the receiver's field, both are marked modified, and there are no return-value links. Summaries are computed once per method and then *applied* at every call site, translating the callee's formal `this` and parameters into the actual receiver and arguments.

10.2. Two field models: concrete fields and virtual fields

A link names a *place inside* a variable, and there are two ways to name such a place:

- For types whose source the analyzer has parsed, the place is a real field: `box.t`, `pair.a`, `bb.t.t`.

- For shallowly analyzed types — most of the JDK — no field is visible. Their contents are represented by *virtual fields*, written with a `§` prefix: `list.§xs` for a list's elements, `map.§kvs` for a map's entries. A virtual field is exactly the *hidden content* of the chapter on More on hidden content, given a name so that links can point at it.

The distinction drives which natures appear: a one-field holder tends to produce assignment links (←, →), a collection produces element links (□, □, ~, □). Otherwise the shapes are identical, and this symmetry is heavily used in the test suite, which pins the same relationship over a source type (`Box<T>`) and a JDK type (`List<X>`) side by side.

One virtual field deserves special mention: `§m`, the *modification component*. It stands for "the mutable aspect of this object, wherever it lives", and lets the engine express that two variables share their modification fate without committing to a structural claim.

10.3. From expressions to a method summary

The linking pipeline runs per method, statement by statement, expression by expression:

1. preparatory analysis (`maddi-modification-prepwork`) computes, per method, the variables, their assignments and reads;
2. the link computer walks the method body; every expression produces a small result: its primary links, links about other variables it touched, and the variables it modified;
3. at every method call, the callee's summary is translated onto the call site — the central operation of the whole system. Formal parameters become arguments, the formal `this` becomes the receiver (including through inheritance: a supertype's `this` maps to the same receiver), and the callee's modification marks transfer to the actual variables;
4. the accumulated links feed a per-method *link graph* on which a fixpoint closure runs (next section);
5. the method's own summary is extracted from the closure, and becomes available to *its* callers.

Functional interfaces get their own machinery, because a lambda's effect happens at a distance: the place that *captures* `x → box.set(x)` is not the place that *applies* it. The capture is wrapped into a synthetic functional-interface variable carrying the lambda's links and modifications; the application site — say `list.forEach(consumer)` — finds that variable and *lifts* the lambda's links onto the concrete receiver: the summary of `forEach` says "the receiver's elements flow into the consumer", and the lifting engine substitutes the lambda's parameter by those elements. When the lambda is not yet known (a method that merely receives a `Consumer` and applies it), the application is recorded symbolically and resolved later, at the caller that supplies the concrete lambda.

10.4. The closure: facts, witnesses, and an incremental fixpoint

Links compose: from `a ← b` and `b □ c.§xs` follows `a □ c.§xs`. Composition is defined by a binary operator on natures (the full table lives in the module's README); composing natures generally *weakens* them — a subset of an overlap is at best an overlap — and the weakest results (□, □) eventually compose to nothing, which keeps the closure finite.

The engine that computes this closure (`IncrementalFixpointEngine`) treats each link as a *fact* and works incrementally: statements add facts, each new fact is propagated forward and backward against the existing closure, and every derived fact carries a *witness* — the pair of facts it was composed from. Witnesses earn their keep when facts are *removed*: when a modification invalidates a link (the `□→~` rewrite below), all facts whose witness chain rests on the removed fact must be recomputed, and only those. This is what makes the engine incremental rather than recompute-the-world.

Two practical notes on this machinery:

- Modification is destructive: calling `list.remove(..)` after `copy.§xs ⊆ list.§xs` was established demotes the subset claim to `~` (shares elements) — the copy still shares content, but the neat containment no longer holds. The rewrite fires once, at the modification statement, and only for links the modified variable itself owns.
- Cycle protection: pathological code shapes (deeply mutually recursive accessors, decompiler-generated monsters) can make the closure explode. The graph walk carries a hard ceiling; hitting it abandons the *source-level* computation for that method and degrades to its shallow summary, keeping the element alive at reduced precision rather than crashing or grinding.

10.5. The shared-variable collapse

The system as described so far has a scaling problem, and solving it is the most substantial recent change to the link engine. Consider a builder-style method that threads one object through many locals: every local is assigned from the previous one, every field of the object is reachable from every local, and the naive link set grows as (locals × fields × natures) — the *part-of link explosion*. On real code bases this reached hundreds of thousands of links for a single method.

The collapse introduces *shared variables*: an equivalence grouping of variables that provably denote the same runtime slot, in two tiers:

- *modification identity*: variables linked by `□` on their `§m` component — they share their modification fate exactly;
- *assignment groups*: chains of pure whole-object assignments (`a = b; c = a;`) collapse into one group with a single representative.

The graph then stores edges against group *representatives* only. What used to be quadratic bookkeeping becomes one edge per relationship, and the closure runs on the collapsed graph.

The price is *reconstruction*: the method summary must be expressed in terms of the method's real parameters, fields and return value — not internal representatives. At extraction time, representative-level facts are projected back onto the group members that matter, guided by one master rule: *direction*. Assignment direction within a group is preserved (the source of an assignment passes knowledge to the recipient, never the reverse), field-level facts are re-homed onto each endpoint of an intra-group link ("setI ← this` mirrors to `setI.i ← this.i□), and a redundancy pass removes links that are transitive consequences of nearer hops. The reconstruction techniques have their own catalogue in the module (`sv-reconstruction-techniques.md`); the summary-level behavior is pinned by the link test suite, which is the de-facto specification.

10.6. Reading the output

For the curious, the fastest way to build intuition is to read summaries off real methods. A few shapes, all current output:

```
X read(Box<X> box)        { return box.get(); }   [-] --> read←0:box.t
X get(List<X> list)       { return list.get(0); } [-] --> get□0:list.§xs
void add(List<X> l, X x)  { l.add(x); }           [0:l*.§xs□1:x, 1:x□0:l*.§xs] --> -
MOD[0:l]
List<X> copy(List<X> l)   { return new ArrayList<>(l); }  [-] --> copy.§xs□0:l.§xs
int size(List<X> list)    { return list.size(); } [-] --> -   MOD[]
```

`0:`/`1:` are parameter indices; `MOD[..]` lists the modified variables. The last line is the pure case: nothing flows, nothing changes — and that, ultimately, is what the whole system exists to prove.

Chapter 11. Convergence: the iterating analyzer

The previous chapter described how links are computed for one method, given the summaries of the methods it calls. This chapter describes the layer above: how the analyzer orders the work, iterates it to a fixpoint, knows when it is done, and what it does when the code refuses to cooperate. Like the previous chapter, it is technical in nature; unlike a design document, everything here describes the engine as it runs today.

11.1. Properties, and the discipline of writing them

Every conclusion the analyzer draws — a method is non-modifying, a field's content is unmodified, a type is immutable at some level — is a *property value* attached to a method, field or type. Three rules govern them:

- Values are written at most once per analysis, and always in the *conservative-to-optimistic* direction: an absent value means "not decided yet", never "assumed fine". A guarded overwrite facility (`TolerantWrite`) exists for the few places that legitimately refine a value (cycle breaking, below), and it enforces monotonicity: a verdict may become more precise, never contradict itself.
- Undecided is a first-class state. An analyzer component that cannot conclude — because a field's type is not yet decided, a supertype is still open, a callee's summary is missing — returns "no value" and will simply be visited again.
- Decisions are local, dependencies are explicit. The immutability of a type depends on its fields' finality and modification status, its methods' modification status, and its supertypes' immutability; each of those is itself a property, computed by its own component.

11.2. Ordering the work

Before iteration starts, the analyzer computes an *analysis order* from the call graph and the type hierarchy: methods before their callers where possible, subtypes and enclosing types grouped, fields between their initializers and their readers. Two prepared ingredients matter beyond ordering:

- *part of construction*: the set of methods only reachable from constructors. Assignments there do not count against a field's effective finality — the essence of the effectively-final rule of the chapter on Final fields. Note that assignments to a nested type's fields may come from the enclosing type, a sibling type, or a lambda anywhere in the primary type; the computation scans the whole primary type.
- *shallow summaries* for everything outside the parsed source: JDK and library types receive their properties from the annotated APIs (below) or conservative defaults.

Java being Java, the call graph has cycles — recursion, mutual recursion, callbacks — so a single pass in analysis order cannot suffice. Hence: iteration.

11.3. The iteration loop

The main loop (`IteratingAnalyzerImpl`) repeatedly runs the per-element analyzers over the analysis order, counting *property changes* — the number of newly decided or refined values in the pass. The first pass visits everything; subsequent passes narrow to a *worklist*: only elements whose inputs changed in the previous pass are dirty and revisited, with dirty targets mapped to their enclosing analysis-order element. On large code bases the worklist typically collapses from tens of thousands of elements to a few hundred within two or three passes.

The loop may run in parallel: independent elements of the analysis order are processed by a small thread pool (bounded by the machine's cores). Parallelism is behaviorally almost — not perfectly — transparent: the vast majority of verdicts are scheduling-independent, and the few known non-confluent shapes (a constructor whose modification verdict depends on whether a sibling was decided in the same pass) are tracked explicitly and resolve to the conservative side.

11.4. Verification and certification

Reaching "no more changes" is necessary but not sufficient: a fixpoint reached through optimistic intermediate states could in principle rest on values that would not be re-derived from scratch. The loop therefore ends with *verification passes*: with all values frozen, every element is recomputed once more and the engine checks that nothing changes. A run that exits this way is *certified*: the published verdicts are a self-consistent fixpoint of the analysis functions, independent of the path that produced them. The engine's proving ground — a set of open-source code bases from a few thousand to fifty thousand elements — is kept certified and crash-free on the default configuration, and every engine change is gated on reproducing byte-identical verdict dumps (an "A/B with zero diff") on at least one of them.

11.5. Cycle breaking

Some clusters never decide on their own. The classic shape: type A's immutability waits on field type B, whose immutability waits on a method whose modification verdict waits on A. Everyone is undecided, nobody moves, and the loop reaches its certification point with a residue of undecided types.

At exactly that point — quiescence, with undecided immutability remaining — the engine activates *cycle breaking*: one more full pass in which the blocked components are allowed to resolve their undecided inputs by strategy rather than waiting. The default strategy floors an undecided supertype at final-fields level and treats absent information as non-modifying; the breaking pass then lets the normal machinery draw its conclusions, and the loop re-certifies including them. On the decompiler code base that motivated the work, immutability went from 380-out-of-450 types undecided to a single undecided type, with the positive verdicts (immutable, immutable-with-hidden-content) appearing for the first time.

Cycle breaking is deliberately conservative about the *strength* of what it concludes: a type whose deep immutability rests on a broken cycle is published at the hidden-content level rather than the full level, because the unbroken evidence is not there.

The supertype path was, for a while, the only one the breaking pass resolved, and the elasticsearch

first contact exposed what that left behind: 51% of its 45,000 types remained undecided. The mechanism was not a cycle at all but a verdict that *will never arrive*: an external, unannotated type has no immutability value, any type holding a non-private field of such a type cannot conclude, and — since "has a field of an undecided type" closes transitively — the undecidedness cascades through the field graph. Test code, which touches unannotated framework types most, made up the bulk of the cluster. The breaking pass now treats these the same way it treats blocked supertypes: a missing field or abstract-method verdict is resolved pessimistically, and the type concludes at final-fields level — field finality being the one thing the analysis has, at that point, already established on its own. The published result is deliberately weak, honestly arrived at, and — unlike an absent verdict — usable by every downstream consumer.

11.6. Annotated APIs as analysis hints

The analyzer ships curated property sets for the JDK — `String` is deeply immutable, `Object` is immutable with hidden content, the collection interfaces are mutable, the functional interfaces carry their link summaries — under the name *analysis hints*. They are loaded after parsing, resolved against the compiled-types universe of the run, and pre-seed the property store so that source-code analysis starts from knowledge rather than pessimism. The difference is not cosmetic: without the hints, every path through a JDK type bottoms out in "undecided", and the immutability side of the analysis barely concludes anything positive; with them, the proving-ground code bases settle into full immutability distributions.

11.7. When code refuses to cooperate

A static analyzer that dies on the first pathological method is useless on real code. The engine's fault-tolerance policy, refined over the corpus work, has three tiers:

- *statement- and method-level containment*: an exception while linking one statement abandons that method's source-level analysis and degrades it to its shallow summary — the element stays alive at reduced precision;
- *cycle protection with a tight ceiling*: run-away closures are cut off early and degrade the same way (an early experiment with a generous ceiling let one module's analysis grind from seconds to half an hour before the tight-ceiling-plus-fallback design was adopted);
- *guards at the boundaries*: unrepresentable link shapes are skipped at the point of production, and every skip is a logged, countable event rather than a silent loss.

Finally, most non-default behaviors — disabling the worklist, disabling parallelism, disabling cycle breaking, dumping the per-element verdict fingerprint — are reachable through environment gates, which is how the A/B verification workflow and the regression hunts of the development process are driven.

11.8. A note on scale

For calibration, on current hardware the analyzer processes, certified, code bases in the range of 50,000 elements (methods, fields, types) in a few minutes, with linking dominating the cost. The performance work follows a strict rule worth stating in a document about correctness: an

optimization is accepted only when the verdict dump is byte-identical before and after. Speed is never allowed to buy verdict changes.

Chapter 12. Support classes

The `maddi-support` library (in whichever version it comes) contains a small selection of support types: the eventually immutable building blocks that you can use in your project, irrespective of whether you want analyzer support or not. The annotations themselves live in a separate library, `maddi-annotation`, which is the single dependency `maddi-support` declares. That split is also a licence boundary: the annotations are Apache-2.0, so that annotating your code commits you to nothing, while the analyzer and its support types are LGPL-3.0.

We discuss a selection of the building blocks here.

12.1. FlipSwitch

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

Example 1, most of `io.codelaser.maddi.support.FlipSwitch`

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

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

    @Mark("t")
    @Modified
    public void set() {
        if (t) throw new IllegalStateException("Already set");
        t = true;
    }

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

    @Mark("t") ①
    @Modified
    public void copy(FlipSwitch other) {
        if (other.isSet()) set();
    }
}
```

① 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. Once it has been done, it can never be 'undone' again.

12.2. SetOnce

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

Example 1, parts of `io.codelaser.maddi.support.SetOnce`

```
@ImmutableContainer(hc=true, after="t")
public class SetOnce<T> {

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

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

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

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

12.3. EventuallyFinal

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

Example 1, `io.codelaser.maddi.support.EventuallyFinal`

```
@ImmutableContainer(hc=true, after="isFinal")
public class EventuallyFinal<T> {
    private T value;
    private boolean isFinal;

    @Independent(hc=true)
    public T get() {
        return value;
    }

    @Mark("isFinal")
    public void setFinal(@Independent(hc=true) 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(@Independent(hc=true) 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 `isFinal` mark test.

12.3.1. EventuallyFinalOnDemand

`EventuallyFinalOnDemand` is `EventuallyFinal` with a loader: the *before* state may carry a `Runnable` that produces the final value on first access. The state transition is the same one, `isFinal`, with the same four state methods carrying the same annotations, so nothing in the previous section changes. Three things are new, and each of them is the only example of its kind in this text.

Example 1, parts of `io.codelaser.maddi.support.EventuallyFinalOnDemand`

```
@ImmutableContainer(hc=true, after="isFinal")
public class EventuallyFinalOnDemand<T> {
    private volatile T value;
    private volatile boolean isFinal;
    private volatile Runnable onDemand;

    @NotModified(after="isFinal") ①
    public T get() {
        if (isFinal) return value;
        Runnable runnable = onDemand;
        if (runnable != null) {
            synchronized (this) {
                Runnable again = onDemand; ②
                if (again != null) {
                    again.run(); ③
                }
            }
        }
        return value;
    }

    @Mark("isFinal")
    public synchronized void setFinal(T value) {
        if (this.isFinal) {
            throw new IllegalStateException("Trying to overwrite final value");
        }
        this.value = value;
        this.isFinal = true;
        this.onDemand = null; ④
    }

    @Only(before="isFinal")
    public synchronized void setOnDemand(Runnable onDemand) {
        assert !isFinal && this.onDemand == null;
        this.onDemand = onDemand;
    }
}
```

① `@NotModified` with an `after` on a **method**, rather than on a field as in the table of Eventual

immutability. Before the transition `get()` may run the loader, which writes; after it, it can only read. This is a property of its own, and it is what a lazy-loading getter that effects the transition earns.

② The loader may have been run, and `onDemand` cleared, by another thread while this one waited for the monitor.

③ The monitor is reentrant, which the loader relies on: it calls `get()` from inside `run()`, on this thread.

④ The blanking of Lazy, in a type that the analyzer itself uses. Only `setFinal` clears `onDemand`, and it sets `isFinal` in the same breath, so no value is ever returned while a loader is still pending.

Note what `get()` does **not** carry: a `@Mark` . The transition happens during the call, but it is effected by the injected `Runnable`, which calls `setFinal` — so the mark is not provable from this type's own body. Compare `Lazy.get()` in Lazy, which marks itself.

`EventuallyFinal` is what a `MethodInfo`, `ParameterInfo` or `FieldInfo` uses to hold its inspection; `TypeInfo` uses this variant, because a type can also arrive by lazy bytecode load, long after the source that mentions it was parsed.

12.4. Freezable

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

Example 1, `io.codelaser.maddi.support.Freezable`

```
@ImmutableContainer(after="frozen") ①
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;
    }

    public void ensureNotFrozen() {
        if (frozen) throw new IllegalStateException("Already frozen!");
    }

    public void ensureFrozen() {
```

```
        if (!frozen) throw new IllegalStateException("Not yet frozen!");
    }
}
```

① Because the type is abstract, `hc=true` is implied.

Note that as discussed in Inheritance, it is important for `Freezable`, as an abstract class, to be immutable: derived classes can never be immutable when their parents are not immutable.

12.5. SetOnceMap

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

Example 1, part of `io.codelaser.maddi.support.SetOnceMap`

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

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

    @Only(before="frozen")
    public void put(@Independent(hc=true) @NotNull K k,
                    @Independent(hc=true) @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);
    }

    @Independent(hc=true)
    @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 analyzer will warn for a potential null pointer exception here, not (yet) making the

connection between `isSet` and `containsKey`.

② Implicitly, the parameter `K k` is `@Independent` , because the method is `@NotModified`.

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

12.6. Lazy

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

Example 1, `io.codelaser.maddi.support.Lazy`

```
@ImmutableContainer(hc=true, after="t")
public class Lazy<T> {

    @NotNull(content=true)
    @Independent(hc=true, after="t")
    @Final(after="t")
    private volatile Supplier<T> supplier;

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

    public Lazy(@NotNull(content=true) @Independent(hc=true) Supplier<T>
supplierParam) { ①
        if (supplierParam == null) throw new NullPointerException("Null not allowed");
        this.supplier = supplierParam;
    }

    @Independent(hc=true)
    @NotNull
    @Modified
    @Mark("t") ②
    public T get() {
        T value = t;
        if (value != null) return value;
        Supplier<T> localSupplier = supplier; ③
        if (localSupplier == null) return t;
        t = Objects.requireNonNull(localSupplier.get()); ④
        supplier = null; ⑤
        return t;
    }

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

```
}
```

① The annotation has traveled from the field to the parameter; therefore the parameter has `@Independent(hc=true)`.

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

③ `supplier` is read once, into a local. Reading the field twice is a race rather than a nicety: two threads find `t` null, the first completes and empties `supplier`, and the second dereferences what it re-reads as null.

④ Here `t`, part of the hidden content, links to `supplier`, as explained in Content linking. The statement also causes the `@NotNull(content=true)` annotation, as defined in Nullable, not null and Identity and fluent methods. It publishes the value *before* the next line empties the supplier, so a reader that sees `supplier == null` has already seen `t`.

⑤ After the transition from mutable to effectively immutable, the field `supplier` moves out of the picture. This is what Kotlin's three `Lazy` implementations do as well, each with its own `initializer = null`.

After calling the marker method `get()`, `t` cannot be assigned anymore, and it becomes `@Final` . The constructor parameter `supplier` is `@Independent(hc=true)`, 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 `@Independent`.

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 the rules of effective immutability by slightly augmenting rule 2:

> **Rule 2**: All fields are either private, of 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 eventual "blanking out" of modifiable fields in immutable types.

Not implemented. The two `@Independent(hc=true)` annotations that this argument arrives at — on the field `supplier`, and on the constructor parameter it travels from — are not computed today: the analyzer concludes `@Dependent` at both positions. It does so whether or not `supplier` is emptied, so the augmented rule 2 is not what is stopping it. `get()` is annotated as printed. The measurement is in `TestBookIndependenceOfSupportTypes`, which runs this listing beside the shipped code and prints both verdicts.

12.7. FirstThen

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

Example 1, `io.codelaser.maddi.support.FirstThen`

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

    public FirstThen(@NotNull @Independent(hc=true) 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("first")
    public void set(@Independent(hc=true) @NotNull T then) {
        Objects.requireNonNull(then);
        synchronized (this) {
            if (first == null) throw new IllegalStateException("Already set");
            this.then = then;
            first = null;
        }
    }

    @Only(before="first")
    @Independent(hc=true)
    @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="first")
```

```
    @Independent(hc=true)
    @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 analyzer 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 analyzer to detect them, we could have a slightly more efficient implementation.

Chapter 13. Support classes in the analyzer

Practice what you preach, and all that. The *maddi* analyzer itself is written in the style this document advocates, and its central data structure is a live example of the two-phase life cycle of Eventual immutability.

The common syntax tree (CST) — the language-agnostic representation shared by the Java and Kotlin front-ends — is built through the builder pattern throughout: every `TypeInfo`, `MethodInfo` and `FieldInfo` starts life as a mutable builder during inspection, and is *committed* into an immutable form before analysis begins. After the commit, the object's structural content — its methods, fields, hierarchy, statements — never changes again, and the analyzer's many threads read it freely without synchronization.

The analysis results themselves live in a property map attached to each info object (`element.analysis()`), and follow the discipline described in Convergence: the iterating analyzer: a property value is written once, absence means "undecided", and the rare legitimate refinements go through a guarded, monotonicity-checking write facility. This is eventual immutability at the granularity of a single property: mutable while the fixpoint iteration runs, frozen from certification onward. Software engineers using the analyzer as a library can rely on the same guarantee as before: once the analysis phase is over, all inspected and analyzed information remains stable.

One deliberate exception proves the rule: info objects are *single-instance* — one `TypeInfo` per fully qualified name and source set — so identity comparison (`==`) is the correct equality throughout the engine, and the immutable-after-commit design is what makes handing these objects across threads safe in the first place.

Chapter 14. Other annotations

The *maddi* 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.

Implementation status varies across this chapter. `@Identity` and `@Fluent` are **computed** by the analyzer, and are described first for that reason.

Everything after them is **not implemented** at present, with one partial exception:

- `@NotNull` / `@Nullable` and `@UtilityClass` are read as **contracts** — from explicit annotations and from annotated APIs — but are never inferred from your code;
- `@Finalizer` is read as a contract, and its **modification semantics** — a finalizer call does not count as a modification of the receiver — are implemented and load-bearing in the modification engine (see Finalizers); the three life-cycle **sequencing rules** below are not enforced;
- `@ExtensionClass` and `@Singleton` exist in `maddi-annotation`, but nothing in the analyzer reads them.

Those sections describe the intended design, and are retained because the definitions still hold. Do not read them as a description of current behaviour.

14.1. Identity and fluent methods

The analyzer marks methods which return 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 1, methods marked `@Identity` and `@Fluent`

```
@FinalFields(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
    @ImmutableContainer
    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();
}
```

14.2. Nullable, not null

Not implemented. Not-null values that arrive from explicit annotations or from annotated APIs are stored and propagated, but the computational inference described in this section — context not-null, field not-null, external not-null — is not part of the analyzer.

Nullability is a standard static code analyzer topic, which we approach from a computational side: the analyzer 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: do parameters occur in a not-null context?
2. field not-null: has the field been assigned a value (or values) that are possibly null? does the field occur in a not-null context?
3. external not-null of parameters linked to fields: once the first two have been computed, warnings can be given when a parameter, assigned to a nullable field, occurs in a not-null context

14.2.1. Higher order not-null

We use the annotation `@NotNull(content=true)` 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 1, `@NotNull` annotations on `Collection`

```
boolean add(@NotNull E e);
boolean addAll(@NotNull(content=true) Collection<? extends E> collection);
@NotNull(content=true) static <E> List<E> copyOf(@NotNull(content=true) Collection<?
extends E> collection);
@NotNull(content=true) 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 1, `@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);
@NotNull(content2=true) 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 `@NotNull(content=true)` , however.

14.3. Finalizers

Partially implemented. `@Finalizer` is a contract (parsed from source annotations and from the annotated APIs, and shown in output), and its modification semantics are implemented: the analyzer exempts finalizer callees from receiver modification, at every layer that judges call sites (`MethodModification`, the shallow method summaries, and the modification-reachability shadow pass). The three life-cycle **sequencing rules** below are not enforced.

14.3.1. Why the modification exemption matters

A finalizer marks the end of an object's life-cycle: after the call, the object may not be used again. Whatever the finalizer does to the dying object is therefore invisible to the rest of the program — there is no "after" in which the change could be observed. The analyzer exploits this: **a call to a `@Finalizer` method never marks its receiver as modified**, whatever the method's own

`@NotModified`/ `@Modified` status.

This is not a corner case; it is what keeps stream-based code analyzable at all. The annotated APIs mark the stream operations — `filter`, `map`, `flatMap`, `collect`, and their relatives — as `@Finalizer` : each consumes the stream it is called on (using the receiver again afterwards is illegal), and none of them carries a `@NotModified` contract. Without the exemption, every one of those calls would conservatively count as a modification of the receiver chain, and since a chain such as `this.elements.stream().map(…)` roots in a field, the false modification would climb to the field and to `this` — making essentially every method containing a stream pipeline `@Modified` , and sinking the immutability of its type. With the exemption, the pipeline is silent about its receivers, and only genuine evidence (the lambda's captures, the collector's target) determines the verdict.

The same reasoning must be applied by **every** component that re-derives modification: when the modification-reachability pass (the post-convergence single writer) judges a call to a method outside the analysis scope, it mirrors the engine's guard exactly — a boundary callee that is a finalizer contributes no receiver-modification evidence, no matter that its conservative no-information default would otherwise read as "modifying". A mismatch here is not benign: it silently poisons whole call chains with false modification, in code as central as the printing of a parameterized type.

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? Secondly, consider "terminal" operations such as `findAny` or `collect` on a stream. They close the stream, after which you are not allowed to use it anymore.

How can the analyzer 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, dependence).

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 1, 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 1, calling the finalizer method

```
@ImmutableContainer
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 1, illegal call

```
@ImmutableContainer
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.

14.3.2. 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 1, 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 symbolizes 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 symbolizes 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 analyzer guarantees that the temporary data is destroyed by severely limiting the scope of the finisher object.

Example 1, 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
    @ImmutableContainer ④
    public EventuallyFinal<String> done(String last) {
        eventuallyFinal.setFinal(last + "; tried " + count);
        return eventuallyFinal;
    }
}
```

① symbolizes the temporary data to be destroyed.

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

③ symbolizes the modifications that act as finishing

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

14.4. Utility classes

Not implemented. `@UtilityClass` is read as a contract, but the analyzer never computes it: the three implications below are not checked.

We use the simple and common definition:

> **Definition**: a **utility class** is an 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 of immutable type.

14.5. Extension classes

Not implemented. `@ExtensionClass` is defined in `maddi-annotation`, but no part of the analyzer reads it, and the criteria below are not applied.

In Java, many classes cannot be extended easily. 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 1, 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 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. 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.

14.6. Singleton classes

Not implemented. `@Singleton` is defined in `maddi-annotation`, but no part of the analyzer reads it, and neither of the two mechanisms below is currently recognized.

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.

Two systems for limiting the number of instances are intended to be recognized: 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 1, first mechanism 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 1, second mechanism 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;
    }
}
```

Appendix A: Annotation overview

When annotating abstract types and methods, or types in the Annotated APIs, observe the following rules.

A method is `@Modified` unless otherwise specified, its parameters are assumed to be `@Modified`, unless they are of immutable type. Modification is the default; the analyzer must *prove* non-modification (see Modification).

Due to the large amount of circular type dependencies in the JDK, combined with the current limitations of the analyzer, the implementation of the Annotated API analyzer requires contracted annotations about independence, immutability, and container on the type. The following combinations are possible with respect to independence and immutability:

- no independence information: (in)dependence according to the immutability value (not immutable → dependent, `@Immutable(hc=true)` → `@Independent(hc=true)`, `@Immutable` → `@Independent`)
- `@Independent(absent=true)`: dependent; this requires that the type is not immutable
- `@Independent(hc=true)`: this is the default for `@Immutable(hc=true)`, `@ImmutableContainer(hc=true)`, so explicitly writing this annotation is only necessary on non-immutable types
- `@Independent`: this is the default for `@Immutable`, `@ImmutableContainer`, so explicitly writing this annotation is only useful for mutable or `@Immutable(hc=true)` types

Recall that abstract types always have hidden content, so the `hc=true` is always implicitly present on `@Independent` and `@Immutable`, `@ImmutableContainer` on the type. We generally do not write `hc=false`, as that is the default value in the annotation. The analyzer will complain when `hc=false` is present on the `@Immutable` annotation of an abstract type.

To support the user, a warning will be raised when the independence value on the type is incompatible with that on its methods, parameters and fields.

In a `@Container` type, all `@Fluent` methods and all void methods are assumed to be `@Modified`. Change this by explicitly marking the method `@NotModified` or `@StaticSideEffects`.

Parameters of a non-modifying method are `@Independent` by default, regardless of an independence annotation on the type. This can be overwritten by `@Independent(absent=true)` or `@Independent(hc=true)` when the method exposes parts of the fields' object graph via its parameters. Parameters of a modifying method are assumed to be dependent when the type is not immutable, and independent when the type is immutable, the hidden content carrying over from immutable to independent.

Dependence is only explicitly written as `@Independent(absent=true)` on a type after analysis, when this type has (eventually) final fields and no modifying methods, as a marker to the user to indicate that it is the independence property that is missing to reach immutability. Marking a non-immutable type with `@Independent` specifies that no parameter or return value can be dependent. The hidden content parameter given by the user is ignored: you will still have to mark any method or parameter which communicates hidden content with `@Independent(hc=true)`.

When a type is immutable, `@Independent` becomes the default independence annotation for methods and parameters. You must still use `@Independent(hc=true)` to indicate communication of hidden content.

Methods marked `@Fluent` are always `@Independent` , because returning the type itself does not expose any additional information.

Following its definition, `@UtilityClass` on a type implies `@Immutable` .

In application mode, the analyzer will regard every class that is never extended as effectively final. This property is only relevant when the type is immutable; the absence of `hc=true` is a marker.

Parameters of "official" functional interface type (i.e., the type is a functional interface type in the package `java.util.function`) have the `@IgnoreModifications` annotation, unless explicitly overwritten by `@Modified` .

The default nullable annotation for parameters and return values of non-primitive type is `@Nullable` .

A factory method is a static method returning an object of the type of the class. Independence of a factory method is always with respect to the method's parameters, rather than to the type. Independence of a factory method's parameters corresponds to the immutability of the parameter type. These two rules also apply to any static method in an immutable type. Note that utility classes are classes that are deeply immutable and cannot be instantiated, so it applies to their static methods.

In general, annotations are inherited on types, methods and parameters. The properties can deviate,

- from `@Modified` to `@NotModified` is possible, from `@NotModified` to `@Modified` is not
- independence can go from left to right in `@Independent(absent=true)` → `@Independent(hc=true)` → `@Independent` , but not from right to left
- a type deriving from an immutable type does not need to be immutable; however, a type deriving from a non-immutable type can never be immutable

When a method has a single statement, returning a constant value, the `@ImmutableContainer("value")` is implicit. Similarly, when a field is explicitly final (it has the `final` modifier) and it has an initializer, then both `@Final` and, if relevant, `@ImmutableContainer("value")`, is implicit.