OOPLs - Inheritance

- Desirable properties
- Models of inheritance
  - Class-based: with single, multiple parents
  - Delegation
  - Mix-ins
- Functionality
  - as code reuse
  - as subtyping

Inheritance

- Data abstraction plus inheritance defines the OO paradigm
- How to model inheritance to achieve flexibility, ease of code reuse, extensibility (esp. over time) and maintain encapsulation?
- Example PLs: Simula, Smalltalk-80, C++, Modula-3, Java,…

Defining Inheritance - Qs

- **Should inheritance be at the level of classes or objects?**
- **How should multiple inheritance be defined?**
- **Is inheritance subtyping or code reuse?**
  - ***Is-a* inheritance versus efficiency in coding**
- **How should modification of inherited attributes be constrained?**

Inheritance- More Qs

"Concepts and Paradigms of OOP", Peter Wegner, OOPS Messenger, vol 1 no 1 Aug 1990.

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

Modifiability of Inheritance

- *Behavior compatibility* - preserves behavior of parent class
    - B *refines* A (preserves and augments A’s properties) ve.0rsus B *is like* A
    - Int (1..10) is subtype of Int
- *Signature compatibility* - can check usages are syntactically correct
    - E.g., using subtypes as parameters
- *Name compatibility* - superclass operation names preserved (possibly refined) in subclass
- *Cancellation* - unrestricted modification of superclass by subclass
    - Can cancel superclass attributes

Desirable properties

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

- Should not expose inheritance of members to clients of a class
    - Compromises encapsulation; superclass can’t change member definitions easily without affecting subclasses
        - Smalltalk-80 allowed complete access to members by subclasses and users
        - C^{++}/Java added *protected* access control

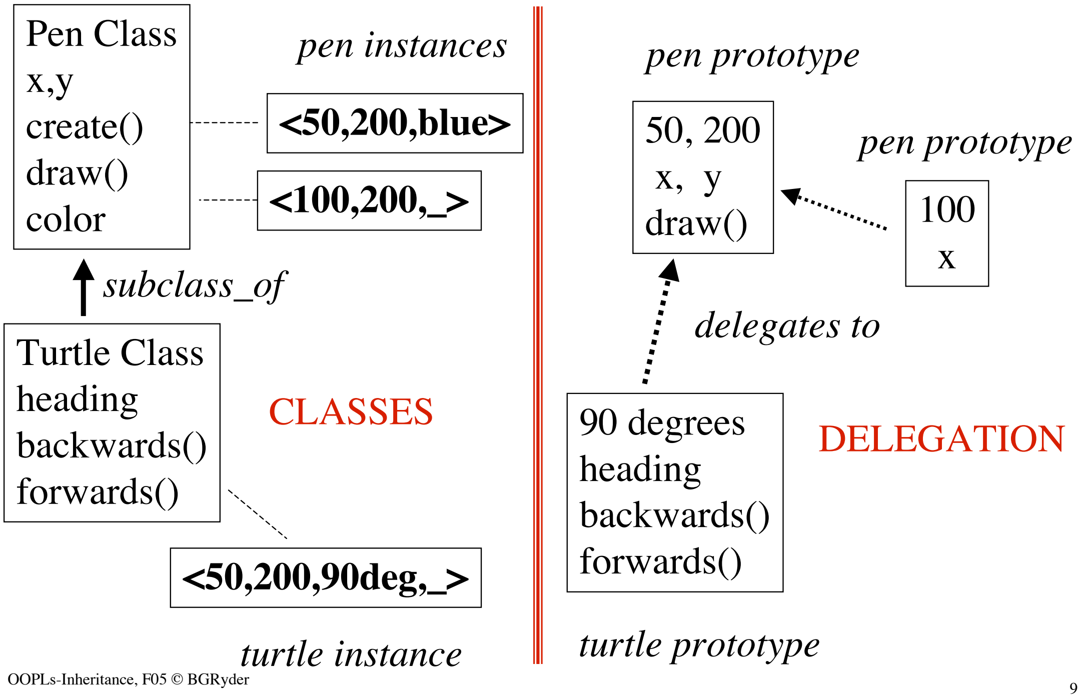
Desirable properties

- Avoid exposure of class hierarchy itself, so class designer can change hierarchy without users 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
    - C^{++} *private* inheritance
    - Smalltalk-80 had *excludes* attribute for subclasses

Inheritance Granularity

- *Class-based* (ST-80, Java, C++)
- *Delegation* - behavior sharing at the level of objects
  - Instances called *prototypes* serve as templates for behavior sharing and cloning of other instances
    - E.g., SELF PL, David Ungar
  - Can share values or operations
  - Exhibit decrease in stored information at cost of greater complexity in executing operations
  - Comparison:
    - Classes use more storage, less complex operations
    - Delegation uses less storage at cost of more complex operations

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

90 degrees
heading
backwards()
forwards()

DELEGATION

turtle prototype

Inheritance Choices in PLs

- **Single (Smalltalk-80) - easier**
- **Multiple (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 (use virtual base class inheritance in C^{++}; use interfaces in Java)**

How can use inheritance?

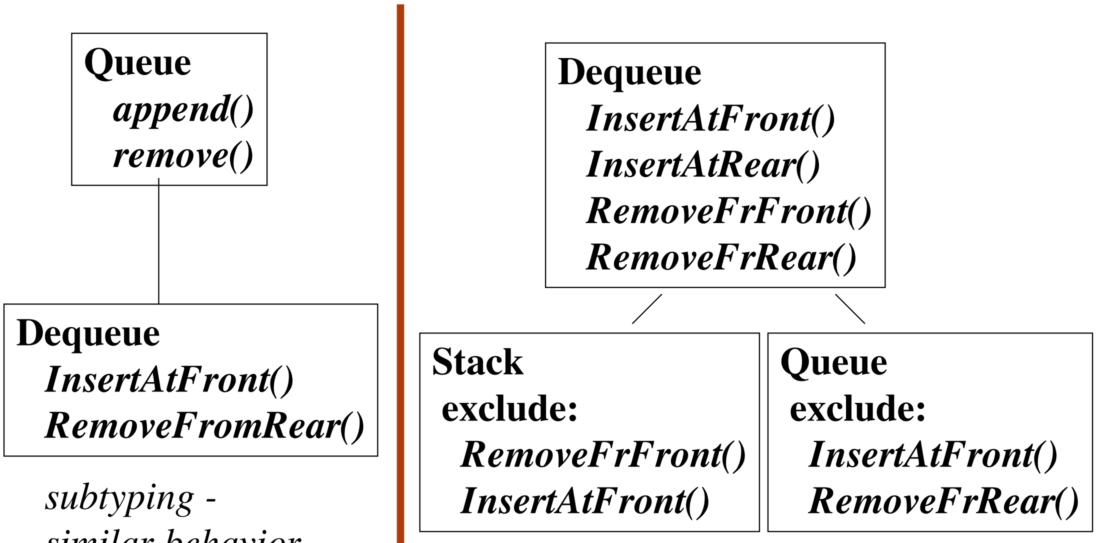
- Many possibilities for why use inheritance
  - Specialization (subtyping, usually assumed in Java, although can have subtyping while redefining implementation: *OrderedSets* vs. *Sets)*
  - Specification (parent has virtual or abstract behavior while concrete behavior is defined in child class)
  - 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

Example

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



Queue
append()
remove()

Dequeue
InsertAtFront()
RemoveFromRear()

subtyping - similar behavior

Dequeue
InsertAtFront()
InsertAtRear()
RemoveFrFront()
RemoveFrRear()

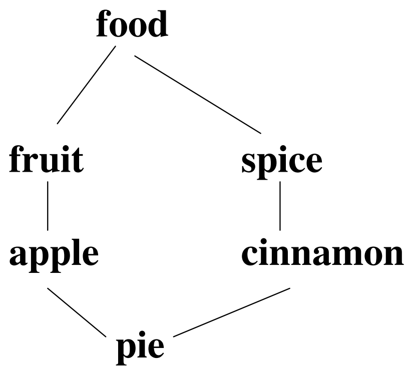
Stack
exclude:
RemoveFrFront()
InsertAtFront()

Queue
exclude:
InsertAtFront()
RemoveFrRear()

as code reuse

Inheritance

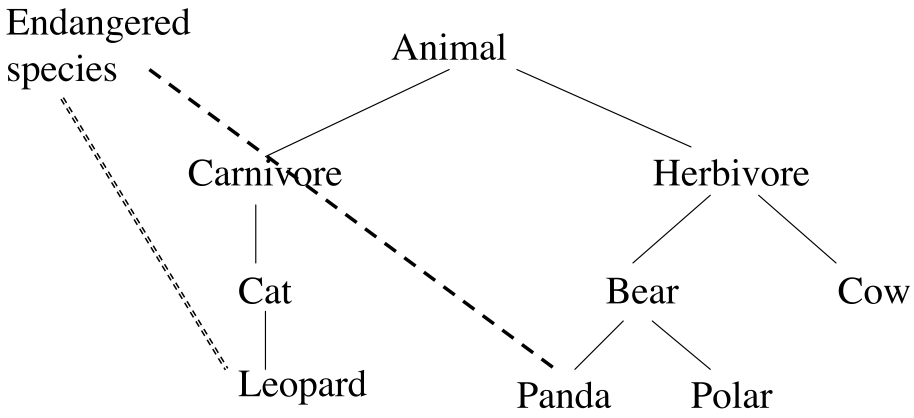
- **Multiple versus single**
  - **Real world is multiple inheritance**
  - **Linearizing lookup**
    - **Problem: interpretation depends on non-local inheritance structure, not robust in face of changes**
  - **No problem if no conflicts**



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

Multiple Inheritance

- **Needed to describe certain complex *is-a* relationships**

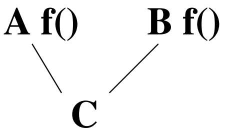


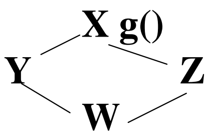
Multiple Inheritance Conflict Resolution

- **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)**

Multiple Inheritance Conflict Resolution

- **Problems:**



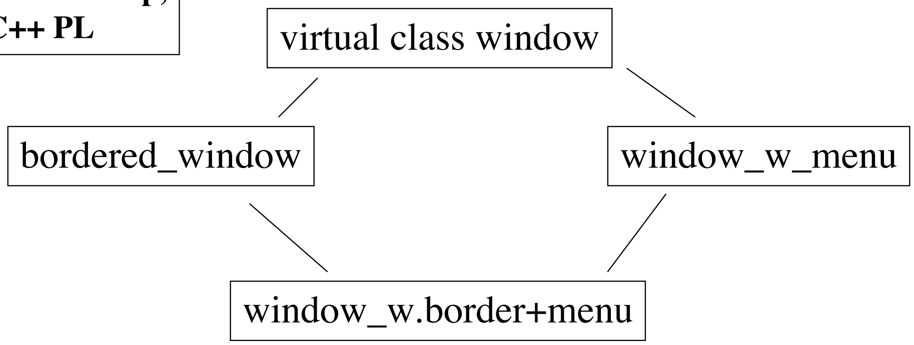


  - **Member clash**
  - **Inheriting more than one copy of same member**
- **Approaches**
  - **Linearize hierarchy so only one parent is "closest" (CLOS, Flavors)**
  - **Throw an exception when same member is applied more than once due to duplicate paths**
  - **Exclude some members to avoid problem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

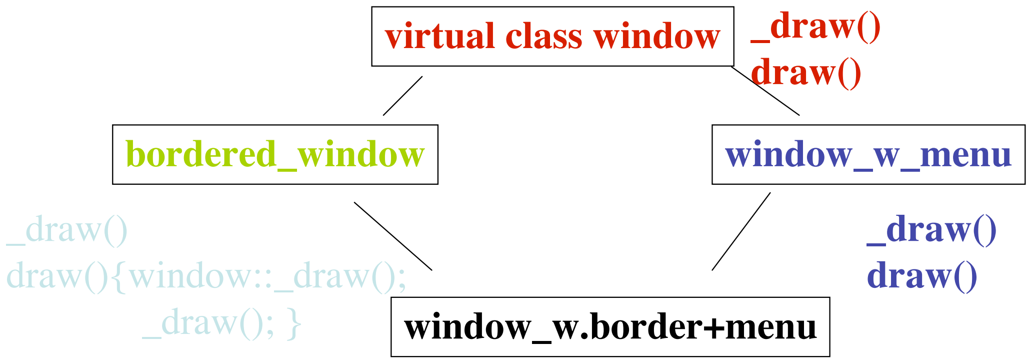
A. Snyder's Mix-in Classes

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

From Stroustrup, The C++ PL



Example



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

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

More on Mixin Inheritance

- **Mixin - 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: mixin 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

```
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!