



CSE 413 Spring 2011

Interfaces, Mixins, & Multiple Inheritance

Based on CSE341, Sp08/Sp11



Overview

- Essence of object-oriented programming: inheritance, overriding, dynamic-dispatch
- Classic inheritance includes specification (types) and implementation (code)
- What about multiple inheritance (>1 superclass)?
 - When does it make sense?
 - What are the issues?



Inheritance Models

- **Single Inheritance: at most 1 superclass**
 - Subclass inherits methods and state from superclass; can override methods, add more methods and instance variables
- **Multiple Inheritance: >1 superclass**
 - Useful – factor different traits/behavior to small classes, then extend several of them
 - But hard to use well (e.g., C++)
 - Typical problem: big, brittle inheritance graph, methods migrate to bloated superclasses over time; becomes (very) hard to make changes

Inheritance Models

- Java-style interfaces: >1 type
 - Doesn't apply to dynamically-typed languages
 - Class “inherits” (has) multiple types, but
 - Only inherits code from one parent class
 - Fewer problems than multiple inheritance
- Mixins: >1 “source of methods”
 - Similarities to multiple inheritance – many of the goodies with fewer(?) problems



Multiple Inheritance

- If single inheritance is so useful, why not allow multiple superclasses?
 - Semantic and implementation complexities
 - Typing issues w/static typing
- Is it useful? Sure:
 - Color3DPoint extends 3DPoint, ColorPoint
- Naïve view: subclass has all fields and methods of all superclasses

Trees, DAGs, and Diamonds

- Class hierarchy forms a graph
 - Edges from subclasses to superclasses
 - Single inheritance: a tree
 - Multiple inheritance: a DAG
- Diamonds
 - With multiple inheritance, may be multiple ways to show that A is a (transitive) subclass of B
 - If all classes are transitive subclasses of e.g. Object, multiple inheritance always leads to diamonds

Multiple Inheritance: Semantic Issues

- What if multiple superclasses define the same message m or field f ?
 - Classic example: Artists, Cowboys, ArtistCowboys
- Options for method m :
 - Reject subclass as ambiguous – but this is too restrictive (esp. w/diamonds)
 - “Left-most superclass wins” – too restrictive (want per-method flexibility) + silent weirdness
 - Require subclass to override m (can use explicitly qualified calls to inherited methods)

Multiple Inheritance: Semantic Issues

- Options for field f : One copy of f or multiple copies?
 - Multiple copies: what you want if Artist::draw and Cowboy::draw use inherited fields differently
 - Single copy: what you want for Color3dPoint x and y coordinates
- C++ provides both kinds of inheritance
 - Either two copies always, or one copy if field declared in same (parent) class

Java-Style Interfaces

- In Java we can define *interfaces* and classes can *implement* them
 - Interface describes methods and types
 - Interface *is* a type – can have variables, parameters, etc. with that type
 - If class C implements interface I, then instances of C have type I but must define everything in I (directly or via inheritance)

Interfaces are all about Types

- In Java, we can have 1 immediate superclass and implement any number of interfaces
- Interfaces provide no methods or fields – no duplication problems
 - If I1 and I2 both include some method *m*, implementing class must provide it somehow
- But this doesn't allow what we want for Color3DPoints or ArtistCowboys
 - No code inheritance/reuse possible



Java Interfaces and Ruby

- Concept is totally irrelevant for Ruby
 - We can already send any message to any object (dynamic typing)
 - We need to get it right (can always ask an object what messages it responds to)

Interfaces vs Abstract Classes

- Interfaces are not needed in C++. Why?
- C++ allows methods and classes to be abstract
 - Specified in class declaration but not provided in implementation (same as Java)
 - Called pure virtual methods in C++
- So a class can extend multiple abstract classes
 - Same as implementing interfaces
- But if that's all you need, you don't need multiple inheritance
 - Multiple inheritance is not just typing

Mixins

- A mixin is a collection of methods
 - No fields, constructors, instances, etc.
- Typically a language with mixins allows 1 superclass and any number of mixins
 - We've seen this in Ruby
- Bad news: less powerful than multiple inheritance (what is in a class, what is in a mixin?)
- Good news: Clear semantics, great for certain idioms (Enumerate, Comparable using each, <=>)