CSE 341: Programming Languages

Dan Grossman
Winter 2008
Lecture 22—Multiple Inheritance, Interfaces, Mixins
Today

Have seen OO’s essence: inheritance, overriding, dynamic-dispatch.

What if we want these things from more than “exactly 1 superclass”?

- **Multiple inheritance**: allow \( > 1 \) superclasses
  - Useful but has some problems (see C++)
- **Java-style interfaces**: allow \( > 1 \) types
  - “Irrelevant” in a dynamically typed language, but fewer problems
- **Mixins**: allow \( > 1 \) “sources of methods”
  - Close to multiple inheritance; almost as useful with fewer (?) problems
  - In Ruby
Multiple Inheritance

If code reuse via inheritance is so useful, why not allow multiple superclasses?

- Because it causes some semantic awkwardness and implementation awkwardness (we’ll discuss only the former)
- (With static typing, there are some more issues)

Is it useful? Sure: A simple example is “3DColorPoint” assuming we already have “3DPoint” and “ColorPoint”.

Naive view: Subclass has all fields and methods of all superclasses
Trees, dags, and diamonds

The “class hierarchy” is a (conceptual) graph with edges from subclasses to superclasses.

Ambiguous phrase: subclass, let’s use immediate-subclass or transitive-subclass when we need to be clear.

• With single inheritance, the class hierarchy is a tree.

• With multiple inheritance, the class hierarchy is a dag.
  – Semantic problems arise from diamonds: Multiple ways to show that class A is a transitive-subclass of some class B.
  – If all classes are transitive-subclasses of something like Object, then multiple inheritance always leads to diamonds.
Multiple Inheritance Semantic Problems

What if multiple superclasses define the same message \( m \) or field \( f \)?

- Classic example: Artists, Cowboys, and ArtistCowboys

Options for \( m \):

- Reject subclass—too restrictive (especially due to diamonds)
- “Left-most superclass wins” (leads to silent weirdness and really want per-method flexiblity)
- Require subclass to override \( m \) (can use directed resends)

Options for \( f \): one copy or two copies?

C++ provides two forms of inheritance:

- One always makes two copies
- One makes one copy if fields were declared by same class
  - Would not work well in Ruby?
Java-style interfaces

(Recall?) in Java, we can define *interfaces* and classes can *implement* them.

- Interface describes methods and their types
  
  ```
  interface Example {
      void m1(int x, int y);
      Foo m2(Example e, String s);
  }
  ```

- Example is a type (can be used for a field, method argument, local variables, etc.)

- If class $C$ implements interface $I$, then instances of $C$ can have type $I$ but $C$ must define everything in $I$ (directly or via inheritance).

- Given an expression of type $I$, it type-checks to send it any message $I$ promises.
Interfaces are a typing thing

In Java, you have 1 immediate-superclass and any number of interfaces you implement.

Because interfaces provide no methods or fields (only types of methods), no duplication problems result!

- No problem if I1 and I2 both “promise” some method m and C implements I1 and I2.

But interfaces do not give us the power we want for making colored 3D points or artist-cowboys.

They’re totally irrelevant in a dynamically typed language like Ruby:

- We are already allowed to send any message to any object
- It is up to us to get it right (“interfaces” more in comments or reflection, e.g., the methods method of Object)
Mixins

A mixin is a collection of methods (no fields, constructors, instances, etc.)

Languages with mixins (e.g., Ruby) typically allow a class to have 1 superclass but any number of mixins.

Bad news: Less powerful than multiple inheritance; have to decide “upfront” what is a class and what is a mixin.

Good news: Clear semantics on methods/fields and works great for certain idioms.
Ruby mixin basics

A *module’s* instance methods are mixed into a class by *including* the module in the class definition.

Method-lookup rules: First class’s methods, then its mixin’s methods (later includes shadow), then immediate-superclass, then immediate-superclass’s mixins, ...

Field rules: It is all one object.
What mixins are good for

We could make Color a mixin and then use it for coloring 2D and 3D points.

- Works fine but often bad style to have mixin methods define fields (could conflict with other fields)

For artist-cowboys, what should the mixin be?

But mixins are extremely elegant for letting classes “get a bunch of methods while defining only a few”.

- All thanks to late-binding!
- Cool examples in Ruby library: Comparable and Enumerable