# Classless Java: Tuning Java Interfaces

**John Q. Open[1] and Joan R. Access[2]**

1 **Dummy University Computing Laboratory**
**Address, Country**
`open@dummyuni.org`

2 **Department of Informatics, Dummy College**
**Address, Country**
`access@dummycollege.org`

## Abstract

Java 8 introduced *default methods*, allowing interfaces to have method implementations. When combined with (multiple) interface inheritance, default methods provide a basic form of multiple inheritance. However, using this combination to simulate multiple inheritance quickly becomes cumbersome, and appears to be quite restricted.

This paper shows that, with a simple language feature, default methods and interface inheritance are in fact very expressive. Our proposed language feature, called *object interfaces*, enables powerful object-oriented idioms, using multiple inheritance, to be expressed conveniently in Java. Object interfaces refine conventional Java interfaces in three different ways. Firstly, object interfaces have their own object instantiation mechanism, providing an alternative to class constructors. Secondly, object interfaces support *abstract state operations*, providing a way to use multiple inheritance with state in Java. Finally, object interfaces allow type refinements that are often tricky to model in conventional class-based approaches. Interestingly, object interfaces do not require changes to the runtime, and they also do not introduce any new syntax (apart from a small annotation): all three features are achieved by reinterpreting existing Java syntax, and are translated into regular Java code without loss of type-safety. Since no new syntax is introduced, it would be incorrect to call object interfaces a language extension or syntactic sugar. So we use the term *language tuning* to characterize this kind of language feature. An implementation of object interfaces using Java annotations and a formalization of the static and dynamic semantics are presented. Moreover the usefulness of object interfaces is illustrated through various examples.

## 1 Introduction

Java 8 introduced *default methods*, allowing interfaces to have method implementations. The main motivation behind the introduction of default methods in Java 8 is *interface evolution*. That is, to allow interfaces to be extended over time, while preserving backwards compatibility. It soon became clear that default methods could also be used to emulate something similar to *traits* [16]. The original notion of traits by Scharli et al. prescribes, among other things, that: 1) a trait provides a set of methods that implement behaviour; and 2) a trait does not specify any state variables, so the methods provided by traits do not access state variables directly. Java 8 interfaces follow similar principles too. Indeed, a detailed description of how to emulate trait-oriented programming in Java 8 can be found in the work by Bono et al. [2]. The Java 8 team designing default methods, was also fully aware of that secondary use of interfaces, but it was not their objective to model traits: “The key goal of adding default methods to Java was "interface evolution", not "poor man's traits"” [9]. As a result they were happy to support the secondary use of interfaces with default methods as long as it did not make the implementation and language more complex.

Still, the design is quite conservative and appears to be quite limited in its current form to model advanced forms of multiple inheritance. Indeed, our own personal experience of


Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

combining default methods and multiple interface inheritance in Java to achieve multiple implementation inheritance is that many workarounds and boilerplate code are needed. In particular, we encountered difficulties because:

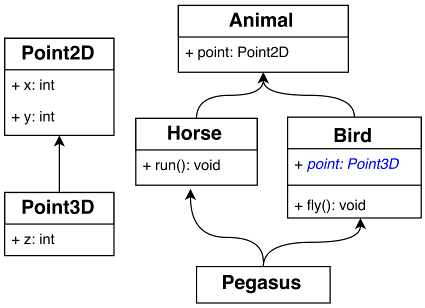
- *Interfaces have no constructors.* As a result classes are still required to create objects, leading to substantial boilerplate code for initialization.
- *Interfaces do not have state.* This creates a tension between using multiple inheritance and having state. Using setter and getter methods is a way out of this tension, but this workaround requires tedious boilerplate classes that latter implement those methods.
- *Useful, general purpose methods, require special care in the presence of subtyping.* Methods such as *clone*, or *fluent* setters [7], not only require access to the internal state of an object, but they also require their types to be refined in subtypes.

Clearly, a way around those difficulties would be to change Java and just remove these limitations. Scala's own notion of traits, for example, allows state in traits. Of course adding state (and other features) to interfaces would complicate the language and require changes in the compiler, and this would go beyond the goals of Java 8 development team.

This paper takes a different approach. Rather than trying to get around the difficulties by changing the language in fundamental ways, we show that, with a simple language feature, default methods and interface inheritance are in fact be very expressive. Our proposed language feature enables powerful object-oriented idioms, using multiple inheritance. We call the language feature *object interfaces*, because such interfaces can be instantiated directly, without the need for an explict class definition. Moreover, object interfaces support *abstract state operations*, providing a way to use multiple inheritance with state in Java. The abstract state operations include various common utility methods (such as getters and setters, or clone-like methods). In the presence of subtyping, such operations often require special care, as their types need to be refined. Object interfaces provide support for type-refinement and can automatically produce code that deals with type-refinement adequately.

Object interfaces do not require changes to the Java runtime or compiler, and they also do not introduce any new syntax (apart from a small annotation). All three features of object interfaces are achieved by reinterpreting existing Java syntax, and are translated into regular Java code without loss of type-safety. Since no new syntax is introduced, it would be incorrect to call object interfaces a language extension or syntactic sugar. So we use the term *language tuning* instead. Language tuning sits in between a lightweight language extension and a glorified library. Language tuning can offer many features usually implemented by a real language extension, but because it does not modify the language syntax pre-existing tools can work transparently on the tuned language. To exploit the full-benefits of language tuning, our prototype implementation of object interfaces uses Java annotations and AST rewriting, allowing existing Java tools (such as IDE's) to work out-of-the-box with our implementation. As a result we could experiment object interfaces with several interesting Java programs, and conduct various case studies.

To formalize object interfaces, we propose Classless Java (CJ): a FeatherweightJava-style [11] calculus, which captures the essence of interfaces with default methods. The semantics of object interfaces is given as a type-directed translation from CJ to itself. In the resulting CJ code all object interfaces are translated into regular CJ (and Java) interfaces with default methods. The translation is proved to be type-preserving, ensuring that the translation does not introduce type-errors. CJ's usefulness goes beyond serving as a calculus to formalize object interfaces. During the development process of CJ, we encountered a bug in the implementation of default methods for the Eclipse Compiler for Java (ECJ). For the




```
interface Animal {} // no points yet!
interface Horse extends Animal{
  default void run() {
    System.out.println("I can run!");
  };
}
interface Bird extends Animal{
  default void fly() {
    System.out.println("I can fly!");
  }
}
interface Pegasus extends Horse, Bird {}
```

**Figure 1** The animal system. Complete animal system on the left, and code for simplified animal system on the right.

program revealing the bug, ECJ behaves differently from both our formalization and Oracle's Java compiler.

To evaluate the usefulness of object interfaces, we illustrate 3 applications and case studies. The first application is a simple solution to the Expression Problem [19], supporting independent extensibility [21], and without boilerplate code. The second application is to show how embedded DSLs using fluent interfaces [7] can be easily defined using object interfaces. Finally, the last application is a larger case study for a simple Maze game implemented with multiple inheritance. For the last application we show that there is a significant reduction in the numbers of lines of code when compared to an existing implementation [2] using plain Java 8. Noteworthy, is the fact that all applications are implemented without defining a single class!

In summary, the contributions of this paper are:

- **Object Interfaces:** A simple feature that allows various powerful multiple-inheritance programming idioms to be expressed conveniently in Java.
- **ClassLess Java (CJ):** A simple formal calculus that models the essential features of Java 8 interfaces with default methods, and can be used to formally define the translation of object interfaces. We prove a type-preservation theorem, and we present a Java program that reveals a bug in the ECJ implementation of default methods.
- **Implementation and Case Studies:** We have a prototype implementation of object interfaces, using Java annotations and AST rewriting. Moreover the usefulness of object interfaces is illustrated through various examples and case studies.
- **Language Tuning** We identify the concept of language tuning and we describe object interfaces as an example. We also discuss how other existing approaches, such as the annotations in project Lombok [23], which can be viewed as language tuning.

## 2 A Running Example: Animals

To propose a standard example, we show `Animals` with a two dimensional `Point2D` representing their `location`. Some kinds of animals are `Horses` and `Birds`. Birds can `fly`, thus their location need to be a three dimensional `Point3D`. Finally, we model `Pegasus` as a kind of `Animal` with the skills of both `Horse`s and `Bird`s.[1]

## 2.1 Simple Multiple Inheritance with Default Methods

Suppose that we want to model an animal system as shown in the left side of Figure 1. `Horse` and `Bird` are subtypes of `Animal`, with methods `run()` and `fly()`, respectively. Pegasus (one of the best known creatures in Greek mythology) can not only *run* but also *fly*! This is the place where *"multiple inheritance"* is needed, because `Pegasus` needs to obtain `fly` and `run` functionality from both `Horse` and `Bird`. Ignoring points, for the moment, a first attempt to model the animal system in Java 8 is given on the right-side of Figure 1. Note that the implementations of the methods `run` and `fly` are defined inside interfaces, using default methods. Moreover, because interfaces support multiple interface inheritance, the interface for `Pegasus` can inherit behaviour from both `Horse` and `Bird`. Although Java interfaces do not allow instance fields, no form of state is needed so far to model the animal system. Therefore this example provides a simple illustratation how a basic form of multiple inheritance, similar to what can be done with traits, can be accomplished in Java 8.

**Instantiation** To use `Horse`, `Bird` and `Pegasus`, some objects must be instantiated first. A first problem with using interfaces to model the animal system is simply that interfaces cannot directly be instantiated: a class is needed. Therefore, we have to first create some classes, such as:

```
class HorseImpl implements Horse {}
class BirdImpl implements Bird {}
class PegasusImpl implements Pegasus {}
```

in order to create animal objects. With those classes, a Pegasus animal, can be created using the class constructor:

```
Pegasus p = new PegasusImpl();
```

There are a couple of annoyances here. Firstly, the sole purpose of the classes is to provide a way to instantiate the objects. Although (in this case) it takes only one line code to provide each of those classes, this code is essentially boilerplate code, which does not add behaviour to the system. Secondly, the namespace gets filled with three additional types. For example, both `Horse` and `HorseImpl` are needed: `Horse` is needed because it needs to be an inferface so that `Pegasus` can use multiple inheritance; and `HorseImpl` is needed to provide object instantiation. Note that, for this very simple animal system, plain Java 8 anonymous classes can be used to avoid these problems. We could have simply instantiated Pegasus using:

```
Pegasus p = new Pegasus() {}; // anonymous class
```

However, as we shall see, once the system gets a little more complicated, the code for instantiation quickly becomes more complex and verbose (even with anonymous classes).

## 2.2 Object Interfaces and Instantiation

To model the animal system with object interfaces all that a user needs to do is to add a **@Obj** annotation to the `Horse`, `Bird`, and `Pegasus` interfaces. For example, for `Bird` and `Pegasus` the code would be as follows:

[1] Some research argues in favour of using subtyping for modelling taxonomies, other research argue against this practice, we do not wish to take sides in this argument, but to provide an engaging example.

```
@Obj interface Bird extends Animal{
  default void fly() {System.out.println("I can fly!");}
}
@Obj interface Pegasus extends Horse, Bird {}
```

The effect of the annotations is that a static *factory* method called `of` is automatically added to the interfaces. With the `of` method a Pegasus object is instantiated as follows:

```
Pegasus p = Pegasus.of();
```

The `of` method provides an alternative to a constructor, which is missing from interfaces. The following code, shows the code correponding to the `Pegasus` interface after the **@Obj** annotation is processed:

```
interface Pegasus extends Horse, Bird { // generated code not visible to users
  static Pegasus of() {return new Pegasus() {};}
}
```

Note that the generated code is transparent to a user, which only sees the original code with the **@Obj** annotation. Compared to the pure Java solution in Section **??**, the solution using object interfaces has the advantage of providing a direct mechanism for object instantiation, which avoids adding boilerplate classes to the namespace.

## 2.3 Object Interfaces with State

The animal system modelled so far, is a simplified version of the system presented in the left-side of Figure 1, and still does not appear to justify the **@Obj** annotation. Lets now move on to modelling the complete animal system. In order to do this, animals will include a notion of points, representing their location in space. As we shall see, modelling stateful components using plain Java 8 quickly becomes cumbersome. The **@Obj** annotation comes to rescue here and allows avoiding significant amounts of boilerplate code.

**`Point2D`: simple immutable data with fields** Two dimensional points should keep track of their coordinates. The usual approach to model points in Java would be to use a class with fields for the coordinates. However here we will illustrate how points are modelled with interfaces, providing a simple example of how to model immutable state with interfaces. Since Java disallows fields inside interfaces, the state is modelled using abstract (getter) methods:

```
interface Point2D{ int x(); int y();}
```

Unfortunatelly, creating a new point object is quite cumbersome, even when using anonymous classes:

```
Point2D p = new Point2D(){ public int x(){return 4;} public int y(){return 2;}}
```

However programmers are not required to use this cumbersome syntax for every object allocation. As programmers do, for ease or reuse, the boring long repetitive code can be encapsulate in a method. A generalization of the `of` static factory method is appropriate in this case:

```
interface Point2D{ int x(); int y();
  static Point2D of(int x, int y){return new Point2D(){
    public int x(){return x;} public int y(){return y;}};}
}
```

**`Point2D` with Object Interfaces** This obvious constructor code can be automatically generated by our **@Obj** annotation. By annotating the interface `Point2D`, the annotation will generate a variation of the shown static method `of`, mimicking the functionality of a simple minded constructor: By looking at the methods that need implementations it first detects what are the fields, then generates an `of` method with one argument for each of them. That is, we can just write

```
@Obj interface Point2D{ int x(); int y();}
```

More precisely, a field or factory parameter is generated for every no-args method requiring implementation (except for methods whose name have special meaning). An example of code using `Point2D` is:

```
Point2D p = Point2D.of(42,myPoint.y());
```

where we return a new point, using `42` as x-coordinate, and taking all the other informations (only `y` in this case) from another point.

**`with` methods in Object Interfaces** The pattern of creating a new piece of data by reusing most information from an old piece of data is very common when programming with immutable data-structures; it is so common that it is supported in our code generation as `with` methods, that is:

```
@Obj interface Point2D {
  int x(); int y(); // getters
  Point2D withX(int val); Point2D withY(int val); // with methods
}
```

Using `with` methods, the point `p` could have been created instead using:

```
Point2D p = myPoint.withX(42);
```

If there is a large number of fields, `with` methods will save programmers from writting large amounts of tedious code that simply copies the values of fields. Moreover, if the programmer wants a different implementation, they just provide it using a **`default`** method. For example:

```
@Obj interface Point2D{
  int x(); int y();
  default Point2D withX(int val){ ... }; default Point2D withY(int val){ ... }}
```

can be used by programmers to define their own behaviour for the `with` methods.

**`Animal` and `Horse`: simple mutable data with fields** Two dimensional points are mathematical entities, thus we chosen to use immutable data structure to model them. However animals are real world entities, and when an animal moves, it is the same animal that now has a different location. We model this with mutable state.

```
interface Animal {
  Point2D location();
  void location(Point2D val);
}
```

Here we declare abstract getter and setter for the mutable "field" `location`. The **@Obj** annotation is not used. This is morally equivalent to an abstract class in full Java; and there is no convenient way to instantiate it. For `Horse`, the **@Obj** annotation is used and a concrete implementation of `run()` method is defined using a default method, which also further illustrates the convenience of *with* methods.

| | Example | Description |
|---|---|---|
| **“fields”/getters** | `int x();` | Retrieves value from field `x`. |
| **withers** | `Point2D withX(int val);` | Clones object; updates field `x` to `val`. |
| **mutable setters** | `void x(int val);` | Sets the field `x` to a new value `val`. |
| **fluent setters** | `Point2D x(int val);` | Sets the field `x` to `val` and returns `this`. |

**Figure 2** Abstract state operations, for a field `x`, allowed by the **@Obj** annotation.

```
@Obj
interface Horse extends Animal {
  default void run() {location(location().withX(location().x() + 20));}
}
```

Creating and using a horse animal is quite simple:

```
Point2D p = Point2D.of(0, 0);
Horse horse = Horse.of(p);
horse.location(p.withX(42));
```

Note how the `of`, `withX` and `location` methods (all generated automatically) provide a basic interface for dealing with animals.

**Summary** Dealing with state (whether mutable or immutable) in object interfaces relies on a notion of abstract state, where only methods to interact with state are available to users. Object interfaces provides support for four different types abstract state operations, which are summarized in Figure 2. The abstract state operations are determined by naming conventions and the types of the methods. Fluent setters are a variant of mutable setters, and are discussed in the case study in Section 7.2.

## 2.4 Object Interfaces and Subtyping

`Birds` are `Animals`, but while `Animals` only need 2D locations, `Birds` need 3D locations. Therefore when the `Bird` interface extends the animal interface, the notion of points needs to be refined. Such kind of refinement usually poses a challenge with typical class-based approaches. Fortunatelly, with object interfaces, we are able to provide a simple and effective solution to that problem.

**Unsatisfactory class-based solutions to field type refinement** In Java if we define an animal class with a field we have a set of unsatisfactory options in front of us:

- Define a `Point3D` field in `Animal`: this is bad since all animals would require more that is needed, and also it may requires the programmer to predict the future, or it may require to adapt the old code to accommodate for new evolutions.
- Define a `Point2D` field in `Animal` and define an extra `int z` field in `Bird`. This solution is very ad-hoc, requires to basically duplicate the difference between `Point2D` and `Point3D` inside of `Bird`. Again, there are many reasons this would be bad, the most dramatic is that it would not scale to a scenario when the programmer of `Bird` and the programmer of `Point3D` are different.
- Redefine getters and setters in `Bird`, always put `Point3D` objects in the field and cast the value out of the `Point2D` field to `Point3D` when implementing the overridden getter. This solution scales to the multiple programmers approach, but requires ugly casts and can be implemented in a wrong way leading to bugs.

Many readers would now think that a language extension is needed. Instead, with object interfaces, another approach is possible.

**Field type refinement with object interfaces** Object interfaces address the challange of type-refinement as follows:

- by covariant method overriding we refine the type of the location field to `Point3D`;
- by *overloading* we define a new setter for the location field with the more precise type;
- using a default method we provide an implementation for the setter called with the old signature.

```
@Obj
interface Bird extends Animal {
  Point3D location();
  void location(Point3D val);
  default void location(Point2D val) { location(location().with(val));}
  default void fly() {
    location(location().withX(location().x() + 40));
  }
}
```

From the type perspective, the key is the covariant method overriding of `location()`. However from the semantic perspective the key is the implementation for the setter with the old signature.

**`Point3D` and properties updater** To implement the old setter in a convenient way, **@Obj** supports one last type of operations: property updater `with` methods. Unlike the `withX` (where `X` stands for a field name) methods presented so far, property updaters take several fields at once, contained in an interface, and copy those fields into fields of another interface. The `Point3D` interface is defined as follows:

```
@Obj
interface Point3D extends Point2D {
  int z();
  Point3D withZ(int z);
  Point3D with(Point2D val);
}
```

By using the new `with` method we may use the information for `z` already stored in the object to forge an appropriate `Point3D` to store. Note how all the informations about what fields sits in `Point3D` and what in `Point2D` is properly encapsulated in the `with` method, and is transparent for the implementer of `Bird`.

**Generated Boilerplate** Just to give a feeling on how much boring code **@Obj** is generating, we show the generated code for the `Point3D` interface Figure 3. The generated code is very repetitive, writing such code by hand can easily induce bugs, for example a distracted programmer may swap the arguments of one of the many calls of `Point3D.of`. Note how `with-` methods are automatically refined in their return type, so that code like `Point3D p=Point3D .of(1,2,3); p=p.withX(42);` will be accepted and behave as expected. If the programmer wishes to suppress this behaviour and keep the signature as it was, it is sufficient to redefine the `with-` methods in the new class repeating the old signature. Again, the philosophy is that if the programmer provides something directly, **@Obj** does not touch it. `with-` methods (functionally) update a field/property at a time. This can be inefficient, and sometime hard to maintain. Often we want to update many fields at the same time, for example using another object as source. Following this idea, the method `with(Point2D)` is an example of

```
interface Point3D extends Point2D{
  Point3D withX(int val); Point3D withY(int val); Point3D withZ(int val);
  Point3D with(Point2D val);
  public static Point3D of(int _x, int _y, int _z){
    int x=_x; int y=_y; int z=_z;
    return new Point3D(){
      public int x(){return x;} public int y(){return y;} public int z(){return z;}
      public Point3D withX(int val){return Point3D.of(val,this.y(),this.z());}
      public Point3D withY(int val){return Point3D.of(this.x(),val,this.z());}
      public Point3D withZ(int val){return Point3D.of(this.x(),this.y(),val);}
      public Point3D with(Point2D val){
        if(val instanceof Point3D){return (Point3D)val;}
        return Point3D.of(val.x(),val.y(),this.z());
        }    };   } }
```

**Figure 3** Generated boilerplate code.

a (functional) properties updater: it takes a certain type and in the current object update all field in that type that match fields in the current type. The idea is that we want as result something that is still like **`this`**, but modified to be as much as possible similar to the parameter. The cast in `with(Point2D)` is trivially safe since it is guarded by an `instanceof` test. The idea is that if the parameter is a subtype of the current exact type, then we can just return the parameter, as something that is just “more” than **`this`**.

## 2.5 Advanced Multiple Inheritance with Object Interfaces

Finally, we can define Pegasus as simply as we did in the simplified (and stateless) version in the right-side of Figure 1:

```
@Obj
interface Pegasus extends Horse, Bird {}
```

Note how even the non trivial pattern for field type refinement is transparently composed, and `Pegasus` has a `Point3D location`. This works because `Horse` do not perform any field type refinement, otherwise we may have to choose/create a common subtype in order for `Pegasus` to exists.

# 3 Interaction of interface methods with interface composition

From Java8 interfaces can have three type of methods: abstract methods, default methods, static methods.

- Static methods are handled in a very clean way: they are visible only on the interface they are explicitly declared. This means that the following code is ill-typed.

  ```
  interface A0 { static int m(){return 1;} }
  interface B0 extends A0 {}
  ...
  B0.m()//ill typed
  ```

  Note how this is different w.r.t. the way static methods are handled in classes. In this way static methods have simply no interaction with interface composition (**`extends`** or **`implements`**).
- Abstract methods composition is accepted when there is one that is the most specific. For example in method `Integer m()` is visible in `C1`.

```
interface A1 { Object m(); }
interface B1 { Integer m();}
interface C1 extends A1,B1 {} //accepted
```

- Default methods conflict with any other default or abstract method. For example the following code is rejected due to conflicting methods.

```
interface A2 { default int m() {return 1;}}
interface B2 { int m(); }
interface C2 { default int m() {return 2;}}
interface D2 extends A2,B2 {} //rejected due to conflicting methods
interface E2 extends A2,C2 {} //rejected due to conflicting methods
```

  Note how this is in contrast with what happens in most trait models, where `D2` would be accepted, and the implementation in `A2` would be part of the behaviour of `D2`.
- The method in the current interface wins over any methods defined in its super-interfaces, provided that the method conform to the subtype of all methods in its super-interfaces, i.e., the method is the most specific one. This also override rejection due to conflicting methods. For example, the following code is accepted, but would be rejected (see before) if the method `m` was not redefined.

```
interface D3 extends A2,B2 { int m(); } //accepted
interface E3 extends A2,C2 { default int m(){return 42;} } //accepted
```

While trying to formally encode the Java specification we have done some tests to clarify corner case behaviour. Consider the following correct declarations:

```
interface A1{T m(); }
interface A2 extends A1{default T m(){ ... } }
interface A3 extends A2{T m(); }

interface B1{default T m(){ ... } }
interface B2 extends B1{T m(); }
interface B3 extends B2{default T m(){ ... } }
```

What happens if we define a new interface `M` extending one $\mathtt{A}_i$ and one $\mathtt{B}_i$? we have 9 cases, that can fit nicely a table:

| M extends | A1 | A2 | A3 |
|---|---|---|---|
| **B1** | conservative error | conflict error | conservative error |
| **B2** | both abstract, accepted | conservative error | both abstract, accepted |
| **B3** | **conservative error** | conflict error | conservative error |

We try to classify the results out of the table:

- **conflict error** happens when the method from $\mathtt{A}_i$ and one $\mathtt{B}_i$ are both implemented. This is also considered an error in most trait models.
- **both abstract, accepted** happens when the method from $\mathtt{A}_i$ and one $\mathtt{B}_i$ are both abstract. This is also considered correct in all trait models.
- **conservative error** happens when the method from $\mathtt{A}_i$ and one $\mathtt{B}_i$ is implemented in only one side. This is different from what we would expect in a trait model, but is coherent with the conservative idea that a method defined in an interface should not silently satisfy a method in another one.

During our experimentation, we found a bug in ECJ (eclipse version of javac): The case **B3**,**A1** is accepted by ECJ4.5.1 and rejected by javac. By email communication with Brian

| | | | |
|---|---|---|---|
| $e$ | $::=$ | $x \mid e.m(\overline{e}) \mid I.m(\overline{e}) \mid I.\mathtt{super}.m(\overline{e}) \mid x{=}e;e' \mid obj$ | expressions |
| $obj$ | $::=$ | $\mathtt{new}\ I()\{\ \overline{field}\ mh_1\{\ \mathtt{return}\ e_1;\}\ \ldots\ mh_n\{\ \mathtt{return}\ e_n;\}\}$ | object creation |
| $field$ | $::=$ | $I\ f{=}x;$ | field declaration |
| $\mathcal{I}$ | $::=$ | $ann\ \mathtt{interface}\ I\ \mathtt{extends}\ \overline{I}\{\ \overline{meth}\ \}$ | interface declaration |
| $meth$ | $::=$ | $\mathtt{static}\ mh\{\ \mathtt{return}\ e;\} \mid \mathtt{default}\ mh\{\ \mathtt{return}\ e;\} \mid mh;$ | method declaration |
| $mh$ | $::=$ | $I_0\ m\ (I_1\ x_1 \ldots I_n\ x_n)$ | method header |
| $ann$ | $::=$ | $\mathtt{@Obj} \mid \mathtt{@ObjWeak} \mid \emptyset$ | annotations |
| $\Gamma$ | $::=$ | $x_1{:}I_1 \ldots x_n{:}I_n$ | environment |

**Figure 4** Grammar of ClassLess Java

Goetz (lead Java8 designer) we have confirmation that the expected behaviour is rejection, and that this is a bug in ECJ.[2]

# 4 Formal Semantics

This section presents a formalization of ClassLess Java: a minimal FeatherweightJava-like calculus which models the essence of Java interfaces with default methods.

## 4.1 Syntax

Figure 4 shows the syntax of ClassLess Java.The syntax formalizes a minimal version of Java 8, focusing on interfaces, default methods and object creation literals. There is no syntax for classes. To help readability we use many metavariables to represent identifiers: $C, x, f$ and $m$; however they all maps to a single set of identifiers as in Java. Expressions consist of conventional constructs such as variables ($x$), method calls ($e.m(\overline{e})$) and static method calls ($I.m(\overline{e})$). For simplicity the degenerate case of calling a static method over the **this** receiver is not considered. A more interesting type of expressions are super calls ($I.\mathtt{super}.m(\overline{e})$), whose semantic is to call the (non static) method $m$ over the **this** receiver, but statically dispatching to the version of the method as visible in the interface $I$. A simple form of field updates ($x{=}e;e'$) is also modelled. In the syntax of field updates $x$ is expected to be a field name. After updating the field $x$ using the value of $e$, the expression $e'$ is executed. To blend the statement based nature of Java and the expression based nature of our language, we consider a method body of the form $\mathtt{return}\ x{=}e;e'$ to represent $x{=}e;\mathtt{return}\ e'$ in Java. Finally, there is an object initialization expression from an interface $I$, where (for simplicity) all the fields are initialized with a variable present in scope. Note how our language is a subset of Java 8. To be compatible with java, the concrete syntax for an interface declaration with empty supertype list would also omit the **extends** keyword. Following standard practise, we consider a global Interface Table ($IT$) mapping from interface names $I$ to interface declarations $\mathcal{I}$.

The environment $\Gamma$ is a mapping from variables to types. As usual, we allow a functional notation for $\Gamma$ to do variable lookup. Moreover, to help us defining auxiliary functions, a functional notation is also allowed for a set of methods $\overline{meth}$, using the method name $m$ as a key. That is, we define $\overline{meth}(m) = meth$ iff there is a unique $meth \in \overline{meth}$ whose name is $m$. For convenience, we define $\overline{meth}(m) = \mathsf{None}$ otherwise; moreover $m \in \mathsf{dom}(\overline{meth})$ iff $\overline{meth}(m) = meth$. For simplicity, we do not model overloading, thus for an interface to be well formed its methods must be uniquelly indentfied by their name.

[2] This bug is fixed in the developer version ECJ???, not yet released as stable.

$$\frac{\begin{array}{c}\Gamma \vdash e \in I_0\\ \forall i \in 1..n \;\; \Gamma \vdash e_i \in _ <: I_i\\ \mathsf{mtype}(m, I_0) = I_1 \ldots I_n \to I\end{array}}{\Gamma \vdash e.m(e_1 \ldots e_n) \in I}\;(\text{T-Invk})
\qquad
\frac{\begin{array}{c}\forall i \in 1..n \;\; \Gamma \vdash e_i \in _ <: I_i\\ \mathsf{mtypeS}(m, I_0) = I_1 \ldots I_n \to I\end{array}}{\Gamma \vdash I_0.m(e_1 \ldots e_n) \in I}\;(\text{T-StaticInvk})$$

$$\frac{\begin{array}{c}\Gamma(\texttt{this}) <: I_0\\ \forall i \in 1..n \;\; \Gamma \vdash e_i \in \_ <: I_i\\ \mathsf{mtype}(m, I_0) = I_1 \ldots I_n \to I\end{array}}{\Gamma \vdash I_0.\texttt{super}.m(e_1 \ldots e_n) \in I}\;(\text{T-SuperInvk})$$

$$\frac{\Gamma(x) = I}{\Gamma \vdash x \in I}\;(\text{T-Var})
\qquad
\frac{\begin{array}{c}\forall i \in 1..k \;\; \Gamma(x_i) <: I_i\\ \forall i \in 1..n \;\; \Gamma, f_1{:}I_1, \ldots, f_k{:}I_k, \texttt{this}{:}I, \Gamma^{mh_i} \vdash e_i \in _ <: I^{mh_i}\\ \mathsf{sigvalid}(mh_1 \ldots mh_n, I) \qquad \mathsf{alldefined}(mh_1 \ldots mh_n, I)\end{array}}{\Gamma \vdash \texttt{new}\ I()\{\ I_1\ f_1\texttt{=}x_1\texttt{;} \ldots I_k\ f_k\texttt{=}x_k\texttt{;}\ mh_1\{\ \texttt{return}\ e_1\texttt{;}\} \ldots mh_n\{\ \texttt{return}\ e_n\texttt{;}\}\} \in I}\;(\text{T-Obj})$$

$$\frac{\begin{array}{c}\Gamma \vdash e \in _ <: \Gamma(x)\\ \Gamma \vdash e' \in I\end{array}}{\Gamma \vdash x\texttt{=}e;e' \in I}\;(\text{T-update})
\qquad
\frac{\begin{array}{c}IT(I) = ann\ \texttt{interface}\ I\ \texttt{extends}\ I_1 \ldots I_n\ \{\ \overline{meth}\ \}\\ \forall \texttt{default}\ mh\{\ \texttt{return}\ e;\} \in \overline{meth},\quad \Gamma^{mh}, \texttt{this}{:}I \vdash e \in _ <: I^{mh}\\ \forall \texttt{static}\ mh\{\ \texttt{return}\ e;\} \in \overline{meth},\quad \Gamma^{mh} \vdash e \in _ <: I^{mh}\\ \mathsf{dom}(I) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})\end{array}}{I\ \text{OK}}\;(\text{T-Intf})$$

**Figure 5** CJ Typing

## 4.2 Typing

Typing statement $\Gamma \vdash e \in I$ reads "in the environment $\Gamma$, expression $e$ has type $I$.". Before discussing the typing rules we discuss some of the used notation. As a shortcut, we write $\Gamma \vdash e \in I <: I'$ instead of $\Gamma \vdash e \in I$ and $I <: I'$.

We omit the definition of the usual traditional subtyping relation between interfaces, that is the transitive and reflexive closure of the declared **extends** relation. [3] The auxiliary notation $\Gamma^{mh}$ trivially extracts the environment from a method header, by collecting the all types and names of the method parameters. The notation $m^{mh}$ and $I^{mh}$ denotes respectivelly, extracting the method name and the return type from a method header. $\mathsf{mbody}(m, I)$, defined in Section **??**, returns the full method declaration as seen by $I$, that is the method $m$ can be declared in $I$ or inherited from another interface. $\mathsf{mtype}(m, I)$ and $\mathsf{mtypeS}(m, I)$ return the type signature from a method (using $\mathsf{mbody}(m, I)$ internally). $\mathsf{mtype}(m, I)$ is defined only for non static methods, while $\mathsf{mtypeS}(m, I)$ only on static ones. We use $\mathsf{dom}(I)$ to denote the set of methods that are defined for type $I$, that is: $m \in \mathsf{dom}(I)$ iff $\mathsf{mbody}(m, I) = meth$.

In Figure 5 we show the typing rules. We discuss the most interesting rules, that is (T-Obj) and (T-Intf). Rule (T-Obj) is the most complex typing rule. Firstly, we need to ensure that all field initializations are type correct, by looking up the type of each variable assigned to a field in the typing environment and verifying that such type is a subtype of the field type. Secondly, we check that all method bodies are well-typed. To do this the enviroment used to check the method body needs to be extended appropriately: we add all

[3] MARCO: find a better place for Notice how there are no classes, thus there is no subclassing. We believe that this approach may scratch an old itching point in the long struggle of subtyping versus subclassing: According to some authors, from a software engineering perspective, interfaces are just a kind of classes. Others consider more opportune to consider interfaces are pure types. In this vision our language would have no subclassing. We do not know how to conciliate those two viewpoints and ClassLess Java design. We do not have Classes purely in the Java sense.

fields and their types; add **this** : $I$; and add the arguments (and types) of the respective method. Now we need to check if the object is a valid extension for that specific interface. This can be logically diveded in two steps: First we check that all method headers are valid with respect to the corresponding method already present in $I$;

- $\mathsf{sigvalid}(mh_1 \ldots mh_n, I) \quad = \quad \forall i \in 1..n \quad mh_i\, \mathtt{;} <: \mathsf{mbody}(m^{mh_i}, I)$

Here we require that for all newly declared methods, there is a method with the same name defined in the interface $I$, and that such method is a subtype of the newly introduced one. We define subtyping between methods in a general form that will be useful also later.

- $I\ m\mathtt{(}I_1 x_1 \ldots I_n x_n\mathtt{)}\mathtt{;} <: I'\, m\mathtt{(}I_1 x'_1 \ldots I_n x'_n\mathtt{)}\mathtt{;} \quad = \quad I <: I'$
- $meth <: \mathbf{default}\ mh\mathtt{\{}\mathbf{return}\ \_\mathtt{;\}} \quad = \quad meth <: mh\mathtt{;}$
- $\mathbf{default}\ mh\mathtt{\{}\mathbf{return}\ \_\mathtt{;\}} <: meth \quad = \quad mh\mathtt{;} <: meth$

Two method headers are subtypes if all the parameter types are the same and the return types are subtypes. That is, we allows return type specialization as introduced in Java5. Default methods are subtypes if their method headers are subtypes.

Finally, all abstract methods in the interface (that is methods that need to be explicitly overridden) have been implemented. That is, we define a method with the same name.

- $\mathsf{alldefined}(mh_1 \ldots mh_n, I) \quad = \quad \forall m$ such that $\mathsf{mbody}(m, I) = mh\mathtt{;} \exists i \in 1..n\ m^{mh_i} = m$

The rule ((T-INTF)) checks that an interface $I$ is correctly typed. First we check that the body of all the default and static methods are well typed. Then we check that $\mathsf{dom}(I)$ is the same of $\mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})$. This is not a trivial check, since $\mathsf{dom}(I)$ is defined using $\mathsf{mbody}$, that is undefined in many cases: notably if a method $meth \in \overline{meth}$ is not compatible with some method in $\mathsf{dom}(I_1) \ldots \mathsf{dom}(I_n)$ or if any method in both $\mathsf{dom}(I_i)$ and $\mathsf{dom}(I_j)$ $(i, j \in 1..n)$ is conflicting.

## 4.3 Auxiliary Definitions

Defining $\mathsf{mbody}$ is not trivial, and requires quite a lot of attention to the specific model of Java Interfaces, and on how it differs w.r.t. Java Class model. $\mathsf{mbody}(m, I)$ denotes the actual method $m$ (body included) that interface $I$ owns. It can either be defined originally in $I$ or in its supertypes, and then passed to $I$ via inheritance.

We use internally a special modifier $\mathsf{conflicted}$ to denote the case of two methods with conflicting implementation.

- $\mathsf{mbody}(m, I_0) \quad = \quad \mathsf{override}(\overline{meth}(m), \mathsf{shadow}(m, \mathsf{needed}(m, \overline{I}\,)))$
  with $IT(I_0) = \quad ann\ \mathbf{interface}\ I_0\ \mathbf{extends}\ \overline{I}\mathtt{\{}\ \overline{meth}\mathtt{\}}$

As you can see, we are delegating the work to three others auxiliary functions: $\mathsf{needed}(\overline{I}), \mathsf{shadow}(m, \overline{I})$ and $\mathsf{override}(meth, meth')$

$\mathsf{needed}$ recovers from the interface table only the "needed" methods, that is, the non static ones that are not transitively reachable by following another, less specific, superinterface chain. Formally:

- $meth \in \mathsf{needed}(m, \overline{I})$
  with $\mathsf{mbody}(m, I) = meth,\ meth$ not a static method, and
  $I \in \overline{I}$ such that $\forall I' \in \overline{I} \setminus I$ not $I' <: I$

$\mathsf{shadow}$ choose the most specific version of a method, that is the unique version available, or a conflicted version from a set of possibilities. We do not model overloading, so it is an error if multiple versions are available with different parameter types. Formally:

- $\mathsf{shadow}() \quad = \quad \mathsf{None}$
- $\mathsf{shadow}(meth) \quad = \quad meth$
- $\mathsf{shadow}(\overline{mh;}) \quad = \quad \mathsf{mostSpecific}(\overline{mh;})$
- $\mathsf{shadow}(\overline{meth}) \quad = \quad \mathsf{conflicted}\, mh;$

  with $\overline{meth}$ not of the form $\overline{mh;}$ and $\mathsf{mostSpecific}(\overline{meth}) \in \{mh;, \texttt{default } mh\{\texttt{return \_;}\}\}$

Where mostSpecific return the most specific method , that is a method whose type is the subtype of all the others. Since method subtyping is a partial ordering, this may be not defined, this in turn makes shadow, and the whole mbody not defined for that specific $m$. Rule (T-INTF) relies on this behaviour.

- $\mathsf{mostSpecific}(\overline{meth}) \quad = \quad meth$

  with $meth \in \overline{meth}$ and $\forall meth' \in \overline{meth} : meth <: meth'$

The override function models how the implementation in an interface can override implementation in the superinterface; even in case of a conflict. Note how we use the special value None, and how (forth case) overriding can solve a conflict.

- $\mathsf{override}(\mathsf{None}, \mathsf{None}) \quad = \quad \mathsf{None}$
- $\mathsf{override}(meth, \mathsf{None}) \quad = \quad meth$
- $\mathsf{override}(\mathsf{None}, meth) \quad = \quad meth$

  with $meth$ not of the form $\mathsf{conflicted}\ mh;$
- $\mathsf{override}(meth, meth') \quad = \quad meth$

  with $meth' \in \{mh;, \texttt{default } mh\{\texttt{return \_;}\}, \mathsf{conflicted}\ mh;\}, meth <: meth'$

## 5 What @Obj Generates

We now show what the **@Obj** annotation generates. We present a formal definition for most of the generated methods; however in our formalism we do not consider casts or `instanceof`, so we do not include the `with` method. For the same reason we do not include `void` returning setters, since they are just a minor variation over the more interesting fluent setters, and they would require special handling just for the conventional `void` type.

Furthermore, we introduce **@ObjWeak**, which is only an annotation for illustrating the translation, hence is not implemented in Lombok. By this, the real translation is divided into two parts: **@Obj** automatically refines the return types only, and **@ObjWeak** generates the constructor method `of`. In the source code, the only annotation **@Obj** combines both of the work and finishes the translation at a time.

### 5.1 Translation Function

Two translation functions for the two parts are presented as follows:

- $[\![\texttt{@Obj interface } I_0 \texttt{ extends } \overline{I}\{\ \overline{meth}\ \}]\!] \quad = \quad \texttt{@ObjWeak interface } I_0 \texttt{ extends } \overline{I}\{\ \overline{meth}\ \overline{meth}'\}$

  with $\overline{meth}' = \mathsf{otherMethods}(I_0, \overline{meth})$

To translate an interface annotated by **@Obj**, we add some other methods for type refinement, then add a new annotation **@ObjWeak** to the interface.

- $[\![\texttt{@ObjWeak interface } I_0 \texttt{ extends } \overline{I}\{\ \overline{meth}\ \}]\!] \quad = \quad \emptyset\ \texttt{interface } I_0 \texttt{ extends } \overline{I}\{\ \overline{meth}\ meth'\}$

  with $\mathsf{valid}(I_0), \texttt{of} \notin \mathsf{dom}(\overline{meth})$ and $meth' = \mathsf{ofMethod}(I_0)$

To translate an interface annotated by **@ObjWeak**, we add the `of` method. However, first of all we check if the interface is valid for annotation:

$\mathsf{valid}(I_0)$ holds if $\forall m \in \mathsf{dom}(I_0)$, if $mh; = \mathsf{mbody}(m, I_0)$, one case is satisfied: $\mathsf{isField}(meth)$, $\mathsf{isWith}(meth, I_0)$ or $\mathsf{isSetter}(meth, I_0)$. That is, we can categorize all the *not implemented* methods in a pattern that we know how to implement.

Moreover, we check that the method `of` is not already defined by the user. In our simplified formalization we consider this to be just an error. In our prototype we keep overloading into account, and so we check that an `of` method with the same signature of the one we would generate is not already present. MARCO: Do we check it? HAOYUAN: Not sure if it is $\texttt{of} \notin \mathsf{dom}(\overline{meth})$ or $\texttt{of} \notin \mathsf{dom}(I_0)$. It is an error if a static method overrides an instance method.

In the following we will write $\texttt{with\#}m$ to append $m$ to `with`, following the camelCase rule, so the first letter of $m$ must be lower-case and is turned in upper-case upon merging. For example `with#foo=withFoo`. Special names $\mathsf{special}(m)$ are `with` and all the identifiers of form $\texttt{with\#}m$.

## 5.2 otherMethods

The definition of $\mathsf{otherMethods}(I_0, \overline{meth})$ is as follows:

HAOYUAN: I think $\texttt{with\#}m \notin \mathsf{dom}(\overline{meth})$ is not enough. The withM method with return type $I_0$ can also be defined already in a super interface. Like:

```
interface A{ int x(); B withX(int x);}
@Obj interface B extends A{}
```

- $I_0\ \texttt{with\#}m\texttt{(}I\ \texttt{\_val);} \in \mathsf{otherMethods}(I_0, \overline{meth}) \quad = \quad \mathsf{isWith}(\mathsf{mbody}(\texttt{with\#}m, I_0), I_0)$
  with $I\ m\texttt{();} \in \mathsf{fields}(I_0)$ and $\texttt{with\#}m \notin \mathsf{dom}(\overline{meth})$
- $I_0\ \_m\texttt{(}I\ \texttt{\_val);} \in \mathsf{otherMethods}(I_0, \overline{meth}) \quad = \quad \mathsf{isSetter}(\mathsf{mbody}(\_m, I_0), I_0)$
  with $I\ m\texttt{();} \in \mathsf{fields}(I_0)$ and $\_m \notin \mathsf{dom}(\overline{meth})$

The methods that we generate in the interface are `with-` and setters. We discover if there is the need of generating such methods by checking if the method is unimplemented in $I_0$. This is needed only if we need to refine the return type. To discover if this is the case, we check if such `with-` or setter is required by $I_0$, but is not already present in the methods directly declared in $I_0$.

- $meth \in \mathsf{fields}(I_0) \quad = \quad \mathsf{isField}(meth)$ and $meth = \mathsf{mbody}(\_, I_0)$
- $\mathsf{isField}(I\ m\texttt{();}) \quad = \quad$ not $\mathsf{special}(m)$
- $\mathsf{isWith}(I'\ \texttt{with\#}m\texttt{(}I\ x\texttt{);}, I_0) = \quad I_0 <: I', \mathsf{mbody}(m, I_0) = I\ m\texttt{();}$ and not $\mathsf{special}(m)$
- $\mathsf{isSetter}(I'\ \_m\texttt{(}I\ x\texttt{);}, I_0) \quad = \quad I_0 <: I', \mathsf{mbody}(m, I_0) = I\ m\texttt{();}$ and not $\mathsf{special}(m)$

## 5.3 ofMethod

We now formally define ofMethod, the function that generates the method `of`, that behaves like a factory. To avoid boring digressions about well known ways to find unique names, for the sake of this formalization we assume that no-args methods do not start with underscore, and we prefix method names with underscore to obtain valid parameter names.

- $\mathsf{ofMethod}(I_0) = \quad \textbf{static}\ I_0\ \texttt{of(}I_1\ \_m_1\texttt{,}\ldots I_n\ \_m_n\texttt{)\{}\ \textbf{return new}\ I_0\texttt{()\{}$
  $I_1\ m_1 = \_m_1\texttt{;}\ldots I_n\ m_n = \_m_n\texttt{;}$
  $I_1\ m_1\texttt{()\{}\textbf{return}\ m_1\texttt{;\}}\ \ldots I_n\ m_n\texttt{()\{}\textbf{return}\ m_n\texttt{;\}}$
  $\mathsf{withMethod}(I_1, m_1, I_0, \overline{e}_1)\ldots\mathsf{withMethod}(I_n, m_n, I_0, \overline{e}_n)$
  $\mathsf{setterMethod}(I_1, m_1, I_0)\ldots\mathsf{setterMethod}(I_n, m_n, I_0)$
  `};}`
  with $\mathsf{fields}(I_0) = I_1\ m_1\texttt{();},\ldots I_n\ m_n\texttt{();}$and $\overline{e}_i = m_1\texttt{,}\ldots\texttt{,}m_{i-1}\texttt{,\_val,}m_{i+1}\texttt{,}\ldots\texttt{,}m_n$

The function $\mathsf{fields}(I_0)$ (defined before) denotes all the fields in the current interface. For methods inside the interface with the form $I_i\ m_i\texttt{();}$

- $m_i$ is the field name, and have type $I_i$.

- $m_i$`()` is the getter, that just return the current field value.
- if a method `with#`$m_i$ is required, then it is implemented by calling the `of` method using the current value for all the fields except for $m_i$. Such new value is provided as parameter. This correspond to the expressions $\overline{e}_i$.
- $\_m_i$`(`$I_i$ `_val)` is the setter. In our prototype we use name $m_i$, here we use the underscore to avoid modelling overloading.

The auxiliary functions are defined below. Note that we do not need to check if some header is a subtype of what we would generate, this is ensured by $\mathsf{valid}(I_0)$.

- $\mathsf{withMethod}(I, m, I_0, \overline{e}) \;=\; I_0$ `with#`$m$`(`$I$ `_val){`**`return`** $I_0$`.of(`$\overline{e}$`);}`
  with $\mathsf{mbody}($`with#`$m, I_0)$ is of form $mh$;
- $\mathsf{withMethod}(I, m, I_0, \overline{e}) \;=\; \emptyset$ otherwise
- $\mathsf{setterMethod}(I, m, I_0) \;=\; I_0$ $\_m$`(`$I$ `_val){` $m$`= _val;`**`return this`**`;}`
  with $\mathsf{mbody}(\_m, I_0)$ is of form $mh$;
- $\mathsf{setterMethod}(I, m, I_0) \;=\; \emptyset$ otherwise

## 5.4 Other features

We have not formally modelled non fluent setters and the `with` method; informally

- For methods inside the interface with the form `void` $m$`(`$I$ $x$`);`:
  - Check if exist method $I$ $m$`();`. If not, generate error (that is, is not $\mathsf{valid}(I_0)$).
  - Generate implemented setter method inside `of`:
    **`public`** `void` $m$`(`$I$`_val){` $m$`=_val;}` Note how there is no need to refine the return type for non fluent setters, thus we do not need to generate the method header in the interface body itself.
- For methods with the form $I'$ `with(`$I$ $x$`);`:
  - $I$ must be an interface type (no classes or primitive types).
  - As for before, check that $I'$ is a supertype of the current interface type $I_0$.
  - Generate implemented `with` method inside `of`:
    **`public`** $I_0$ `with(`$I$`_val){`
    **`if`**`(_val instanceof` $I_0$`){`**`return`** `(`$I_0$`)_val;}`
    **`return`** $I_0$`.of(`$e_1 \ldots e_n$`);}`
    where with $m_1 \ldots m_n$ fields of $I_0$, $e_i =$`_val.`$m_i$`()` if $I$ has a $m_i$`()` method; otherwise $e_i = m_i$.
  - If needed, as for `with-` and setters, generate the method header with refined return type in the interface.

MARCO: insert somewhere description of fluent setter [?]. This allows for convenient and chains of setters, as we will show later MARCO: insert forward reference when available.

## 5.5 Results

Informally, in a type correct interface table, if we translate a specific interface, then the resulting table is type correct, or there is a subtype of the translated interface in the original table. **THEOREM.** For a given $\mathcal{I}_0 \ldots \mathcal{I}_n$ interface table such that $\forall \mathcal{I} \in \mathcal{I}_0 \ldots \mathcal{I}_n, \mathcal{I}$ OK, if $\mathcal{I}_0$ has **@Obj**, $\mathsf{valid}(I_0)$ and `of` $\notin \mathsf{dom}(I_0)$, there is no $I <: I_0$, there is no **`new`** $I_0$`(){..}`, then in the interface table $[\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n$
$\forall \mathcal{I} \in [\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n$ $\mathcal{I}$ OK.

To understand this theorem statement, we need to understand three kind of guarantees that we can offer for safety:

- *Self coherence*: the generated code itself is well-typed; type errors are not present in code the user have not wrote. In our case it means that either **@Obj** produces in controlled way an understandable error, or the class can be successfully annotate and the generated code is well typed. We guarantee *Self coherence*.
- *Client coherence*: all the client code (as for example method calls) that is well typed without the generation/instrumentation process is well typed also after the generation. That is, the annotation do not remove any functionality, is just adding more behaviour. We guarantee *Client coherence*.
- *Heir coherence*: Interfaces (and in general classes) inheriting from instrumented code are well typed if they was well typed without the instrumentation. This would require to not add any (default or abstract) method to the annotated interfaces, including type refinement. We do not guarantee *Heir coherence*. Indeed consider the following example

```
interface A { int x(); A withX(int x); }
@Obj interface B extends A {}
interface C extends B { A withX(int x); }
```

By the translation rule, **@Obj** would generate in `B` a method "`B withX(int x);`". This would break `C`.

To prove the theorem we introduce two lemmas below. The complete proof is available in Appendix A.1, A.3 and A.4.

MARCO: you need to use the Lemma/theorem macros **LEMMA 2.** If $\mathcal{I}_0$ OK, then $[\![\mathcal{I}_0]\!]$ OK. This is what we defined as self coherence before.

## 6 Implementation

Our implementation is based on an extension to Lombok. The Lombok project [23] is a Java tool that aims at removing (or reducing) Java boilerplate code via annotations. There are a number of annotations provided by the original Lombok, including **@Getter**, **@Setter**, **@ToString** for generating getters, setters and `toString` methods, respectively. Furthermore, Lombok provides a number of interfaces for users to create custom transformations, as extensions to the original framework. A transformation is based on a handler, which acts on the AST from parsing the annotated node and returns a modified AST for analysis and generation afterwards. Such a handler can either be a Javac handler or an Eclipse handler.

The annotation we created is **@Obj**. In Eclipse, with an interface annotated by **@Obj**, the automatic annotation processing is performed transparently and the information of the interface from compilation is captured in the "Outline" window. This includes all the methods inside the interface as well as the generated ones. The custom transformation is easy and convenient to use. For example this means that the IDE functionality for content assist and autocomplete will work for the newly generated methods. The biggest reasons to use Lombok rather than using a conventional Java annotation processor are:

- Lombok modifies the generation process of the class files, by directly modifying the AST. Neither the source code is modified nor new Java files are generated.
- Moreover, and probably more importantly, Lombok is capable of generating code *inside* a class/interface. This is the ability that conventional Java annotation processors do not provide.

**Limitations** Our prototype implementation using Lombok has certain limitations:

- The prototype does not support separate compilation yet. Currently all related interfaces have to appear in a single Java file. Therefore, changes to a single interface would require re-compiling the whole file. This compilation limitation is not caused by our algorithm. It is a Lombok implementation related issue: in Lombok it is hard to capture a type declaration from its reference, even harder when the type declaration is in other files (we have not found a way to do this yet).
- At this stage our implementation only realizes the Eclipse handler and our experiments are all conducted in Eclipse. The implementation for `javac` is missing.
- The current implementation does not take type-parameters into consideration, thus it does not support generics yet.

**Comparison with other Lombok annotations** The Lombok project provides a set of predefined annotations, including constructor generators similar as ours (e.g., **@NoArgsConstructor**, **@RequiredArgsConstructor** and **@AllArgsConstructor**). They generate various kinds of constructors for *classes*, with or without constructor arguments. This set of annotations is of great use, especially when used together with other features provided in Lombok (e.g., **@Data**). Moreover, the implementation of these annotations in Lombok gives us hints on how to implement **@Obj**. However, none of these annotations can model what we are doing with **@Obj**- generating constructor-methods (**of**) for *interfaces*. Apart from constructors, **@Obj** also provides other convenient features (including generating fluent setters, type refinement,etc), which the base Lombok project does not provide. Finally, while **@Obj** is formalized, none of Lombok's annotations have been studied in a formal way.

# 7 Case Studies

In this section we conduct three case studies which reveals various advantages using **@Obj** annotation. The first case study provides a simple way to solve the Expression Problem, while supporting multiple, independent extensions in Java. The second case study show how to model an embedded DSL for SQL languages with fluent interfaces. Finally, the third case study models a simple game, and compares our implementation with an existing one, showing that the amount of code is reduced significantly using **@Obj**.

## 7.1 A Trivial Solution to the Expression Problem with Object Interfaces

The *Expression Problem* (EP) [19] is a well-known problem about modular extensibility issues in software evolution. Recently, a new solution [20], MARCO: repharesed using only covariant type refinement, has been shown. When such solution is modelled with interfaces and default methods, it can even provide independent extensibility: the ability to assemble a system from multiple, independently developed, extensions. Unfortunatelly, the required instantiation code makes a plain Java solution verbose and cumbersome to use. The **@Obj** annotation is enough to remove the boilerplate code, making the presented approach very appealing.

**Initial System** YANLIN: code figure updated. In the formulation of the EP there is an initial system that models arithmetic expressions with only literals and addition, and an initial operation `eval` for expression evaluation.
As shown in Figure **??**, `Exp` is the common super-interface with an evaluation operation `eval()` inside. Sub-interfaces `Lit` and `Add` extend interface `Exp` with default implementations for the `eval` operation. The number field `x` of a literal is represented as a getter method `x()` and expression fields (`e1` and `e2`) of an addition as getter methods `e1()` and `e2()`.

```
interface Exp { int eval(); }
@Obj interface Lit extends Exp {
 int x();
 default int eval() { return x(); }
}
@Obj interface Add extends Exp {
 Exp e1(); Exp e2();
 default int eval() {
  return e1().eval() + e2().eval();
 }
}
```

```
interface ExpP extends Exp { String print(); }
@Obj interface LitP extends Lit, ExpP {
   default String print() {return "" + x();}
}
@Obj interface AddP extends Add, ExpP {
   ExpP e1(); ExpP e2();//return type refined!
   default String print() {
   return "(" + e1().print() + " + " + e2().
        print() + ")";
 }}
```

**Figure 6** The Expression Problem. Initial system on the left, and code for adding print operation on the right.

**Adding a New Type of Expressions** In the object oriented paradigm, it is easy to add new types of expressions. For example, the following code shows how to add subtraction.

```
@Obj interface Sub extends Exp {Exp e1(); Exp e2();
  default int eval() {return e1().eval() - e2().eval();}}
```

MARCO: text under need to be improved **Adding a New Operation** The difficulty of the EP in object-oriented languages arises from adding new operations. For example, adding a pretty printing operation, would typically involve changing all of the existing code. However, a solution to the EP forbids this and it also forbids using casts. In other words, it should be possible to add the pretty printing operation in a type-safe and modular way. It turns out that, this can be done easily, using our object interfaces approach. The code in Figure **??** shows how to add the new operation `print`.

The interface `ExpP` extending interface `Exp` is defined with the extra method `print()`. Interfaces `LitP` and `AddP` are defined with default implementations of `print()`, extending base interfaces `Lit` and `Add`, respectively. Importantly, note that in `AddP`, the types of the "*fields*" (i.e. the getter methods) `e1` and `e2` are refined. If the types were not refined then the body of the `print` method in `AddP` would fail to type-check.

**Independent Extensibility** To show that our approach supports independent extensibility [21], we first define a new operation `collectLit` (which collects all literal components in an expression) on expressions. For space reasons, we omit the definitions of the methods:

```
interface ExpC extends Exp { List<Integer> collectLit(); }
@Obj interface LitC extends Lit, ExpC {
   default List<Integer> collectLit() { ... }}
@Obj interface AddC extends Add, ExpC {
   ExpC e1(); ExpC e2(); //return type refined!
   default List<Integer> collectLit() { ... }}
```

Now we combine the two extensions (`print` and `collectLit`) together:

```
interface ExpPC extends ExpP, ExpC {}
@Obj interface LitPC extends ExpPC, LitP, LitC {}
@Obj interface AddPC extends ExpPC, AddP, AddC {ExpPC e1(); ExpPC e2();}
```

`ExpPC` is the new expression interface supporting `print` and `collectLit` operations; `LitPC` and `AddPC` are the extended variants. Notice that except for the routine of **extends** clauses, no glue code is required. Return types of `e1,e2` are also automatically refined to `ExpPC`.

Note that the code for instantiation is automatically generated by **@Obj**. So, for example creating a simple expression of type `ExpPC` is as simple as:

```
ExpPC e8 = AddPC.of(LitPC.of(3), LitPC.of(4));
```

In contrast, in a pure Java solution, the tedious instantiation code would need to be defined manually.

## 7.2 Embedded DSLs with Fluent Interfaces

Since the style of fluent interfaces was invented in Smalltalk as method cascading, more and more languages came to support fluent interfaces, including JavaScript, Java, C++, D, Ruby, Scala, etc. For most languages, to create fluent interfaces, programmers have to either handwrite everything or create a wrapper around the original non-fluent interfaces using **this**. In Java, there are several libraries (including jOOQ, op4j, fluflu, JaQue, etc) providing useful fluent APIs. However most of them only provide a fixed set of predefined fluent interfaces. Fluflu enables the creation of a fluent API and implements control over method chaining by using Java annotations. However methods that returns **this** are still hand-written.

The **@Obj** annotation can also be used to create fluent interfaces. When creating fluent interfaces with **@Obj**, there are two main advantages:

1. Instead of forcing programmers to hand write code using **return this**, our approach with **@Obj** annotation removes this verbosity and automatically generate fluent setters.
2. Along the extension direction, the return types of fluent setters are automatically refined.

We use embedded DSLs of two simple SQL query languages to illustrate. The first query language `Database` models `select`, `from` and `where` clauses:

```
@Obj interface Database {
  String select(); Database select(String select);
  String from();  Database from(String from);
  String where(); Database where(String where);
  static Database of() {return of("", "", "");}
}
```

The main benefit that fluent methods give us is the convinience of method chaining:

```
Database query1 = Database.of().select("a, b").from("Table").where("c > 10");
```

Note how all the logic for the fluent setters is automatically provided by the **@Obj** annotation.

**Extending the Query Language** The previous query language can be extended with a new feature `orderBy` which orders the result records by a field that users specify. With **@Obj** programmers just need to extend the interface `Database` with new feature, and the return type of fluent setters in `Database` is automatically refined to `ExtendedDatabase`:

```
@Obj interface ExtendedDatabase extends Database {
  String orderBy(); ExtendedDatabase orderBy(String orderBy);
  static ExtendedDatabase of() {return of("", "", "","");}
}
```

This way, when a `ExtendedDatabase` query created, all the fluent setters return the correct type, and not the old `Database` type, which would prevent calling `orderBy`.

```
ExtendedDatabase query2 = ExtendedDatabase.of().select("a, b").from("Table").
    where("c > 10").orderBy("b");
```

### 7.3 A Maze Game

The last case study is a simplified variant of Maze game, which is often used [8, 2] to evaluate code reuse ability related to inheritance and design patterns. In the game, there is a player with the goal of collecting as many coins as possible. She may enter a room with several doors to be chosen among. This is a good example because it involves code reuse (different kinds of doors inherit a common type, with different features and behaviour), multiple inheritance (a special kind of door may require features from two other door types) and it also shows how to model operations `symmetric sum`, `override` and `alias` as trait-oriented programming. The game has been implemented using plain Java 8 and default methods by Bono et. al [2], and the code for that implementation is available online. We reimplemented the game using **@Obj**. Due to space constraints, we omit the code here. The following table summarizes the number of lines of code and classes/interfaces in each implementation:

| | SLOC | # of classes/interfaces |
|---|---|---|
| Bono et al. | 335 | 14 |
| Ours | 199 | 11 |
| Reduced by | 40.6% | 21.4% |

The **@Obj** annotation allowed us to reduce the interfaces/classes used in Bono et al.'s implementation by 21.4% (from 14 to 11). The reductions was due to the replacement of instantiation classes with generated `of` methods. The number of source lines of code (SLOC) was reduced by 40% due to both the removal of instantiation overhead and generation of getters/setters. To ensure a fair comparison, we used the same coding style as Bono et al.'s.

## 8 Related Work

In this section we discuss related work and comparison to Classless Java.

### 8.1 Multiple Inheritance in Object Oriented Languages

Many authors have argued in favour or against multiple inheritance. Multiple inheritance provides expressive power, but it is difficult to model and implement, and can create programs that are hard to reason about. These difficulties include the famous diamond (fork-join) problem [3, 15], conflicting methods, etc. To conciliate the need for expressive power and the need for simplicity, many models have been proposed in over the years, including C++ virtual inheritance, mixins [3], traits [16], and hybrid model such as CZ [13]. They provide novel programming architecture models in the OO paradigm. In terms of restrictions set on these models, C++ virtual inheritance aims at a more general model, mixins added some restrictions on the model, and trait model is the most restricted one (excluding states, instantiation, etc). BRUNO: How about Malayeri and Aldrich's paper? shouldn't that be discussed? YANLIN: Newly added discussion on CZ. please double check.

**C++ Approach.** C++ tries to provide a general solution to multiple inheritance. Virtual inheritance in C++ provides another solution to multiple inheritance (especially the diamond problem by keeping only one copy of the base class) [5], however suffers from object initialization problem as pointed out by Malayeri et al. [13]. It bypasses all constructor calls to virtual superclasses, which would potentially cause serious semantic errors. BRUNO: What happens in our approach for the same case?YANLIN: explained below please double-check. In our approach, **@Obj** annotation processor has the full control of object initialization mechanism, and this mechanism is clean and transparent to users. Moreover, if not satisfied with the default generated `of` method, customized factory methods are also allowed to be defined.

**Mixins.** Mixins are a more restricted model than the C++ approach. Mixins allow to name components that can be applied to various classes as reusable functionality units. However, they suffer from linearisation: the order of mixin application is relevant in often subtle and undesired ways. This hinders their usability and the ability of resolving conflicts: the linearisation (total ordering) of mixin inheritance cannot provide a satisfactory resolution in all cases and restricts the flexibility of mixin composition. To fight those limitations, an algebra of mixin operators is introduced [1], but this raised the complexity of the approach, especially when constructors and fields are considered [22]. Scala traits are in fact more like linearised mixins. Scala avoids the object initialization problem by disallowing constructor parameters, causing no ambiguity in cases such as diamond problem. However this approach has limited expressiveness, and suffers from all the problems of linearised mixin composition. Other languages, such as Python, also use linearized mixins. Java interfaces and default methods does not use linearisation: the semantics of Java **extends** clause in interfaces is unordered and symmetric.

BRUNO: rrevise latter? However, in pure Java, there is no mechanism for creating objects in interfaces. Also, our approach supports proper constructor mechanism.

**CZ.** Malayeri and Aldrich proposed a model CZ [13] which aims to do multiple inheritance without diamond problem. They divide inheritance into two separate concepts: inheritance dependency (using `require`) and implementation inheritance (using `extends`). Using a combination of `requires` and `extends`, a program with diamond inheritance can be transformed to one without diamonds. Also in this approach, fields and multiple inheritance can coexist. However as shown in the example in the article, the untangling of inheritance also untangles class structure. In CZ, not only the number of classes (from 4 to 6 in the stream example), but also the class hirarchy complexity increases. Our approach does not complicate the structure, and (conceptual) fields coexist with multiple inheritance.

**Traits and Java's default methods** Simplifying the mixins approach, traits [16] draw a strong line between units of reuse (traits) and object factories (classes). In this model, traits as units of reusable code, contain only methods as reusable entities. Thus, no state and state initialization are considered. Classes act as object factories, requiring functionalities from multiple traits. Java 8 interfaces with default methods are closely related to traits: concrete method implementation are allowed (via the **default** keyword) inside interfaces. The introduction of default methods opens the gate for various flavours of multiple inheritance in Java, using interfaces. Traits offer a trait algebra with operations like sum, alias and exclusion, provided for explicit conflict resolution. Former work by Bono et al. [2]. provides details on mimicking the trait algebra through Java 8 interfaces. We briefly recall the main points of their encoding; however we propose a different representation of **exclusion**. The author of [2] agree that our revised version is cleaner, typesafe and more direct.

- **Symmetric sum** can be obtained by simple multiple inheritance between interfaces.

  ```
  interface A { int x(); } interface B { int y(); } interface C extends A, B {}
  ```

- **Overriding** a conflict is obtained by specifying which super interface take precedence.

  ```
  interface A { default int m() {return 1;} }
  interface B { default int m() {return 2;} }
  interface C extends A, B { default int m() {return B.super.m();} }
  ```

- **Alias** is creating a new method delegating to the existing super interface.

```
interface A { default int m() {return 1;} }
interface B extends A { default int k() {return A.super.m();} }
```

- **Exclusion**: exclusion is also supported in Java, where method declarations can hide the default methods correspondingly in the super interfaces.

```
interface A { default int m() {return 1;} }
interface B extends A { int m(); }
```

There are also proposals for extending Java (before Java8) with traits. For example, FeatherTrait Java (FTJ) [12] by Liquori et al. extends the calculus of Featherweight Java (FJ) [11] with statically-typed traits, adding trait-based inheritance in Java. Except for few, mostly syntactic details, their work can be emulated/rephrased with Java8 interfaces. There are also extensions to the original trait model, which have operations that default methods and interfaces cannot model, such as method renaming (as in [Reppy2006]), which breaks structural subtyping.

**Traits vs Object Interfaces** We consider object interfaces to be an alternative to traits or mixins. In the trait model two concepts (traits and classes) coexist and cooperate. Some authors [**?**]YANLIN: There are many trait related papers of ferruccio damiani, please check whether citing this one is ok. see this as good language design fostering good software development by helping programmers to think about the structure of their programs. However, other authors see the need of two concepts and the absence of state as drawbacks of this model [13]. Object interfaces are units of reuse, and at the same time they provide factory methods for instantiation and support state. Our approach promotes the use of interfaces instead of classes, in order to rely on the modular composition offered by interfaces. Since Java was designed for classes, a direct class-less programming style is verbose and feels unnatural. However, annotation driven code generation is enough to overcome this difficulty, and the resulting programming style encourages modularity, composability and reusability, by keeping a strong object oriented feel. In that sense, we promote object interfaces as being both units of reusable code and object factories. Our practical experience is that, in Java, separating the two notions leads to alot of boilerplate code, and is quite limiting when multiple inheritance with state is required. The use of abstract state operations avoids the key difficulties associated with multiple inheritance and state, while still being quite expressive. Moreover the ability to support constructors adds additional expressivity, which is not available in approaches such as Scala's traits/mixins.

## 8.2 ThisType/MyType/Extensibility

In certain situations, object interfaces allow automatic refinement for *return types*. This is part of a bigger topic in class based languages: expressing and preserving type recursion and (nominal/structural) subtyping at the same time.

One famous attempt in this direction is provided by *MyType* [4], representing the type of **this**, changing its meaning along with inheritance. However when invoking a method with MyType in a parameter position, the exact type of the receiver must be known. This is a big limitation in class based object oriented programming, and is exasperated by the interface-based programming we propose: no type is ever going to be exact since classes are not explicitly used. A recent article [14] lights up this topic, proposing two new features: exact statements and nonheritable methods. Both are related to our work: 1) any method generated inside of the `of` method is indeed non-inheritable since there is no class name to extend from; 2)exact statements (a form of wild-card capture on the exact run-time type) could capture the "exact type" of an object even in a class-less environment. Admittedly, MyType greatly enhances the expressiveness and extensibility of object-oriented programming languages. We are using covariant-return types to simulate some uses of MyType. But our approach only works on method positive positions, whereas MyType is more general, as it works on any positions. Nevertheless our approach is still useful for modeling fluent interfaces and solving expression problems,etc.

## 8.3 Meta-programming Competes with Language Extensions

BRUNO: needs to be polished better, I think BRUNO: Mention/Compare with Language Virtualization work in Scala. They also re-interpret syntax. The most obvious solution to adding features to a language is language extension. It is often implemented as syntactic extensions that can be desugared to the base language. For example, the Scala compiler was extended to directly supports XML syntax. However, this approach does not support combining multiple extensions into one. We are de facto creating a fork in the language, and rarely the new fork gain enough traction to become the main language release. On this topic we mention SugarJ [6] - a Java-based extensible language allowing programmers to extend it with custom features by definitions in meta-DSLs (SDF, Stratego, etc).BRUNO: so, what's the point that we want to make with SugarJ?YANLIN: we were talking about lightweight language sugar, so SugarJ is a comparison to ours. Where using SugarJ requires learning various DSLs like SDF, stratego, etc, in comparison, our approach is just using Java.

On the other side, when the base language has a flexible enough syntax and a fast and powerful enough reflection mechanism, we may just need to play with operator overloading and other language tricks to discover that the language feature we need can be expressed as a simple library in our language. For example, consider SQLAlchemy in python.

Java-like languages tend to sit in the middle of two extremes: libraries can not influence the type system, so many solutions valid in python or other languages are not be applicable, or may be applicable at the cost of losing safety.

Here compile time code generation comes at the rescue: if for a certain feature (**@Obj** in our case) it is possible to use the original language syntax to *express-describe* any specific instantiation of such feature (annotating a class and providing getters), then we can insert in the compilation process a tool that examins and enrichs the code before compilation. No need to modify the original source; for example we can work on temporary files. Java is a particular good candidate for this kind of manipulation since it already provide ways to define and integrate such tools in its own compilation process: in this way there is no need of temporary files, and there is a well defined way of putting multiple extensions together.

Other languages offer even stronger support to safe code manipulation: Template Haskell [18], F# (type providers) [4] and MetaFjig (Active Libraries) [17] all allow to execute code at compile time and to generate code/classes that are transparently integrated in the program that is being generated/processed/compiled. In particular, MetaFjig offers a property called *meta-level-soundness*. In short this property ensures by construction that library code (even if wrong or non nonsensical) would never generate ill-typed code. This is roughly equivalent to what we state and manually proof in Lemma 2 for our particular transformation. Since MetaFjig is not working on annotated classes, there is no direct equivalence on the overall theorem of safety we shown.

[4] http://research.microsoft.com/fsharp/

## 8.4 Formalization of Java8

We provide a simple formalization for a subset of Java including default/static interface methods and object initialization literals (often called anonymous local inner classes). A similar formalization was drafted by Goetz and Field [10] to formalize defender (default) methods in Java. However this formalization is limited to model exactly one method inside classes/interfaces.BRUNO: explain why having multiple methods is significantly more challenging. Our code generation generates several methods that may interacts with each other and methods defined by user. Thus implementing and formalizing multiple methods inside interfaces adds complexity greatly. MARCO: I do not see this as a good explanation, I think we may just remove the 'one method limitation discussion'. Is their work modelling interactions between abstract and default methods? like making a default method abstract again?

# 9 Future work

## 9.1 Qualifiers in Methods

The biggest limitation of our approach is the inability to model qualifiers for class methods (private, protected, synchronized, etc.). For example, the absence of the support for private/protected methods in Java8 interfaces forces all members of interfaces to be public, including static methods. Since we use abstract methods to encode the state, our state is always all public. Still, because the state can only be accessed by methods, it is impossible for the user to know if a certain method maps directly to a field or if it has a default implementation. If the user wants a constructor that does not directly maps to the fields, (as for secondary constructors in Scala) he can simply define its own `of` method and delegate on the generated one, as in

```
@Obj interface Point{ int x(); int y();
  static Point of(int val){return Point.of(val,val);} }
```

However, the generated `of` method would also be present and public. If a future version of Java was to support *static private methods in interfaces* we could extend our code generation to handle also encapsulation. Currently, it is possible to use a public nested class with private static methods inside, but this is ugly and cumbersome. One possibility is that the annotation processor takes methods with a `@Private` annotation, and turns it into static private methods of a nested class. In this extension, also the `of` method could be made private following the same pattern.

## 9.2 State initialization

As discussed before, the user can trivially define its own `of` method, and initialize a portion of the state with default values. However, the initialization code would not be reusable, and subinterfaces would have to repeat such initialization code. If a field has no setters, a simple alternative is to just define the "field" as a default method as in

```
@Obj interface Box{ default int val(){return 0;} }
```

If setters are required, a possible extension of our code expansion could recognize a field if the getter is provided and the setter is required, and could generate the following code:

```
interface Box{
   default int val(){return 0;} //provided
```

```
void val(int _val);//provided
static Box of(){return new Box(){//generated
   int val=Box.super.val();
   public int val(){return val;}
   public void val(int _val){val=_val;}
};}}
```

We are unsure about the value of this workaround because of its trickiness. In order to define a state with initialization, users have to defining a method trusting that it will be overridden later with a behavioural change, but such change is visible only after the first time the setter is called. Moreover, this code would cache the result instead of re-computing it every time. This can be very tricky in a non functional setting.

## 9.3 Clone, toString, equals and hashCode

Methods originally defined in Java class `Object`, as `clone` and `toString`, can be supported by our approach, but they need special care. If an interface annotated with **@Obj** asks an implementation for `clone`, `toString`, `equals` or `hashCode` we can easily generate one from the fields.[5] However, if the user wishes to provide his own implementation, since the method is also implemented in `Object`, a conflict arises. The generated code can resolve the conflict inside `of`, by implementing the method and delegating it to the user implementation, thus

```
@Obj interface Point{ int x(); int y();
  default Point clone(){ return Point.of(0,0);}//user defined clone
}
```

Would expand into

```
interface Point{ int x(); int y();
  default Point clone(){ return Point.of(0,0);}//user defined clone
  public static Point of(int _x, int _y){
    return new Point(){...
      public Point clone(){ return Point.super.clone();}
    };}}
```

# 10 Conclusion

Before Java 8, concrete methods and static methods where not allowed to appear in interfaces. Java 8 allows static interface methods and introduces *default methods*, which allow for implementation insides interfaces. This had an important positive consequence that was probably overlooked by the Java design team: the concept of class (in java) is now redundant and unneeded. We define a subset of Java, called ClassLess Java, where programs and (reusable) libraries can be easily defined and used. To avoid for some syntactic boilerplate caused by Java not being originally designed to work without classes, we introduce a new annotation:**@Obj** provide default implementations for various methods (e.g. getters, setters, with-methods) and a mechanism to instantiate objects. **@Obj** annotation helps programmers to write less cumbersome code while coding in ClassLess Java; indeed we think the obtained gain is so high that ClassLess Java with **@Obj** annotation can be less cumbersome than full Java.BRUNO: May need rewriting

[5] In particular, for clone we can do automatic return type refinement as we do for `with-` and fluent setters. Note how this would solve most of the Java ugliness related to `clone` methods.

## References


1. Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(02):91–132, 2002.
2. Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-oriented programming in java 8. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ'14, 2014.
3. Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, 1990.
4. Kim B Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(02):127–206, 1994.
5. Margaret A Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, 1990.
6. Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *OOPSLA'11*, 2011.
7. Martin Fowler. Fluentinterface at `http://martinfowler.com/bliki/FluentInterface.html`, December 2005.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
9. Brian Goetz. Allow default methods to override object's methods. Discussion on the lambda-dev mailing list, 2013. `http://mail.openjdk.java.net/pipermail/lambda-dev/2013-March/008435.html`.
10. Brian Goetz and Robert Field. Featherweight defenders: A formal model for virtual extension methods in java, 2012.
11. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3), May 2001.
12. Luigi Liquori and Arnaud Spiwack. Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 30(2), March 2008.
13. Donna Malayeri and Jonathan Aldrich. *CZ: multiple inheritance without diamonds*. ACM, 2009.
14. Chieri Saito and Atsushi Igarashi. Matching mytype to subtyping. *Science of Computer Programming*, 2013.
15. Markku Sakkinen. Disciplined inheritance. In *ECOOP'89*, 1989.
16. Nathanael Scharli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP'03*, 2003.
17. Marco Servetto and Elena Zucca. Metafjig: a meta-circular composition language for java-like classes. In *ACM Sigplan Notices*, 2010.
18. Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *ACM SIGPLAN Haskell Workshop*, 2002.
19. Philip Wadler. The Expression Problem. Email, November 1998. Discussion on the Java Genericity mailing list.
20. Yanlin Wang and Bruno C. d. S. Oliveira. The expression problem, trivially! In *HKU CS Tech Report*, 2015. `http://www.cs.hku.hk/research/techreps/document/TR-2015-08.pdf`.
21. Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *FOOL'05*, 2005.
22. Elena Zucca, Marco Servetto, and Giovanni Lagorio. Featherweight Jigsaw - A minimal core calculus for modular composition of classes. In *ECOOP'09*, 2009.
23. Reinier Zwitserloot and Roel Spilker. Project lombok. `http://projectlombok.org`.

# A Appendix

## A.1 THEOREM

**THEOREM.** For a given $\mathcal{I}_0 \ldots \mathcal{I}_n$ interface table such that $\forall \mathcal{I} \in \mathcal{I}_0 \ldots \mathcal{I}_n, \mathcal{I}$ OK, if $\mathcal{I}_0$ has **@ObjWeak**, $\mathsf{valid}(I_0)$ and $\mathtt{of} \notin \mathsf{dom}(I_0)$, then in the interface table $[\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n$ $\forall \mathcal{I} \in [\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n\ \mathcal{I}$ OK.

For convenience, assume that

$$\begin{aligned} \mathcal{I}_0 &= \texttt{@ObjWeak}\ \textbf{\texttt{interface}}\ I_0\ \textbf{\texttt{extends}}\ \overline{I}\,\{\ \overline{meth}\ \} \\ [\![\mathcal{I}_0]\!] &= \emptyset\ \textbf{\texttt{interface}}\ I_0\ \textbf{\texttt{extends}}\ \overline{I}\,\{\ \overline{meth}\ meth'\} \end{aligned}$$

## A.2 LEMMA 1 and Proof

**LEMMA 1.** For any expression $e$ under an interface table $\mathcal{I}_0 \ldots \mathcal{I}_n$ with $\Gamma \vdash e \in I$, if $\mathcal{I}_0$ has **@ObjWeak** and $[\![\mathcal{I}_0]\!]$ is well-defined, then under the interface table $[\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n$, $\Gamma \vdash e \in \_ <: I$.

**Proof.** The proof is based on induction. By the grammar shown in Figure 4, there are three cases for an arbitrary expression $e$:

- A variable or a field update. The type preservation for $e$ is ensured by induction.
- A method call (normal, static or super). Such a method won't be "removed" by the translation and is still there, and the types remain unchanged. The only work **@ObjWeak** does is adding a static method `of` to the interface, however, a pre-condition of the translation is $\mathtt{of} \notin \mathsf{dom}(I_0)$, so adding `of` method has no way to affect any method call.
- An object creation. Adding the `of` method doesn't introduce unimplemented methods to an interface, moreover, the static method is not inheritable, hence after translation such an object creation still type checks and has the right type by induction.

◀

## A.3 LEMMA 2 and Proof

**LEMMA 2.** If $\mathcal{I}_0$ has **@ObjWeak** and is OK, then $[\![\mathcal{I}_0]\!]$ OK.

**Proof.** By the rule (T-INTF) in Figure 5, we divide the proof into two parts.
**Part I.** For each default or static method in the domain of $[\![\mathcal{I}_0]\!]$, the type of the return value is compatible with the method's return type.

Since $\mathcal{I}_0$ OK, and by **LEMMA 1**, all the existing default and static methods are well typed in $[\![\mathcal{I}_0]\!]$, except for the new method `of`. It suffices to prove that it still holds for $\mathsf{ofMethod}(I_0)$.

By the definition of $\mathsf{ofMethod}(I_0)$, the return value is an object

$$\textbf{\texttt{return new}}\ I_0\texttt{()\{...\}}$$

To prove it is of type $I_0$, we use the typing rule (T-OBJ).

- All field initializations are type correct. By the definition of $\mathsf{ofMethod}(I_0)$ in Section **??**, the fields $m_1, \ldots, m_n$ are initialized by `of`'s arguments, and types are compatible.
- All method bodies are well-typed.

- Typing of the $i$-th getter $m_i$.

  $\Gamma, m_i : I_i, \texttt{this} : I_0 \vdash m_i \in I_i$

  We know that $I_i = I^{mh_i}$ since the $i$-th getter has its return type the same as the corresponding field $m_i$.
- Typing of the `with-` method of an arbitrary field $m_i$. By Section **??**, if the `with-` method of $m_i$ is well-defined, it has the form

  $I_0$ `with#`$m_i$`(`$I_i$ `_val){return` $I_0$`.of(`$\overline{e}_i$`);}`

  $\overline{e}_i$ is obtained by replacing $m_i$ with `_val` in the list of fields, and since they have the same type $I_i$, the arguments $\overline{e}_i$ are compatible with $I_0$`.of` method. Hence

  $\Gamma, \overline{m_i : I_i}, \texttt{this} : I_0,\ \texttt{\_val} : I_i \vdash I_0\texttt{.of(}\overline{e}_i\texttt{)} \in I_0$

  We know that $I_0 = I^{mh_i}$ by the return type of `with#`$m_i$ shown as above.
- Typing of the $i$-th setter $\_m_i$. If the $\_m_i$ method is well-defined, it has the form

  $I_0$ $\_m_i$`(`$I_i$ `_val){` $m_i$`= _val;return this;}`

  By (T-UPDATE), the assignment "$m_i$`= _val;`" is correct since $m_i$ and `_val` have the same type $I_i$, and the return type is decided by **`this`**.

  $\Gamma, \texttt{this} : I_0,\ \texttt{\_val} : I_i \vdash \texttt{this} \in I_0$

  We know that $I_0 = I^{mh_i}$ by the return type of $\_m_i$ shown as above.
  HAOYUAN: Can we write $\Gamma, \texttt{this} : I_0,\ \texttt{\_val} : I_i, I^{mh_i} = I_0 \vdash \texttt{this} \in I_0 = I^{mh_i}$?
  HAOYUAN: Can we write `{` $m_i$`= _val;return this;}` $\in I_0$?

- All method headers are valid with respect to the domain of $I_0$. Namely

$$\mathsf{sigvalid}(mh_1 \dots mh_n, I)$$

  - For the $i$-th getter $m_i$,

$$\begin{array}{rl} & I_i\ m_i\texttt{()\{...\}}\ \in \mathsf{ofMethod}(I_0) \\ \text{implies} & I_i\ m_i\texttt{();} \in \mathsf{fields}(I_0) \\ \text{implies} & I_i\ m_i\texttt{();} = \mathsf{mbody}(m_i, I_0) \\ \text{implies} & I_i\ m_i\texttt{();} <: \mathsf{mbody}(m_i, I_0) \end{array}$$

  - For the `with#`$m_i$ method,

$$\begin{array}{rl} & I_0\ \texttt{with\#}m_i\texttt{(}I_i\ \texttt{\_val)\{...\}}\ \in \mathsf{ofMethod}(I_0) \\ \text{implies} & \mathsf{mbody}(\texttt{with\#}m_i, I_0) \text{ is of form } mh; \\ \text{with} & \mathsf{valid}(I_0) \\ \text{implies} & \mathsf{isWith}(\mathsf{mbody}(\texttt{with\#}m_i, I_0), I_0) \\ \text{implies} & I_0\ \texttt{with\#}m_i\texttt{(}I_i\ \texttt{\_val);} <: \mathsf{mbody}(\texttt{with\#}m_i, I_0) \end{array}$$

  - For the $i$-th setter $\_m_i$,

$$\begin{array}{rl} & I_0\ \_m_i\texttt{(}I_i\ \texttt{\_val)\{...\}}\ \in \mathsf{ofMethod}(I_0) \\ \text{implies} & \mathsf{mbody}(\_m_i, I_0) \text{ is of form } mh; \\ \text{with} & \mathsf{valid}(I_0) \\ \text{implies} & \mathsf{isSetter}(\mathsf{mbody}(\_m_i, I_0), I_0) \\ \text{implies} & I_0\ \_m_i\texttt{(}I_i\ \texttt{\_val);} <: \mathsf{mbody}(\_m_i, I_0) \end{array}$$

- All abstract methods in the domain of $I_0$ have been implemented. Namely

$$\mathsf{alldefined}(mh_1 \ldots mh_n, I)$$

Here we simply refer to $\mathsf{valid}(I_0)$, since it guarantees each abstract method to satisfy $\mathsf{isField}$, $\mathsf{isWith}$ or $\mathsf{isSetter}$. But that object includes all implementations for those cases, hence it is of type $I_0$ by (T-OBJ).

**Part II.** Next we check that in $[\![\mathcal{I}_0]\!]$,

$$\mathsf{dom}([\![\mathcal{I}_0]\!]) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth}) \cup \mathsf{dom}(meth')$$

Since $\mathcal{I}_0$ OK, we have $\mathsf{dom}(\mathcal{I}_0) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})$, and hence it is equivalent to prove

$$\mathsf{dom}([\![\mathcal{I}_0]\!]) = \mathsf{dom}(I_0) \cup \mathsf{dom}(meth')$$

This is obvious since a pre-condition of the translation is $\texttt{of} \notin \mathsf{dom}(I_0)$, so $meth'$ doesn't overlap with $\mathsf{dom}(I_0)$. The definition of $\mathsf{dom}$ is based on $\mathsf{mbody}$, and here the new domain $\mathsf{dom}([\![\mathcal{I}_0]\!])$ is only an extension to $\mathsf{dom}(I_0)$ with the `of` method, namely $meth'$. Also note that after translation, there are still no methods with conflicted names, since the `of` method was previously not in the domain, hence $[\![\mathcal{I}_0]\!]$ is well-formed, which finishes our proof. ◀

## A.4 Proof of THEOREM

**Proof.** **LEMMA 2** already proves that $[\![\mathcal{I}_0]\!]$ is OK. On the other hand, for any $\mathcal{I} \neq \mathcal{I}_0$, by **LEMMA 1**, we know that all its methods are still well-typed, and the generated code in translation of **@ObjWeak** is only a static method `of`, which has no way to affect the domain of $\mathcal{I}$, so after translation rule (T-INTF) can still be applied, which finishes our proof. ◀