

# Concepts of Object-Oriented Programming

**Peter Müller**

Chair of Programming Methodology

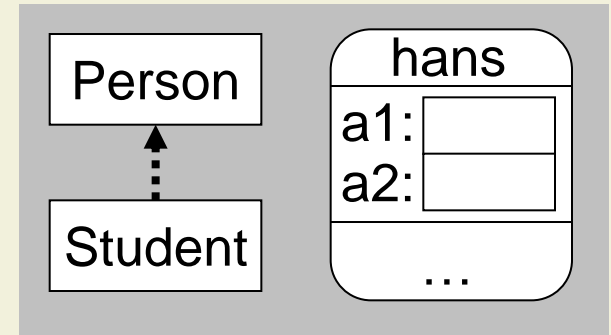
Autumn Semester 2013

**ETH** zürich

# Reuse

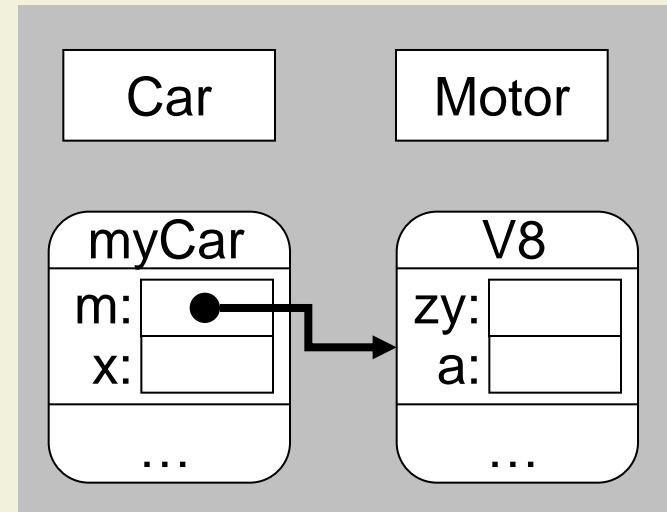
## ■ Inheritance

- Only **one object** at run time
- Relation is fixed at compile time
- Often coupled with subtyping



## ■ Aggregation

- Establishes **“has-a” relation**
- **Two objects** at run time
- Relation can change at run time
- **No subtyping** in general



# 3. Inheritance

3.1 Inheritance and Subtyping

3.2 Dynamic Method Binding

3.3 Multiple Inheritance

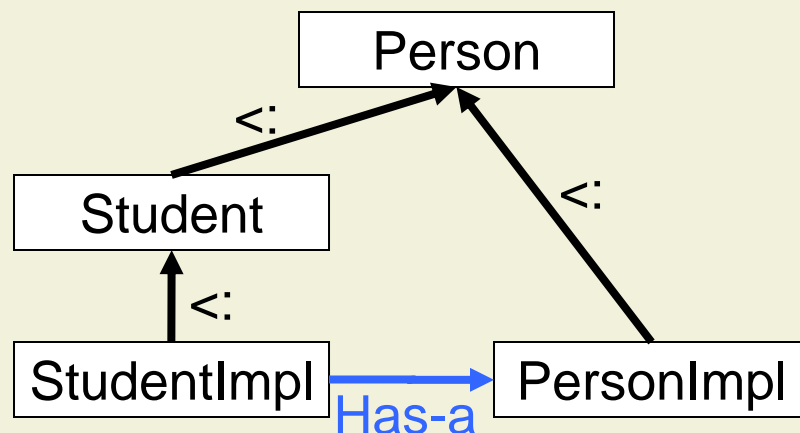
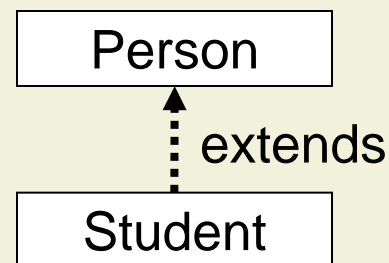
3.4 Traits

# Inheritance versus Subtyping

- **Subtyping** expresses **classification**
  - Substitution principle
  - Subtype polymorphism
- **Inheritance** is a means of **code reuse**
  - Specialization
- Inheritance is **usually coupled** with subtyping
  - Inheritance of all methods leads to structural subtypes
  - Coupling is also a useful default for nominal subtyping
- Terminology: **Subclassing** = Subtyping + Inheritance

# Simulation of Subclassing with Delegation

- Subclassing can be simulated by a combination of subtyping and aggregation
  - Useful in languages with single inheritance
- OO-programming can do without inheritance, but not without subtyping
- Inheritance is **not a core concept**



# Simulation of Subclassing: Example

```
interface Person {  
    void print( );  
}
```

```
interface Student extends Person {  
    int getRegNum( );  
}
```

Subtyping

```
class PersonImpl  
    implements Person {  
    String name;  
    void print( ) { ... }  
    PersonImpl( String n ) { name = n; }  
}
```

Subtyping

```
class StudentImpl implements Student {  
    Person p;  
    int regNum;  
    StudentImpl( String n, int rn ) { p = new PersonImpl( n ); regNum = rn; }  
    int getRegNum( ) { return regNum; }  
    void print( ) { p.print( ); System.out.println( regNum ); }  
}
```

Aggregation

Subtyping

Delegation

Specialization

# Subtyping, Inheritance, and Subclassing

```
class Circle {  
    Point center;  
    float radius;  
  
    float getRadius( ) { ... }  
    float area( ) { ... }  
}
```

```
class Ellipse {  
    Point center;  
    float radiusA;  
    float radiusB;  
  
    float getRadiusA( ) { ... }  
    float getRadiusB( ) { ... }  
  
    float area( ) { ... }  
    void rotate( ) { ... }  
}
```

- How to define the subtype, inheritance, and subclass relationship between these two classes?

# Circles and Ellipses: Subtyping

- A circle **is an** ellipse!

```
class Circle <: Ellipse {  
    Point center;  
    float radius;  
    float getRadius( ) { ... }  
    float area( ) { ... }  
}
```

```
class Ellipse {  
    Point center;  
    float radiusA;  
    float radiusB;  
    float getRadiusA( ) { ... }  
    float getRadiusB( ) { ... }  
    float area( ) { ... }  
    void rotate( ) { ... }  
}
```

- We need to **enrich Circle's interface**
  - Methods getRadiusA and getRadiusB return radius
  - Method rotate does nothing for Circle



# Circles and Ellipses: Inheritance

- An ellipse has more features than a circle

```
class Circle {  
    Point center;  
    float radius;  
    float getRadius( ) { ... }  
    float area( ) { ... }  
}
```

```
class Ellipse inherits Circle {  
  
    float radiusB;  
    float getRadiusB( ) { ... }  
    float area( ) { ... }  
    void rotate( ) { ... }  
}
```

- Reuse center, radius, and getRadius
- Add extra fields and operations to Ellipse
- Override method area

# Circles and Ellipses: Subclassing

- Subclassing includes subtyping
  - We must have an “is-a” relation

```
class Circle extends Ellipse {  
    // invariant radiusA == radiusB
```

```
    Circle( int r ) {  
        radiusA = r;  
        radiusB = r;  
    }  
}
```

Possibly override  
rotate to improve  
performance

```
class Ellipse {  
    Point center;  
    float radiusA;  
    float radiusB;  
  
    float area( ) {  
        return radiusA * radiusB * 3.14;  
    }  
  
    void rotate( ) {  
        // swap radiusA and radiusB  
    }  
}
```

# Sets and Bounded Sets

```
class Set {  
    int size; // number of elements  
  
    ...  
  
    void add( Object o ) {  
        // add o to set  
    }  
  
    boolean contains( Object o ) { ... }  
}
```

```
class BoundedSet {  
    int size; // number of elements  
    int capacity; // maximum number  
  
    ...  
  
    void add( Object o ) {  
        // add o if there is still space  
    }  
  
    boolean contains( Object o ) { ... }  
}
```

- How to define the subtype, inheritance, and subclass relationship between these two classes?

# Subtyping: BoundedSet <: Set

- BoundedSet specializes add method
- Precondition of add is strengthened
- Clients using Set might fail when using a BoundedSet
- BoundedSet is not a behavioral subtype of Set

```
class Set {  
    ...  
    // requires true  
    // ensures contains( o )  
    void add( Object o ) { ... }  
}
```

```
class BoundedSet extends Set {  
    int size, capacity;  
    // requires size < capacity  
    // ensures contains( o )  
    void add( Object o ) {  
        if ( size < capacity ) super.add( o );  
    }  
}
```

# Subtyping: BoundedSet <: Set (cont'd)

```
class Set {  
  ...  
  // requires true  
  // ensures result => contains( o )  
  boolean add( Object o ) {  
    ...;  
    return true;  
  }  
}
```

Does not  
specify result

```
class BoundedSet extends Set {  
  int size, capacity;  
  
  // requires true  
  // ensures result => contains( o )  
  // ensures result == old(size < capacity)  
  boolean add( Object o ) {  
    if ( capacity <= size ) return false;  
    return super.add( o );  
  }  
}
```

- Clients cannot rely on the properties of unbounded set (have to test for result of add)

# Subtyping: Set <: BoundedSet

- Set must respect BoundedSet's invariant and history constraint
- Set.add cannot increase capacity when full
- Set is not a behavioral subtype of BoundedSet

```
class BoundedSet {  
    int size, capacity;  
  
    // invariant size <= capacity  
    // constraint old( capacity ) == capacity  
  
    // requires size < capacity  
    // ensures contains( o )  
    void add( Object o ) { ... }  
}
```

```
class Set extends BoundedSet {  
    // requires true  
    // ensures contains( o )  
    void add( Object o ) { ... }  
}
```

# Subtyping: Set <: BoundedSet (cont'd)

- **Hack**: Assign a very high number to capacity in Set
- To maintain invariant, Set.add still requires precondition
- At least for static verification, Set behaves still like a bounded set

```
class BoundedSet {  
    int size, capacity;  
  
    // invariant size <= capacity  
    // constraint old( capacity ) == capacity  
  
    // requires size < capacity  
    // ensures contains( o )  
    void add( Object o ) { ... }  
}
```

```
class Set extends BoundedSet {  
    // requires size < capacity  
    // ensures contains( o )  
    void add( Object o ) { ... }  
}
```

# Discussion

- The presented classes for Set and BoundedSet are not behavioral subtypes
  - Syntactic requirements are met
  - Semantic requirements are not met
  
- Large parts of the implementation are identical
  - This code should be reused



# Solution 1: Aggregation

- BoundedSet **uses** Set
- Method calls are **delegated** to Set
- **No subtype relation**
  - No polymorphism
  - No behavioral subtyping requirements

```
class Set {  
    ...  
    void add( Object o ) { ... }  
    int   size( )       { ... }  
}
```

```
class BoundedSet {  
    Set rep;  
    int capacity;  
  
    void add( Object o ) {  
        if (rep.size( ) < capacity) rep.add( o );  
    }  
  
    int size( ) { return rep.size( ); }  
}
```

# A Variant of the Problem

- Aggregation seems okay for Set and BoundedSet
- Similar examples require subtyping
- Polygons and Rectangles
  - Polygon: Unbounded set of vertices
  - Rectangle: Bounded set of (exactly four) vertices
  - A rectangle is a polygon!

```
class Polygon {  
    Vertex[ ] vertices;  
  
    ...  
    void addVertex( Vertex v ) { ... }  
}
```

```
class Rectangle  
    extends Polygon {  
    // vertices contains 4 vertices  
  
    ...  
    void addVertex( Vertex v ) {  
        // unsupported operation  
    }  
}
```

Not what  
we want

## Solution 2: Creating New Objects

```
class Polygon {  
    Vertex[ ] vertices;  
  
    ...  
    // requires true  
    // ensures result.hasVertex( v )  
    Polygon addVertex( Vertex v ) {  
        ... // add v to vertices  
        return this;  
    }  
}
```

```
class Rectangle extends Polygon {  
    // vertices contains 4 vertices  
  
    ...  
    // requires true  
    // ensures result.hasVertex( v )  
    Polygon addVertex( Vertex v ) {  
        return new Pentagon(  
            vertices[ 0 ], vertices[ 1 ],  
            vertices[ 2 ], vertices[ 3 ], v );  
    }  
}
```

```
void foo ( Polygon[ ] p, Vertex v ) {  
    for( int i=0; i < p.length; i++ ) { p[ i ].addVertex( v ).display( ); }  
}
```

# Solution 2 for BoundedSet

```
class Set {  
    ...  
    // requires true  
    // ensures result.contains( o )  
    Set add( Object o ) {  
        ... // add o  
        return this;  
    }  
}
```

```
class BoundedSet extends Set {  
    int size, capacity;  
    // requires true  
    // ensures result.contains( o )  
    Set add( Object o ) {  
        if ( size < capacity )  
            return super.add( o );  
        else {  
            Set res = new Set( );  
            res.addAll( this );  
            res.add( o );  
            return res;  
        }  
    }  
}
```

# Discussion of Solution 2

- `BoundedSet.add` may return `Set` object
- No problem for polymorphic client code
- Error-prone for clients of `BoundedSet`
  - Dynamic type checks necessary

`BoundedSet:`

```
Set add( Object o )
```

```
static Set union( Set a, Set b ) {  
    Set res = new Set( );  
    for( Object e: a ) { res = res.add( e ); }  
    for( Object e: b ) { res = res.add( e ); }  
    return res;  
}
```

```
BoundedSet bs = ...;  
bs = ( BoundedSet ) bs.add( "X" );  
int c = bs.getCapacity( );
```

- Most likely not what users of `BoundedSet` want

# Solution 3: Weak Superclass Contract

- Behavioral subtyping is **relative to a contract**
- Introduce superclass with very **weak contract**
- Strengthen subtype contracts via postconditions

```
abstract class AbstractSet {  
    // invariant true  
    // constraint true  
    ...  
    // requires true  
    // ensures true  
    void add( Object o ) { // add o to set }  
}
```

How to reason about client code?

```
class BoundedSet extends AbstractSet {  
    // requires true  
    // ensures old( size < capacity ) & contains( o )  
    void add( Object o )  
    { if ( size < capacity ) super.add( o ); }  
}
```

How to show that call establishes postcondition?

# Discussion of Solution 3

- Static contracts specify a given method implementation
- Used only for statically-bound calls
  - super-calls
- No behavioral subtyping needed

```
abstract class AbstractSet {  
    ...  
    // requires true  
    // ensures true  
    // static requires true  
    // static ensures contains( o )  
    void add( Object o ) { // add o to set }  
}
```

```
class BoundedSet extends AbstractSet {  
    // requires true  
    // ensures old( size < capacity ) => contains( o )  
    void add( Object o )  
    { if ( size < capacity ) super.add( o ); }  
}
```

# Solution 4: Inheritance w/o Subtyping

- Some languages support **inheritance without subtyping**
  - C++:  
private and protected inheritance
  - Eiffel: non-conforming inheritance

- **No polymorphism**

```
void foo( BoundedSet b ) {  
    Set s = b; // compile-time error  
}
```

C++

```
class Set {  
public:  
    // requires true  
    // ensures contains( o )  
    void add( int o ) { ... }  
    bool contains( int o ) { ... }  
    ...  
}
```

C++

```
class BoundedSet : private Set {  
public:  
    void add( int o ) { ... }  
    Set::contains  
    ...  
}
```

Make method  
public

Override  
method



# Aggregation vs. Private Inheritance

- Both solutions allow code reuse without establishing a subtype relation
  - No subtype polymorphism
  - No behavioral subtyping requirements
- Aggregation causes more overhead
  - Two objects at run-time
  - Boilerplate code for delegation
  - Access methods for protected fields
- Private inheritance may lead to unnecessary multiple inheritance

# 3. Inheritance

3.1 Inheritance and Subtyping

3.2 Dynamic Method Binding

3.3 Multiple Inheritance

3.4 Traits

# Method Binding

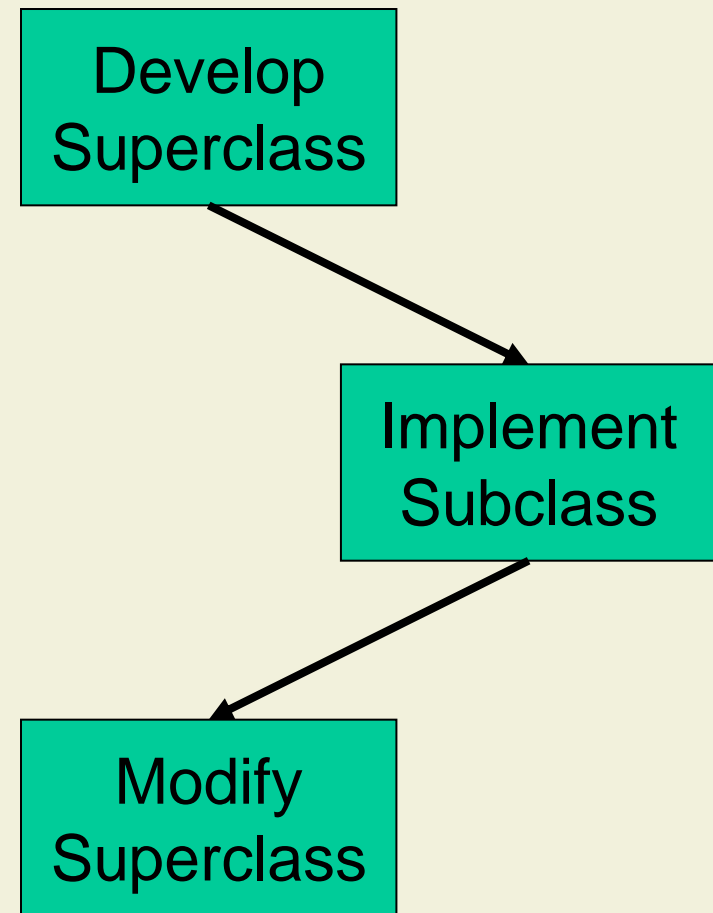
- Static binding:  
At compile time, a method declaration is selected for each call based on the static type of the receiver expression
- Dynamic binding:  
At run time, a method declaration is selected for each call based on the dynamic type of the receiver object

# Static vs. Dynamic Method Binding

- Dynamic method binding enables specialization and subtype polymorphism
- However, there are important drawbacks
  - **Performance**: Overhead of method look-up at run-time
  - **Versioning**: Dynamic binding makes it harder to evolve code without breaking subclasses
- Defaults
  - Dynamic binding: Eiffel, Java, Scala, dynamically-typed languages
  - Static binding: C++, C#

# Fragile Baseclass Scenario

- Software is not static
  - Maintenance
  - Bugfixing
  - Reengineering
- Subclasses can be affected by changes to superclasses
- How should we apply inheritance to make our code robust against revisions of superclasses?



# Example 1: Selective Overriding

```
class Bag {  
    ...  
    int getSize( ) {  
        // count elements  
    }  
  
    void add( Object o )  
    { ... }  
  
    void addAll( Object[ ] arr ) {  
        for( int i=0; i < arr.length; i++ )  
            add( arr[ i ] );  
    }  
}
```

```
class CountingBag extends Bag {  
    int size;  
  
    int getSize( )  
    { return size; }  
    void add( Object o )  
    { super.add( o ); size++; }  
}
```

```
Object[ ] oa = ... // 5 elements  
CountingBag cb =  
    new CountingBag( );  
cb.addAll( oa );  
System.out.println( cb.getSize( ) );
```

# Example 1: Selective Overriding (cont'd)

```
class Bag {  
    ...  
    int getSize( ) {  
        // count elements  
    }  
  
    void add( Object o )  
        { ... }  
  
    void addAll( Object[ ] arr ) {  
        // add elements of arr  
        // directly (not using add)  
    }  
}
```

```
class CountingBag extends Bag {  
    int size;  
  
    int getSize( )  
        { return size; }  
    void add( Object o )  
        { super.add( o ); size++; }  
}
```

```
Object[ ] oa = ... // 5 elements  
CountingBag cb =  
                new CountingBag( );  
cb.addAll( oa );  
System.out.println( cb.getSize( ) );
```

# Example 1: Discussion

```
class Bag {
```

```
...
```

```
int getSize
```

```
... // cou
```

```
}
```

```
// requires true
```

```
// ensures  $\forall i. 0 \leq i < \text{arr.length}$ :
```

```
// contains( arr[ i ] )
```

```
void addAll( Object[ ] arr ) {
```

```
  for( int i=0; i < arr.length; i++ )
```

```
    add( arr[ i ] );
```

```
}
```

```
}
```

Subclass: Using inheritance, rely on interface documentation, not on implementation

Superclass: Do not change calls to dynamically-bound methods

```
class CountingBag extends Bag {
```

```
  int size;
```

```
  // invariant size==super.getSize( )
```

```
  ...
```

```
  void add( Object o )
```

```
  { super.add( o ); size++; }
```

```
  void addAll( Object[ ] arr ) {
```

```
    for( int i=0; i < arr.length; i++ )
```

```
      add( arr[ i ] );
```

```
  }
```

```
}
```

Subclass: Override all methods that could break invariants



## Example 2: Unjustified Assumptions

```
class Math {
```

```
    float squareRt( float f ) {  
        return  $\sqrt{f}$ ;  
    }
```

```
    float fourthRt( float f ) {  
        return  $\sqrt{\sqrt{f}}$ ;  
    }  
}
```

```
class MyMath extends Math {
```

```
    float squareRt( float f ) {  
        return  $-\sqrt{f}$ ;  
    }  
}
```

```
MyMath m = new MyMath( );  
System.out.println  
    ( m.fourthRt( 16 ) );
```

## Example 2: Unjustified Assumptions (c'd)

```
class Math {  
    // requires  $f \geq 0$   
    // ensures  $\text{result}^2 = f$   
    float squareRt( float f ) {  
        return  $\sqrt{f}$ ;  
    }  
    // requires  $f \geq 0$   
    // ensures  $\text{result}^4 = f$   
    float fourthRt( float f ) {  
        return squareRt( squareRt( f ) );  
    }  
}
```

Rely on interface documentation of dynamically-bound method, not on implementation

```
class MyMath extends Math {  
    // requires  $f \geq 0$   
    // ensures  $\text{result}^2 = f$   
    float squareRt( float f ) {  
        return  $-\sqrt{f}$ ;  
    }  
}
```

Superclass: Do not change calls to dynamically-bound methods

```
MyMath m = new MyMath( );  
System.out.println  
    ( m.fourthRt( 16 ) );
```

# Example 3: Mutual Recursion

```
class C {  
    int x;  
  
    void inc1( ) {  
        x = x + 1;  
    }  
  
    void inc2( ) {  
        x = x + 1;  
    }  
}
```

```
class CS extends C {  
  
    void inc2( ) {  
        inc1( );  
    }  
}
```

```
CS cs = new CS( );  
cs.x = 5;  
cs.inc2( );  
System.out.println( cs.x );
```

## Example 3: Mutual Recursion (cont'd)

```
class C {  
    int x;  
    // requires true  
    // ensures x = old( x ) + 1  
    void inc1( ) {  
        inc2( );  
    }  
    // requires true  
    // ensures x = old( x ) + 1  
    void inc2( ) {  
        x = x + 1;  
    }  
}
```

Superclass: Do not change  
calls to dynamically-bound  
methods

```
class CS extends C {  
    // requires true  
    // ensures x = old( x ) + 1  
    void inc2( ) {  
        inc1( );  
    }  
}
```

Subclass: Avoid  
specializing classes  
that are expected to be  
changed (often)

```
CS cs = new CS( );  
cs.x = 5;  
cs.inc2( );  
System.out.println( cs.x );
```

```
class DiskMgr {  
  
    void cleanUp( ) {  
        ... // remove temporary files  
    }  
}
```

```
class MyMgr extends DiskMgr {  
    void delete( ) {  
        ... // erase whole hard disk  
    }  
}
```

```
MyMgr mm = new MyMgr( );  
...  
mm.cleanUp( );
```

## Example 4: Additional Methods (cont'd)

```
class DiskMgr {  
    void delete( ) {  
        ... // remove temporary files  
    }  
  
    void cleanUp( ) {  
        delete( );  
    }  
}
```

Superclass: Do not change calls to dynamically-bound methods

```
class MyMgr extends DiskMgr {  
    void delete( )  
        ... // erase whole hard disk  
    }  
}
```

Subclass: Avoid specializing classes that are expected to be changed (often)

```
MyMgr mm = new MyMgr( );  
...  
mm.cleanUp( );
```

## Example 4: Additional Methods (cont'd)

```
class DiskMgr {  
    virtual void delete( ) {  
        ... // remove temporary files  
    }  
  
    void cleanUp( ) {  
        delete( );  
    }  
}
```

C#

```
class MyMgr : DiskMgr {  
    new void delete( ) {  
        ... // erase whole hard disk  
    }  
}
```

C#

```
MyMgr mm = new MyMgr( );  
...  
mm.cleanUp( );
```

C#

- In C#, methods are bound statically by default
- Potential overrides must be declared as either **override** or **new**
  - Prevents accidental overriding

# Example 5: Additional Methods

```
class Super {
```

```
}
```

Java

```
class Sub extends Super {
```

```
    void foo( Object o ) { ... }
```

```
    void bar( double i ) { ... }
```

```
}
```

Java

```
Sub s = new Sub( );
```

```
s.foo( "Java" );
```

```
s.bar( 5 );
```

Java



## Example 5: Additional Methods (cont'd)

```
class Super {  
    void foo( String o ) { ... }  
    void bar( int i ) { ... }  
}
```

Java

```
class Sub extends Super {  
    void foo( Object o ) { ... }  
    void bar( double i ) { ... }  
}
```

Java

```
Sub s = new Sub( );  
s.foo( "Java" );  
s.bar( 5 );
```

Java

- Overloading resolution in Java chooses **most specific** method declaration
- Adding methods to a superclass may affect clients of subclasses
  - Even without overriding

## Example 5: Additional Methods (cont'd)

```
class Super {  
    void foo( string o ) { ... }  
    void bar( int i ) { ... }  
}
```

C#

```
class Sub : Super {  
    void foo( object o ) { ... }  
    void bar( double i ) { ... }  
}
```

C#

```
Sub s = new Sub( );  
s.foo( "C#" );  
s.bar( 5 );
```

C#

- Overloading resolution in C# chooses **most specific** method declaration **in the class of the receiver**
  - Then superclass, etc.
- Adding methods to a superclass does not affect overloading resolution

# Summary: Rules for Proper Subclassing

- Use subclassing only if there is an “is-a” relation
  - Syntactic and **behavioral** subtypes
- Do not rely on implementation details
  - Use **precise documentation** (**contracts** where possible)
- When evolving superclasses, **do not mess around with dynamically-bound methods**
  - Do not add or remove calls, or change order of calls
- Do not specialize superclasses that are expected to change often

# Binary Methods

- Binary methods take receiver and one explicit argument
- Often behavior should be specialized depending on the dynamic types of both arguments
- Recall that covariant parameter types are not statically type-safe

```
class Object {  
    boolean equals( Object o ) {  
        return this == o;  
    }  
}
```

```
class Cell {  
    int val;  
    boolean equals( Cell o ) {  
        return this.val == o.val;  
    }  
}
```

# Binary Methods: Example

- **Dynamic binding for specialization based on dynamic type of receiver**
- How to specialize based on dynamic type of explicit argument?

```
class Shape {  
    Shape intersect( Shape s ) {  
        // general code for all shapes  
    }  
}
```

```
class Rectangle extends Shape {  
    Shape intersect( Rectangle r ) {  
        // efficient code for two rectangles  
    }  
}
```

# Solution 1: Explicit Type Tests

- Type test and conditional for specialization based on dynamic type of explicit argument
- Problems
  - Tedious to write
  - Code is not extensible
  - Requires type cast

```
class Rectangle extends Shape {  
    Shape intersect( Shape s ) {  
        if( s instanceof Rectangle ) {  
            Rectangle r = ( Rectangle ) s;  
            // efficient code for two rectangles  
        } else {  
            return super.intersect( s );  
        }  
    }  
}
```

## Solution 2: Double Invocation

```
class Shape {  
    Shape intersect( Shape s )  
    { return s.intersectShape( this ); }  
  
    Shape intersectShape( Shape s )  
    { // general code for all shapes }  
  
    Shape intersectRectangle( Rectangle r )  
    { return intersectShape( r ); }  
}
```

- Additional dynamically-bound call for specialization based on dynamic type of explicit argument

```
class Rectangle extends Shape {  
    Shape intersect( Shape s )  
    { return s.intersectRectangle( this ); }  
  
    Shape intersectRectangle( Rectangle r )  
    { // efficient code for two rectangles }  
}
```

## Solution 2: Double Invocation (cont'd)

Corresponds to  
Node and Visitor

Corresponds to  
Node.accept

- Double invocation is also called  
**Visitor Pattern**

```
class Shape {  
    Shape intersect( Shape s )  
    { return s.intersectShape( this ); }  
  
    Shape intersectShape( Shape s )  
    { // general code for all shapes }  
  
    Shape intersectRectangle( Rectangle r )  
    { return intersectShape( r ); }  
}
```

Corresponds to  
Visitor.visitX

- Problems
  - Even more tedious to write
  - Requires modification of superclass  
(not possible for equals method)



# Solution 3: Multiple Dispatch

- Some research languages allow method calls to be bound based on the **dynamic type of several arguments**
- Examples: CLU, Cecil, Fortress, MultiJava

```
class Shape {  
    Shape intersect( Shape s ) {  
        // general code for all shapes  
    }  
}
```

```
class Rectangle extends Shape {  
    Shape intersect( Shape@Rectangle r ) {  
        // efficient code for two rectangles  
    }  
}
```

Static type  
of r

Dispatch  
on r

## Solution 3: Multiple Dispatch (cont'd)

- Multiple dispatch is statically type-safe

```
Shape client( Shape s1, Shape s2) {  
    return s1.intersect( s2 );  
}
```

Calls `Rectangle.intersect`  
only if `s1` and `s2` are of  
type `Rectangle`

- Problems

- Performance overhead of method look-up at run time
- Extra requirements are needed to ensure there is a “unique best method” for every call

# Binary Methods: Summary

- The behavior of binary methods often depends on the dynamic types of both arguments
- Type tests
  - One single-dispatch call and one case distinction
- Double invocation (Visitor Pattern)
  - Two single-dispatch calls
- Multiple dispatch
  - One multiple-dispatch call

# 3. Inheritance

3.1 Inheritance and Subtyping

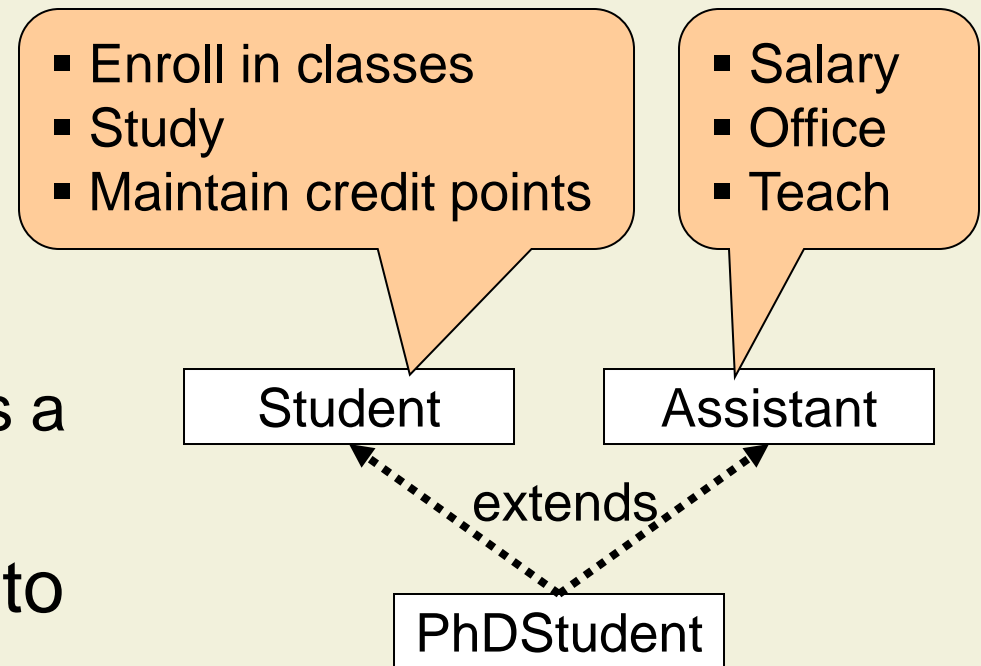
3.2 Dynamic Method Binding

3.3 Multiple Inheritance

3.4 Traits

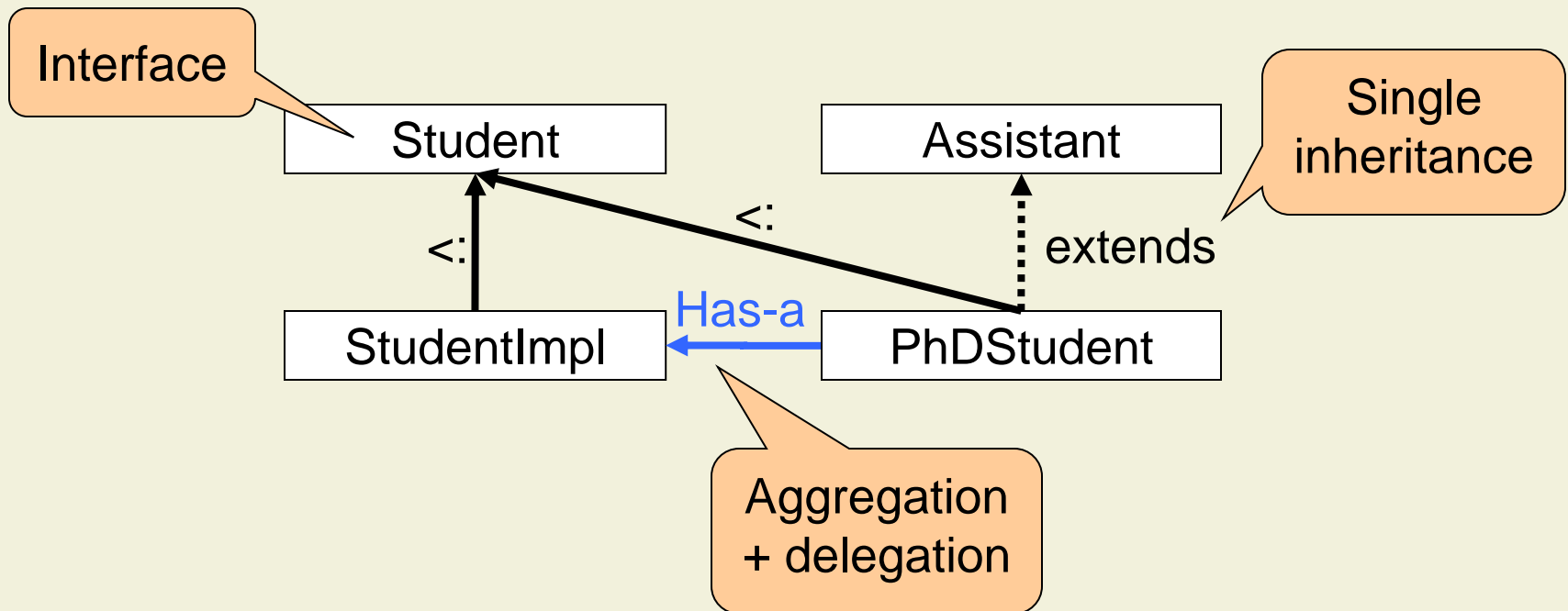
# Motivation

- All object-oriented languages support multiple subtyping
  - One type can have several supertypes
  - Subtype relation forms a DAG
- Often it is also useful to **reuse code from several superclasses** via multiple inheritance



# Simulating Multiple Inheritance

- Java and C# support only single inheritance
- Multiple inheritance is simulated via delegation
  - Not elegant



# Problems of Multiple Inheritance

- Ambiguities

- Superclasses may contain fields and methods with identical names and signatures
- Which version should be available in the subclass?

- Repeated inheritance (diamonds)

- A class may inherit from a superclass more than once
- How many copies of the superclass members are there?
- How are the superclass fields initialized?

# Ambiguities: Example

```
class Student {  
    public:  
        Professor* mentor;  
        virtual int workLoad( ) { ... }  
        ... };
```

```
class Assistant {  
    public:  
        Professor* mentor;  
        virtual int workLoad( ) { ... }  
        ... };
```

```
class PhDStudent :  
    public Student, public Assistant {  
};
```

```
void client( PhDStudent p ) {  
    int w = p.workLoad( );  
    p.mentor = NULL;  
}
```

Which method  
should be called?

Which field  
should be  
accessed?



# Ambiguity Resolution: Explicit Selection

```
class Student {  
    public:  
        Professor* mentor;  
        virtual int workLoad( ) { ... }  
        ... };
```

C++

```
class Assistant {  
    public:  
        Professor* mentor;  
        virtual int workLoad( ) { ... }  
        ... };
```

C++

```
class PhDStudent :  
    public Student, public Assistant {  
};
```

C++

```
void client( PhDStudent p ) {  
    int w = p.Assistant::workLoad( );  
    p.Student::mentor = NULL;  
}
```

- Subclass has two members with identical names
- Ambiguity is resolved by client
- Clients need to know implementation details

# Ambiguity Resolution: Merging Methods

Overrides both  
inherited methods

```
class PhDStudent {  
    public Student, public Assistant {  
public:  
    virtual int workLoad( ) {  
        return Student::workLoad( ) +  
            Assistant::workLoad( );  
    }  
};
```

Correspond to  
super-calls in Java

```
void client( PhDStudent p ) {  
    int w = p.workLoad( );  
}
```

- Related inherited methods can often be merged into one overriding method
- Usual rules for overriding apply
  - Type rules
  - Behavioral subtyping

# Merging Unrelated Methods

```
class Student {  
public:  
    virtual bool test( ) { // take exam }  
    ... };
```

C++

```
class Assistant {  
public:  
    virtual bool test( ) { // unit test }  
    ... };
```

Clients can call  
Assistant::test

```
class PhDStudent :  
    public Student, public Assistant {  
public:  
    virtual bool test( )  
    { return Student::test( ); }  
};
```

C++

Violates  
behavioral  
subtyping

- Unrelated methods cannot be merged in a meaningful way
  - Even if signatures match
- Subclass should provide both methods, but with different names

# Ambiguity Resolution: Renaming

```
class Student
feature
  test: BOOLEAN do ... end
end
```

Eiffel

```
class Assistant
feature
  test: BOOLEAN do ... end
end
```

Eiffel

```
class PhDStudent inherit
  Student
  rename test as takeExam
  redefine takeExam end
  Assistant
end
```

Eiffel

- Inherited methods can be renamed
- Dynamic binding takes renaming into account

```
client( s: Student ): BOOLEAN
do
  Result := s.test( )
end
```

For PhDStudent  
bound to takeExam

- C++/CLI provides similar features

# Repeated Inheritance: Example

```
class Person {  
    Address address;  
    ...  
};
```

C++

```
class Student : public Person {  
    ...  
};
```

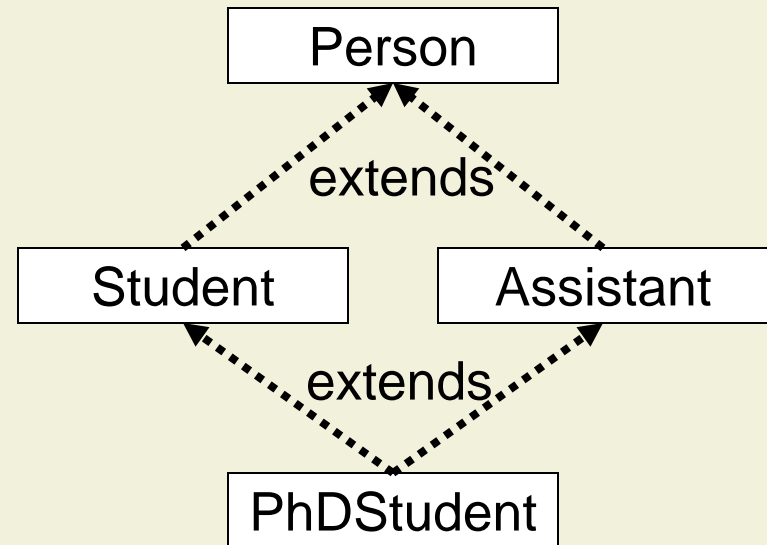
C++

```
class Assistant : public Person {  
    ...  
};
```

C++

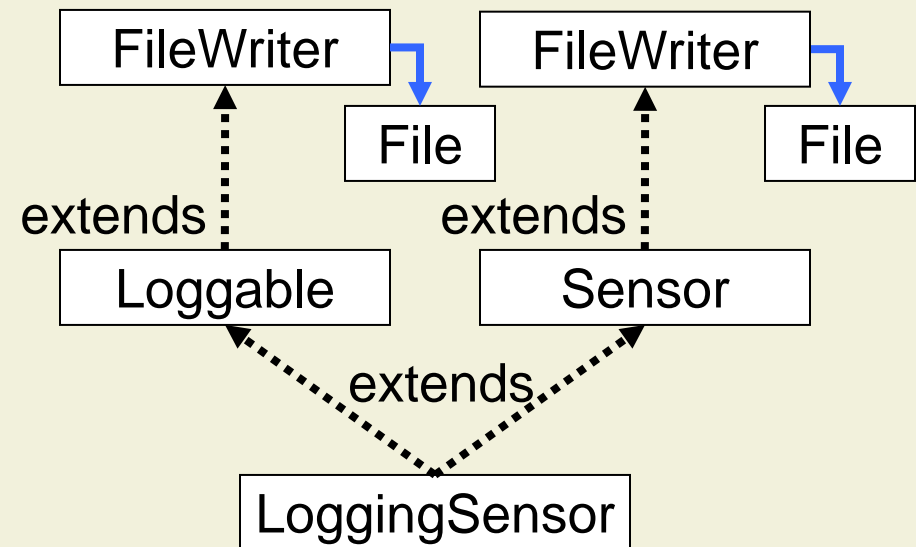
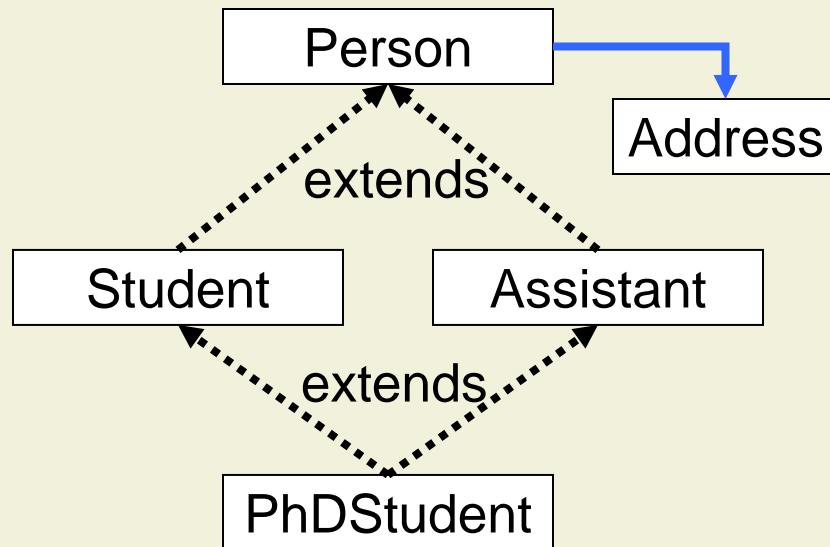
```
class PhDStudent :  
    public Student, public Assistant {  
};
```

C++



- How many address fields should PhDStudent have?
- How are they initialized?

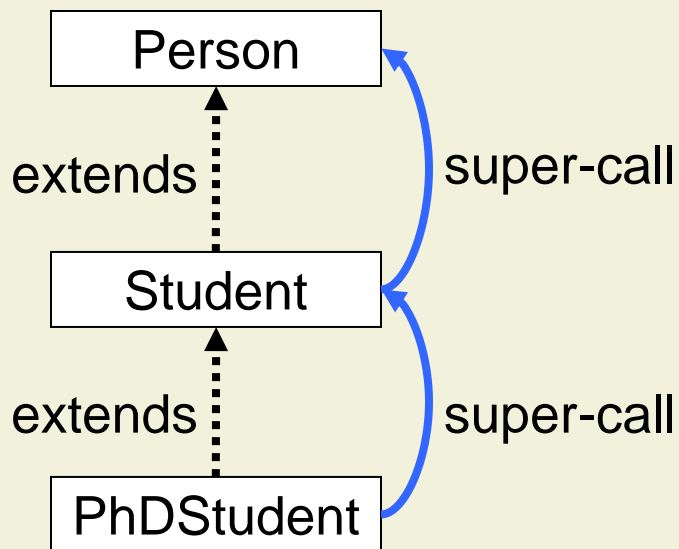
# How Many Copies of Superclass Fields?



- Eiffel: default
- C++: virtual inheritance
- Eiffel: via renaming
- C++: non-virtual inheritance

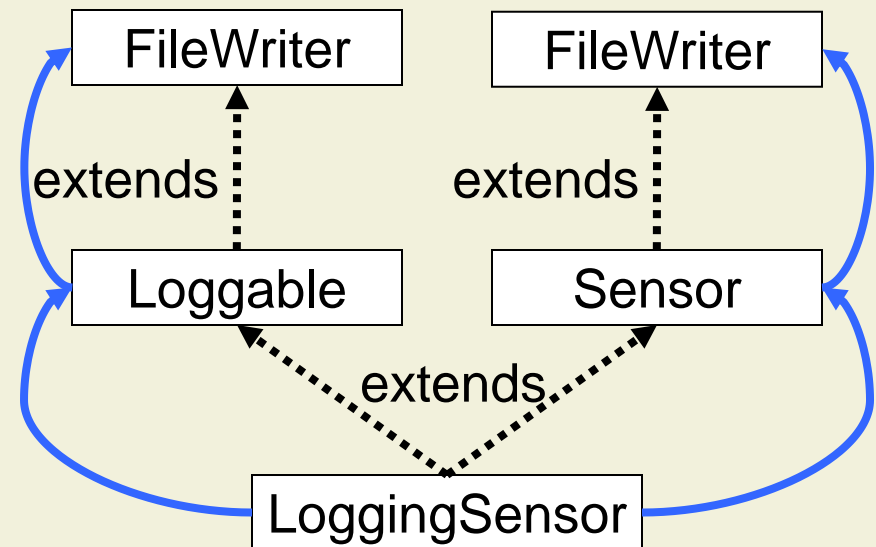
# Inheritance and Object Initialization

- **Superclass fields** are initialized **before subclass fields**
  - Helps preventing use of uninitialized fields, e.g., in inherited methods
- Order is typically implemented via mandatory call of superclass constructor at the beginning of each constructor



# Initialization and Non-Virtual Inheritance

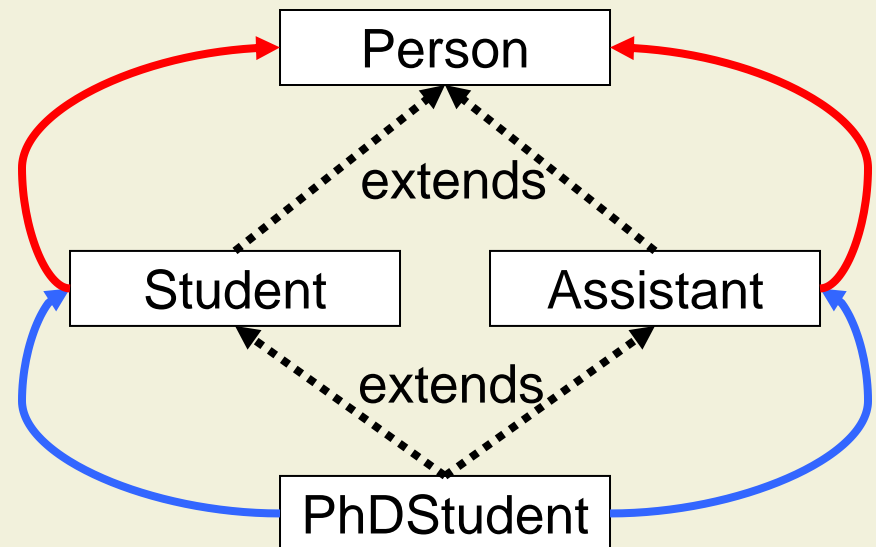
- With non-virtual inheritance, there are **two copies** of the superclass fields
- Superclass **constructor is called twice** to initialize both copies
  - Here, create two file handles for two files





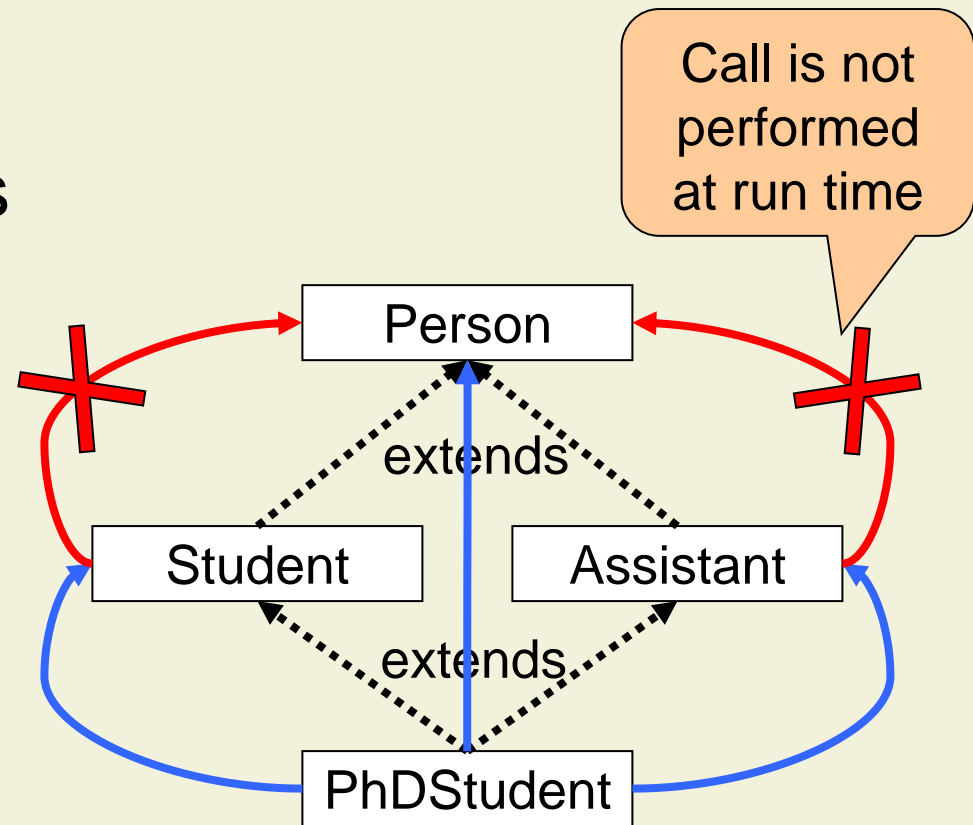
# Initialization and Virtual Inheritance

- With virtual inheritance, there is **only one copy** of the superclass fields
- Who gets to call the superclass constructor?



# Initialization: C++ Solution

- **Constructor** of repeated superclass is **called only once**
- **Smallest subclass** needs to call the constructor of the virtual superclass directly



# C++ Solution: Example

```
class Person {  
    Address* address;  
    int workdays;  
public:  
    Person( Address* a, int w ) {  
        address = a;  
        workdays = w;  
    };  
};
```


```
class Student : virtual public Person {  
public:  
    Student( Address* a ) : Person( a, 5 ) { };  
};
```

```
class Assistant: virtual public Person {  
public:  
    Assistant( Address* a ) : Person( a, 6 ) { };  
};
```

```
class PhDStudent : public Student, public Assistant {  
public:  
    PhDStudent( Address* a ) : Person( a, 7 ), Student( a ), Assistant( a ) { };  
};
```

# C++ Solution: Discussion

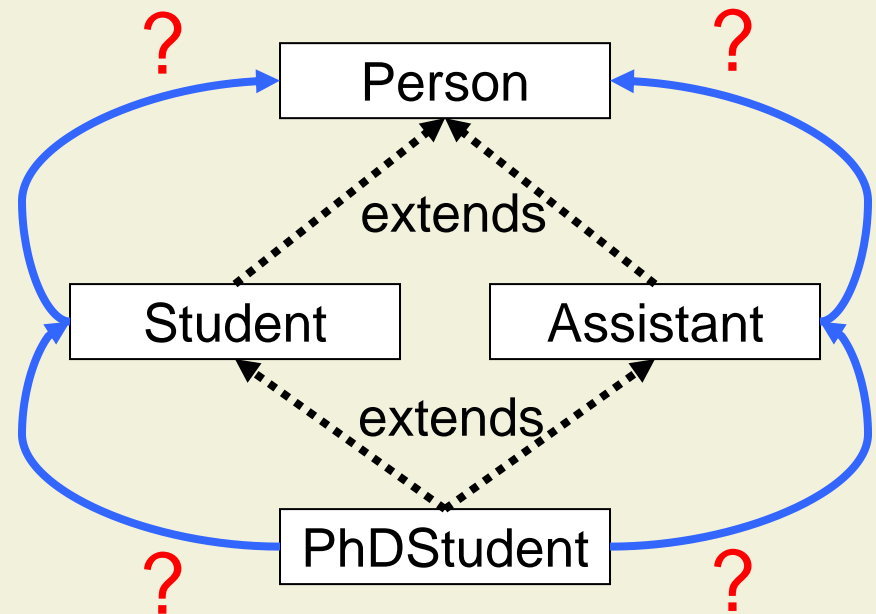
```
class Student : virtual public Person {  
public:  
    Student( Address* a ) : Person( a, 5 ) {  
        assert( workdays == 5 );  
    };  
};
```



- Non-virtual inheritance is the default
  - Virtual inheritance leads to run-time overhead
  - **Programmers need foresight!**
- Constructors **cannot rely on the virtual superclass constructors** they call
  - For instance, to establish invariants

# Initialization: Eiffel Solution

- Eiffel **does not force** constructors to call superclass constructors
- Programmer has **full control** over calls to superclass constructors



# Eiffel Solution: Discussion

## No call of superclass constructor

- Subclasses have to initialize inherited fields
  - Code duplication
- Subclasses need to understand superclass implementation

## Policy: Always call superclass constructor

- Constructors of repeated superclasses get called twice
- What if these super-calls have different arguments?
- Problematic if constructor has side-effects

# Renaming Revisited

```
class Person  
feature  
  foo: BOOLEAN do ... end  
end
```

Eiffel

```
class Student inherit  
  Person rename foo as bar end  
end
```

Eiffel

```
class Assistant inherit Person  
end
```

Eiffel

```
class PhDStudent inherit  
  Student redefine bar select bar end  
  Assistant  
... end
```

Eiffel

```
client( p: Person ): BOOLEAN  
do  
  Result := p.foo( )  
end
```

For PhDStudent,  
call foo or bar?

# Multiple Inheritance

## Pros

- Increases expressiveness
- Avoids overhead of delegation pattern

## Cons

- Ambiguity resolution
  - Explicit selection
  - Merging
  - Renaming
- Repeated inheritance
  - Complex semantics
  - Initialization
  - Renaming
- Complicated!



# 3. Inheritance

3.1 Inheritance and Subtyping

3.2 Dynamic Method Binding

3.3 Multiple Inheritance

3.4 Traits

# Mixins and Traits

- Mixins and traits provide a form of reuse
  - Methods and state that can be **mixed into various classes**
  - Example: Functionality to persist an object
- Main applications
  - Making thin interfaces thick
  - Stackable specializations
- Languages that support mixins or traits: Python, Ruby, Scala, Squeak Smalltalk
  - We will focus on Scala's version of traits

# Scala: Trait Example

```
class Cell {  
  var value: Int = 0  
  
  def put( v: Int ) = { value = v }  
  def get: Int = value  
}
```

Scala

```
trait Backup extends Cell {  
  var backup: Int = 0;  
  
  override def put( v: Int ) = {  
    backup = value  
    super.put( v )  
  }  
  def undo = { super.put( backup ) }  
}
```

Scala

```
object Main1 {  
  def main( args: Array[ String ] ) {  
    val a = new Cell with Backup  
    a.put( 5 )  
    a.put( 3 )  
    a.undo  
    println( a.get )  
  }  
}
```

Scala

# Scala: Declaration of Traits

Traits may have fields

Traits may override superclass methods

Traits may declare methods

```
trait Backup extends Cell {  
  var backup = 0;  
  
  override def put( v: Int ) = {  
    backup = value  
    super.put( v )  
  }  
  def undo = { super.put( backup ) }  
}
```

Scala

Traits extend exactly one superclass (and possibly other traits)

# Scala: Mixing-in Traits

```
class FancyCell extends Cell with Backup {  
  ...  
}
```

Traits can be mixed-in when classes are declared

```
def main( args: Array[String] ) {  
  val a = new Cell with Backup  
  ...  
}
```

Traits can be mixed-in when classes are instantiated

- Class must be a subclass of its traits' superclasses
  - Otherwise we would get multiple inheritance

# Traits and Types

- Each trait defines a type
  - Like classes and interfaces
  - Trait types are abstract
- Extending or mixing-in a trait introduces a subtype relation

```
trait Backup extends Cell {  
  ...  
}
```

Scala

```
class FancyCell  
  extends Cell with Backup {  
  ...  
}
```

Scala

```
val a: Backup = new FancyCell  
val b: Cell = a
```

Scala

# Example: Thin and Thick Interfaces

- Traits can extend thin interfaces by additional operations
- Allows very specific types with little syntactic overhead
  - See structural subtyping

```
class ThinCollection {  
  def add( s: String ) = { ... }  
  def contains( s: String ): Boolean = { ... }  
}
```

```
trait AddAll extends ThinCollection {  
  def addAll( a: Array[String] ) = {  
    val it = a.iterator  
    while( it.hasNext ) { add( it.next ) }  
  }  
}
```

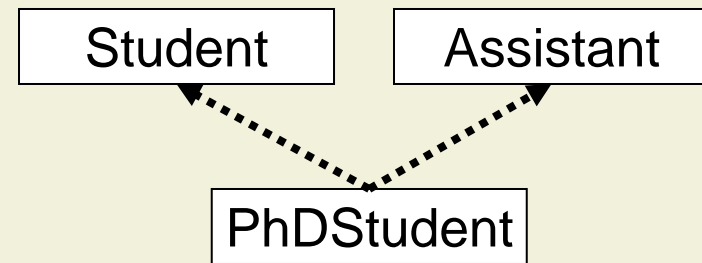
```
def client ( p: ThinCollection with AddAll, a: Array[String] ) = { p.addAll( a ) }
```

# Ambiguity Resolution

```
trait Student {  
  var mentor: Professor  
  def workLoad: Int = 5  
}
```

```
trait Assistant {  
  var mentor: Professor  
  def workLoad: Int = 6  
}
```

```
class PhDStudent  
  extends AnyRef  
  with Student  
  with Assistant { }
```



- Ambiguity is resolved by **merging**
  - No scope-operator like in C++
  - No renaming like in Eiffel



# Ambiguity Resolution (cont'd)

```
trait Student {  
  def workLoad: Int = 5  
}
```

```
trait Assistant {  
  def workLoad: Int = 6  
}
```

- Subclass overrides both mixed-in methods
- Does not work for mutable fields

```
class PhDStudent extends AnyRef with Student with Assistant {  
  override def workLoad: Int = {  
    super[ Student ].workLoad +  
    super[ Assistant ].workLoad  
  }  
}
```

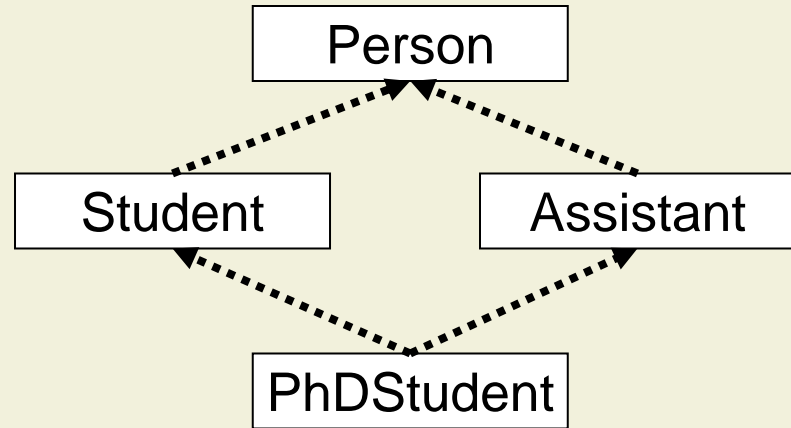
# Ambiguity Resolution and Diamonds

```
class Person {  
  def workLoad: Int = 0  
}
```

```
trait Student extends Person {  
  override def workLoad: Int = 5  
}
```

```
trait Assistant extends Person {  
  override def workLoad: Int = 6  
}
```

```
class PhDStudent  
  extends Person  
  with Student  
  with Assistant { }
```

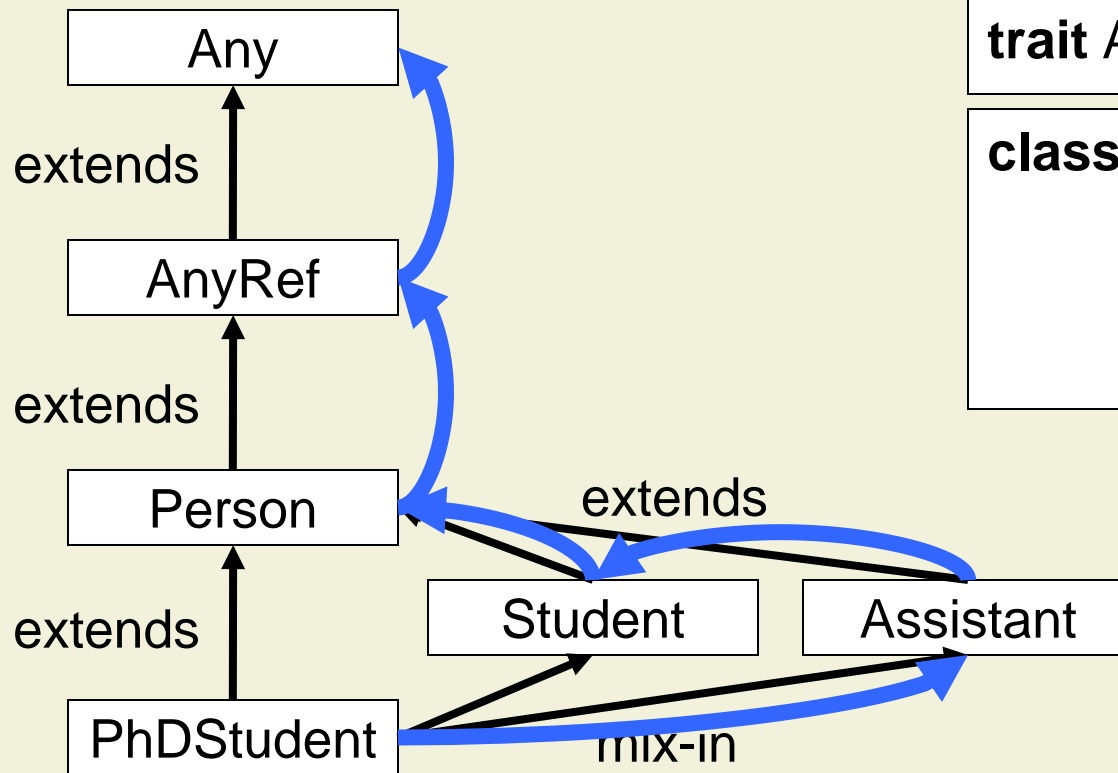


- If two inherited methods override a common superclass method, merging is **not** required
- What is the behavior of `workLoad` in `PhDStudent`?

# Linearization

- The key concept to understanding the semantics of Scala traits
- Bring supertypes of a type in a linear order
- For a class C, compute order from back to front:
  1. Linearize superclass of C
  2. Linearize supertraits of C (in the order of with-clauses)Do not include types that have been linearized already
- Overriding and super-calls are defined according to this linear order

# Linearization Example



```
class Person
```

```
trait Student extends Person
```

```
trait Assistant extends Person
```

```
class PhDStudent  
    extends Person  
    with Student  
    with Assistant
```

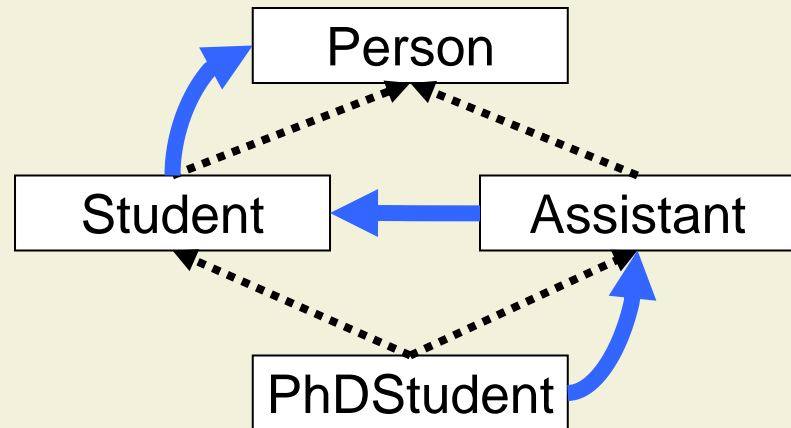
# Overriding and Super-Calls

```
class Person {  
  def workLoad: Int = 0  
}
```

```
trait Student extends Person {  
  override def workLoad: Int = 5  
}
```

```
trait Assistant extends Person {  
  override def workLoad: Int = 6  
}
```

```
class PhDStudent  
  extends Person  
  with Student  
  with Assistant { }
```



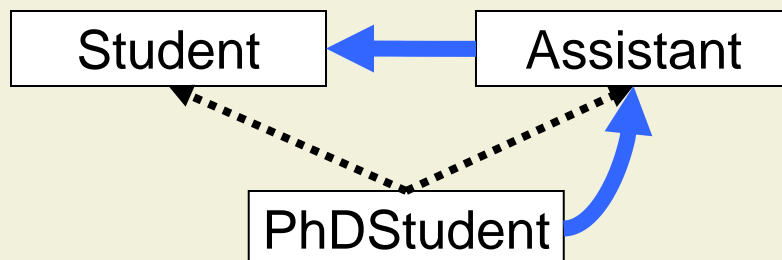
- PhDStudent's workLoad method is inherited from Assistant
  - Assistant's workLoad overrides Student's
  - Student's workLoad overrides Person's

# Overriding and Super-Calls (cont'd)

```
trait Student  
{ def workLoad: Int = 5 }
```

```
trait Assistant  
{ def workLoad: Int = 6 }
```

```
class PhDStudent extends AnyRef  
  with Student with Assistant {  
  def override workLoad: Int =  
    super.workLoad  
}
```



- PhDStudent's workLoad overrides methods from Assistant and Student
  - Super-call refers to predecessor in the linear order, Assistant
- Now Assistant's and Student's workLoad **do not override each other**
  - No super-calls allowed

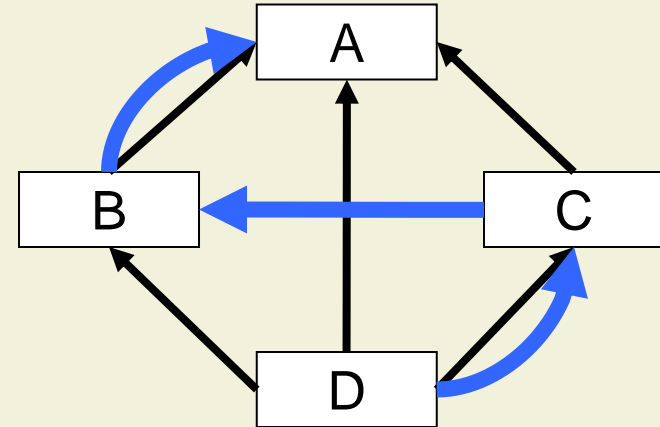
# Repeated Inheritance

```
class A {  
  var f = 0  
  def foo = println( "A::foo" )  
}
```

```
trait B extends A {  
  override def foo = println( "B::foo" )  
}
```

```
trait C extends A {  
  override def foo = println( "C::foo" )  
}
```

```
class D extends A with B with C {  
}
```



- Subclass inherits only **one copy** of repeated superclass
  - Like Eiffel and virtual inheritance in C++

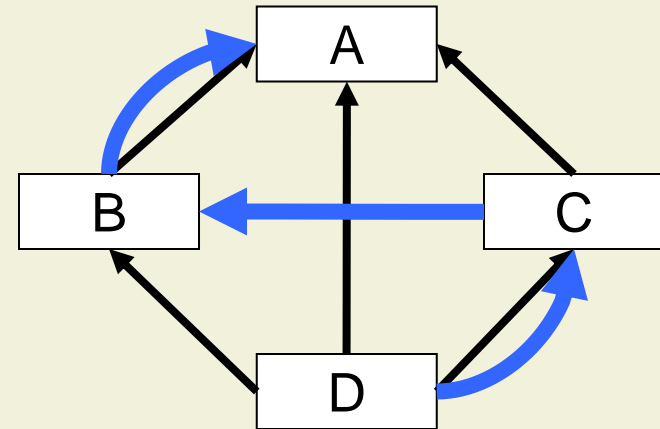
# Initialization Order

```
class A {  
  println( "Constructing A" )  
}
```

```
trait B extends A {  
  println( "Constructing B" )  
}
```

```
trait C extends A {  
  println( "Constructing C" )  
}
```

```
class D extends A with B with C {  
  println( "Constructing D" )  
}
```

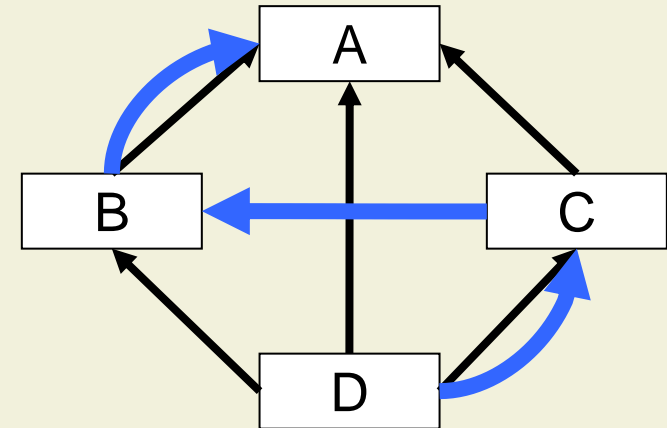


- Classes and traits are initialized in the reverse linear order



# Initialization of Repeated Superclasses

- Each constructor is called **exactly once**
  - Good if constructor has side-effects
- Arguments to superclass constructors are supplied **by immediately preceeding class** in the linearization order



```
class A( x: Int ) {  
  println( "Constructing A" + x )  
}
```

```
trait B extends A { ... }
```

```
trait C extends A { ... }
```

```
class D extends A( 5 )  
  with B with C { ... }
```

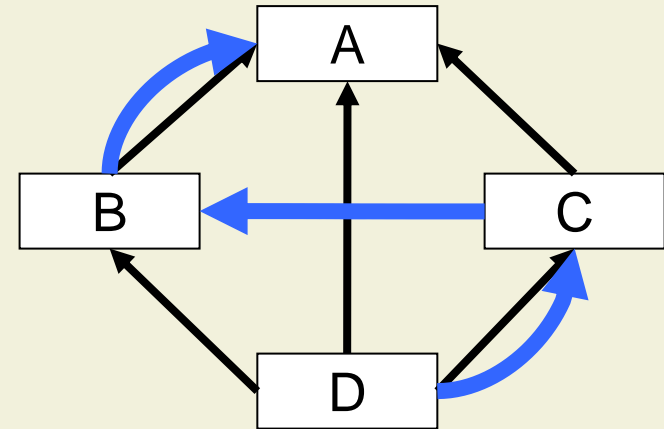
# Initialization of Repeated Superclasses

```
class A( x: Int ) {  
  println( "Constructing A" + x )  
}
```

```
trait B extends A { ... }
```

```
trait C extends A { ... }
```

```
class D extends A( 5 )  
  with B with C { ... }
```



- Each constructor is called **exactly once**
  - Good if constructor has side-effects
- Arguments to superclass constructors are supplied **by preceding class** in the linearization order

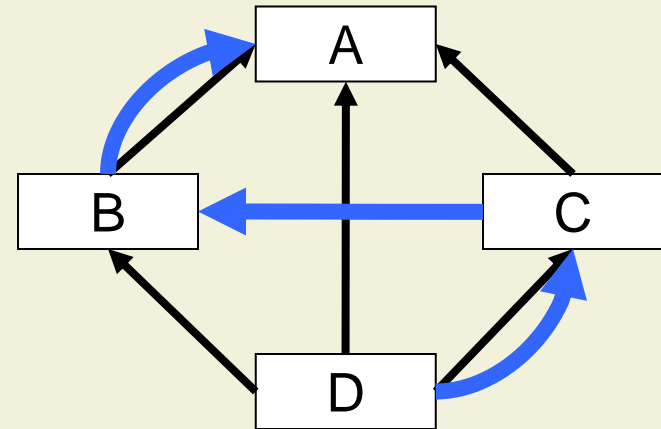
# Overriding and Super-Calls Revisited

```
class A {  
  def foo = println( "A::foo" )  
}
```

```
trait B extends A {  
  override def foo =  
    { println( "B::foo" ); super.foo }  
}
```

```
trait C extends A {  
  override def foo =  
    { println( "C::foo" ); super.foo }  
}
```

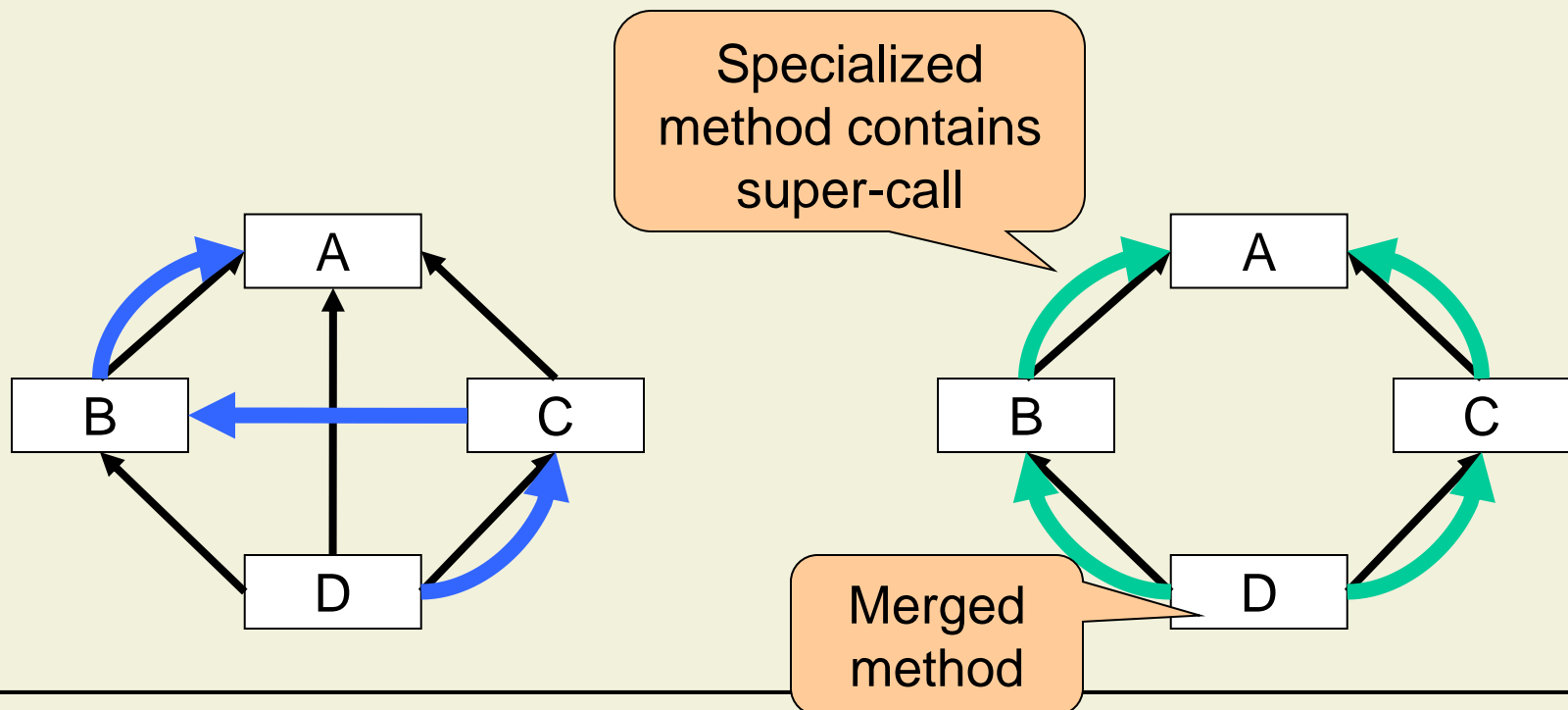
```
class D extends A with B with C { }
```



```
def client ( d: D ) = { d.foo }
```

# Stackable Specializations

- With traits, specializations can be combined in flexible ways
- With multiple inheritance, methods of repeated superclasses are called twice



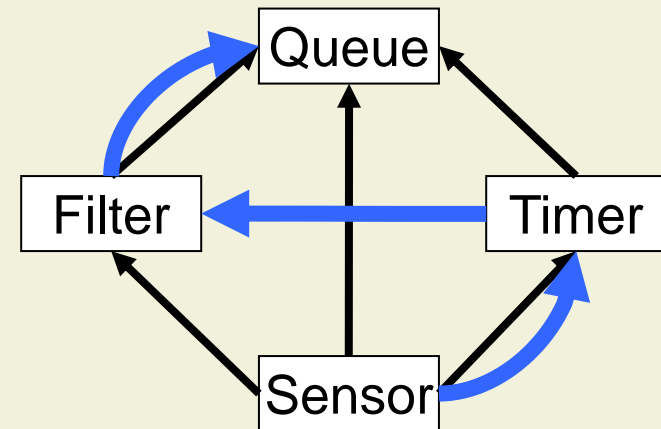
# Stackable Specializations: Example

```
class Queue {  
  ...  
  def put( x: Data ) { ... }  
}
```

```
trait Timer extends Queue {  
  override def put( x: Data )  
  { x.SetTime( ... ); super.put( x ) }  
}
```

```
trait Filter extends Queue {  
  override def put( x: Data )  
  { if( x.Time > ... ) super.put( x ) }  
}
```

```
class SensorData extends Queue  
  with Filter with Timer { }
```



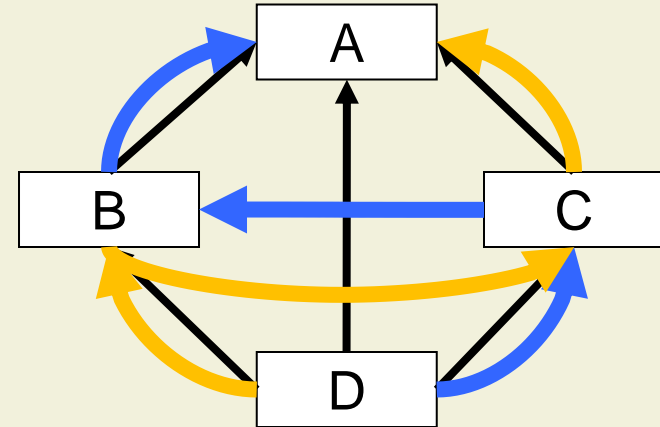
# Traits and Behavioral Subtyping

```
trait B extends A {  
  override def foo =  
    { println( "B::foo" ); super.foo }  
}
```

```
trait C extends A {  
  override def foo =  
    { println( "C::foo" ); super.foo }  
}
```

```
class D extends A with B with C { }
```

```
class D extends A with C with B { }
```



- Overriding of trait methods depends on order of mixing
- Behavioral subtyping could be checked only when traits are mixed in

# Reasoning About Traits

- Traits are very dynamic, which complicates static reasoning
- Traits do not know **how their superclasses get initialized**
- Traits do not know **which methods they override**
- Traits do not know **where super-calls are bound to**

```
trait B extends A {  
  override def foo =  
    { println( "B::foo" ); super.foo }  
}
```

```
trait C extends A {  
  override def foo =  
    { println( "C::foo" ); super.foo }  
}
```

# Traits: Summary

- Traits partly solve problems of multiple inheritance
  - Linearization resolves some issues with ambiguities and initialization
- Other problems remain
  - Resolving ambiguities between unrelated methods
  - Initializing superclasses
- And new problems arise
  - No specification inheritance between trait methods
  - What to assume about superclass initialization and super-calls
- Traits pose several research challenges



# References

- Leonid Mikhajlov, Emil Sekerinski: *A Study of the Fragile Base Class Problem*. LNCS 1445, Springer-Verlag, 1998
- Donna Malayeri and Jonathan Aldrich: *CZ: Multiple Inheritance without Diamonds*. OOPSLA 2009
- James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley: *The Java Language Specification*. 2013  
<http://docs.oracle.com/javase/specs/>
- Bertrand Meyer: *Eiffel: The Language*. Prentice Hall, 1991
- Bjarne Stroustrup: *The C++ Programming Language*. Addison Wesley, 2013
- Martin Odersky, Lex Spoon, and Bill Venners: *Programming in Scala*. Artima, 2008