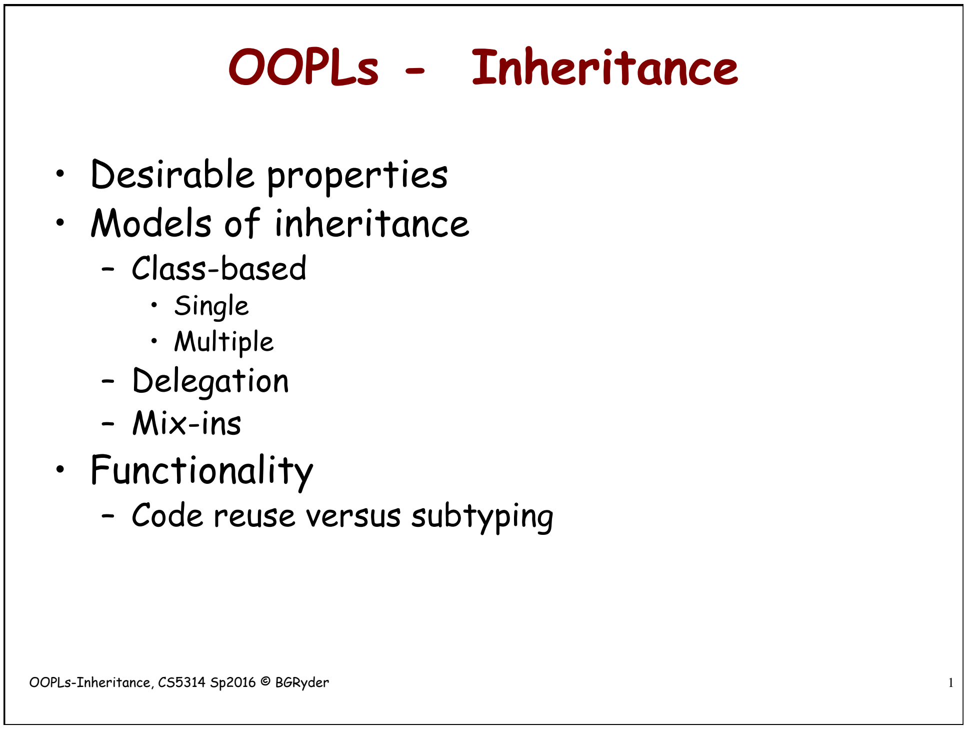
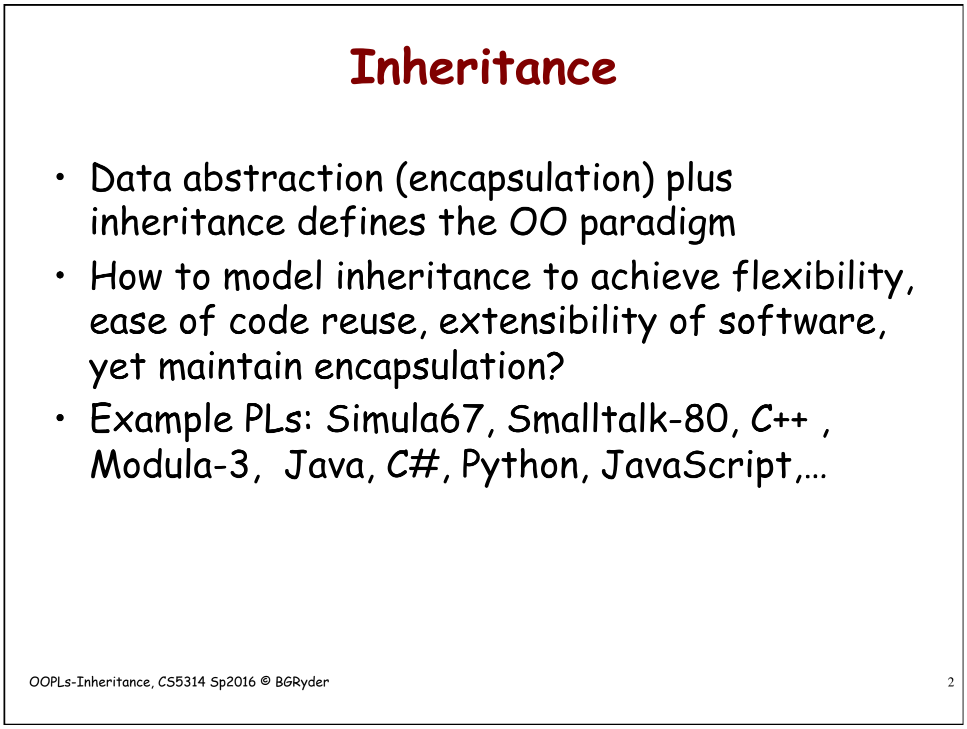


OOPLs - Inheritance

- Desirable properties
- Models of inheritance
  - Class-based
    - Single
    - Multiple
  - Delegation
  - Mix-ins
- Functionality
  - Code reuse versus subtyping



Inheritance

- Data abstraction (encapsulation) plus inheritance defines the OO paradigm
- How to model inheritance to achieve flexibility, ease of code reuse, extensibility of software, yet maintain encapsulation?
- Example PLs: Simula67, Smalltalk-80, C++ , Modula-3, Java, C#, Python, JavaScript,…

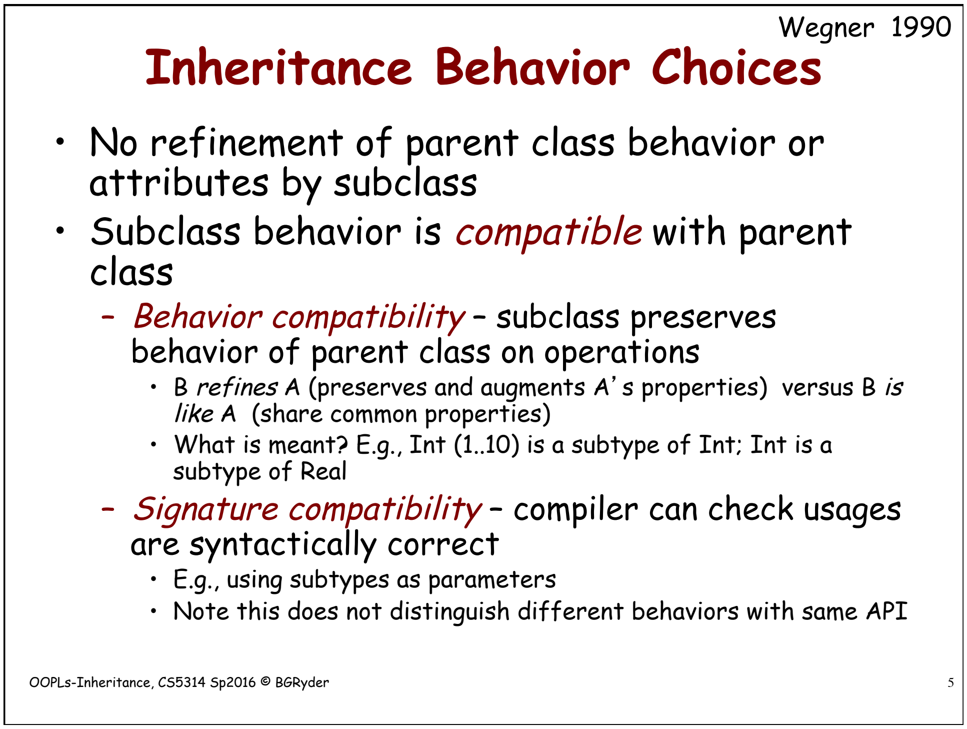
Defining Inheritance - Qs

- Should inheritance be at the level of classes or objects?
- How should multiple inheritance be defined?
- Is inheritance a form of subtyping or just code reuse?
  - *Is-a* inheritance versus efficiency in coding (e.g., interfaces)
- How should modification of inherited properties be constrained?

Inheritance- More Qs

"Concepts and Paradigms of OOP", Peter Wegner, OOPS Messenger, vol 1 no 1 Aug 1990. (expanded from an OOPSLA89 keynote)

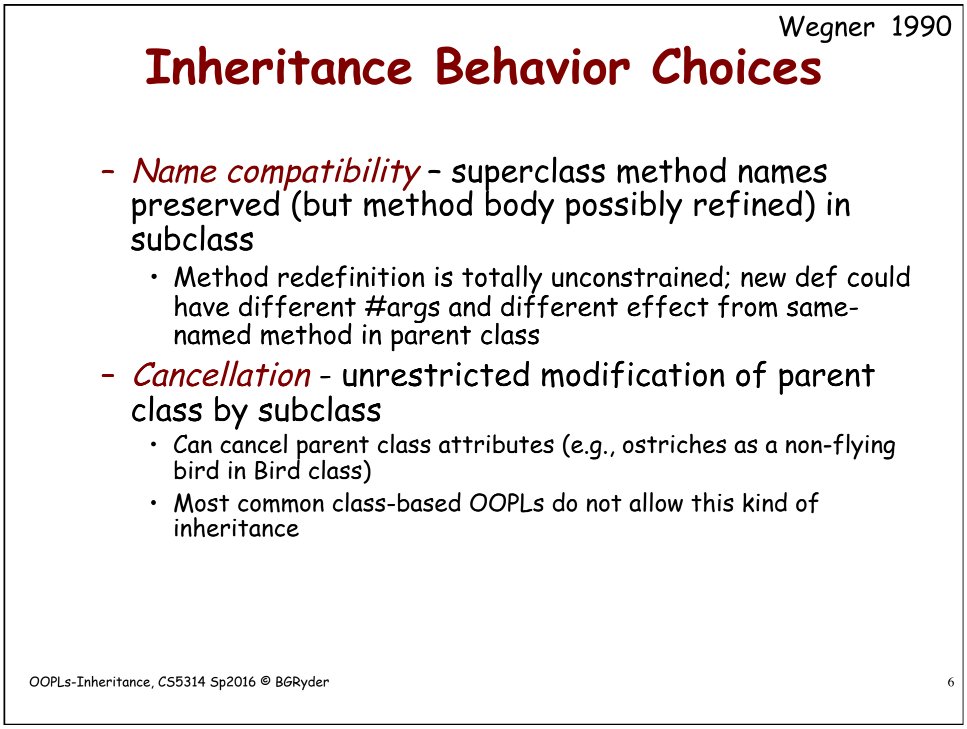
- A mechanism for sharing code and behavior
- Should we modify inherited attributes?
- Do we inherit at the level of classes or instances (i.e., delegation)?
- How is multiple inheritance to be defined and managed?
- What should be inherited? behavior? code? both?



Wegner 1990

Inheritance Behavior Choices

- No refinement of parent class behavior or attributes by subclass
- Subclass behavior is *compatible* with parent class
  - *Behavior compatibility* – subclass preserves behavior of parent class on operations
    - B *refines* A (preserves and augments A's properties) versus B *is like* A (share common properties)
    - What is meant? E.g., Int (1..10) is a subtype of Int; Int is a subtype of Real
  - *Signature compatibility* – compiler can check usages are syntactically correct
    - E.g., using subtypes as parameters
    - Note this does not distinguish different behaviors with same API



Wegner 1990

Inheritance Behavior Choices

- *Name compatibility* – superclass method names preserved (but method body possibly refined) in subclass
  - Method redefinition is totally unconstrained; new def could have different #args and different effect from same-named method in parent class
- *Cancellation* - unrestricted modification of parent class by subclass
  - Can cancel parent class attributes (e.g., ostriches as a non-flying bird in Bird class)
  - Most common class-based OOPLs do not allow this kind of inheritance

Wegner 1990

Granularity of Inheritance

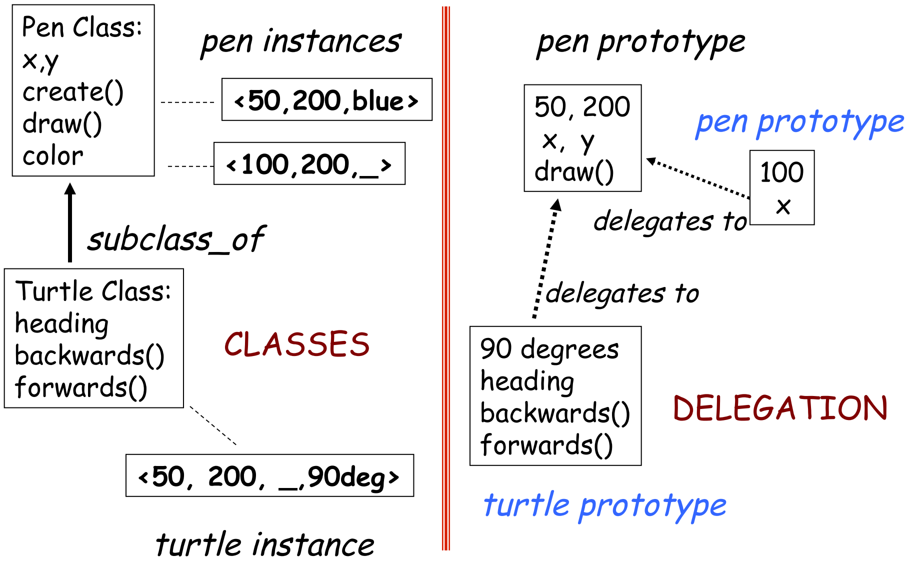
- Class-based inheritance
  - All objects share same attributes and behavior
  - Sometimes have different shared behaviors provided by multiple classes as parent classes
- Delegation – inheritance at object-level
  - Objects delegate nonlocal operations to parent instances called prototypes (e.g., JavaScript)
  - Prototypes are templates and instances themselves and have both sharable properties and methods
  - Useful for types that have only a single instance
  - E.g., SELF PL designed by David Unger in 1980s

Wegner 1990

Inheritance Granularity

- *Class-based* (ST-80, Java, C++. C#)
- *Delegation* - behavior and value sharing at the level of objects
  - Especially good for things that will have only one instance
- Tradeoffs
  - Claim is that delegation may introduce more complexity in executing operations, but may be more storage efficient (e.g., turtle example p 41)
  - Storage is distributed among the prototype objects (i.e., prototype objects can themselves inherit from other prototype objects)

Delegation Example, Liebermann, OOPSLA' 86)



Pen Class:
x,y
create()
draw()
color

pen instances

<50,200,blue>

<100,200,_>

subclass_of

Turtle Class:
heading
backwards()
forwards()

CLASSES

<50, 200, _,90deg>

turtle instance

pen prototype

50, 200
x, y
draw()

pen prototype

100
x

delegates to

delegates to

90 degrees
heading
backwards()
forwards()

DELEGATION

turtle prototype

Desirable Properties for Class-based Inheritance

A. Snyder, "Inheritance and the Development of Encapsulated SW Components", HICSS20, 1987

- Two kinds of users of class attributes and methods: subclasses and external clients
  - Must consider different sorts of sharing/access
  - Only want external clients to see APIs of methods, no rep type, no instance vars to preserve encapsulation and to allow redesign of the class implementation and rep type

Snyder 1987

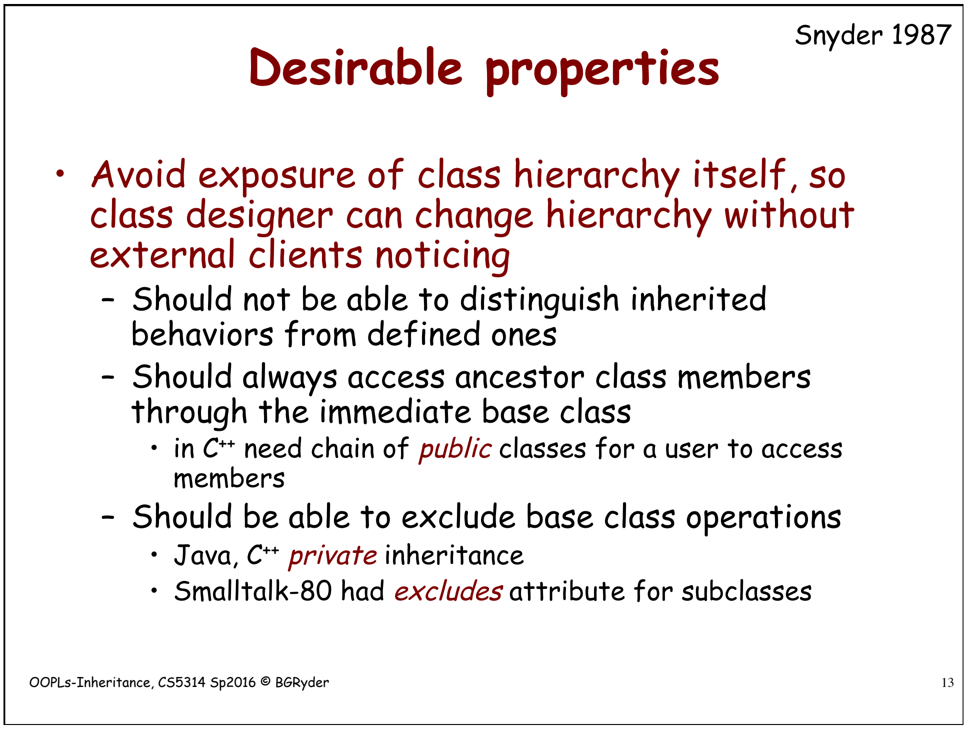
Desirable Properties

- May want to allow descendent classes to have full access to instance variables
  - Problem: Smalltalk-80 & Objective-C allowed full access to instance variables of class by a descendent class
    - But then how allow change to the rep type in ancestor class?
    - Soln: Require descendent class to use ancestor class access operations for inherited state

Snyder 1987

Desirable Properties

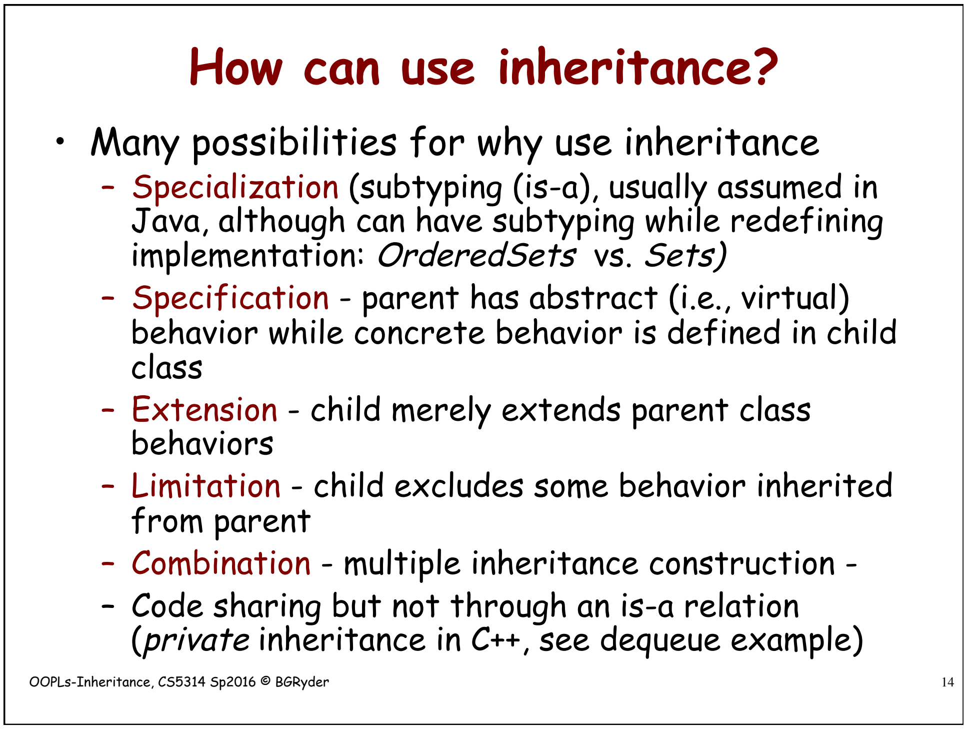
- Should not expose inheritance of members to external clients of a class
  - Smalltalk-80 allowed complete access to members by external clients (and subclasses) compromising encapsulation
  - C^{++}/Java added *protected* access control to instance variables



Snyder 1987

Desirable properties

- Avoid exposure of class hierarchy itself, so class designer can change hierarchy without external clients noticing
  - Should not be able to distinguish inherited behaviors from defined ones
  - Should always access ancestor class members through the immediate base class
    - in C^{++} need chain of *public* classes for a user to access members
  - Should be able to exclude base class operations
    - Java, C^{++} *private* inheritance
    - Smalltalk-80 had *excludes* attribute for subclasses



How can use inheritance?

- Many possibilities for why use inheritance
  - Specialization (subtyping (is-a), usually assumed in Java, although can have subtyping while redefining implementation: *OrderedSets* vs. *Sets)*
  - Specification - parent has abstract (i.e., virtual) behavior while concrete behavior is defined in child class
  - Extension - child merely extends parent class behaviors
  - Limitation - child excludes some behavior inherited from parent
  - Combination - multiple inheritance construction -
  - Code sharing but not through an is-a relation (*private* inheritance in C++, see dequeue example)

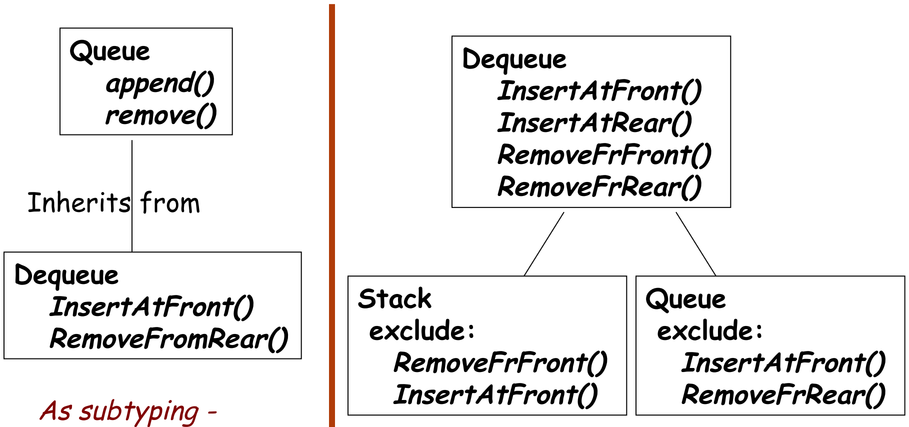
Inheritance

- As subtyping
  - Inheriting implementation and external specification
  - S is subtype of T if all operations on type T objects are meaningful on S objects;
    - Behavioral substitutability
- As code reuse
  - Inheriting only implementation; not necessarily an *is-a* relation
  - Building new components from old
  - E.g., interfaces in Java – common functionality, but not typical class inheritance

Example

Dequeue is subtype of both Stack and Queue but inherits from neither

- Two ways to define *queue* and *dequeue*

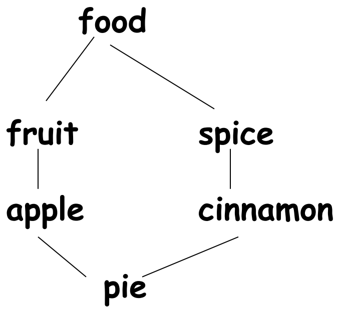


As subtyping - similar behavior with added methods

As code reuse - inheritance with exclusion

Inheritance

- Multiple versus single
  - Real world is multiple inheritance
  - Linearizing lookup
    - Problem: interpretation depends on non-local inheritance structure, not robust in face of changes
  - No problem if no conflicts



food

fruit

spice

apple

cinnamon

pie

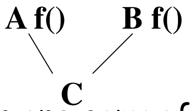
Linearized:pie, apple, fruit, cinnamon, spice, food

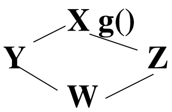
Class-based Inheritance Choices in PLs

- Single inheritance (Smalltalk-80 adapted from Simula) - easier
- Multiple inheritance (C^{++}, Java)
  - Problem: how to avoid inheriting more than one copy of multiply inherited instance variables or member functions from same ancestor through more than one path?
    - Can linearize hierarchy for lookup purposes (Clos, Flavors)
    - Can exclude some inherited members (CommonObjects, C^{++})
    - Can define it away at user option (accomplish multiple inheritance by use virtual base class inheritance in C^{++} ; use interfaces in Java)

Multiple Inheritance Conflict Resolution

- Problems:
  - Member clash
  - Inheriting more than one copy of same member

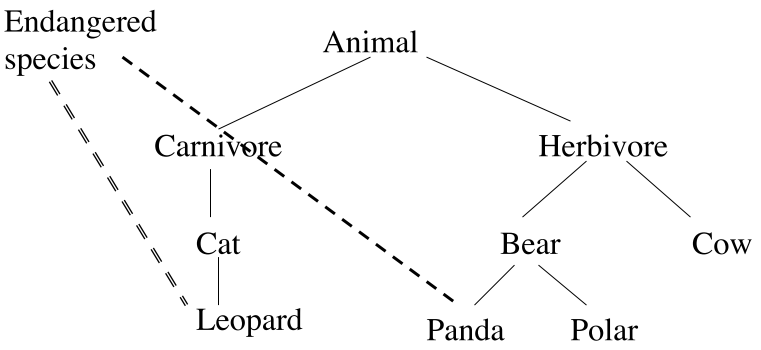




- Approaches
  - Linearize hierarchy so only one parent is "closest" (CLOS, Flavors)
  - Throw an exception when same member is applied more than once due to duplicate paths
  - Exclude some members to avoid problem (C++)

Multiple Inheritance

- Needed to describe certain complex *is-a* relationships (non-overlapping attributes)



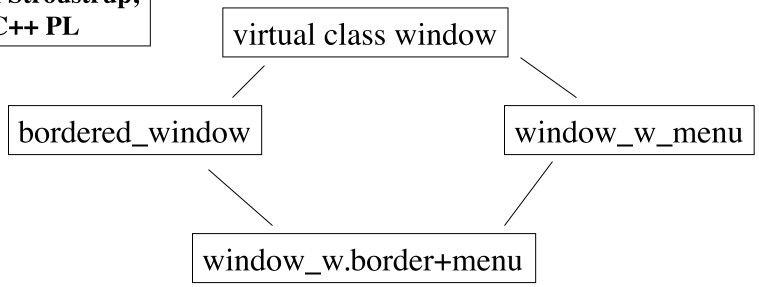
Multiple Inheritance Conflict Resolution

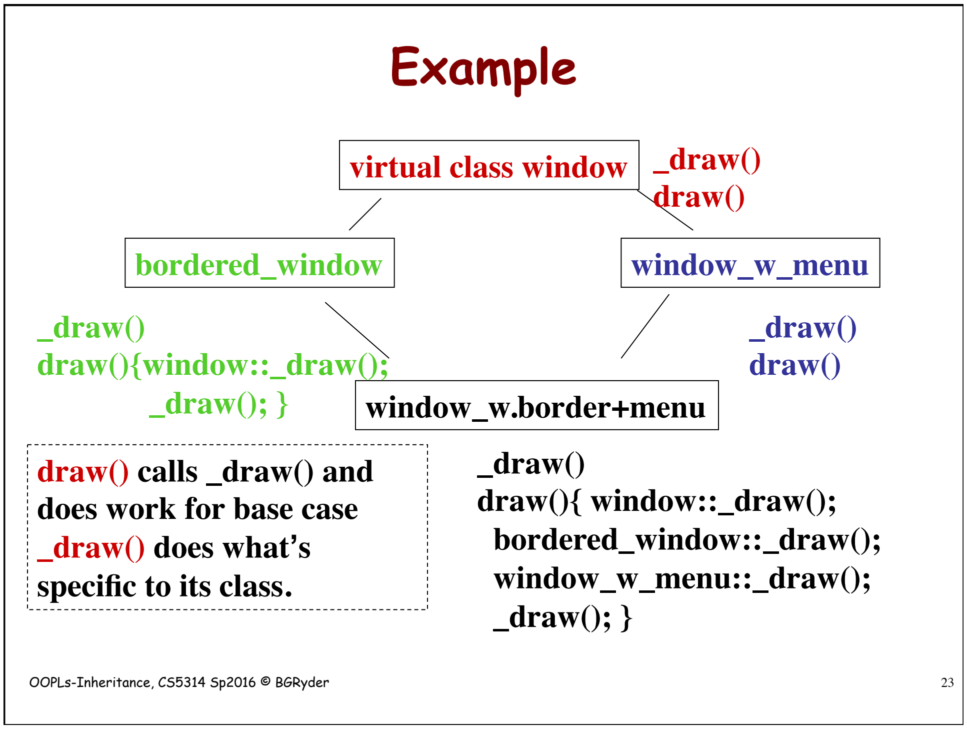
- Actual solutions
  - Disallow multiple inheritance (ST-80)
  - Allow inheritance of indistinguishable components but only one of them (set at defn time) (CLOS, C++)
  - Take approach #2 but pick inherited member at use time (C++, <baseclass>::f())
  - Combine inherited components into one new component (like flattening the hierarchy) (Flavors)

A. Snyder's Mix-in Classes

- Use of disjoint parent classes with desired behaviors
- Reminiscent of Java's interfaces

From Stroustrup, The C++ PL





Example

virtual class window

_draw()
draw()

bordered_window

_draw()
draw(){window::_draw();
_draw(); }

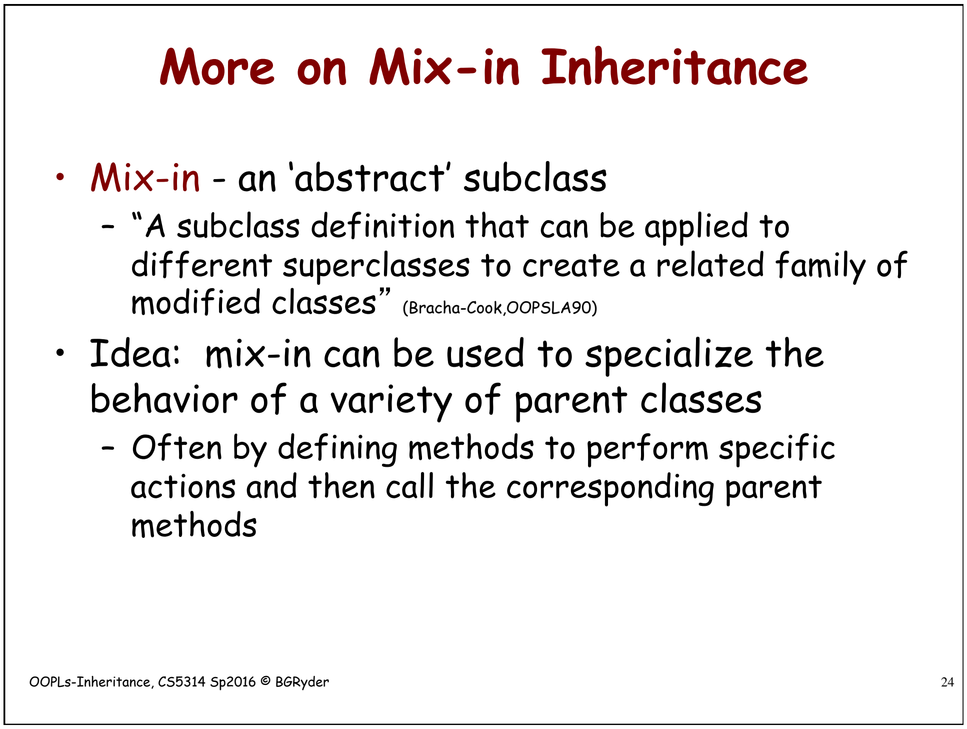
window_w_menu

_draw()
draw()

window_w.border+menu

_draw()
draw(){ window::_draw();
bordered_window::_draw();
window_w_menu::_draw();
_draw(); }

draw() calls _draw() and does work for base case _draw() does what's specific to its class.



More on Mix-in Inheritance

- Mix-in - an 'abstract' subclass
  - "A subclass definition that can be applied to different superclasses to create a related family of modified classes" (Bracha-Cook,OOPSLA90)
- Idea: mix-in can be used to specialize the behavior of a variety of parent classes
  - Often by defining methods to perform specific actions and then call the corresponding parent methods

cf http://csis.pace.edu/~bergin/patterns/multipleinheritance.html

Java Example

Use of child is effectively use of delegation

```
class Parent
{public P(int value) {this.val = value;}
  public int getvalue(){return this.val;}
 public toString() {return “” + this.val;}
 private int val;
}
class Other
{public Other(int value){..}
 public void f(){…}
}
interface OtherInterface
{ void f();}
class OtherChild extends Other implements OtherInterface
{public OtherChild(int value) { super(value);}
}
```

```
class ParentChild extends Parent
implements OtherInterface
{ public ParentChild(..)
      {child = new OtherChild(..);…
}
public void f(){child.f();}
private final OtherInterface child;
```

We have merged the implementations of 2 classes - Parent, Other -- without modifying either one!