Slide references for 686

Plan

- Basic Principles and OOPLs
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Metaclasses
  - Classless OOPLs
  - Implementation techniques for compilers of OOPL
  - Other Interobject relationships-- association, part-whole, inheritance, instance-of, friends, nested classes,...
- Use of the above
- Design Patterns
- Frameworks and meta-patterns
- Refactoring patterns
- Analysis patterns
- Processes – Use cases, CRC, and other techniques
- Middleware architectures
- Metrics

Evaluation

- A few programming assignments – P/NP (feedback) - 5
- 2 Quizes - 15
- Midsem, endsem -- 70
- Term project in group (implementation or paper reading) – 10

Abstraction

- Data abstractions – examples: primitive types, structured types
- Control abstractions – examples: functions, control constructs
- Object abstraction: data (hidden) + control (exported)
- Limitations of non-oop abstractions in combining the two?
  - Data cannot be kept hidden inside function bodies
    - Static? --- cannot be shared across multiple function
      - Try an implementation of objects with multiple function —use function pointers
    - Globals-- accessible to other functions—breakage of encapsulation (data cannot be hidden)
    - Multiple instantiation needs parameterization-->data has to be global

Object abstraction in OOPLs

- core constructs: class, interface
- Class = {data, public interface, their implementations}
- Classless OOPLs---cloning for instantiation
- An Example
  - Object Counter –
    - interface={inc,dec,set,reset,val}: Abstract data type
    - Focus on externally observable behavior
    - Not on internal implementation (during conceptualization)

Encapsulation

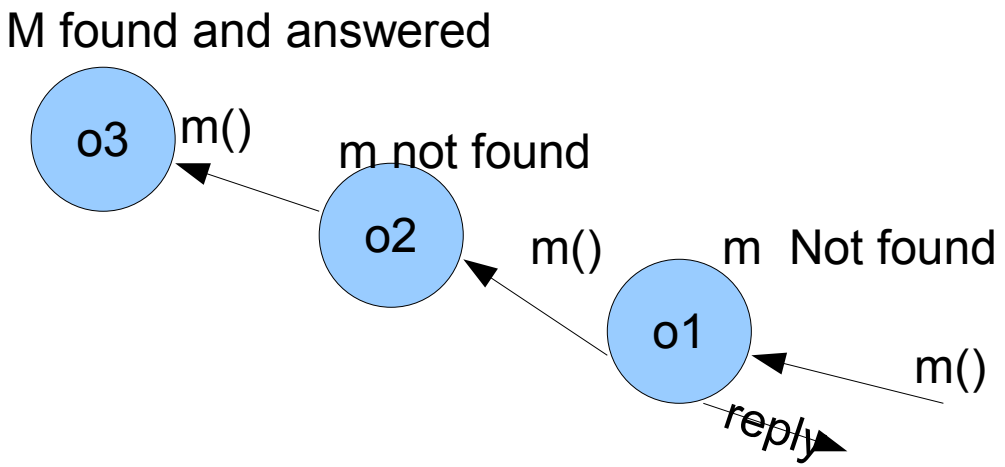
- To make sure that the only way to access or interact with an object is through the intended abstraction
- An old principle – applied to obj abstractions
  - e.g. Locals in files, functions
  - Other examples: human beings, function libraries – local members/control flow are hidden, files with local variable
- How does this principle manifest in OOPLs?
  - The distinction between Private members and public members
- Required when you implement abstraction
  - i.e. It has more to do with implementation than a conceptualization. Abstraction deals with conceptualization.

Breakage of Encapsulation

- When is encapsulation considered as broken:
  - Abstraction no longer works
  - Bypass abstraction and manipulate
  - Flaws in design – e.g. Top pointer public
  - Flaw/feature in language – exploited e.g. Viruses, buggy code using pointers
  - Type safe computation – compile time and runtime
    - Exception handling

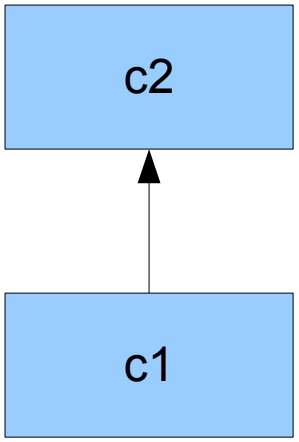
Inheritance and delegation reuse mechanisms

- Relation between 2 classes
- Between two objects== delegation model
  - Delegation is meant to obtain the same effect as that of inheritance when we operate at object level in classless languages
-



Delegation Model

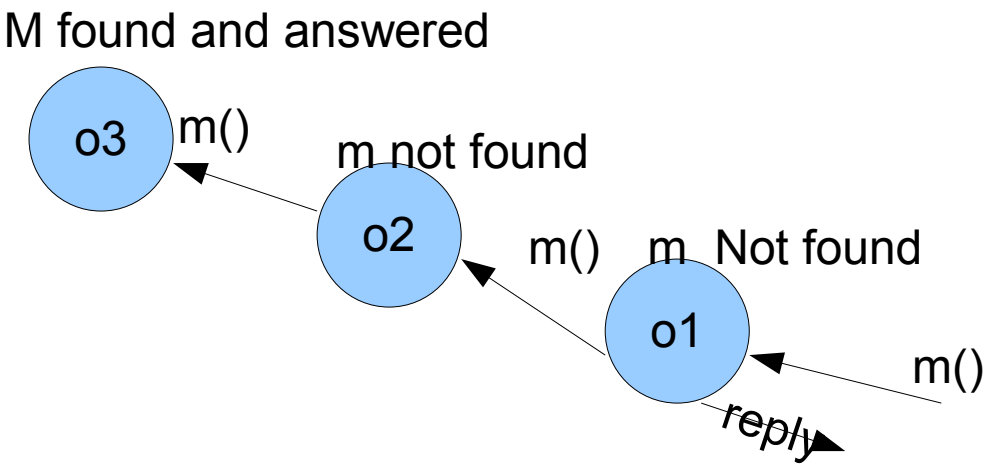
Inheritance between classes



C1={h,k}
C2={f,g}
o1 = new C1;
o2 = new C2;

is o2 parent of o1?

:no. o1 and o2 are independent instances. But o1 has its own internal o2



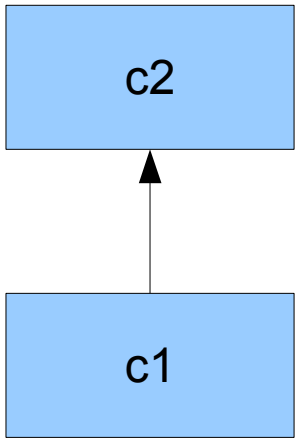
Delegation Model

o1={j,k}
o2={l}
o3={m}
o2 is parent of o1;
o3 is parent of o2;
o1.m() --> will this work?

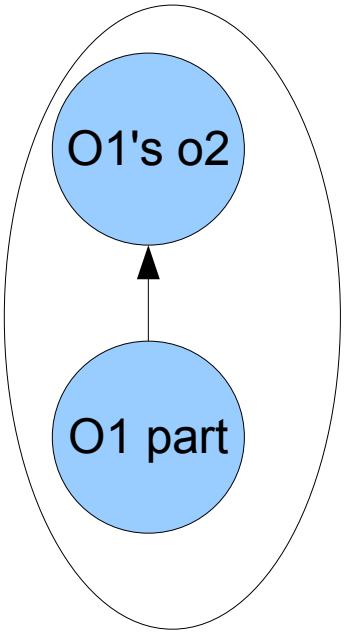
Inheritance vs. delegation

- Inheritance
  - Between classes
  - In Class-based languages
  - Every instance of derived class has internal parent chain
  - Cannot share parent objects, but can share parent classes
  - Reuse of parent class
- Delegation
  - Between objects
  - In prototype-based languages
  - Chaining of objects is explicit
  - Can Share parent instances
  - Reuse of parent object

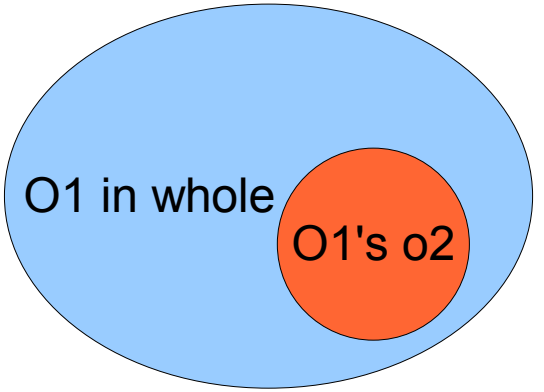
Internal parent objects in inheritance



An integrated assembly for instance of C1



Alternative way of picturing the same assembly

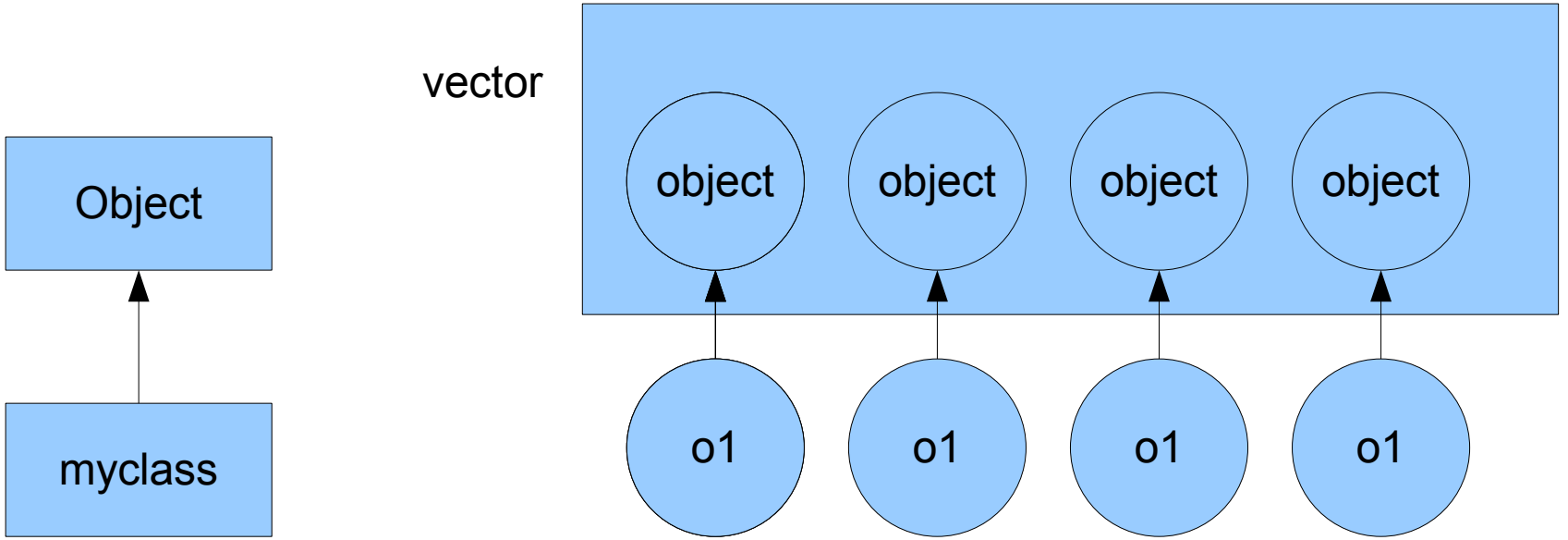


O1 = new C1

o1's o2 can be extracted if needed --> widening

from such o2, the associated o1 can be extracted back--> narrowing

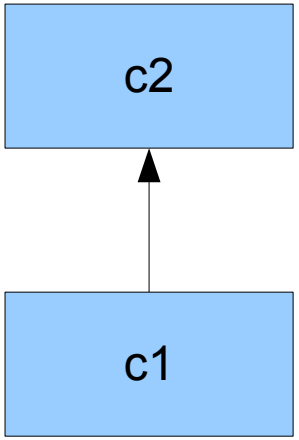
Example of narrowing



A vector holds instances of type object (by widening)

from this vector the actual objects can be extracted for use --> narrowing

Inheritance for reusing parent's members as they are: pure extension.



Defn C2 = {f.,g}
Defn C1 = {h, k}

Effectively, if C1 : subclass of C2, then
C1 = {f,g,h,k}

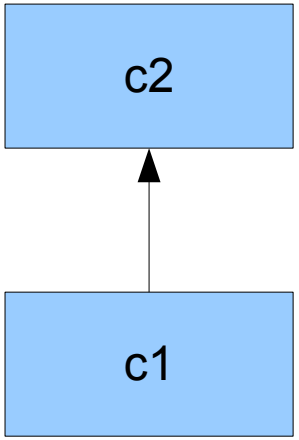
in other words, C1 is an extension of C2, i.e. C2 has been extended—you have 2 more functions

why not simply edit C2 and add these functions
-- useful in modeling?
-- independent instances of c2 will be affected

objects of both the classes are needed

Can be some members be removed?

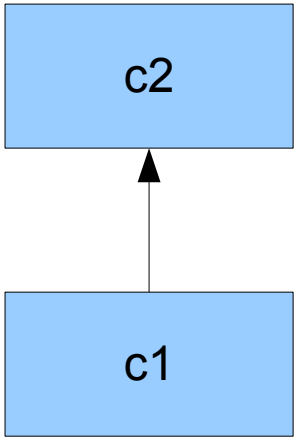
Defn C2 = {f.,g}
Defn C1 = {h, k}



can we say, C1 is subclass of C2, but without C2:f ?

--- most OO languages do not permit this feature
-- for type safe widening

Inheritance with Specialization: Can some members be changed?-yes



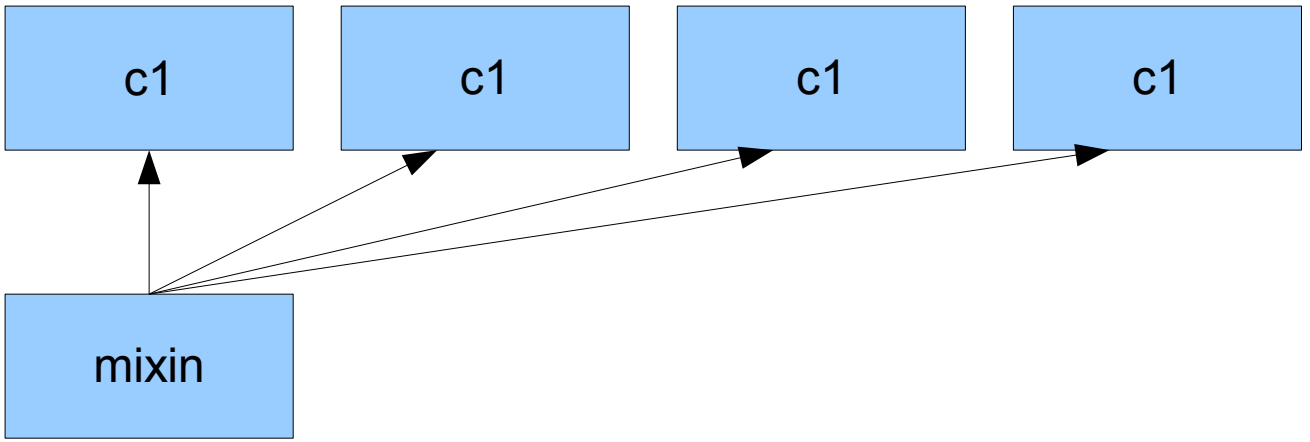
Defn C2 = {f.,g}
Defn C1 = {h, k, f}

can we say, C1 is subclass of C2, with C2:f changed ? : yes
java like syntax
for an equivalent prog in c++, use pointers

```
C1 c1 = new C1;
C2 c2 = new C2;
c1.f                    which f?   C1::f
c2.f                    which f?   C2::f
c2 = c1;
c2.f                    which f?    C1::f
```

Observe that though the 2 statements are same, static binding is not done

Inheritance for mixins



Mixin has 4 internal components.
e.g. A PC with motherboard, memory, graphics card and inbuilt network card

Contracts

- Between parties (at least 2)
- Contract = abstraction / full behavior = ADT specification
- Interface = syntactic contract
    - Member function names
    - Input parameter types
    - Return types
    - Exceptions
    - Name of the interface
- Object's contract: object itself and its environment
- Design by contract method by Meyer, inventor/designer of Eiffel language

Design by contract

- Between full ADT description
- Assertions
  - Preconditions
    - Parameter values
    - Local state
  - Postconditions
    - Value to be returned
    - Current local state
    - Old state (state-1) before this call was accepted
  - Invariants
    - Class invariant == true throughout the lifetime of the instance
- Example of stack

What if a contract violation is detected?

- Who detects?
  - The runtime environment
- Exception is thrown
  - e.g. Precondition violation exception
- Who benefits from pre-conditions?
  - Implementation of the object
    - In what way? -- no need to check for pre-conditions
      - Write the pure abstraction logic
    - Who ensures pre-conditions?
      - Parameters: caller ensures
      - Local state: previous postcondition/initial condition
- Who benefits from post-conditions?
  - Caller
  - Who ensures them?
    - Server object/the object/service provider
      -

3 levels of contract specifications

- Best: full description
- Syntactic – interface type descriptions
- Assertions: pre/post conditions, invariants: design-by-contract
- C++: use assert macro—before and after the method body (core code); and do not check for the assertions in the core code of the method—to benefit from contracts
  - Terminate upon failure of assertion
  - For graceful degradation: exception handling is used (as in Eiffel)
- Defensive Programming

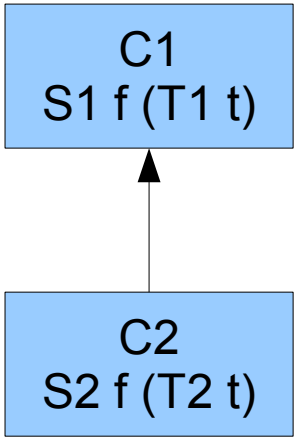
Defensive/contract oriented programming/development

- Develop interfaces
- then develop contract specifications
- Compile them
- Then write the body of the methods == you got the class now!
- Work with the class
  - Your contract code (assertions) work against logical errors in methods bodies

Contracts and inheritance?

- What's an acceptable refinement in inheritance?
- Builder's dilemma
  - Original: 1,00,000 --> 3BHK
  - New? 2,00,000 -->3BHKFurn
  - 1,00,000 -->3BHKFurn
  - 50,000-1,00,000 --> 3BHK or 3BHKFurn
  - 2,00,000 ---> 1BHK
  - Should preconditions be allowed to become weaker? stronger?
  - Should postconditions be allowed to become stronger? weaker?

Type systematic view



C1::f and C2::f are virtual/dynamically bound functions

consider following code:

```
C1 *obj = new .....
T1 * t = new .....

 obj --> f (t);
```

We have following 4 combinations

	T1	T2
C1		
C2		

covariance

- C1 *obj = another.k()
- T1 *t = other.g()
- obj->f(t)
- K returns an instance of C2
  - G returns an instance of T1
  - Compiler passes the code
  - Runtime error
- Can you construct such an example with contravariance?
- Not allowing covariance is too restrictive

contravariance

- Type safe but too restrictive
  - Asking developers of new classes to use old or older parameter types
  - Therefore Eiffel supports covariance and uses runtime type checking to prevent type unsafe combinations

Invariance of parameter types

- If there is a slightest change in parameter types
  - Don't analyze relationships between those parameter types
  - Simply consider the two functions as entirely different ones – they only happen to have same name and that;s all-- overloading
- Overloading is not dynamic binding
  - At compile time you can resolve functions
  - No need to wait till runtime
  - Overloading is called syntactic polymorphism

Return types

- Covariance is safe
- Contra: unsafe
  - C1 *obj = new C2
  - S1 *s = obj.f()
    - S2 C2:f()
    - If S1 is subclass of S2

polymorphism

- Why is subclass a subtype?
- Reuse argument
  - 1. reuse code written in terms of the superclass(super-type)
    - In what context?
      - In an environment which provides instance of subclasses
  - 2. reuse member functions of superclass
    - In what context?
      - By not implementing/overriding in subclass. i.e. Reused in subclass
      - Reuse the contracts

'this'

- Sharing of member function implementations
  - i.e. One per class

    vs.
- Embedding implementations inside objects
  - i.e one per object

'this'

C1 *o1 = new C1

C2 *o2 = new C2

C1* o3 = new C2

o1 -> f

o2 -> f

o3 ->f