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
  – Why? Factor different traits/behavior into 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?
  – Semantics are often awkward (next few slides)
  – Static type checking is harder (not discussed)
  – Efficient implementation is harder (hints next time)
• Is it useful? Sure:
  – Color3DPoint extends 3DPoint, ColorPoint
  – StudentAthlete extends Student, Athlete
• Naïve view: subclass has all fields and methods of all superclasses; avoids copying code
Trees, DAGs, and Diamonds

- Class hierarchy forms a graph
  - Nodes are classes
  - Edges from subclasses to superclasses
  - Single inheritance: a tree
  - Multiple inheritance: a DAG (but no cycles allowed)
- Diamonds
  - With multiple inheritance, may be multiple ways to show that Y is a (transitive) subclass of X
  - 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: \texttt{Artists, Cowboys, ArtistCowboys}
    • All have a \texttt{draw} method
    • The \texttt{draw} methods access a (the?) \texttt{pocket} instance variable
Multiple Inheritance: Methods

• If V and Z both define method m, which one does Y inherit? What does super mean?
  – Can use directed resends: Z::m
• What if X defines m that Z overrides but V does not?
  – Can do the same thing, but often we want Z to “win” (e.g., ColorPt3D wants Pt3D’s overrides)
Multiple Inheritance: Methods

- Some 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: Fields

- 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 (e.g., both use a `pocket` variable)
  - 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 – program can create 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

- A Java class can have implement any number of interfaces (and also has one superclass – Object if nothing else declared)
- 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)
  - We don’t type-check implementers
Why no interfaces in C++?

- C++ allows methods and classes to be *abstract*
  - Specified in class declaration but with no implementation (same as Java)
  - Called pure virtual methods in C++
- Abstract classes can be extended but not instantiated
- 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 (just) collection of methods
  - Less than a class: no fields, constructors, instances, etc.
  - More than an interface: methods have implementations
- Languages with mixins typically allow one superclass and any number of mixins (e.g., Ruby)
Mixin Semantics

- Including a mixin makes its methods part of the class
  - Mixins extend or override in the order they are included in the class definition
  - More powerful than helper methods because mixin methods can access methods and instance variables not defined in the mixin using self

- Not quite as powerful as multiple inheritance, but...
- Clear semantics, great for certain idioms
  \(\text{Enumerate} \text{ and} \ \text{Comparable} \ \text{using} \ \text{each,} \ \langle=\rangle\)
Next time

• Implementing inheritance, dynamic dispatch

• Then on Friday: wrapup, review, the end.