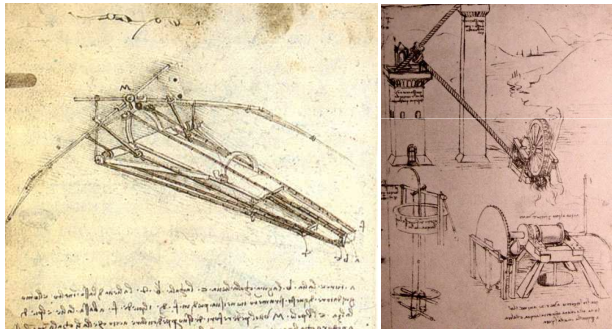
HY540 – Advanced Topics in Programming Language Development



Lecture 3

Inheritance – introduction (1/4)

- *What is it?*
  - A way of defining new classes, called *derived*, *on the image* of other existing classes, known as *bases*
    - Thus inheritance was originally introduced as a class-oriented concept
  - The way derived classes can deviate from *the image* of their bases is not in detail agreed across all languages
    - But it is always true that derived classes are seen as specializations of their base, meaning they play as subcategories

Inheritance – introduction (2/4)

- *Why was it introduced?*
  - As a reuse discipline since frequently objects are needed in a system being apparent variations of other objects (sometimes also coexisting in the same system design)
    - So, if objects relate with a *look alike* relationship, somehow their respective class should relate as well
  - Consequently, inheritance formalized the fact that object variations may be supported only through corresponding class variations
    - And clearly, variations may built upon previously defined variations
    - Thus, the notion of hierarchical class taxonomies (derivation trees) has naturally appeared

Inheritance – introduction (3/4)

- *What relationships it defines?*
  - Among classes
    - *subtyping* relationship through *subclassing*
    - ***A*** subclass of ***B*** $\Rightarrow$ ***A*** subtype of ***B***
  - Among objects
    - *substitutability* relying on subclassing
    - ***A*** subclass of ***B*** $\Rightarrow$ ***A*** objects are eligible as ***B*** arguments

Inheritance – introduction (4/4)

- *How it affects the code?*
  - Hierarchies are disciplined structures
    - So they serve as a code organization paradigm
  - Hierarchies may become very rigid
    - Meaning update flexibility can be limited
  - Subtyping is always a dependency
    - Thus hierarchies are strongly coupled, meaning modularity may be sometimes put in severe danger
  - Substitutability is the real gold
    - *Reuse is overall judged by how many functions support substitutability and how many classes are eligible in them*

Roadmap

- *Next we discuss*
  - Subclasses
  - Virtual tables and late binding
  - Multiple inheritance
    - Shared and repeated inheritance
    - Member resolution ordering
  - Typed object protocols

Subclasses (1/12)

- A subclass defines a class by describing extensions or changes to an existing class called its base class
  - Fields from the base class are implicitly replicated in the subclass and new ones may be also added
  - Methods from a base class may be either replicated in a subclass, by default, or explicitly overridden by similarly named and typed methods
- By the *subclass relation* we mean the partial order induced by the subclass declarations

Subclasses (2/12)

- Subclassing is a *binary partial order* relation among classes:
  - $A \leq A$ *reflexive*
  - $A \leq B \wedge B \leq A \Rightarrow A = B$ *antisymmetric*
  - $A \leq B \wedge B \leq C \Rightarrow A \leq C$ *transitive*
- *Inheritance is* the *sharing* of base attributes by the subclasses including both fields and the code for methods
  - So *inheritance is an effect of subclassing*, although we tend to say *A* inherits from *B* to mean *A* is a subclass of *B*
  - Some subclasses effectively inherit nothing

Subclasses (3/12)

- By overriding we mean a subclass method that *is treated* as an updated version of a base class method, and is subject to rules set by the language
  - May allow override methods explicitly declared as *overridable* (like with *virtual* functions in C++, Java and C#)
    - Methods qualified once as overridable *always* carry their qualification to all subclasses
  - May treat all methods as overridable (Smalltalk)
    - Every method re-implemented by the subclass is considered to override its respective base method

Subclasses (4/12)

- Within a subclass there must be an unambiguous way to refer to either the subclasses fields or to the inherited fields any level above
  - any occurrence of ***self*** within a method ***m*** is interpreted with respect to the current class
    - the same holds when ***self*** is implied in the language (i.e., it can be used optionally)
  - invocation of an original method ***m*** from a direct base class is allowed though ***super.m***
    - alternative ways of qualification to invoke “upwards” a method are supported in different languages
      - ***resend*** (Self), ***call-next-method*** (CLOS), ***Base::*** qualifier (C++)

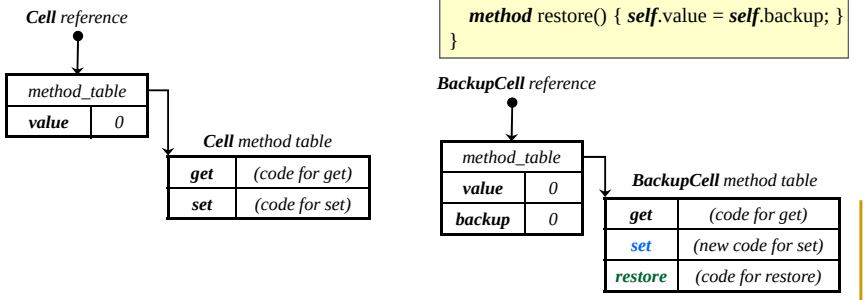
Subclasses (5/12)

- The implementation of compile-time method lookup relies on the assembly of method tables
  - the methods of the base class are merged appropriately with the methods of the subclass
    - an overridden method appears only once with its most recent version
  - method **a.*m*** lookup for object ***a*** is performed using the method table of *TypeOf(a)*
    - the method table per subclass is created at compile-time
    - is stored at runtime in memory
    - is referenced by every instance of a class by a reserved constant hidden field of the instance

Subclasses (6/12)

```
class Cell {
   var value: integer=0;
   method get(): integer { return self.value; }
   method set(n: integer) { self.value=n; }
}
```

```
subclass BackupCell of Cell {
   var backup: integer=0;
   override set(n: integer) {
            self.backup = self.value;
            super.set(n);
   }
   method restore() { self.value = self.backup; }
}
```



Subclasses (7/12)

Assume reference semantics for objects in the code below:
var *normalCell* ***:*** *Cell* ***= new*** *Cell;*
var *backupCell* : *BackupCell* ***= new*** *BackupCell;*
function *f* (a: *Cell*) { … }

normalCell = backupCell; Is this legal?
f(backupCell); Is this legal too?

An instance of class *BackupCell* is assigned to a variable holding instances of class *Cell*.

Similarly, an instance of class *BackupCell* is passed as an argument to a function *f* expecting instances of *Cell*.

- The previous code is legal due to the fundamental property of subclassing called *subtype polymorphism*:
  - *A' subclass of A* $\wedge$ *a' instance of A'* $\Rightarrow$ *a' instance of A*
  - *We define a subtype relationship* <: *as follows*
    1. *a: A* $\wedge$ *A* <: *B* $\Rightarrow$ *a: B*
       - ***subsumption****: a value is subsumed from type A to type B*
    2. *TypeOf(A')* <: *TypeOf(A)* $\Leftrightarrow$ *A* subclass of *B*
       - ***subclassing-is-subtyping****: also known as inheritance-is-subtyping*

Subclasses (8/12)

function *f* (x: *Cell*) { *x.set(23);* } *f(backupCell);* What is the meaning of *x.set(23)* in *f* for this call?

- With the introduction of subsumption we need reexamine the meaning of method invocation
  - The *declared type* of *x* is *Cell*
  - Its actual value is *backupCell* instance of *BackupCell*
  - So its *actual type* is *BackupCell*
- There are two possible options
  - *static dispatch* *x.set(23)* executes *Cell.set*
    - Based on compile-time information available for *x*
  - *dynamic dispatch* *x.set(23)* executes *BackupCell.set*
    - Based on the runtime value of *x*

Subclasses (9/12)

- Dynamic dispatch (*late binding*) is found in most class-based languages (some have variations) and is considered fundamental for object abstraction
  - each object knows how to behave autonomously, so the context does not need to examine the object to decide which operation to apply
- Some languages may also support static dispatch usually in two ways
  - upon invocation of non-overridable qualified methods
  - through syntactic constructs imposing static dispatch
    - like ***super.m*** identifying statically a specific method
    - or when qualified access to base methods is used

Subclasses (10/12)

- An interesting consequence of late binding is that *subsumption should have no runtime effect on objects*
- More specifically, assume an application of subsumption from *BackupCell* to *Cell* converts a *BackupCell* to *Cell* by cutting-off its additional attributes (*backup* field and *restore* method)
  - Then a dynamically dispatched invocation of *set* would normally fail because access to the *backup* field is a memory error
- Languages supporting stack-based objects, i.e. value copy semantics, have to deal with such truncation issues for assignment and argument passing
  - for example, the C++ language supports subsumption only for pointers and references but never for stack objects

Subclasses (11/12)

- Subsumption reduces static knowledge about the true type of an object
  - Let a root class with no attributes, where all classes are subclasses of it
  - Then, by subsumption, any object can be are regarded as a member of the root class, but is useless since it has no attributes
- In our examples, by subsuming a *BackupCell* object to a *Cell* object we loose the ability to access *backup* and *restore* attributes
  - However, these attributes are still used by the body of the overriding method *set*
  - So *attributes forgotten by subsumption can still be used thanks to late binding*

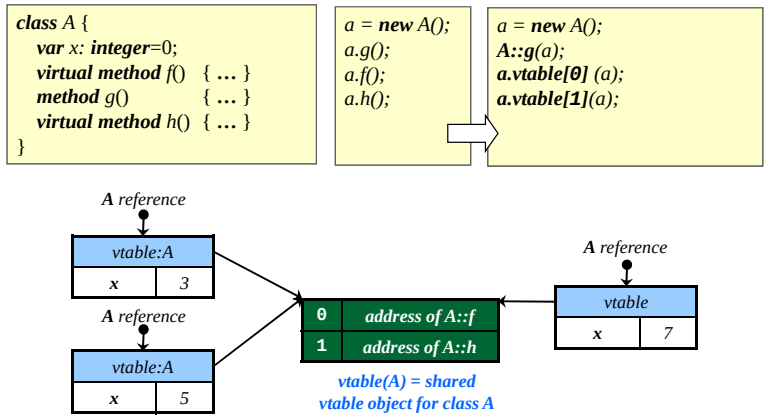
Subclasses (12/12)

- In a purist view of OO methodology *late binding is the only mechanism to take advantage of attributes forgotten by subsumption*
  - The purist approach implies that the language forbids to revert from a subsumed view to the original object value, not even query the original view – i.e., can't have down casting
  - In our example, given a *Cell* object whose actual value is a *BackupCell* object, we should not be able to examine it is a *BackupCell* object and regain it
    - *examine* ⇔ *type information (runtime)*
    - *regain* ⇔ *type casting (downcasting)*
- The idea is that clients should be revealed only the details visible by the classes they use
  - Once we give clients a *Cell* object *restore* is inaccessible
  - Commonly broken in many languages through downcasting

Virtual tables and late binding (1/6)

- The implementation of dynamic dispatch is possible through a special way of handling method tables during runtime
  - Commonly called *virtual method tables*, *virtual function tables*, *virtual tables* or shortly *vtables*
  - They concern only overridable methods (those qualified as *virtual* in some languages)
- *We add a new vtable in a new class* with all new overridable methods it introduces
  - if none, no vtable is added
- Every instance of a class has a reserved field being a pointer to its current vtable
- All access to overridable methods inside a class are made exclusively by referring to the vtable

Virtual tables and late binding (2/6)



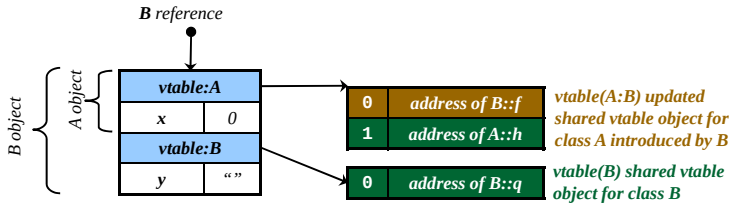
Virtual tables and late binding (3/6)

- When a class ***B*** refines a method ***f*** inherited from its base class ***A*** then during compilation:
  - we produce a new version of the vtable for ***A***
    - with the address of the refined ***f*** stored in the vtable cell corresponding to ***f***
  - we generate extra code in the ***B*** constructor
    - which assigns to the vtable field of the ***A*** base object the address of the new vtable

Virtual tables and late binding (4/6)

```
class A {
   var x: integer=0;
   virtual method f()  { … }
   method g()          { … }
   virtual method h()  { … }
}
```

```
subclass B of class A {
   var y: string="";
   override method f() { … }
   method g()                { … }
   virtual method q()        { … }
}
```



Virtual tables and late binding (5/6)

```
in B constructor {
      vtable:A = vtable(A:B)
      vtable:B = vtable(B)
      …
}
```

```
F(b:B) {
  b.g();
  b.f();
  b.h();
  b.q();
}
```

```
assume actually b : B
B::g(b);                  statically
b.vtable(A)[0](b);        late B::f
b.vtable(A)[1](b);        late A::h
b.vtable(B)[0](b);        late B::q
```

Polymorphism with dynamic dispatching

```
function foo (a:A)  { a.f(); }
      =>
function foo (a:A) { a.vtable(A)[0](a); }
      =>
function foo (a:A) { B::f(a); } when a:B
```

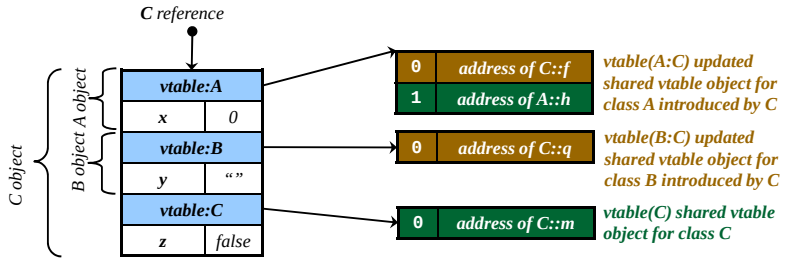
Next we study an example for multiple inheritance

Virtual tables and late binding (6/6)

```
class A {
   var x: integer=0;
   virtual method f()  { … }
   method g()          { … }
   virtual method h()  { … }
}
```

```
class B {
   var y: string="";
   override method f(){ … }
   method g()          { … }
   virtual method q()  { … }
}
```

```
subclass C of class A, B {
   var z: boolean=false;
   override method f()       { … }
   override method q()       { … }
   virtual method  m()       { … }
}
```



Multiple inheritance (1/17)

- Multiple inheritance means a subclass is defined as the concurrent extension or specialization of many (multiple) base classes
  - ***A** subclass of **B_1, B_2,…,B_N***
- It allows reuse and adaptation in a far easier way compared to single inheritance
  - But it is harder to implement
  - And even more hard to deploy it without sacrificing modularity
  - That is why some languages drop it (Java, C#)

Multiple inheritance (2/17)

- In single inheritance no class may appear twice in an *inheritance chain*
  - ***$A_1 \leftarrow A_2 \leftarrow \ldots \leftarrow A_N \Rightarrow$ every A_i appears only once***
    - **Since no class is allowed to inherit from itself**
- In multiple inheritance a class may appear multiple times in an *inheritance tree*
  - assume two base classes with a common base class
    - ***A** subclass of **D*** and ***B** subclass of **D***
    - ***C** subclass of **A, B***
    - what is the object model of *subclass **C**?*

Multiple inheritance (3/17)

- If every distinct occurrence of a class ***D*** in the inheritance tree ***T*** of a subclass ***C*** contributes separately, and equivalently, to the ***C*** object model then:
  - the presence of ***D*** in the object model of ***C*** is repeated as many times as ***D*** appears in ***T***
  - this form is called *repeated inheritance*
    - in *Eifel* you can specify as part of the subclass what you want to share and what you want to replicate
    - in *C++* repeated inheritance is the default for all members

Multiple inheritance (4/17)

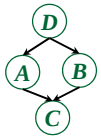
- If all distinct occurrences of a class ***D*** in the inheritance tree ***T*** of a subclass ***C*** contribute only once to the ***C*** object model then:
  - there is a single presence of ***D*** in the object model of ***C*** despite the multiple appearances of ***D*** in ***T***
  - this form is called *shared inheritance*
    - in *Eifel* you can eventually emulate sharing by collapsing repeated members explicitly, but it can be painful
    - in *C++* shared inheritance is feasible with virtual base classes
    - the issue does not apply to *Java*, *C#* and *Objective-C* with single inheritance only

Multiple inheritance (5/17)

- Shared inheritance affects the way the vtable should be handled for the shared object
- We study an example showing how the vtable should be managed

```
class D {
   var x: integer=0;
   virtual method f()  { … }
   virtual method g()  { … }
}
```

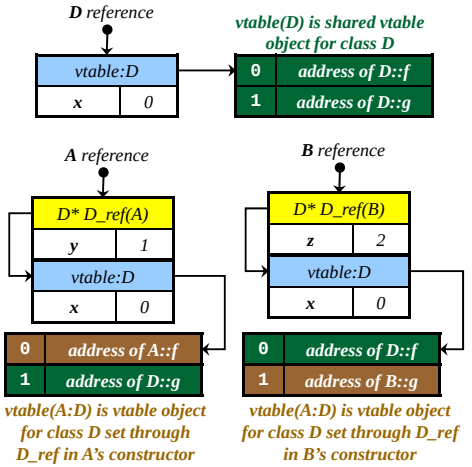
```
subclass A of shared class D {
   var y: integer=1;
   override method f(){ … }
}
```




```
subclass B of shared class D {
   var z: integer=2;
   override method g(){ … }
}
```

```
subclass C of class A, B {
   var w: integer=3;
   override method f() { … }
   virtual method  q()  { … }
}
```

Multiple inheritance (6/17)

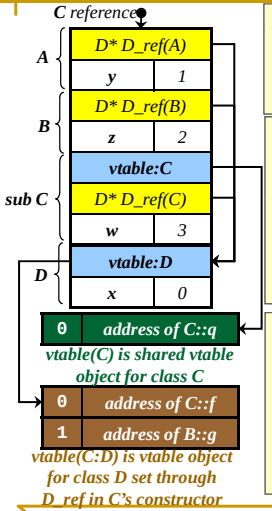


While parsing ***A*** or ***B*** classes and producing their memory region models we cannot put ***D*** inside since, then, any subclass of both ***A*** and ***B*** would have two copies of the shared base class ***D*** in its model. So, we apply the following trick:

(a) include a pointer to the memory region of ***D*** in ***A*** and ***B***, since ***D*** is shared

(b) when making ***A*** or ***B*** instances in the program, as part of their required memory we include an extra block for a ***D*** object and set its address at the ***A*** or ***B D_ref*** field. This way the memory for a complete ***A*** or ***B*** it is still continuous.

Multiple inheritance (7/17)




```
A_constructor (A* this, D* d){
   this->D_ref = d
   this->D_ref->vtable:D = vtable(A:D)
}
```

```
C_constructor (C* this, D* d){
   this->vtable:C = vtable(C);
   A_constructor(this + offset(C:A), d);
   B_constructor(this + offset(C:B), d);
   this->D_ref = d
   D_ref->vtable:D = vtable(C:D)
}
```

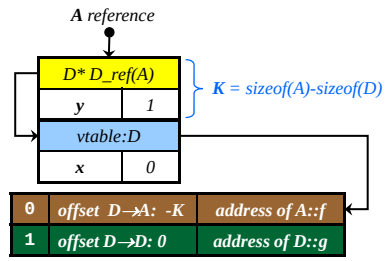
```
ref A a = new A;          Heap (low→high)
a=getmem[sizeof(A)]
A_constructor(a, a+sizeof(A)-sizeof(D));

val B b;                  Stack (high→low)
a = top - offset(B);
B_constructor(a, a+sizeof(B)-sizeof(D));
```

Clearly shared base classes introduce an extra reference field (*memory overhead*) in every object, while they impose that every access to the base object passes through this reference field (*performance overhead*). Additionally, down-casting / cross-casting from a reference of a shared base class to any derived class will generally fail (even with specialized runtime downcasting operators) - reserving extra space in the shared object per derived object is considered unacceptable.

Multiple inheritance (8/17)

- When calling an overridden method f with a base subobject $A\ a$ then
  - the call is made as $a.vtable(A)[j](a)$
  - where j is the vtable position corresponding to f
- Since the refined method is called in the context of a subclass subobject we need pass its address as *self* rather than merely a
  - *se we need restore what was forgotten by subsumption*
- We introduce a way to convert: the base callee subobject to the derived subobject owning the invoked method (most-refined version)

Multiple inheritance (9/17)

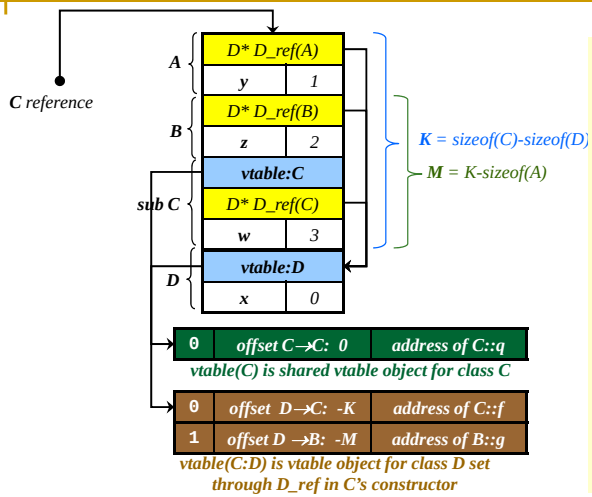


vtable(A:D) is vtable object for class D set through D_ref in A's constructor

***function** foo (d:D) { **d.f();** }*
⇒
*{ **d.vtable(D)[θ].address(a+d.vtable(D)[θ].offset);** }*
⇒
*{ **A::f(d-K);** } when d:A*
⇒
*{ A::f(**A object encompassing D d object**) }*

- We produce extended vtables encompassing pairs of address and offset
- The offset is added to the callee when passed as an argument to invocations to convert it to the correct derived subobject in the context of which the call is actually made
- The offset is the distance among*: the starting address of the subobject whose vtable is updated from the object whose class actually updates the vtable*

Multiple inheritance (10/17)



vtable(C) is shared vtable object for class C

vtable(C:D) is vtable object for class D set through D_ref in C's constructor

Βλέπουμε ότι ακόμη και για τα shared subobjects θέτουμε offsets, ενώ ακόμη δεν γνωρίζουμε εάν ένα ***D*** subobject ανήκει σε ένα ***C*** main object ή σε ένα main object ενός subclass ***E*** του ***C***.

Ωστόσο παρατηρούμε ότι στη δεύτερη περίπτωση απλώς ο constructor του ***E*** θα έκανε overwrite τον vtable του ***D***, τον οποίον θα είχε θέσει η κλήση του ***C*** constructor που προηγούνταν, με τα σωστά offsets του ***D*** στο πλαίσιο ενός ***E*** main object. Άρα η μέθοδος γενικεύεται σωστά.

Multiple inheritance (11/17)

- Since a class can inherit from multiple base classes an issue is how *member resolution* is applied when there are name conflicts amongst the inherited parts
  - Let class ***A*** with field ***x*** and method ***f***
  - Let class ***B*** also with field ***x*** and method ***f***
  - Consider ***subclass C of A, B*** with a method ***g*** including unqualified (i.e. default) reference to inherited field ***x*** and method ***f***
  - Then which of the following two cases apply?
    - $x \equiv A(x) \wedge f \equiv A(f)$
    - $x \equiv B(x) \wedge f \equiv B(f)$
- The language should unambiguously define how name references in subclasses resolve to members found in bases classes in multiple inheritance with a *member resolution ordering* method

Multiple inheritance (12/17)

- There are two fundamental requirements any member resolution ordering method must satisfy
  - should by compliant with the notions of *refinement* and *most recent*
    - ☞ always choose the newest versions appearing within the bottommost subclasses
  - should preserve the corresponding ordering within any of the base classes when seen in isolation, thus it should be *monotonic*
    - ☞ the priorities of inherited members should be respected by subclasses

☞ *The first is obvious, now lets study why the second one is critical as well*

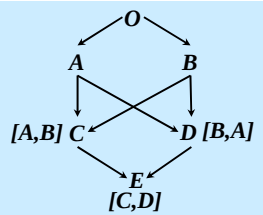
Multiple inheritance (13/17)

class O { *virtual method f* }
subclass A of O { *override method f* }
subclass B of O { *override method f* }

subclass C of A, B {}
Ordering is: **C→B→A→O**
function g (c:C) { c.f() ≡ B.f(c) }

subclass D of B, A {}
Ordering is: **D→A→B→O**
function h (d:D) { d.f() ≡ A.f(d) }

subclass E of C, D {}
Ordering 1: **E→D→C→A→B→O**
Breaks ordering of C
Ordering 2: **E→D→C→B→A→O**
Breaks ordering of D
***for **subclass of** $T_1,\ldots,T_N$ we assume T_{i+1} to be more recent than T_i



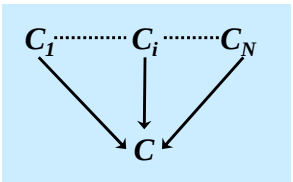
E e = new E;
{ g(e); h(e); } *breaks the semantics of either* **g** *or* **h**
With ordering 2
g(e) ≡ B.f(e) *preserved*
h(e) ≡ B.f(e) *broken*
With ordering 1
g(e) ≡ A.f(e) *broken*
h(e) ≡ A.f(e) *preserved*
***this problem has no solution, so this scheme can't be implemented (some languages allow it but the problem still exists).

Multiple inheritance (14/17)

- When the method resolution order (MRO) for an inheritance tree preserves monotonicity it is called linearized
- Definition – ***linearization***
  - Given an inheritance tree of class ***C*** and any direct or indirect base classes ***A*** and ***B***
  - ***If*** ***A*** precedes ***B*** in the linearization of ***C***
  - ***Then*** ***A*** precedes ***B*** in the linearization of any of the subclasses of ***C***
- The first known algorithm for class tree linearization was introduced as part of the Dylan programming language and is called *C3*
  - *Notably, when no linearization can be produced for an inheritance tree the respective class is rejected*
    - i.e., a semantic compile error is posted

Multiple inheritance (15/17)

```
linearize(C, C1,…,Cn) {        /* Ci more recent than Ci+1 */
  return [C] + merge(
                 linearize(C1),
                 …,
                 linearize(Cn),
                 [C1,…,Cn]
               );
}
```



Definition of C is:
C: C_N,…C₁ if rightmost more recent
C[C₁,…C_N] if leftmost more recent

Multiple inheritance (16/17)

```
merge (L1,…,Ln) {                /* lists of class names */
  L=[], A={L1,…,Ln}, ok = true
  while ok ∧ A not empty do {
    ok = false
    for each Li in A ∧ until not ok do {
      x = Li.head
      if x ∉ in any Lj.tail of Lj∈A then {
        ok = true
        L.append(x)
        Remove x from all Lj∈A
        Remove all empty Lj from A
      }
    }
  }
  if not ok then
    { error "rejected" return nil }
  else return L
}
```

L =head + tail
head: element
tail: list

Assume we have two lists as arguments L_1=[A,B] and L_2=[B,A]. The *merge* function will never make to empty these lists since:

(a) When testing L_1 and head *A* it cannot remove *A* since it is in the tail of L_2. So, L_1 remains as it is until *B* is removed from L_2.

(b) When testing L_2 with head *B*, it cannot remove *B* since it is a tail of L_1, so it waits until *A* is removed from L_2.The mutual dependency will never allow change L_1 and L_2.

The previous scenario reflects the case of merging two lists of base classes having conflicting linearization lists.

Multiple inheritance (17/17)

- *Criticism*
  - Tends to exacerbate the tight-coupling problem amongst bases and derived classes in a combinatorial fashion
    - a subclass may depend on a tree of classes
    - while in single inheritance it depends on a list of classes
    - so the domino of changes from base class refactoring may turn more easily to disastrous
  - Ambiguities causing inheritance schemes to be rejected may be too expensive to rectify (i.e., may require changing base classes)
    - usually programmers end-up with a desirable class schema to address some actual development need
    - likely the need will still be there, meaning they can only modify the design by repairing the inheritance tree

Typed protocols (1/9)

- The support of polymorphism has been based on the notion of subsumption
  - an object can be subsumed from a subtype to a supertype
  - thus polymorphism was based on subtyping
  - which, for us, is until now synonymous to subclassing
- We now present another approach to subtyping without subclassing
  - The idea is to introduce compatibility requirements as opposed to subclassing requirements

Typed protocols (2/9)

```
class Min {
   var n: integer;
  method max (other: Min):Min;
}
class MinMax {
   var n: integer;
  method min (other: MinMax):MinMax;
  method max (other: MinMax):MinMax;
}
```

```
protocol Min [type T] {
   var n: integer;
  method min (other: T):T;
}
protocol MinMax [type T] {
   var n: integer;
  method min (other: T):T;
  method max (other: T):T;
}
```

- Classes *Min* and *Max* are not related via subclassing although intuitively they are related with subtyping
- We define an operator on types named ***protocol*** which accepts a type and produces another type. For example:
  - *Min-protocol:* $T \rightarrow Min[T]$
  - *MinMax-protocol:* $T \rightarrow MinMax[T]$
- Now we define a higher-order subtype relationship $\angle$: among protocol type operators as follows:
  - $P \angle: P' \Leftrightarrow P[T] <: P'[T] \quad \forall$ type T
  - Meaning, subprotocols actually produce subtypes, thus it holds:
  - $MinMax \angle: Min \Leftrightarrow MinMax[T] <: Min[T]$
- So, with protocols we subtype their outcomes, although originally unrelated

Typed protocols (3/9)

- Clearly we need ways to express two things:
  - how a class can actually define that it obeys (offers or implements) a given protocol
    - *class Point* ***implements protocol Min*** *{…}*
    - the previous does not state that *Point* is convertible (subsumed) to *Min*, but that *Point* offers what is required by *Min*
  - how to express that a type parameter should match a specific protocol
    - *protocol Sorter [****T matches protocol Min****] {…}*
    - the previous states that *Sorter* is an operator on types implementing the *Min* protocol

Typed protocols (4/9)

- Consider a polymorphic function on protocols defined with a loose syntax as *f (Min[T]){…}* or *f(protocol Min){…}*
- Then any object of a class obeying the *Min-protocol* or a subprotocol of it is eligible to *f*
- Notably, eligible classes need not be related with a subclass relationship, but have only the limited obligation to offer *Min* functionality
- This approach displays the following advantages:
  - Easier programming of ***f*** since the involved concerns are reduced (protocols are lightweight)
  - Making a class compliant to a protocol involves dependency with a lightweight entity with no effects or ambiguities
  - Subprotocols are just extensions (supersets) of the base protocol, thus inheritance is simple set union, turning even multiple protocol inheritance to trivial

Typed protocols (5/9)

- You experience protocols in most languages with varying syntax and semantics
- *interfaces*
  - In Java and C# a class may implement multiple interfaces
  - In Java you can provide only one implementation of a common method (i.e. appearing in many interfaces)
  - In C# you can implement methods selectively per implemented interface (qualifying with the interface name)
- *abstract shared classes*
  - in C++, lots of pure virtual functions, used exclusively with virtual qualified inheritance
- *traits and concepts*
  - in C++, both relate to a way of expressing and imposing a protocol at compile time
- *protocols*
  - Objective-C, semantically similar to Java

Typed protocols (6/9)

- An important issue is whether the language imposes classes to early-commit on a protocol, or allows to retrofit (adapt) classes on a protocol without affecting the class as such
- In Java and C# which support interfaces automatic retrofitting is not possible, though retrofitting is allowed with extra derived (adapters) classes
  - ✌ *we will show how this could be possibly accomplished by extending a little the language*
- In C++ retrofitting is possible through concepts
  - interface-like constraints on template type parameters enabling better type checking

Typed protocols (7/9)

```
concept LessThanComparable<typename T> { // C++
  bool operator<(const T&, const T&);
};
interface LessThanComparable<T> // C# and Java
  { bool less_than(T x); }

class Foo implements LessThanComparable<Foo> {// Java
  public boolean less_than(Foo x) { comparison logic }
}
class Foo : LessThanComparable<Foo> {// C#
  public bool less_than (Foo x) {comparison logic }
}

class Foo {… }; // C++, class ignorant of concept
concept_map LessThanComparable<Foo> {// C++, retrofitting
 bool operator<(const Foo& x, const Foo& y)
    {comparison logic }
};
```

Typed protocols (8/9)

- *Lets define the syntax and semantics in C# and Java to retrofit classes to interfaces*
- Given already defined class ***A*** and interface ***I*** we introduce the notation ***interface_map {…}*** to denote an anonymous decorator-class definition like '`extends A implements I {…}`' subclass of ***A*** implementing ***I***
- The decorator-class has a constructor with a single argument of instance ***A*** which can be used to refer to the decorated object
- Multiple interfaces may also be implemented with a single ***interface_map A<…>***
- Let any generic function on interfaces `<T extends I<T>> f(T)` (Java) or `f(T)where T: I<T>` (C#), in general defined as ***f(T: I<T>)***
  - `extends` and `T: I<T>` are simply syntax, implying no subtyping

Typed protocols (9/9)

```
public class Foo {…}; // Java, ignorant of interface
interface_map LessThanComparable<Foo> {// Java, retrofitting
  private Foo x;
  public LessThanComparable (Foo _x)
     { x = _x; }
  public bool less_than (Foo y) {comparison logic }
}
```

- Then for any ***a:A*** object ***f(a) ≡ f(new I<A>(a))***
- The technique implements the decoration pattern
- The compiler automatically constructs the correct decorator, if existing, to fit the required interface
- But ***I<A>*** **subclass of** ***A*** though ***new I<A>(a).super ≠ a*** meaning we1 need special type casting for such decorators and possibly resolution of ***this*** (requires extensions on the VM)
- *An issue is where the retrofitting fits (its an anonymous class, right?)*

Lessons learned (1/2)

- The key feature of *subclassing-is-subtyping* is the notion of subsumption
  - a derived object is subsumed to a base object (*auto upcasting*) by respecting method refinements through dynamic dispatching (*late binding*)
- Generic functions can be written via *polymorphism* by exploiting subsumption
  - using supertype arguments, meaning eligible values are implicitly constrained via subtyping
- The primary mechanics behind dynamic dispatching for single or multiple inheritance are virtual tables
  - compiled statically - *method tables*
  - maintained dynamically - *singletons, shared*
  - referred massively - *upon invocation*
  - reverting subsumption - *upon dynamic dispatching*

Lessons learned (2/2)

- *Shared* multiple inheritance introduces an extra overhead to subclasses, while disabling down casting
  - *Repeated* multiple inheritance as the only mechanism is insufficient since many scenarios require base sharing rather than replication
- Method resolution ordering through *linearization* respects the notions of refinement and most recent
  - but not all class schemas are allowed
- Typed *protocols* allow genericity without subtyping
  - classes may conform to multiple protocols
  - unrelated classes may conform to the same protocol
  - protocols may extend existing protocols
  - protocols themselves are typeless

Reuse flash forward

- Via *abstraction*
  - using *subtype* polymorphism
    - polymorphic functions
    - abstract classes
- Via *genericity*
  - using *protocol* polymorphism
    - generic functions
    - generic classes
- What to choose for language design?
  - Favor *generic functions* - more open
  - Favor *class retrofitting* - more orthogonal
  - Favor *information hiding* - more modular