

# CSE-505: Programming Languages

## Lecture 25 — Multiple Inheritance and Interfaces

Zach Tatlock  
2016

### Diamond Issues

If  $C$  extends  $C1$  and  $C2$  and  $C1, C2$  have a common superclass  $D$  (perhaps transitively), our class hierarchy has a diamond

- ▶ If  $D$  has a field  $f$ , should  $C$  have one field  $f$  or two?
- ▶ If  $D$  has a method  $m$ ,  $C1$  and  $C2$  will have a clash
- ▶ If subsumption is coercive (changing method-lookup), how we subsume from  $C$  to  $D$  affects run-time behavior (incoherent)

Diamonds are common, largely because of types like `Object` with methods like `equals`

## Multiple Inheritance

Why not allow `class C extends C1, C2, ... { ... }`  
(and  $C \leq C1$  and  $C \leq C2$ )?

What everyone agrees: C++ has it and Java doesn't

All we'll do: Understand some basic problems it introduces and how interfaces get most of the benefits and some of the problems

Problem sources:

- ▶ Class hierarchy is a dag, not a tree (not true with interfaces)
- ▶ Subtype hierarchy is a dag, not a tree (true with interfaces)

### Multiple Inheritance, Method-Name Clash

If  $C$  extends  $C1$  and  $C2$ , which both define a method  $m$ , what does  $C$  mean?

Possibilities:

1. Reject declaration of  $C$  (Too restrictive with diamonds)
2. Require  $C$  to override  $m$  (Possibly with *directed resends*)
3. "Left-side" ( $C1$ ) wins (Must decide if upcast to "right-side" ( $C2$ ) coerces to use  $C2$ 's  $m$  or not)
4.  $C$  gets both methods (Now upcasts definitely coercive and with diamonds we lose coherence)
5. Other?

## Implementation Issues

This isn't an implementation course, but many semantic issues regarding multiple inheritance have been heavily influenced by clever implementations

- ▶ In particular, accessing members of `self` via compile-time offsets...
- ▶ ... which won't work with multiple inheritance unless upcasts "adjust" the `self` pointer

That's one reason C++ has different kinds of casts

Better to think semantically first (how should subsumption affect the behavior of method-lookup) and implementation-wise second (what can I optimize based on the class/type hierarchy)

## Digression: Casts

A "cast" can mean many things (cf. C++).

At the language level:

- ▶ upcast: no run-time effect until we get to static overloading
- ▶ downcast: run-time failure or no-effect
- ▶ conversion: key question is round-tripping
- ▶ "reinterpret bits": not well-defined

At the implementation level:

- ▶ upcast: usually no run-time effect but see last slide
- ▶ downcast: usually only run-time effect is failure, but...
- ▶ conversion: same as at language level
- ▶ "reinterpret bits": no effect (by definition)

## Least Supertypes

Consider if  $e_1$  then  $e_2$  else  $e_3$  (or in C++/Java,  $e_1 ? e_2 : e_3$ )

- ▶ We know  $e_2$  and  $e_3$  must have the same type

With subtyping, they just need a common supertype

- ▶ Should pick the least (most-specific) type
- ▶ Single inheritance: the closest common ancestor in the class-hierarchy tree
- ▶ Multiple inheritance: there may be no least common supertype

Example:  $C1$  extends  $D1, D2$  and  $C2$  extends  $D1, D2$

Solutions: Reject (i.e., programmer must insert explicit casts to pick a common supertype)

## Multiple Inheritance Summary

- ▶ Method clashes (what does inheriting  $m$  mean)
- ▶ Diamond issues (coherence issues, shared (?) fields)
- ▶ Implementation issues (slower method-lookup)
- ▶ Least supertypes (may be ambiguous)

Complicated constructs lead to difficult language design

- ▶ Doesn't necessarily mean they are bad ideas

Now discuss *interfaces* and see how (and how not) multiple interfaces are simpler than multiple inheritance...

## Interfaces

An interface is *just a (named) (object) type*. Example:

```
interface I { Int get_x(); Bool compare(I); }
```

A class can *implement* an interface. Example:

```
class C implements I {  
  Int x;  
  Int get_x() {x}  
  Bool compare(I i) {...} // note argument type  
}
```

If  $C$  implements  $I$ , then  $C \leq I$

Requiring *explicit* “implements” hinders extensibility, but simplifies type-checking (a little)

Basically,  $C$  implements  $I$  if  $C$  could extend a class with all *abstract* methods from  $I$

## Interfaces, continued

Subinterfaces (interface  $J$  extends  $I$  { ...}) work exactly as subtyping suggests they should

An unnecessary addition to a language with abstract classes and multiple inheritance, but what about single inheritance and multiple interfaces:

```
class C extends D implements I1,I2,...,In
```

- ▶ Method clashes (no problem, inherit from  $D$ )
- ▶ Diamond issues (no problem, no implementation diamond)
- ▶ Implementation issues (still a “problem”, different object of type  $I$  will have different layouts)
- ▶ Least supertypes (still a problem, this *is* a typing issue)

## Using Interfaces

Although it requires more keystrokes and makes efficient implementation harder, it may make sense (be more extensible) to:

- ▶ Use interface types for all fields and variables
- ▶ Don't use constructors directly: For class  $C$  implementing  $I$ , write:

```
I makeI(...) { new C(...) }
```

This is related to “factory patterns”; constructors are behind a level of indirection

It is using named object-types instead of class-based types