Jam

A Smooth Extension of Java with Mixins

D. Ancona, G. Lagorio, and E. Zucca

Dipartimento di Informatica e Scienze dell'Informazione
University of Genova

Home page:

`http://www.disi.unige.it/person/LagorioG/jam`

What is a mixin?

The notion originated in the late 80's in the OOP community (CLOS and Flavors).

> A *mixin* is a subclass parametric in the superclass

What is our goal?

We want to study mixins in the context of a real-world *strongly typed* language.

In particular, we have extended Java.

The problem

Suppose to have a class:

`class H1 extends P1` { *decs* }

... and the need to extend another class `P2` in the same way. In plain Java we have to duplicate *decs*:

`class H2 extends P2` { *decs* }

Drawbacks:

- code duplication
- inability to use `H1` and `H2` in a uniform way (*decs* is *not* a type)

Mixins enhance software reuse

How is possible to define `H1` and `H2` without duplicating *decs*?

`mixin M` { *decs* }

The mixin `M` is not associated with a fixed superclass.

```
class H1 = M extends P1 ;
class H2 = M extends P2 ;
```

The classes `H1` and `H2` are defined instantiating `M` on `P1` and `P2`, respectively.

A mixin in Jam

```
mixin Undo {
⇒   inherited String getText() ;
⇒   inherited void setText(String s) ;
    String lastText ;
    void setText(String s) {
        lastText = getText() ;
        super.setText(s) ;
    }
    void undo() {
        setText(lastText) ;
    }
}
```

Being in a strongly-typed context we have to "describe" the generic parent class in order to be able to type-check the mixin alone.

Mixin instantiation

The mixin `Undo` can be correctly instantiated only on classes implementing the two methods

`String getText()` and `void setText(String s)`

```
class Textbox extends Component {
    ...
    String getText() { ... }
    void setText(String s) { ... }
}
```

```
⇒  class TextboxWithUndo =
⇒                Undo extends Textbox {}
```

Mixins are types

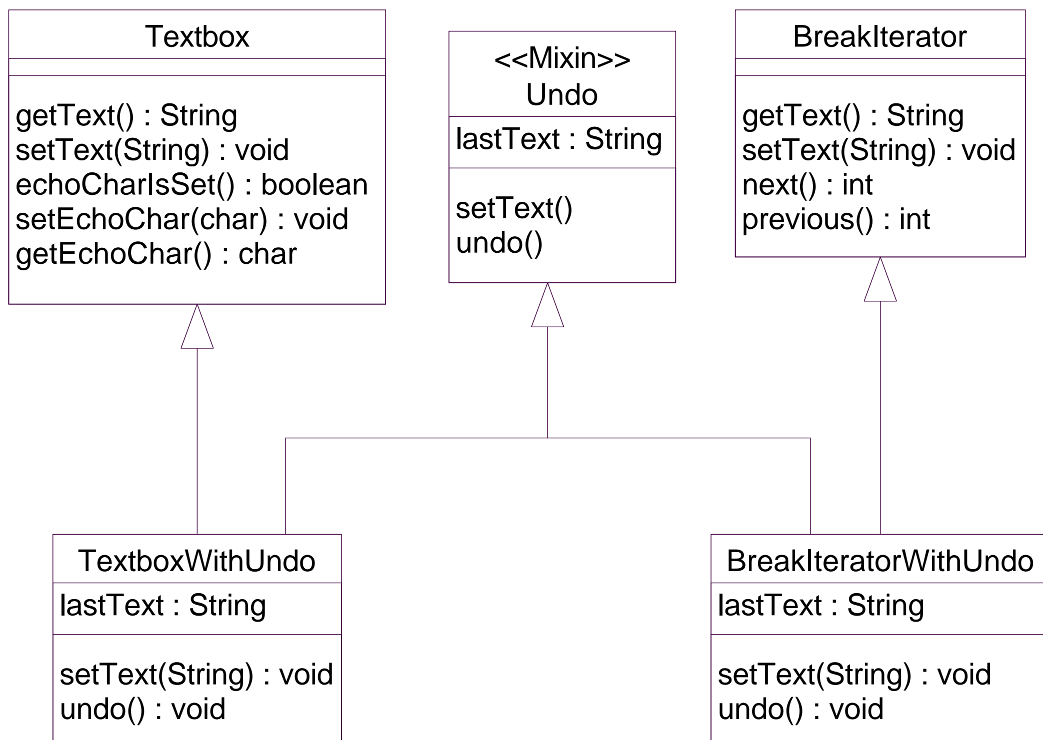
Analogously to classes/interfaces, the declaration of a mixin `M` introduces a new *type* `M`.

However, like abstract classes/interfaces, *object creation* is forbidden for mixins.

```
void g(Undo u) {
    u.setText("foo") ;
    u.setText("bar") ;
    System.out.println(u.lastText) ;
    System.out.println(u.getText()) ;
}
```

Mixin types and multiple inheritance

Mixin types allow to recover some of the expressive power of multiple inheritance avoiding its complexity.



Design guidelines

In defining a type system for mixins in Jam we want:

- new types: mixin types

- a clear, intuitive semantics

- type soundness

Satisfying *all* these requirements is not trivial.

An idea: copy semantics

The semantics of Jam can be expressed by translation into Java.

Copy principle:

```
mixin M { decs }
class H = M extends P ≈
class H extends P { decs }
```

Copy principle describes only visible behavior, not implementation details.

Type soundness

According to our goals, the following class declaration:

```
class H = M extends P;
```

is statically correct only if:

- `P` implements inherited components in `M`;
- combining components in `P` and `M` we obtain a correct Java class

Type checking at: $\begin{cases} \text{mixin declaration} \\ \text{mixin instantiation} \end{cases}$

Type issues

- instantiation time checks $\begin{cases} \text{Java rules for overriding} \\ \text{Java rules for overloading} \end{cases}$

- a method *defined* in a mixin should be considered to be more specific than an *inherited* one because the Java overloading resolution (extended to mixin types)

- `this` in mixin:
  - type of `this` in a mixin `M` is `M`
  - *but* type of `this` in a mixin instance `H` is `H`
  - $H \leq M$ but problems arise in overloading resolution

Benefits of the approach

- a copy semantics defined in terms of translation is *intuitive* from the point of view of Java programmers;

- *type safeness* for Jam can be easily proved by showing that the translation into Java preserves well-typed expressions;

- a translation into Java leads to a simple prototype *implementation* of Jam, which makes Jam programs executable on any Java Virtual Machine.

Formal results: sketch of the translation

A translation $[\![\ ]\!]$ has been formally defined for a large subset of Jam.

Jam		Java
`mixin M` $\{$ *inh def* $\}$	$\xrightarrow{[\![\ ]\!]}$	`interface Parent$M` $\{$ *inh* $\}$ `interface M extends Parent$M` $\{$ *def* $\}$
`class H = M extends P`	$\xrightarrow{[\![\ ]\!]}$	`class H extends P implements M` $\{$ *def* $\}$

`Parent$M` corresponds to the *inherited* part, `M` to the *mixin type*.
Note that, as required, `H`$\leq$`P` and `H`$\leq$`M`.

`Parent$M` is needed for correctly dealing with overloading resolution.

An example: the translation of a mixin declaration

```
interface Parent$Undo {
    String getText() ;
    void setText(String s) ;
}
interface Undo extends Parent$Undo {
    String Undo_$get$_lastText() ;
    String Undo_$set$_lastText(String newValue) ;
    void setText(String s) ;
    void undo() ;
}
```

Formal results: soundness of the type system

The *type system* defined for Jam is inspired to that defined for Java by Drossopoulou and Eisenbach.

That is, we have split the program in an environment and a set of body declarations:

$$\text{Program} \begin{cases} \text{Environment } \Gamma \\ \\ \text{Body declarations} \\ BD_1, \ldots, BD_n \end{cases}$$

The translation $[\![\ ]\!]$ has been extended in order to map Jam type judgments into Java type judgments.

Examples of judgments

Java Judgments:

$\Gamma \vdash \diamond$

$\Gamma \vdash \{BD_1, \ldots, BD_n\}\diamond_{\textsf{Body-Decls}}$

$\Gamma \vdash C\ \textsf{is}_c\ K\ KST\ FST\ MST$

$\Gamma \vdash C \leq_c C'$

$\Gamma \vdash T \leq T'$

$\Pi, \Gamma, T, K \vdash E : \langle T, ET \rangle$

$\ldots$

Jam Judgments:

$\Gamma \vdash M\ \textsf{is}_m\ FST\ MST\ \texttt{inherited}\ FST'\ MST'$

$\Gamma \vdash C \lhd_m M$

$\ldots$

Proof of soundness

Theorem: the translation preserves the static semantics (every correct Jam program is translated into a correct Java program)

$$\Gamma \vdash_{\mathsf{Jam}} \diamond \Rightarrow [\![\Gamma]\!] \vdash_{\mathsf{Java}} \diamond$$

$$\Gamma \vdash_{\mathsf{Jam}} \{BD_1, \ldots, BD_n\} \diamond_{\mathsf{Body\text{-}Decls}} \Rightarrow$$
$$[\![\Gamma]\!] \vdash_{\mathsf{Java}} \{ [\![BD_1]\!]_{\Gamma, \mathit{MBD}}, \cdots$$
$$[\![BD_n]\!]_{\Gamma, \mathit{MBD}} \} \diamond_{\mathsf{Body\text{-}decls}}$$

where: $\mathit{MBD} = \mathit{MixinBodyDeclarations}(BD_1, \ldots, BD_n)$

. . . inductively for each judgment . . .

Current implementation

A Jam to Java translator (called `jamc`) has been implemented in Java.

$\texttt{foo.jam} \xrightarrow{\texttt{jamc}} \texttt{foo.java} \xrightarrow{\texttt{javac}} \texttt{foo.class}$

The translator is driven by the formal translation $[\![\ ]\!]$ and makes Jam an upward-compatible extension of Java 1.0 (except for two new keywords).

Type-checking of Jam input files is not yet fully implemented, therefore most static errors are caught later by the Java compiler.

The source files for `jamc` are available at the following address:
`http://www.disi.unige.it/person/LagorioG/jam`

Related Work

Smalltalk with mixins (Bracha and Griswold, '96):

- implemented, but no types!

Java with mixins (Flatt, Krishnamurthi, and Felleisen, '98):

- only a theoretical toy language not implemented;
- semantics is not "by copy";
- composition of mixins supported.

Further development

Composition of mixins:

Jam does not allow `mixin M3 = M2 extends M1`.

`M3 extends P` should be equivalent to `M2 extends (M1 extends P)`.

Flexible matching:

```
mixin M {
    inherited void f(H h, H h) ;
}
class C {
    void f(P p, H h) {}
}
```

In Jam `M` cannot be instantiated on `C`, since it uses *exact matching* for inherited components. *Flexible matching* would allow contravariance on argument types and covariance on return types.

Jam

mixin

Overriding and mixin instantiation

When instantiating a mixin Java rules for overriding have to be checked:

```
class P {int m(){return 1 ;} }

mixin M {void m(){} }

class H = M extends P ; // not correct!
```

According to the copy semantics, the Jam class `H` would be translated into a not correct Java class!

This error can be caught only when instantiating `M`.

The same considerations apply for the `throws` clause.

Overloading and mixin instantiation

When instantiating a mixin, Java rules for resolving overloading have to be checked:

```
class P {
    void m(String s){}
    void m(Integer i){}
}
mixin M { ...  m(null); ...  }
class H = M extends P ; // ambiguous!
```

According to the copy semantics, the Jam class `H` would be translated into a not correct Java class!

Possible solution: forbid instantiation on classes having methods with same name and number of parameters and signatures which differs only for some reference types.

Overloading with mixin types

When invoking a method on an object having a mixin type, overloading resolution can be tricky.

```
class A {}
class B extends A {}
class Parent {
    void f(B b) {} }
class Heir extends Parent {
    void f(A a) {} }
mixin M {
    inherited void f(B b) ;
    void f(A a) {}
}
class Test {
    void test(Heir h, B b, M m) {
        h.f(b) ; // ambiguous!
        m.f(b) ; // ambiguous?
    }
}
```

Use of `this` in mixins

The object `this` has type `M` in `M` and type `H`≤`M` in `H`!

```
class A {
    static int f(M m) { ...}
    static boolean f(H h) { ...}
}
mixin M {
    void g() {
        int i = A.f(this) ;
    }
} class H = M extends Object {}

// translation of H
class H {
    void g() {
        int i = A.f(this) ; // boom!
    }
}
```

Quite drastic solution: forbid `this` in method and constructor invocations inside mixins.

Mixins vs parametric types

- Orthogonal extensions

- Disadvantages of the mixin-based approach: no way to refer in `M` to the generic:
  - heir `H`
  - parent `P`

On the other hand, you cannot simulate mixins with current extensions based on parametric types:

```
class <T> Mixin extends T { bla bla ...
```

is not allowed.