# Concepts of Object-Oriented Programming

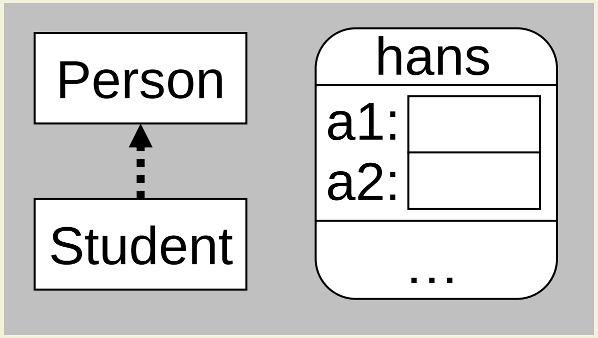
**Peter Müller**
Programming Methodology Group

Autumn Semester 2024

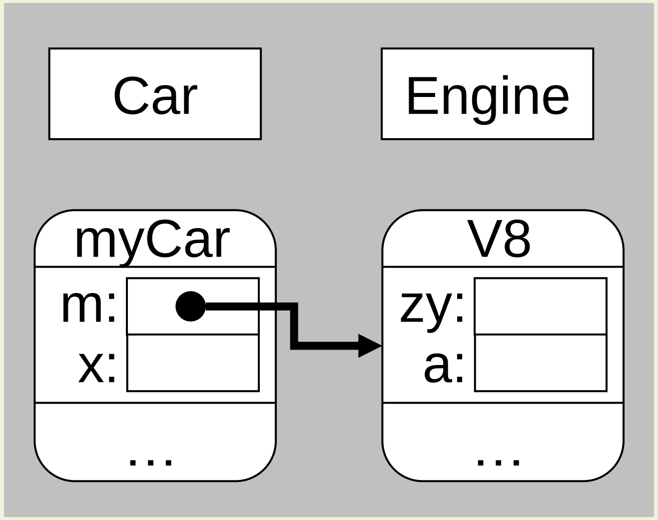
ETH zürich

# 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

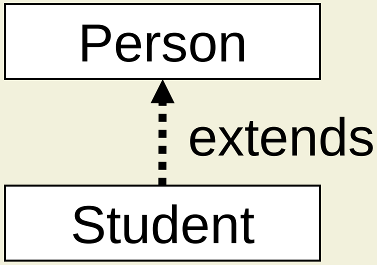
# Inheritance versus Subtyping

- 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

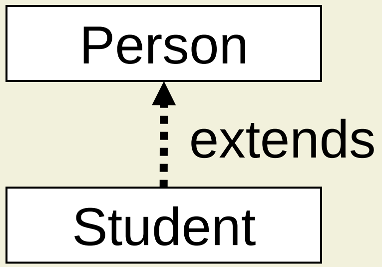
- Terminology: Subclassing = Subtyping + Inheritance

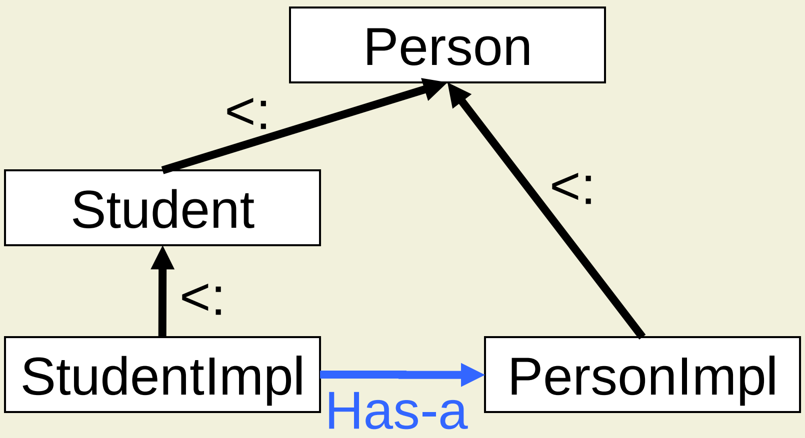
# Simulation of Subclassing with Delegation



# 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( );
}
```

# Simulation of Subclassing: Example

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

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

# Simulation of Subclassing: Example

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

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

Subtyping

# 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; }
}
```

# 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

# 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 ); }
}
```

# 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 ); }
}
```

Subtyping

# 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

# 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 ); }
}
```

Subtyping

Aggregation

Delegation

# 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 (Person p;)

Subtyping (implements Student)

Delegation (p.print( );)

Specialization (System.out.println( regNum );)

# Subtyping, Inheritance, and Subclassing

- In practical examples, it is often not obvious how to use subtyping and inheritance
- Example: immutable types, whose objects do not change their state after construction
- Advantages
  - No unexpected modifications of shared objects
  - No thread synchronization necessary
  - No inconsistent states

```
class ImmutableCell {
  private int value;

  ImmutableCell( int value ) {
    this.value = value;
  }

  int get( ) {
    return value;
  }

  // no setter
}
```

# Immutable Types: Subtyping

```
class ImmutableCell {
  int value;
  ImmutableCell( int value ) { … }
  int get( ) { … }
  // no setter
}
```

```
class Cell {
  int value;
  Cell( int value ) { … }
  int get( ) { … }
  void set( int value ) { … }
}
```

- What should be the subtype relation between mutable and immutable types?

# Proposal 1: Immutable <: Mutable

```
class ImmutableCell <: Cell {
  ImmutableCell( int value ) { … }
  …
}
```

```
class Cell {
  int value;
  Cell( int value ) { … }
  int get( ) { … }
  void set( int value ) { … }
}
```

- Not possible because mutable type has wider interface

# Proposal 1: Immutable <: Mutable

```
class ImmutableCell <: Cell {
  ImmutableCell( int value ) { … }
  void set( int value ) {
    // throw exception
   }
}
```

```
class Cell {
  int value;
  Cell( int value ) { … }
  int get( ) { … }
  void set( int value ) { … }
}
```

- Not possible because mutable type has wider interface

# Proposal 2: Mutable <: Immutable

```
class ImmutableCell {
  int value;

  … // no setter
}
```

```
class Cell <: ImmutableCell {
  Cell( int value ) { … }
  void set( int value ) { … }
}
```

# Proposal 2: Mutable <: Immutable

```
class ImmutableCell {
  int value;

  … // no setter
}
```

```
class Cell <: ImmutableCell {
  Cell( int value ) { … }
  void set( int value ) { … }
}
```

- Mutable type has wider interface
  - Also complies with structural subtyping

# Proposal 2: Mutable <: Immutable

```
class ImmutableCell {
  int value;
  // constraint old( value ) == value
  … // no setter
}
```

```
class Cell <: ImmutableCell {
  Cell( int value ) { … }
  void set( int value ) { … }
}
```

- Mutable type has wider interface
  - Also complies with structural subtyping
- But: Mutable type does not specialize behavior

# Proposal 2: Mutable <: Immutable

```
class ImmutableCell {
  int value;
  // constraint old( value ) == value
  … // no setter
}
```

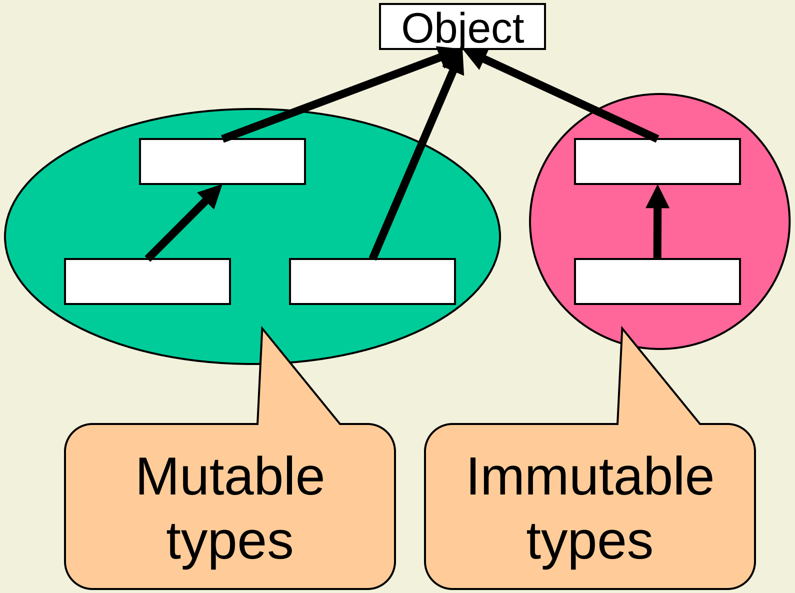
```
class Cell <: ImmutableCell {
  Cell( int value ) { … }
  void set( int value ) { … }
}
```

```
foo( ImmutableCell c ) {
  int cache = c.get( );
  …
  assert cache == c.get( );
}
```

- Mutable type has wider interface
  - Also complies with structural subtyping
- But: Mutable type does not specialize behavior

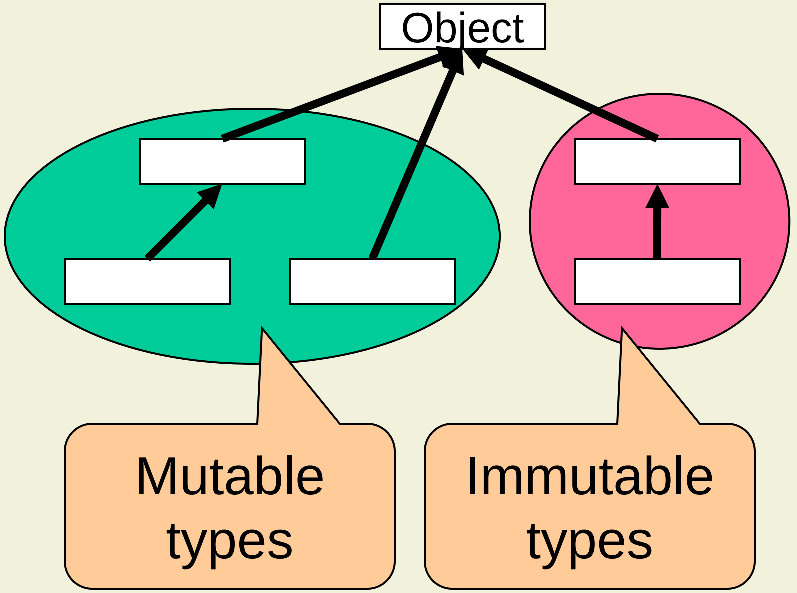
# Clean Solution: No Subtyping

- Clean solution
  - No subtype relation between mutable and immutable types
  - Only exception: Object, which has no history constraint



# Clean Solution: No Subtyping

- Clean solution
  - No subtype relation between mutable and immutable types
  - Only exception: Object, which has no history constraint
- Java API contains immutable types that are subtypes of mutable types
  - AbstractCollection and Iterator are mutable
  - All mutating methods are optional



# Discussion

- The presented classes for Cell and ImmutableCell 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
  - Subclassing is not an option because the types should not be subtypes

# Solution 1: Common Superclass

```
abstract class AbstractCell {
  int value;
  // constraint true
  int get( ) { … }
}
```

- Place reused code in a common superclass
- Polymorphic client code may use superclass


ETH zürich

# Solution 1: Common Superclass

```
abstract class AbstractCell {
  int value;
  // constraint true
  int get( ) { … }
}
```

```
class Cell extends AbstractCell {
  Cell( int value ) { … }
  void set( int value ) { … }
}
```

- Place reused code in a common superclass
- Polymorphic client code may use superclass

# Solution 1: Common Superclass

```
abstract class AbstractCell {
  int value;
  // constraint true
  int get( ) { … }
}
```

```
class Cell extends AbstractCell {
  Cell( int value ) { … }
  void set( int value ) { … }
}
```

```
class ImmutableCell
          extends AbstractCell {
  // constraint old( value ) == value
  ImmutableCell( int value ) { … }
}
```

- Place reused code in a common superclass
- Polymorphic client code may use superclass
- Be careful when strengthening invariants or history constraints over inherited fields

# Solution 2: Aggregation

- ImmutableCell uses Cell
  - Method calls are delegated to Cell
- No subtype relation
- Be careful when sharing underlying Cell object
- Run-time overhead

```
class Cell {
  int value;
  Cell( int value ) { … }
  int get( ) { … }
  void set( int value ) { … }
}
```

```
class ImmutableCell {
  final Cell rep;
  // constraint old( rep.value ) == rep.value
  ImmutableCell( int value ) { … }
  int get( ) { return rep.get( ); }
  // no setter
}
```

# Solution 3: Inheritance w/o Subtyping

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

```
class ImmutableCell {
public:
  int get( ) { … }
  …
}
```
C++

```
class Cell : private ImmutableCell {
public:
  void set( int value ) { … }
  ImmutableCell::get
  …
}
```
C++

# Solution 3: Inheritance w/o Subtyping

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

```
class ImmutableCell {
public:
  int get( ) { … }
  …
}
```
C++

```
class Cell : private ImmutableCell {
public:
  void set( int value ) { … }
  ImmutableCell::get
  …
}
```
C++

Make method public


ETH zürich

# Solution 3: Inheritance w/o Subtyping

- Some languages support inheritance without subtyping
  - C++: private and protected inheritance
  - Eiffel: non-conforming inheritance
- No (visible) subtype relation

```
class ImmutableCell {
public:
  int get( ) { … }
  …
}
```
C++

```
class Cell : private ImmutableCell {
public:
  void set( int value ) { … }
  ImmutableCell::get
  …
}
```
C++

Make method public

```
void foo( Cell c ) {
  ImmutableCell ic = c;
}
```
C++

Compile-time error



ETH zürich

# 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

# 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

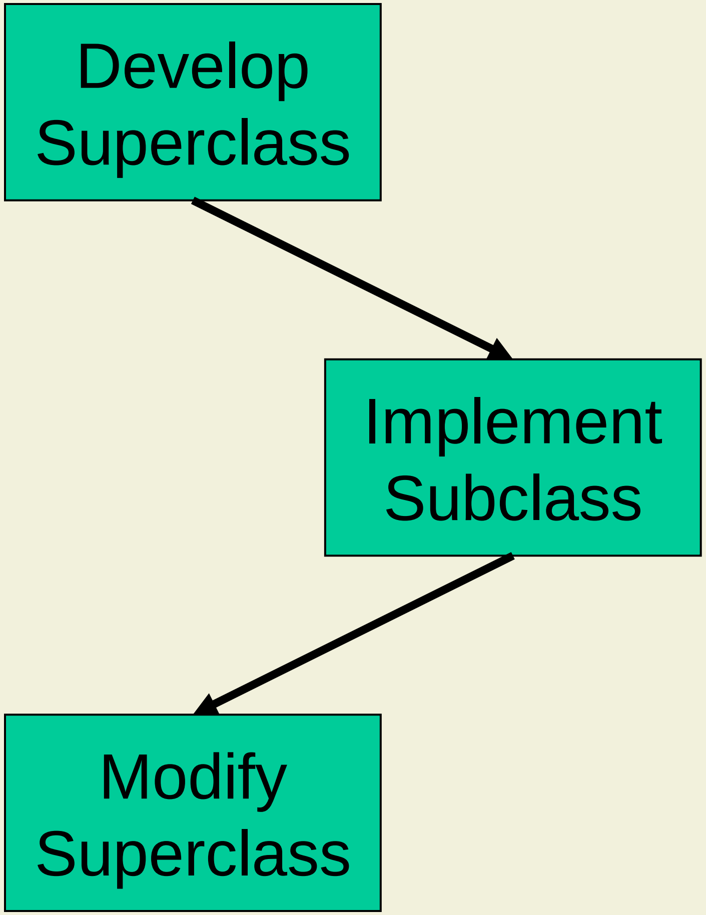
- Dynamic method binding enables specialization and subtype polymorphism
- However, there are important drawbacks
  - Performance: Overhead of method look-up at run time, less static information available for optimizations
  - Versioning: Dynamic binding makes it harder to evolve code without breaking 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?

# 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++; }
}
```

# 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( ) {
    … // count elements
  }

  // requires true
  // ensures ∀i. 0 <= i < arr.length:
  //                   contains( arr[ i ] )
  void addAll( Object[ ] arr )   {
    for( int i=0; i < arr.length; i++ )
      add( arr[ i ] );
  }
}
```

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

# Example 1: Discussion

```
class Bag {
  …
  int getSiz
    … // cou
  }

  // requires true
  // ensures ∀i. 0 <= i < 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

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

}
```

# Example 1: Discussion

```
class Bag {
  …
  int getSiz
    … // cou
  }

  // requires true
  // ensures ∀i. 0 <= i < 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

```
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 1: Discussion

```
class Bag {
  …
  int getSiz
    … // cou
  }

  // requires true
  // ensures ∀i. 0 <= i < 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 √ f;
  }

  float fourthRt( float f ) {
   return √√ f;
  }
}
```

```
class MyMath extends Math {

  float squareRt( float f ) {
    return –√ f;
  }
}
```

# Example 2: Unjustified Assumptions

```
class Math {

  float squareRt( float f ) {
    return √ f;
  }

  float fourthRt( float f ) {
    return √√ f;
  }
}
```

```
class MyMath extends Math {

  float squareRt( float f ) {
    return –√ f;
  }
}
```

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

# Example 2: Unjustified Assumptions (c'd)

```
class Math {

  float squareRt( float f ) {
    return √ f;
  }

  float fourthRt( float f ) {
   return squareRt( squareRt( f ) );
  }
}
```

```
class MyMath extends Math {

  float squareRt( float f ) {
    return –√ f;
  }
}
```

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

# Example 2: Unjustified Assumptions (c'd)

```
class Math {
  // requires f >= 0
  // ensures result ^ 2 = f
  float squareRt( float f ) {
    return √ f;
  }
  // requires f >= 0
  // ensures result ^ 4 = f
  float fourthRt( float f ) {
   return squareRt( squareRt( f ) );
  }
}
```

```
class MyMath extends Math {
  // requires f >= 0
  // ensures result ^ 2 = f
  float squareRt( float f ) {
    return –√ f;
  }
}
```

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

# Example 2: Unjustified Assumptions (c'd)

```
class Math {
  // requires f >= 0
  // ensures result ^ 2 = f
  float squareRt( float f ) {
    return √ f;
  }
  // requires f >= 0
  // ensures 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 >= 0
  // ensures result ^ 2 = f
  float squareRt( float f ) {
    return –√ f;
  }
}
```

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

# Example 2: Unjustified Assumptions (c'd)

```
class Math {
  // requires f >= 0
  // ensures result ^ 2 = f
  float squareRt( float f ) {
    return √ f;
  }
  // requires f >= 0
  // ensures result ^ 4 = f
  float fourthRt( float f ) {
   return squareRt( squareRt( f ) );
  }
}
```

```
class MyMath extends Math {
  // requires f >= 0
  // ensures result ^ 2 = f
  float squareRt( float f ) {
    return –√ f;
  }
}
```

Superclass: Do not change calls to dynamically-bound methods

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

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

# 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( );
  }
}
```

# 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;

  void inc1( ) {
    inc2( );
  }

  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;
  }
}
```

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

```
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( );
  }
  // require
  // ensure
  void inc
    x = x + 1;
  }
}
```

Superclass: Do not change calls to dynamically-bound methods

```
class CS extends C {
  // requires true
  // ensures x = ( 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 );
```

# Example 4: Additional Methods

```
class DiskMgr {




  void cleanUp( ) {
    … // remove temporary files
  }
}
```

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

# Example 4: Additional Methods

```
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( );
  }
}
```

```
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
  }
}
```

```
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 wh  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 {
   void delete( ) {
    … // remove temporary files
  }

  void cleanUp( ) {
    delete( );
  }
}
```
C#

```
class MyMgr : DiskMgr {
  void delete( ) {
    … // erase whole hard disk
  }
}
```
C#

```
MyMgr mm = new MyMgr( );
…
mm.cleanUp( );
```
C#

# Example 4: Additional Methods (cont'd)

```
class DiskMgr {
  void delete( ) {
   … // remove temporary files
  }

  void cleanUp( ) {
   delete( );
  }
}
```
C#

```
class MyMgr : DiskMgr {
  void delete( ) {
   … // erase whole hard disk
  }
}
```
C#

```
MyMgr mm = new MyMgr( );
…
mm.cleanUp( );
```
C#

- In C#, methods are bound statically by default

# Example 4: Additional Methods (cont'd)

```
class DiskMgr {
  virtual void delete( ) {
    … // remove temporary files
  }

  void cleanUp( ) {
    delete( );
  }
}
```
C#

```
class MyMgr : DiskMgr {
  void delete( ) {
    … // erase whole hard disk
  }
}
```
C#

```
MyMgr mm = new MyMgr( );
…
mm.cleanUp( );
```
C#

- In C#, methods are bound statically by default

# 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

# Example 5: Additional Methods (cont'd)

```java
class Super {
  void foo( String o ) { … }
  void bar( int i ) { … }
}
```
Java

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

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

- Overloading resolution in Java chooses most specific method declaration

# 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
  - When the client is re-compiled

# 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#

# 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.


ETH zürich

# 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)

- 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 ); }
}
```

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

# Solution 2: Double Invocation (cont'd)

Corresponds to Node and Visitor

```
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 ); }
}
```

- Double invocation is also called Visitor Pattern
- Problems
  - Even more tedious to write
  - Requires modification of superclass (not possible for equals method)

# Solution 2: Double Invocation (cont'd)

Corresponds to Node and Visitor

Corresponds to Node.accept

```
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 ); }
}
```

- Double invocation is also called Visitor Pattern
- Problems
  - Even more tedious to write
  - Requires modification of superclass (not possible for equals method)

# Solution 2: Double Invocation (cont'd)

Corresponds to Node and Visitor

Corresponds to Node.accept

```
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

- Double invocation is also called Visitor Pattern

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

# Solution 3: Overloading plus Dynamic

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

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

```
static Shape intersect( Shape s1, Shape s2 ) {
  return s1.intersect( s2 );
}
```
C#

# Solution 3: Overloading plus Dynamic

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

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

Overloads Shape's method

```
static Shape intersect( Shape s1, Shape s2 ) {
  return s1.intersect( s2 );
}
```
C#

# Solution 3: Overloading plus Dynamic

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

C#

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

C#

Overloads Shape's method

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

Since the receiver has static type Shape, this call is statically resolved to Shape's method

# Solution 3: Overloading plus Dynamic

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

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

Overloads Shape's method

```
static Shape intersect( Shape s1, Shape s2 ) {
  return ( s1 as dynamic ).intersect( s2 );
}
```
C#

# Solution 3: Overloading plus Dynamic

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

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

Overloads Shape's method

```
static Shape intersect( Shape s1, Shape s2 ) {
  return ( s1 as dynamic ).intersect( s2 );
}
```

Since the argument has static type Shape, this call is dynamically resolved to Shape's method

# Solution 3: Overloading plus Dynamic

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

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

Overloads Shape's method

```
static Shape intersect( Shape s1, Shape s2 ) {
  return ( s1 as dynamic ).intersect( s2 as dynamic );
}
```
C#

# Solution 3: Overloading plus Dynamic

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

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

Overloads Shape's method

```
static Shape intersect( Shape s1, Shape s2 ) {
  return ( s1 as dynamic ).intersect( s2 as dynamic );
}
```

Dynamic resolution depends on dynamic types of both arguments

# Solution 3: Overloading plus Dynamic (c'd)

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

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

```
static Shape intersect( Shape s1, Shape s2 ) {
  return
  ( s1 as dynamic ).intersect( s2 as dynamic );
}
```
C#

- Concise
- No change to superclass required
- Problems
  - Not entirely type safe
  - Overhead for run-time checks

# Solution 4: 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
  }
}
```

# Solution 4: 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

# Solution 4: 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 4: Multiple Dispatch (cont'd)

- Multiple dispatch is statically type-safe

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

# Solution 4: 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

# Solution 4: 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
- Overloading plus dynamic
  - Dynamic resolution based on dynamic argument types
- Multiple dispatch
  - One multiple-dispatch call

# 3. Inheritance

# Information Hiding

- Hide implementation details
- Reduce dependencies between modules
- Classes can be understood, reused, and modified in isolation

```
class SymbolTable {
  private Dictionary d;

  public void add( String k, String v ) {
    d.put( k, v );
  }

  public String lookup( String k ) {
    return d.atKey( k );
  }
}
```

# The Client Interface of a Class

- Class name
- Type parameters and their bounds
- Super-class
- Super-interfaces
- Signatures of exported methods and fields
- Client interface of direct superclass

```
class SymbolTable
        extends Dictionary<String,String>
        implements Map<String,String> {
  public int size;

  public void add( String key, String value )
   { put( key, value ); }

  public String lookup( String key )
        throws IllegalArgumentException {
   return atKey( key );
  }
}
```

# Other Interfaces

- Subclass interface
  - Efficient access to superclass fields
  - Access to auxiliary superclass methods

```
public class DList {
  protected Node first, last;
  private int modCount;
  protected void modified( )
    { modCount++; }
  …
}
```

# Other Interfaces

- Subclass interface
  - Efficient access to superclass fields
  - Access to auxiliary superclass methods
- Friend interface
  - Mutual access to implementations of cooperating classes
  - Hiding auxiliary classes

```
package coop.util;
public class DList {
  protected Node first, last;
  private int modCount;
  protected void modified( )
   { modCount++; }
  …
}
```

```
package coop.util;
/* default */ class Node {
  /* default */ Object elem;
  /* default */ Node next, prev;
  … }
```

# Other Interfaces

- Subclass interface
  - Efficient access to superclass fields
  - Access to auxiliary superclass methods
- Friend interface
  - Mutual access to implementations of cooperating classes
  - Hiding auxiliary classes
- And others

```
package coop.util;
public class DList {
  protected Node first, last;
  private int modCount;
  protected void modified( )
    { modCount++; }
  …
}
```

```
package coop.util;
/* default */ class Node {
  /* default */ Object elem;
  /* default */ Node next, prev;
  … }
```

# Expressing Information Hiding

- Java: Access modifiers
  - **public** client interface
  - **protected** subclass + friend interface
  - Default access friend interface
  - **private** implementation

- Eiffel: Clients clause in feature declarations
  - **feature** { ANY } client interface
  - **feature** { T } friend interface for class T
  - **feature** { NONE } implementation (only "**this**"-object)
  - All exports include subclasses

# Method Selection in Java (JLS1)

```
class T {
  public void m( ) { ... }
}
```

```
class S extends T {
  public void m( ) { ... }
}
```

```
class U extends S { }
```

```
T v = new U( );
v.m( );
```

# Method Selection in Java (JLS1)

- At compile time:
  1. Determine static declaration

```
class T {
  public void m( ) { ... }
}
```

```
class S extends T {
  public void m( ) { ... }
}
```

```
class U extends S { }
```

```
T v = new U( );
v.m( );
```

# Method Selection in Java (JLS1)

- At compile time:
  1. Determine static declaration
  2. Check accessibility

```
class T {
  public void m( ) { ... }
}
```

```
class S extends T {
  public void m( ) { ... }
}
```

```
class U extends S { }
```

```
T v = new U( );
v.m( );
```

# Method Selection in Java (JLS1)

- At compile time:
  1. Determine static declaration
  2. Check accessibility
  3. Determine invocation mode (virtual / nonvirtual)

```
class T {
  public void m( ) { ... }
}
```

```
class S extends T {
  public void m( ) { ... }
}
```

```
class U extends S { }
```

```
T v = new U( );
v.m( );
```

# Method Selection in Java (JLS1)

- At compile time:
  1. Determine static declaration
  2. Check accessibility
  3. Determine invocation mode (virtual / nonvirtual)
- At run time:
  4. Compute receiver reference

```
class T {
  public void m( ) { ... }
}
```

```
class S extends T {
  public void m( ) { ... }
}
```

```
class U extends S { }
```

```
T v = new U( );
v.m( );
```

# Method Selection in Java (JLS1)

- At compile time:
  1. Determine static declaration
  2. Check accessibility
  3. Determine invocation mode (virtual / nonvirtual)
- At run time:
  4. Compute receiver reference
  5. Locate method to invoke (based on dynamic type of receiver object)

```
class T {
  public void m( ) { ... }
}
```

```
class S extends T {
  public void m( ) { ... }
}
```

```
class U extends S { }
```

```
T v = new U( );
v.m( );
```

# Rules for Overriding: Access

# Rules for Overriding: Access

```
class Super {
  …
  protected void m( ) { … }
}
```

```
class Sub extends Super {
            void m( ) { … }
}
```

# Rules for Overriding: Access

```
class Super {
  …
  protected void m( ) { … }
}
```

```
class Sub extends Super {
            void m( ) { … }
}
```

In class Super or Sub:

```
public void test( Super v ) {
  v.m( );
}
```

# Rules for Overriding: Access

- Access Rule:
  The access modifier of an overriding method must provide at least as much access as the overridden method

```
class Super {
  …
  protected void m( ) { … }
}
```

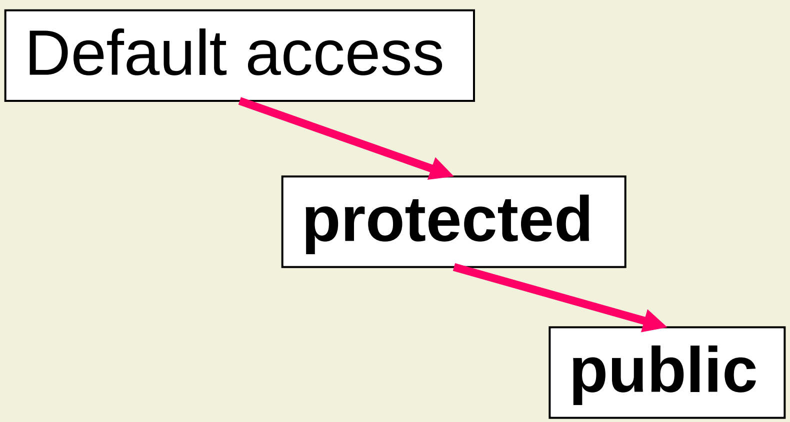
```
class Sub extends Super {
          void m( ) { … }
}
```

In class Super or Sub:

```
public void test( Super v ) {
  v.m( );
}
```

# Rules for Overriding: Access

- Access Rule:
  The access modifier of an overriding method must provide at least as much access as the overridden method




```
class Super {
  …
  protected void m( ) { … }
}
```

```
class Sub extends Super {
          void m( ) { … }
}
```

In class Super or Sub:

```
public void test( Super v ) {
  v.m( );
}
```

# Rules for Overriding: Access

- Access Rule:
  The access modifier of an overriding method must provide at least as much access as the overridden method

Default access → **protected** → **public**

```
class Super {
  …
  protected void m( ) { … }
}
```

```
class Sub extends Super {
  public      void m( ) { … }
}
```

In class Super or Sub:

```
public void test( Super v ) {
  v.m( );
}
```

# Rules for Overriding: Hiding

```
class Super {
  …
  private void m( )
    { System.out.println(“Super”); }
  public void test( Super v )
    { v.m( ); }
}
```

```
class Sub extends Super {
  public void m( )
    { System.out.println(“Sub”); }
}
```

# Rules for Overriding: Hiding

```
class Super {
  …
  private void m( )
    { System.out.println("Super"); }
  public void test( Super v )
    { v.m( ); }
}
```

```
class Sub extends Super {
  public void m( )
    { System.out.println("Sub"); }
}
```

```
Super v = new Sub( );
v.test( v );
```

# Rules for Overriding: Hiding

- Override Rule:
  A method Sub.m overrides the superclass method Super.m only if Super.m is accessible from Sub
- If Super.m is not accessible from Sub, it is hidden by Sub.m
- Private methods cannot be overridden

```
class Super {
  …
  private void m( )
   { System.out.println("Super"); }
  public void test( Super v )
   { v.m( ); }
}
```

```
class Sub extends Super {
  public void m( )
   { System.out.println("Sub"); }
}
```

```
Super v = new Sub( );
v.test( v );
```

# Problems with Default Access Methods

```
package PT;
public class T {
  void m( ) { ... }
}
```

```
package PS;
public class S extends PT.T {
  public void m( ) { ... }
}
```

In package PT:

```
T v = new PS.S( );
v.m( );
```

# Problems with Default Access Methods

- S.m does not override T.m (T.m is not accessible in S)
- T.m and S.m are different methods with same signature
- Static declaration for invocation is T.m
- At run time, S.m is selected and invoked

```
package PT;
public class T {
  void m( ) { ... }
}
```

```
package PS;
public class S extends PT.T {
  public void m( ) { ... }
}
```

In package PT:

```
T v = new PS.S( );
v.m( );
```

# Corrected Method Selection (JLS2)

- Dynamically selected method must override statically determined method

# Corrected Method Selection (JLS2)

- Dynamically selected method must override statically determined method

- At compile time:
  1. Determine static declaration
  2. Check accessibility
  3. Determine invocation mode (virtual / nonvirtual)

# Corrected Method Selection (JLS2)

- Dynamically selected method must override statically determined method

- At compile time:
  1. Determine static declaration
  2. Check accessibility
  3. Determine invocation mode (virtual / nonvirtual)

- At run time:
  4. Compute receiver reference
  5. Locate method to invoke that overrides statically determined method

# Problems with Protected Methods

```
package PT;
public class T {
  protected void m( ) { ... }
}
```

```
package PS;
public class S extends PT.T {
  protected void m( ) { ... }
}
```

```
package PT;
public class C {
 public void foo( ) {
   T v = new PS.S( );
   v.m( );                 }
}
```

# Problems with Protected Methods

- S.m overrides T.m
- Static declaration is T.m, which is accessible for C
- At run time, S.m is selected, which is not accessible for C
- **protected** does not always "provide at least as much access" as **protected**

```
package PT;
public class T {
  protected void m( ) { ... }
}
```

```
package PS;
public class S extends PT.T {
  protected void m( ) { ... }
}
```

```
package PT;
public class C {
  public void foo( ) {
    T v = new PS.S( );
    v.m( );                    }
}
```

# Problems with Protected Methods

- S.m overrides T.m
- Static declaration is T.m, which is accessible for C
- At run time, S.m is selected, which is not accessible for C
- **protected** does not always "provide at least as much access" as **protected**

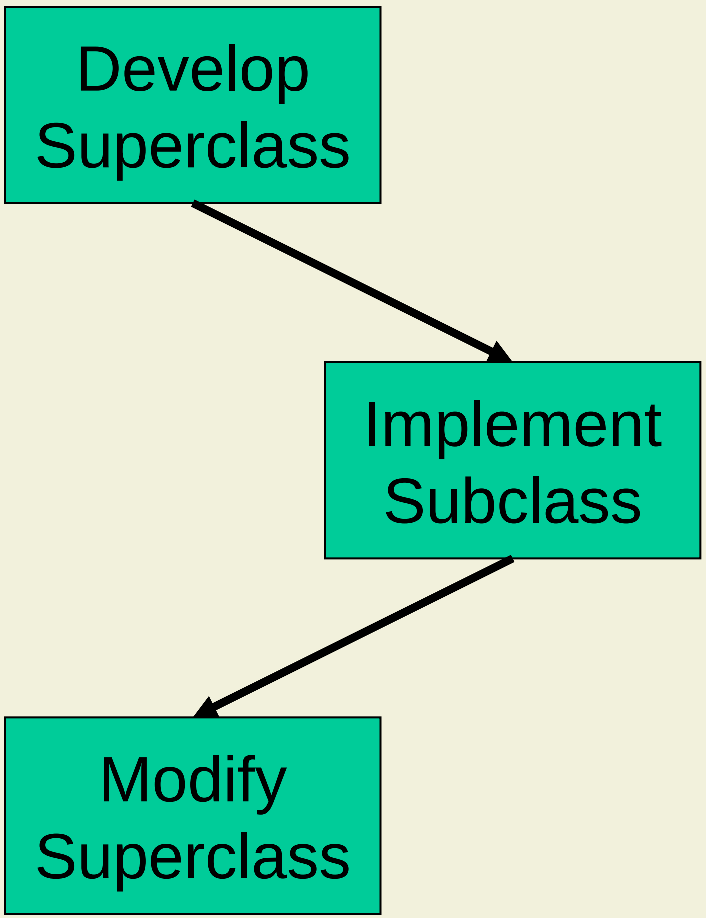
```
package PT;
public class T {
  protected void m( ) { ... }
}
```

```
package PS;
public class S extends PT.T {
  protected void m( ) { ... }
}
```

**public** would be safe

```
package PT;
public class C {
 public void foo( ) {
   T v = new PS.S( );
   v.m( );              }
}
```

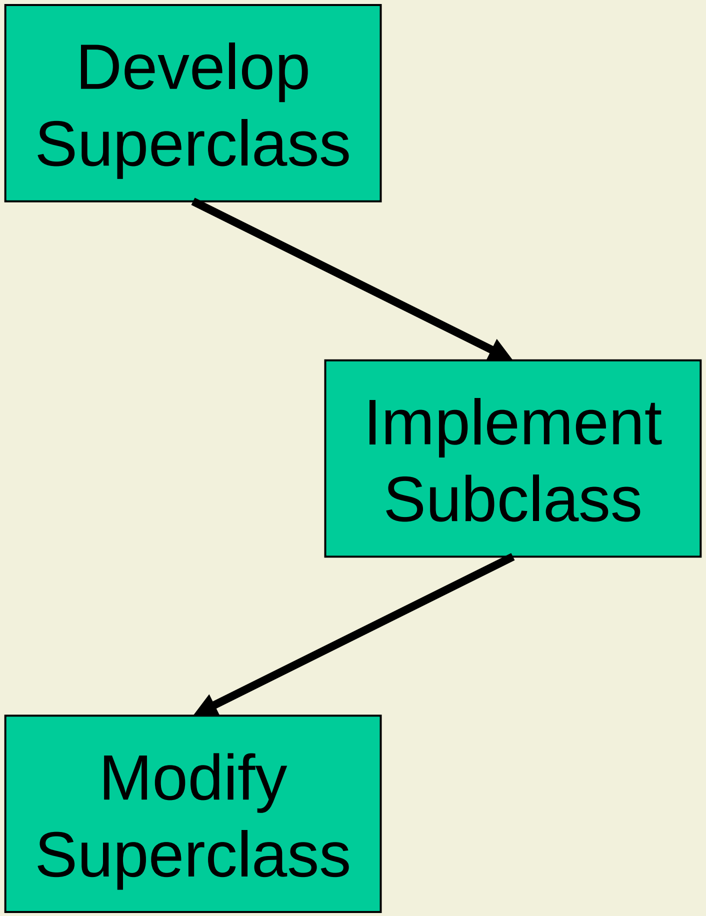
# Another Fragile Baseclass Problem




```
class C {
  int x;
  public      void inc1( )
    { this.inc2( ); }
  private     void inc2( )
    { x++; }
}
```

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

# Another Fragile Baseclass Problem

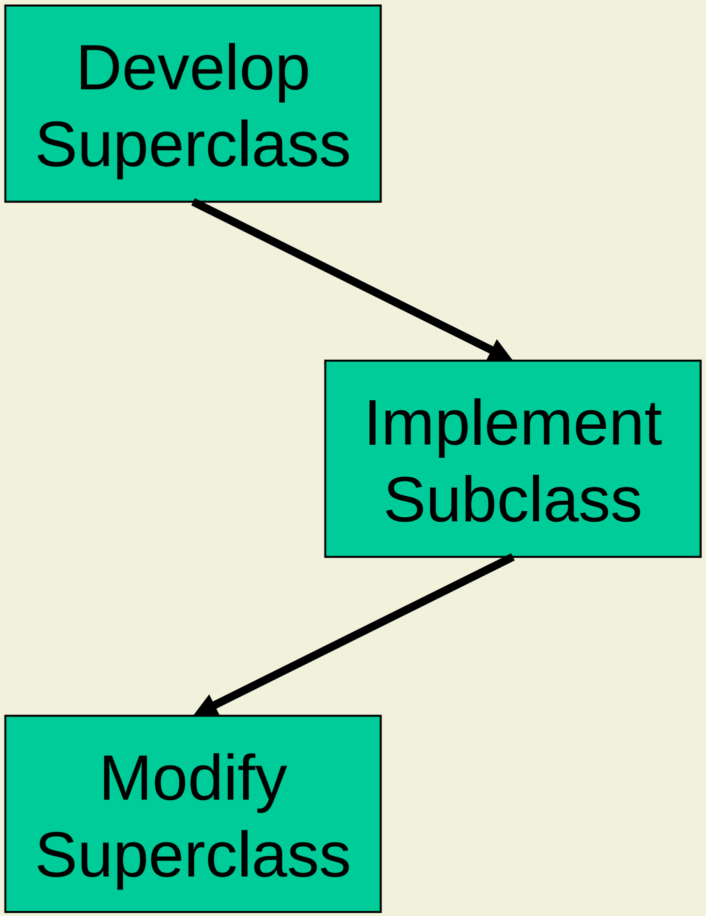



```
class C {
  int x;
  public    void inc1( )
   { this.inc2( ); }
  private   void inc2( )
   { x++; }
}
```

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

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

# Another Fragile Baseclass Problem

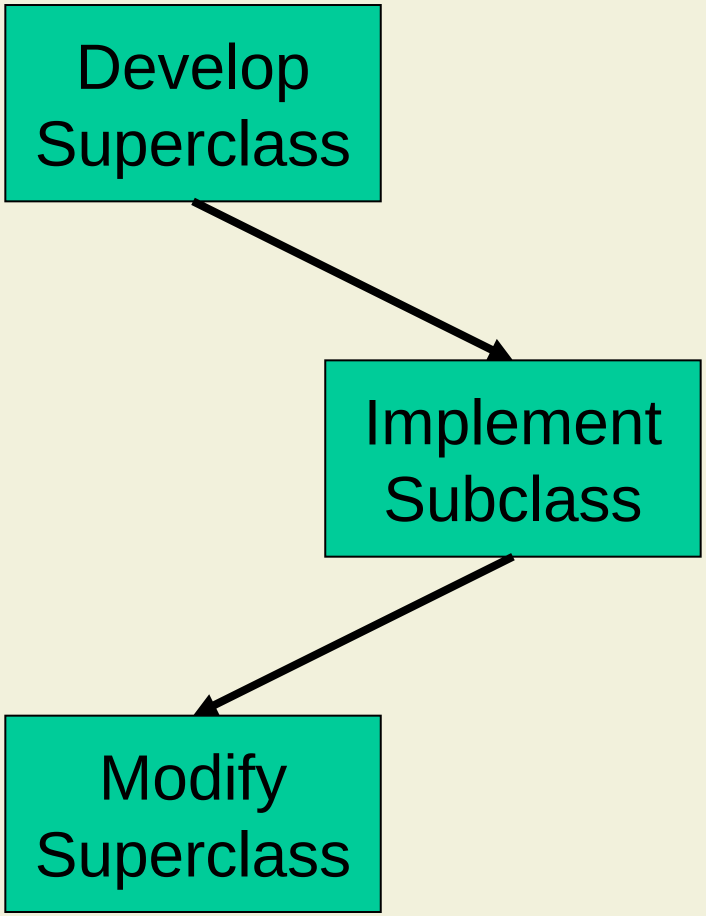



```
class C {
  int x;
  public     void inc1( )
    { this.inc2( ); }
  protected void inc2( )
    { x++; }
}
```

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

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

# Another Fragile Baseclass Problem




```
class C {
  int x;
  public      void inc1( )
    { this.inc2( ); }
  protected void inc2( )
    { x++; }
}
```

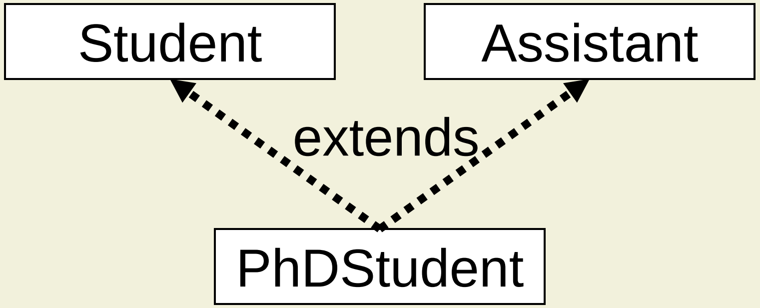
```
class CS extends C {
  public      void inc2( )    { inc1( ); }
}
```

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

# 3. Inheritance

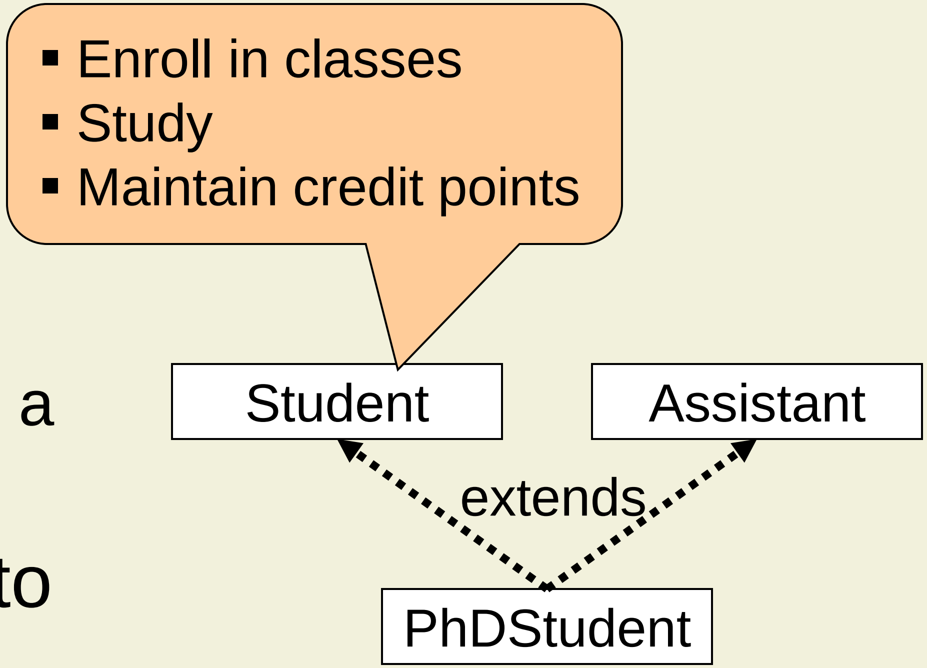
# Motivation

- 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



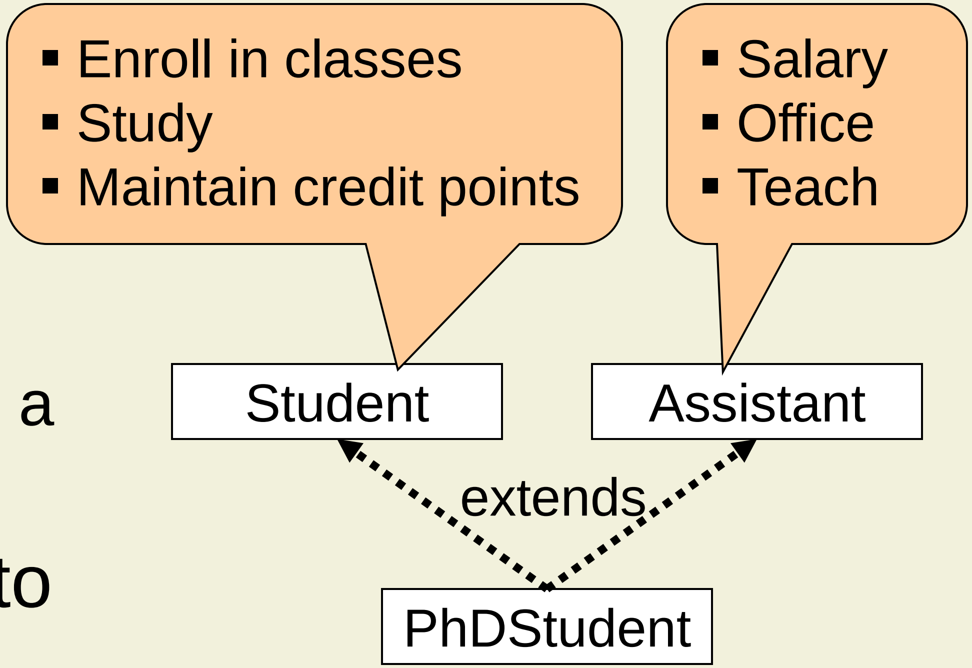
# Motivation

- 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



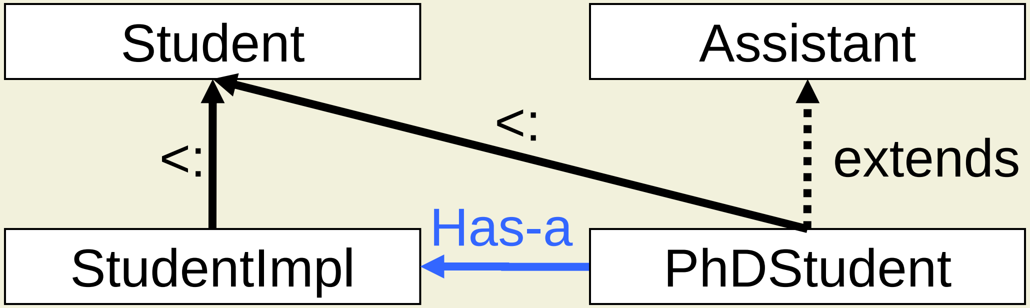
# Motivation

- 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



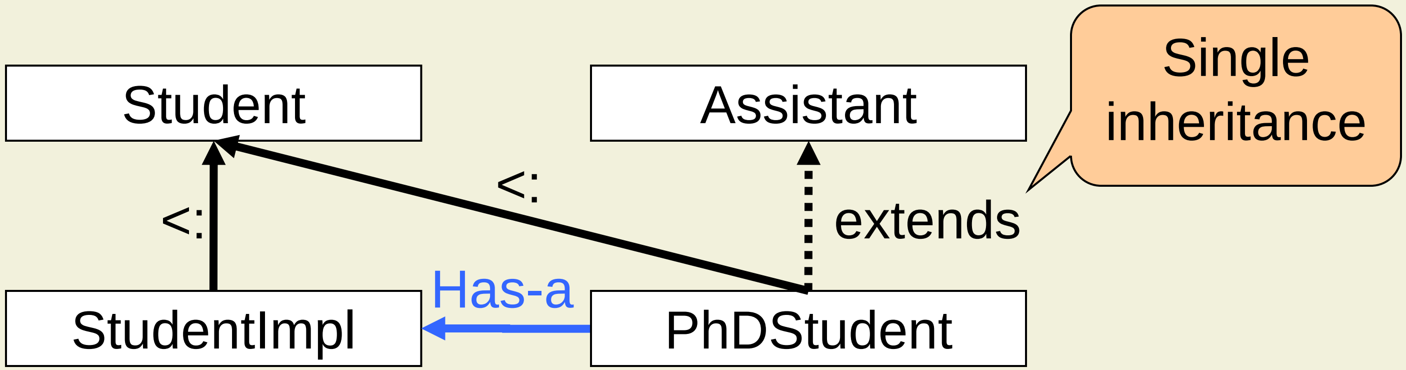
# Simulating Multiple Inheritance

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



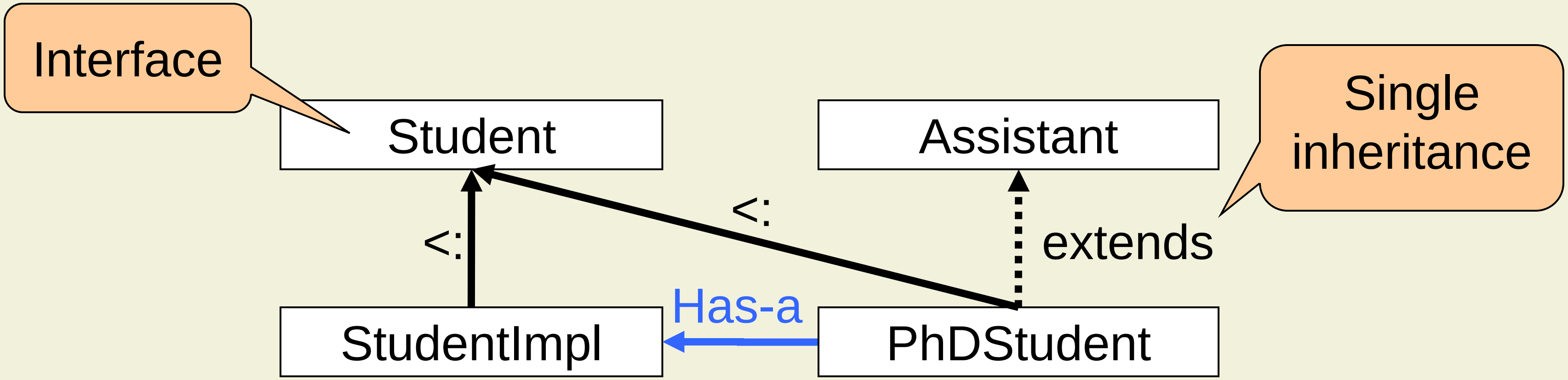
# Simulating Multiple Inheritance

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



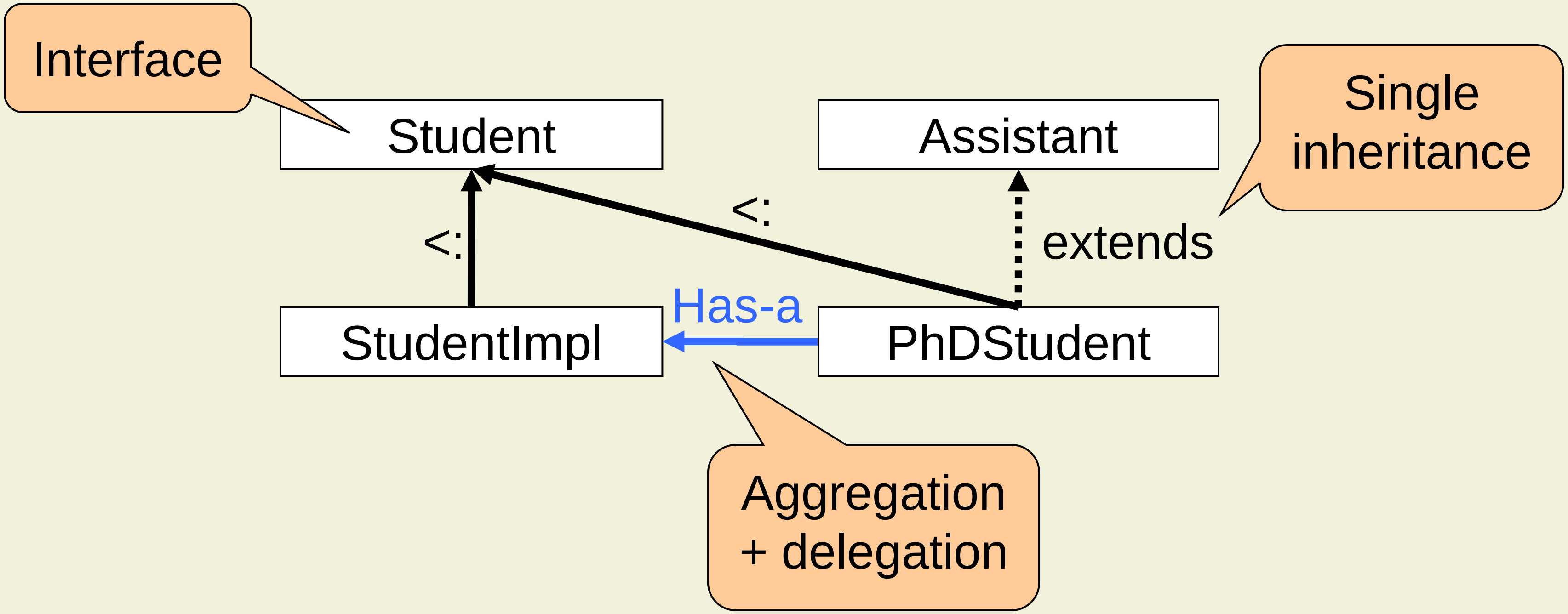
# Simulating Multiple Inheritance

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



# 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

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

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

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

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

# Ambiguities: Example

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

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

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

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

Which method should be called?

# Ambiguities: Example

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

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

C++
```
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

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

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

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

```
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

C++

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

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

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



ETH zürich

# Ambiguity Resolution: Merging Methods

Overrides both inherited methods

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

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

- Related inherited methods can be merged into one overriding method

- Usual rules for overriding apply
  - Type rules
  - Behavioral subtyping

# Ambiguity Resolution: Merging Methods

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

Overrides both inherited methods

Correspond to super-calls in Java

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

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

# Merging Unrelated Methods

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

C++
```
class Assistant {
public:
  virtual bool test( ) { // unit test }
  … };
```

# Merging Unrelated Methods

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

C++
```
class Assistant {
public:
  virtual bool test( ) { // unit test }
  … };
```

- Unrelated methods cannot be merged in a meaningful way
  - Even if signatures match

# Merging Unrelated Methods

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

C++
```
class Assistant {
public:
  virtual bool test( ) { // unit test }
  … };
```

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

- Unrelated methods cannot be merged in a meaningful way
  - Even if signatures match

# Merging Unrelated Methods

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

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

Clients can call Assistant::test

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

- Unrelated methods cannot be merged in a meaningful way
  - Even if signatures match

# Merging Unrelated Methods

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

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

Clients can call Assistant::test

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

Violates behavioral subtyping

- Unrelated methods cannot be merged in a meaningful way
  - Even if signatures match

# Merging Unrelated Methods

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

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

Clients can call Assistant::test

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

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

Eiffel
```
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
```

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

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

- C++/CLI provides similar features

# Ambiguity Resolution: Renaming

Eiffel
```
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
```

- 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

```cpp
class Person {
  Address address;
  …
};
```

C++

```cpp
class Student : public Person {
  …
};
```

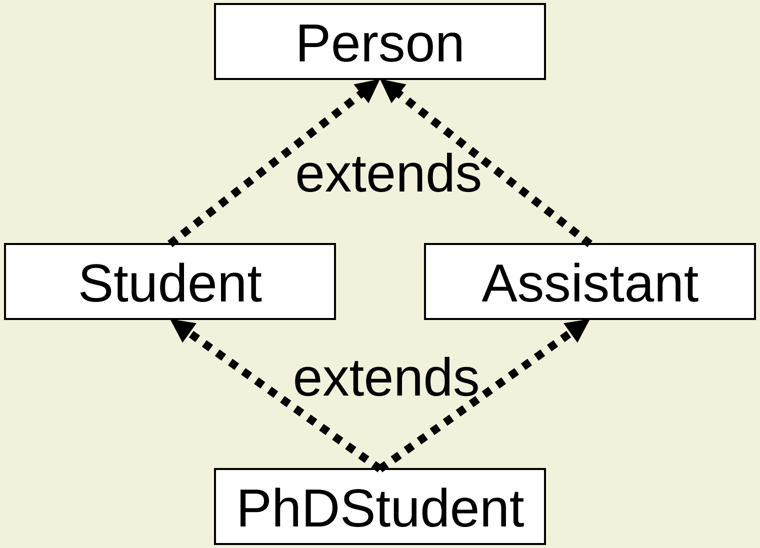
C++

```cpp
class Assistant : public Person {
  …
};
```

C++

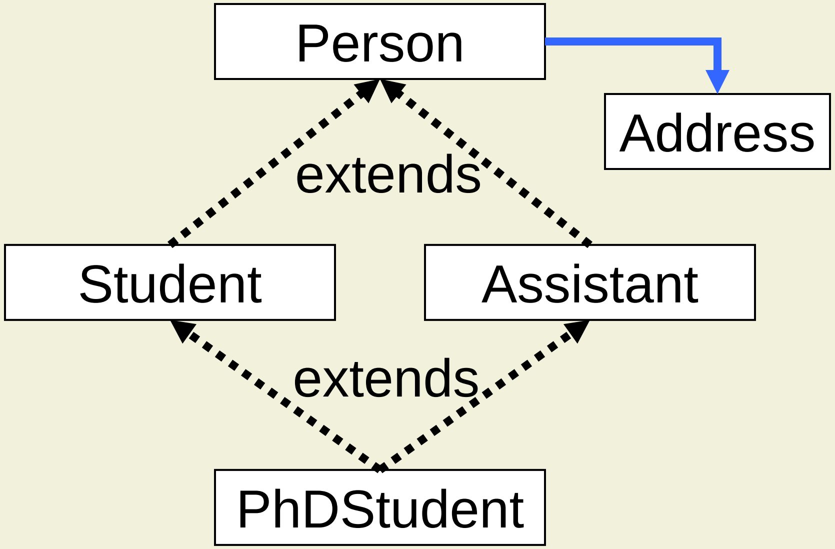
```cpp
class PhDStudent :
      public Student, public Assistant {
};
```

C++



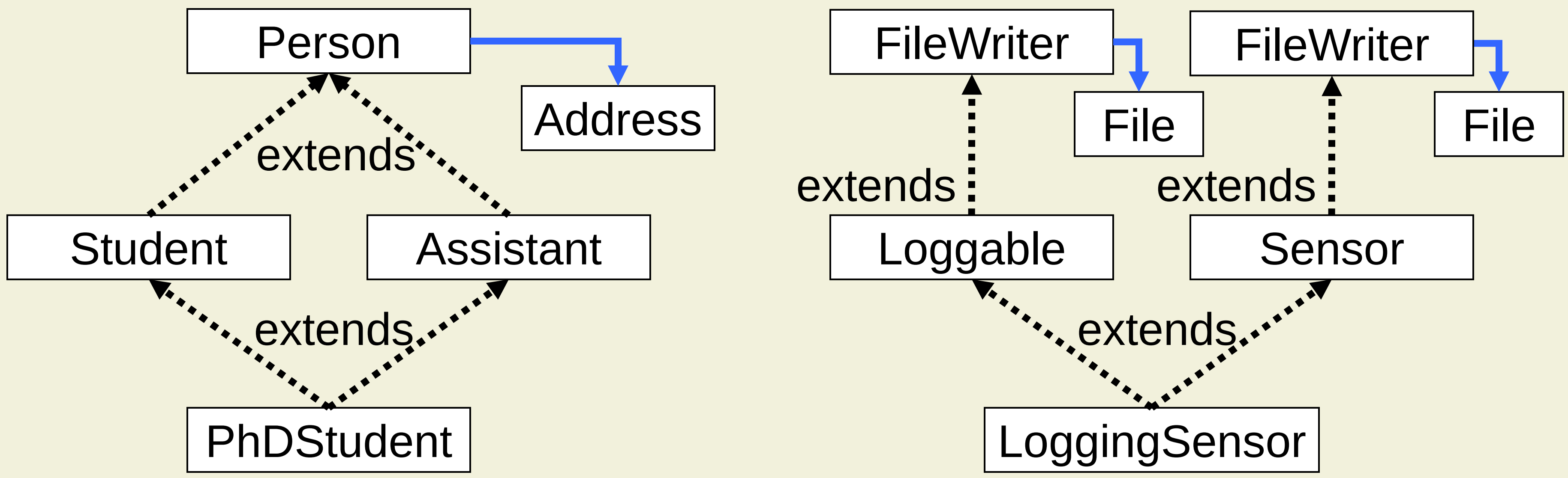

- How many address fields should PhDStudent have?
- How are they initialized?

# How Many Copies of Superclass Fields?




- Eiffel: default
- C++: virtual inheritance

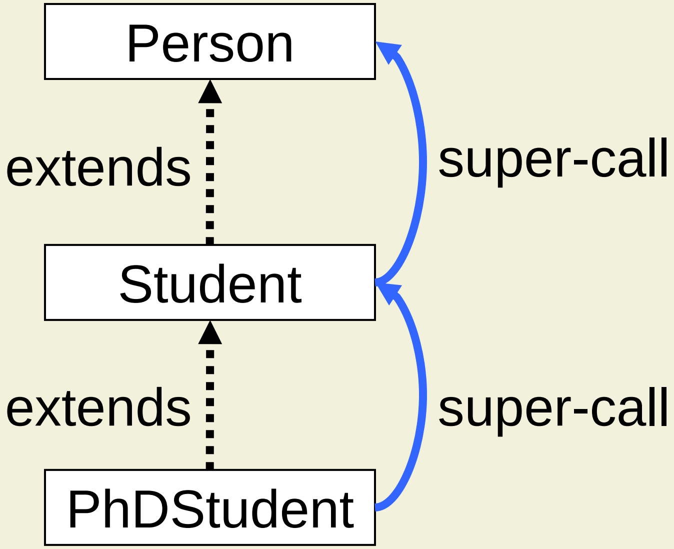
# How Many Copies of Superclass Fields?




- Eiffel: default
- C++: virtual inheritance

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

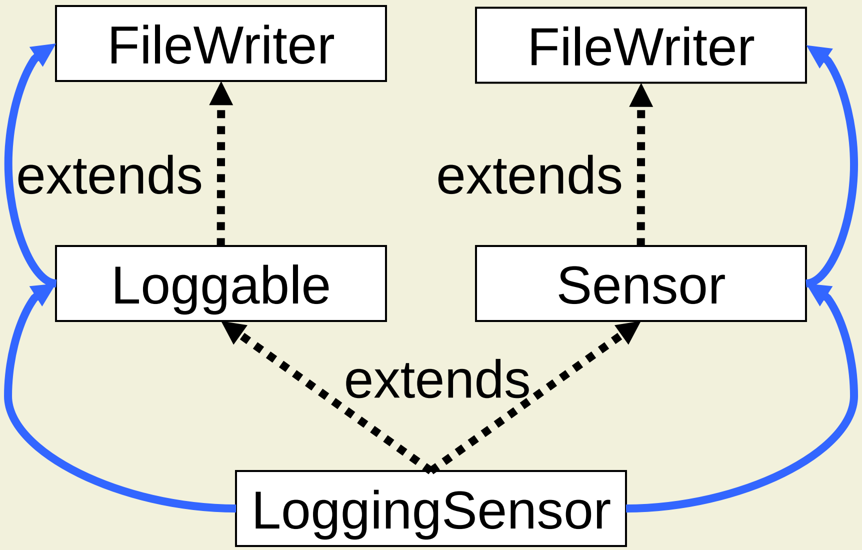
# Inheritance and Object Initialization




- 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

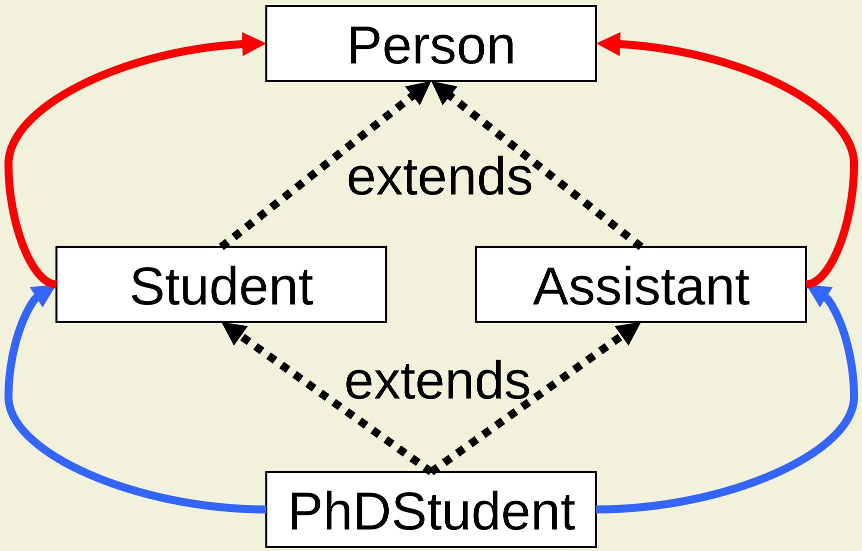
- With non-virtual inheritance, there are two copies of the superclass fields
- Superclass constructor is called twice to initialize both copies
  - Here, create two file handles for two files



# Initialization and Virtual Inheritance

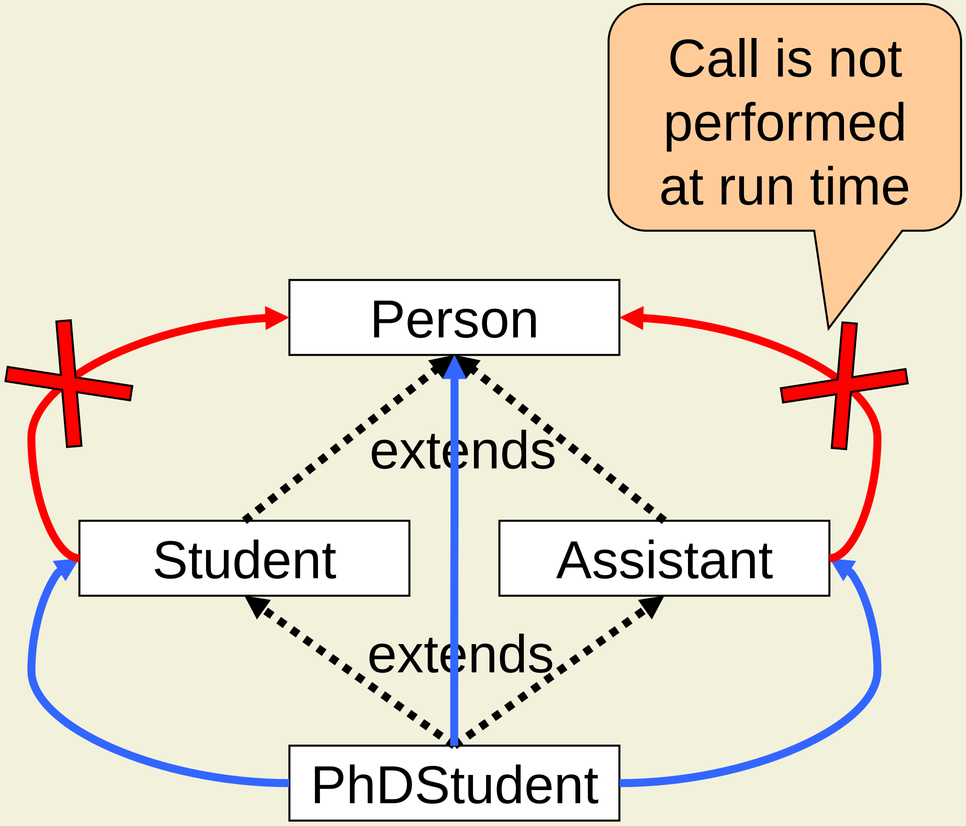
- With virtual inheritance, there is only one copy of the superclass fields

- 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 ) {

  };
};
```

- Non-virtual inheritance is the default
  - Virtual inheritance leads to run-time overhead
  - Programmers need foresight!

# C++ Solution: Discussion

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

Might fail

- 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

# Multiple Inheritance

**Pros**

- Increases expressiveness
- Avoids overhead of delegation pattern

**Cons**

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

# 3. Inheritance





ETH zürich

# 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

```scala
class Cell {
  var value: Int = 0

  def put( v: Int ) = { value = v }
  def get: Int = value
}
```

Scala

```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

```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

```
trait Backup extends Cell {
  var backup = 0;

  override def put( v: Int ) = {
    backup = value
    super.put( v )
  }
  def undo = { super.put( backup ) }
}
```
Scala

# Scala: Declaration of Traits

Traits extend exactly one superclass (and possibly other traits)

```
trait Backup extends Cell {
  var backup = 0;

  override def put( v: Int ) = {
    backup = value
    super.put( v )
  }
  def undo = { super.put( backup ) }
}
```

Scala

# Scala: Declaration of Traits

Traits may have fields

Traits extend exactly one superclass (and possibly other traits)

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

Scala

# Scala: Declaration of Traits

Traits extend exactly one superclass (and possibly other traits)

Traits may have fields

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

Scala

Traits may declare methods

# Scala: Declaration of Traits

Traits may have fields

Traits may override superclass methods

Traits may declare methods

Traits extend exactly one superclass (and possibly other traits)

```
trait Backup extends Cell {
 var backup = 0;

 override def put( v: Int ) = {
  backup = value
  super.put( v )
 }
 def undo = { super.put( backup ) }
}
```

Scala

# Scala: Mixing-in Traits

```
class FancyCell extends Cell with Backup {
  …
}
```
Scala

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

# Scala: Mixing-in Traits

```
class FancyCell extends Cell with Backup {
  …
}
```
Scala

Traits can be mixed-in when classes are declared

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

# 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

# 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' direct superclasses
  - To avoid multiple inheritance among classes

# 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

```scala
trait Backup extends Cell {
  …
}
```
Scala

```scala
class FancyCell
     extends Cell with Backup {
  …
}
```
Scala

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

# Example: Thin and Thick Interfaces

- Traits can extend thin interfaces by additional operations

```
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 ) }
  }
}
```

# 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 ) }
  }
}
```

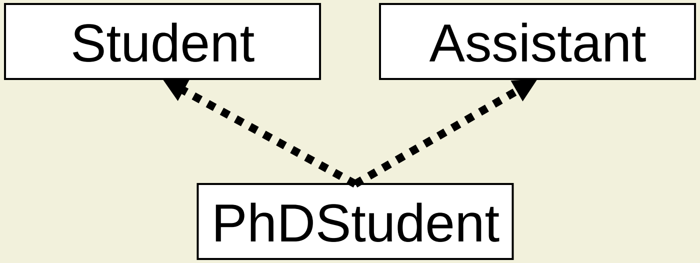
```
def client ( p: ThinCollection with AddAll, a: Array[String] ) = { p.addAll( a ) }
```

# 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
  }
}
```

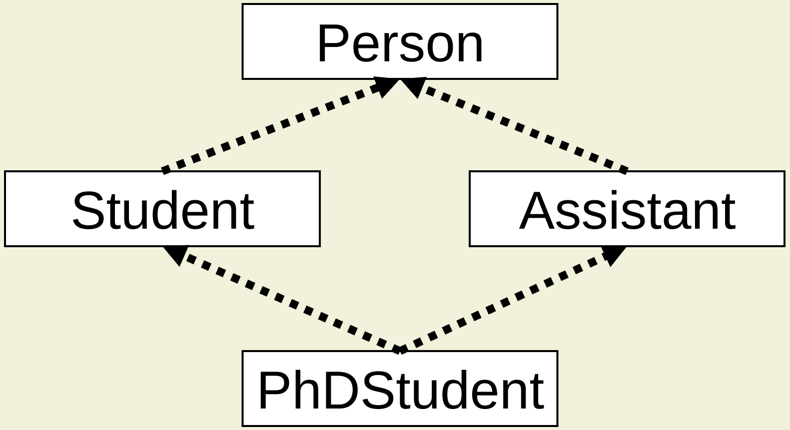
# Ambiguity Resolution and Diamonds

```
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 types in a linear order
  - Define overriding and super-calls according to this order
- For a class or trait

  C **extends** C' **with** $C_1$ … **with** $C_n$

  the linearization L(C) is

  C, L($C_n$) • … • L($C_1$) • L(C')
- Do not include types more than once

  $\varepsilon \bullet B = B$

$$(a, A) \bullet B = \begin{cases} a, (A \bullet B) & \text{if } a \notin B \\ A \bullet B & \text{otherwise} \end{cases}$$

# Linearization Example

```
class Person
```

```
trait Student extends Person
```

```
trait Assistant extends Person
```

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

# Linearization Example

```
class Person
```

```
trait Student extends Person
```

```
trait Assistant extends Person
```

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

For a class or trait

C **extends** C' **with** $C_1$ … **with** $C_n$

the linearization L(C) is

C, $L(C_n) \bullet \ldots \bullet L(C_1) \bullet L(C')$

# Linearization Example

```
class Person
```

```
trait Student extends Person
```

```
trait Assistant extends Person
```

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

L(Person) = Person

For a class or trait

C **extends** C' **with** $C_1$ … **with** $C_n$

the linearization L(C) is

C, L($C_n$) • … • L($C_1$) • L(C')

# Linearization Example

```
class Person
```

```
trait Student extends Person
```

```
trait Assistant extends Person
```

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

For a class or trait

C **extends** C’ **with** $C_1$ … **with** $C_n$

the linearization L(C) is

C, $L(C_n) \bullet \ldots \bullet L(C_1) \bullet L(C')$

L(Person) = Person

L(Student) = Student, Person

# Linearization Example

**class** Person

**trait** Student **extends** Person

**trait** Assistant **extends** Person

**class** PhDStudent
**extends** Person
**with** Student
**with** Assistant

For a class or trait

C **extends** C' **with** $C_1$ … **with** $C_n$

the linearization L(C) is

C, L($C_n$) • … • L($C_1$) • L(C')

L(Person) = Person

L(Student) = Student, Person

L(Assistant) = Assistant, Person

# Linearization Example

```
class Person
```

```
trait Student extends Person
```

```
trait Assistant extends Person
```

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

For a class or trait

C **extends** C' **with** $C_1$ … **with** $C_n$

the linearization L(C) is

C, L($C_n$) • … • L($C_1$) • L(C')

L(Person) = Person

L(Student) = Student, Person

L(Assistant) = Assistant, Person

L(PhDStudent) = PhDStudent, L(Assistant) • L(Student) • L(Person)

# Linearization Example

**class** Person

**trait** Student **extends** Person

**trait** Assistant **extends** Person

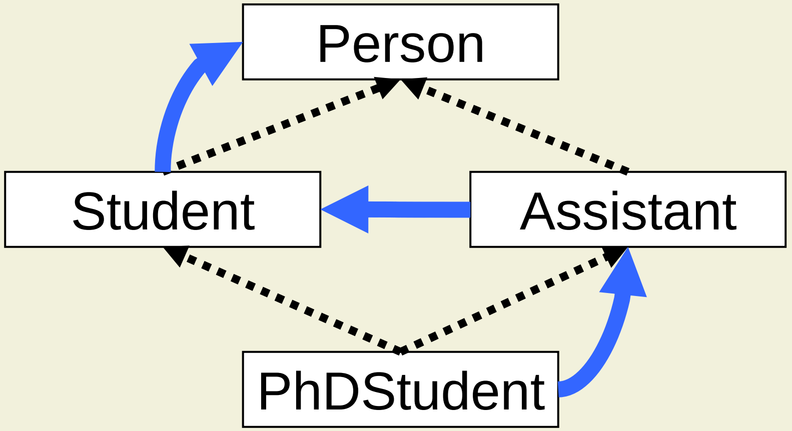
**class** PhDStudent
**extends** Person
**with** Student
**with** Assistant

For a class or trait

C **extends** C' **with** $C_1$ … **with** $C_n$

the linearization L(C) is

C, $L(C_n) \bullet \ldots \bullet L(C_1) \bullet L(C')$




L(Person) = Person

L(Student) = Student, Person

L(Assistant) = Assistant, Person

L(PhDStudent) = PhDStudent, L(Assistant) • L(Student) • L(Person)

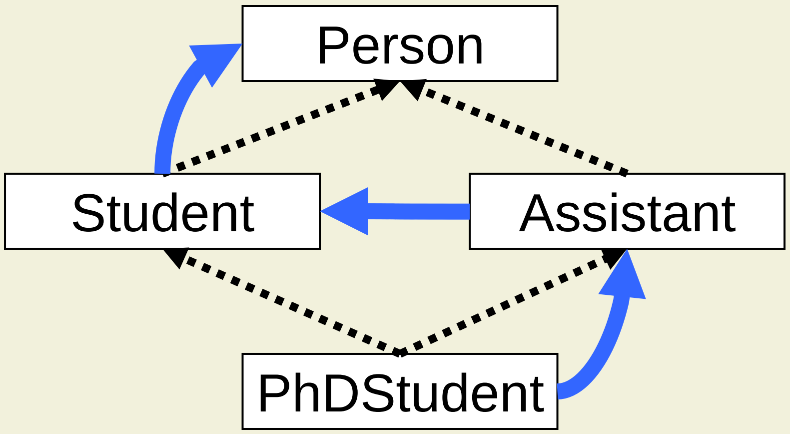
# Overriding

```
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 { }
```



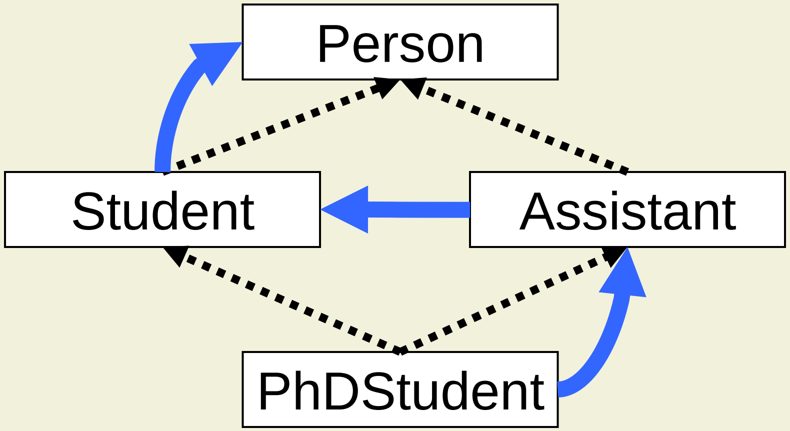
# Overriding

```
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

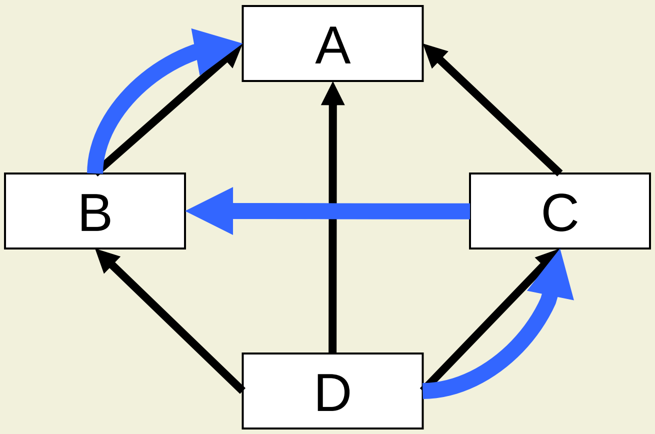
# Repeated Inheritance

```
class A {
  var f = 0
}
```

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

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

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




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

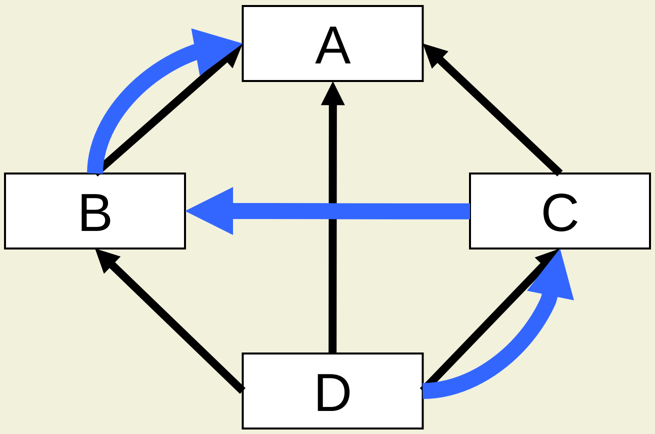
# Initialization Order

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

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

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

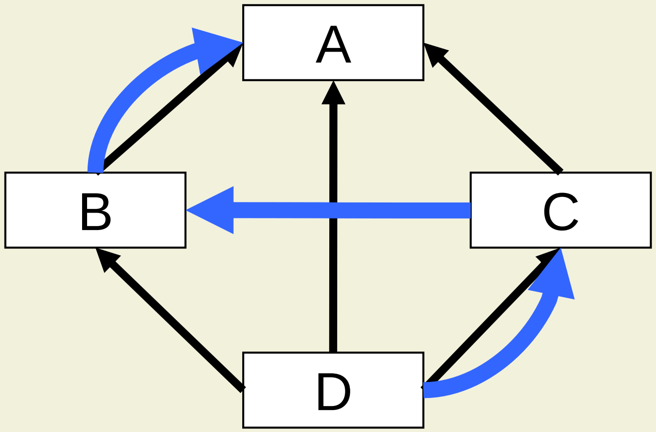
```
class D extends A with B with C {
  println( "Constructing D" )
}
```




- Classes and traits are initialized in the reverse linear order

# Initialization of Repeated Superclasses

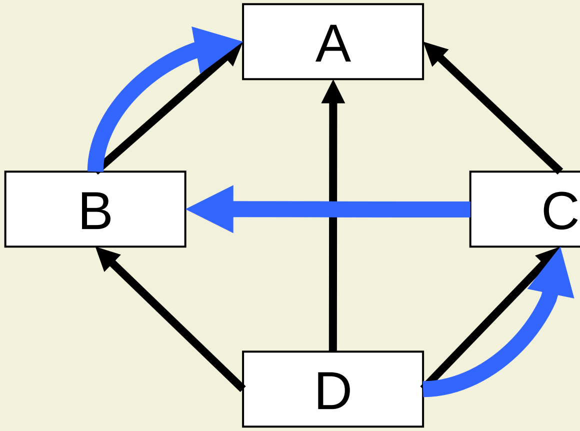
- Each constructor is called exactly once
  - Good if constructor has side-effects



# 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 { … }
```

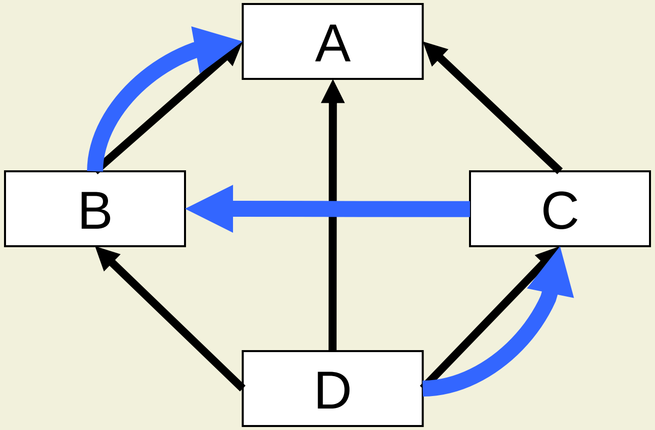
# Overriding and Super-Calls

```
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 }
}
```

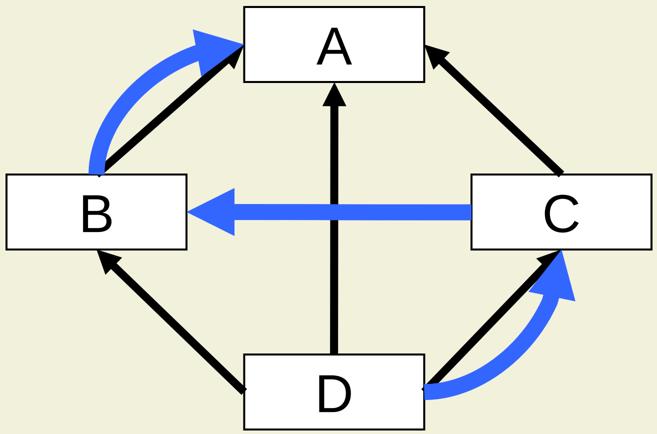
```
class D extends A with B with C { }
```




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

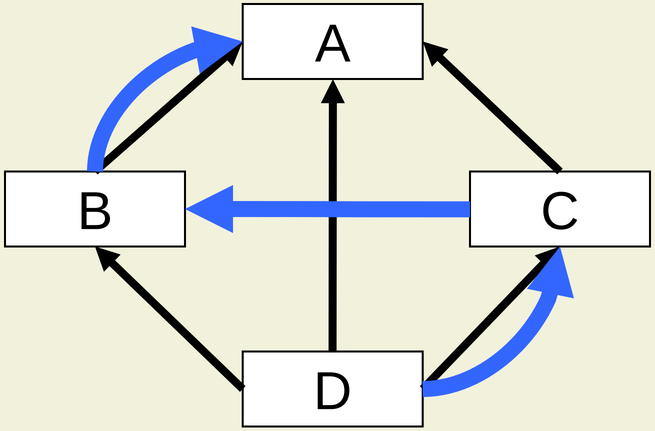
# Stackable Specializations

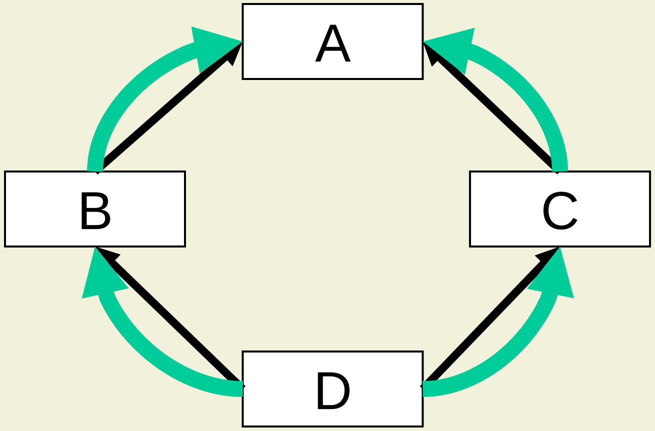
- With traits, specializations can be combined in flexible ways



# Stackable Specializations

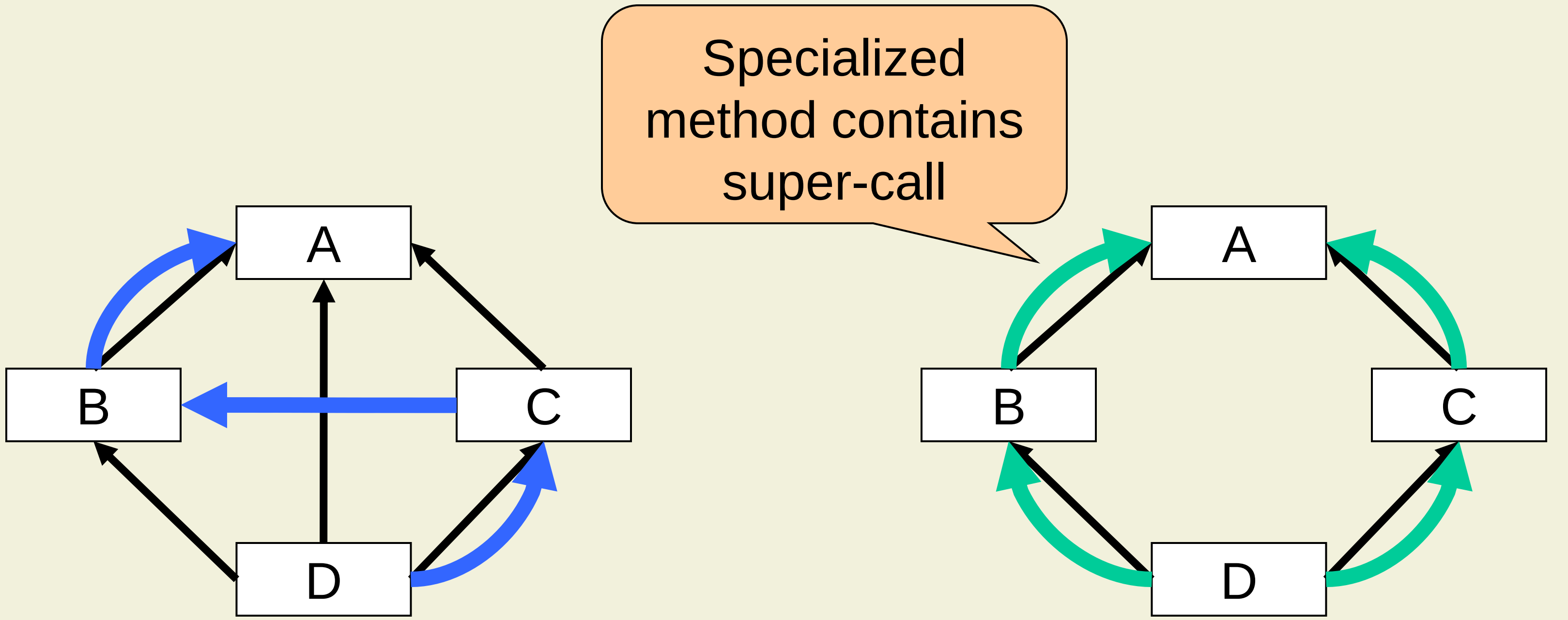
- With traits, specializations can be combined in flexible ways
- With multiple inheritance, methods of repeated superclasses are called twice



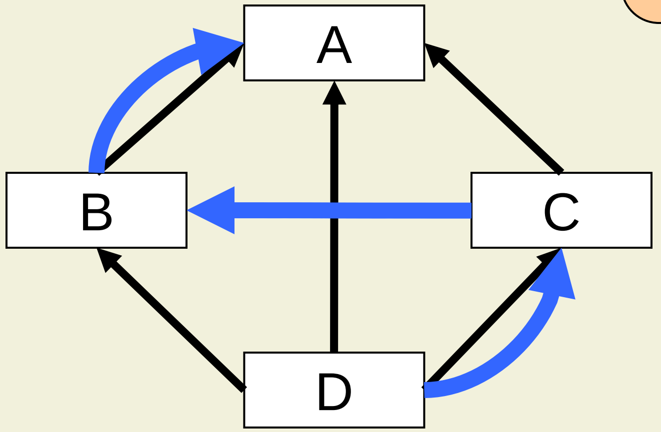
# Stackable Specializations

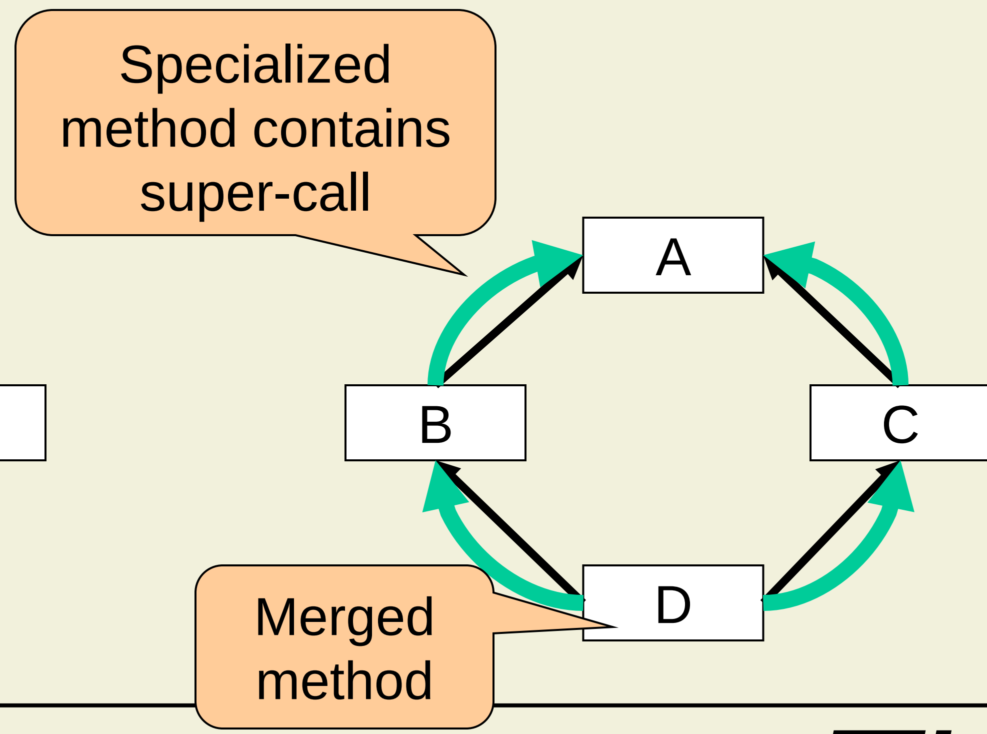
- With traits, specializations can be combined in flexible ways
- With multiple inheritance, methods of repeated superclasses are called twice



# Stackable Specializations

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



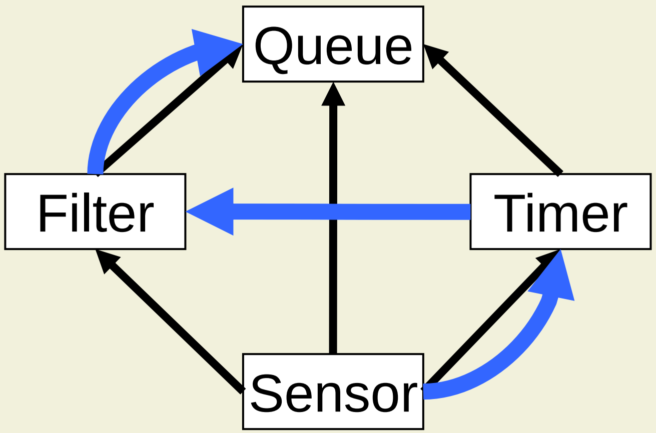
# Stackable Specializations: Example

```
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 ) }
}
```

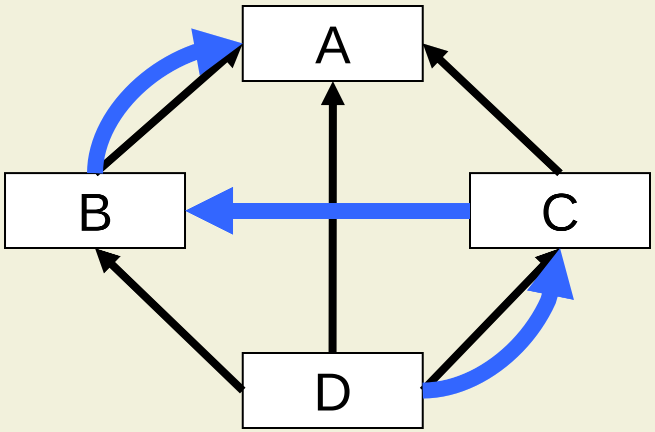
```
class Sensor extends Queue
              with Filter with Timer { }
```



# 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 }
}
```

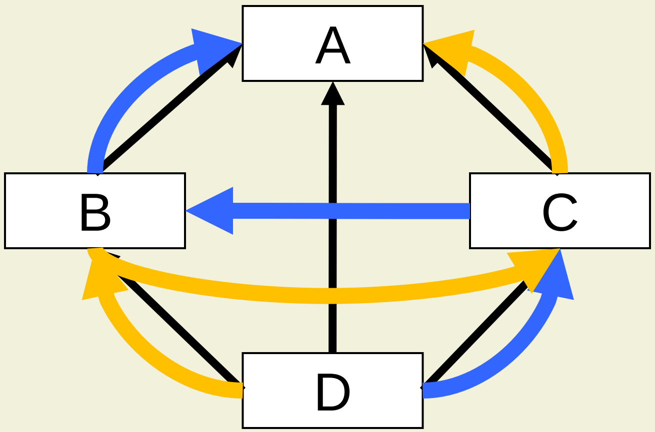



```
class D extends A with B with C { }
```

# 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 { }

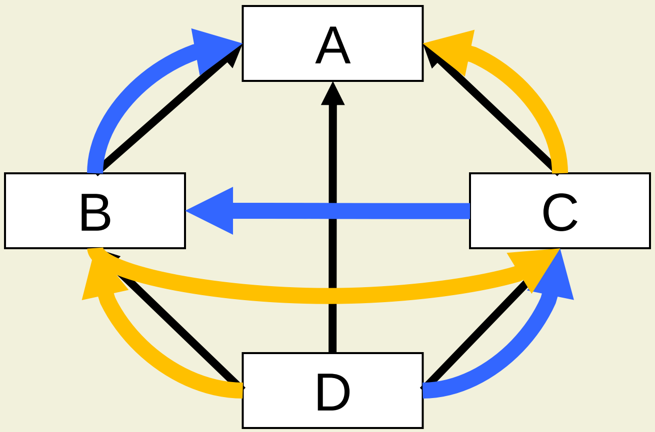
# Traits and Behavioral Subtyping

```
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

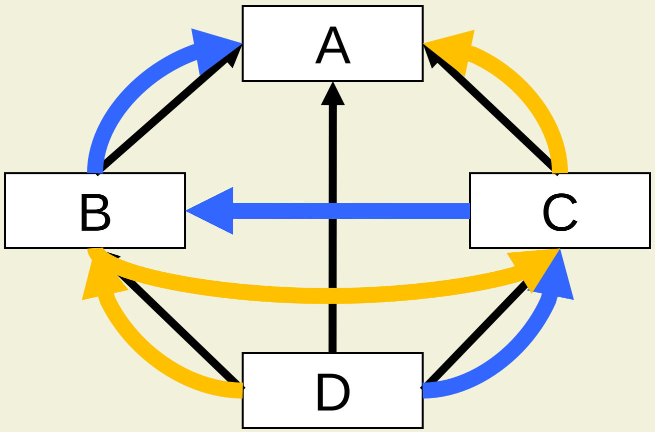
# Traits and Behavioral Subtyping

```
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


ETH zürich

# 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 }
}
```

# Linearization: Summary

- Linearization partly solves problems of multiple inheritance
  - Solves some issues with ambiguities and initialization
- Other problems remain
  - Resolving ambiguities between unrelated methods
- And new problems arise
  - Behavioral subtyping cannot be checked modularly
  - What to assume about superclass initialization and super-calls

# 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
- 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