
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Paul Corazza

Lecture 3: Inheritance and Composition

Reflecting the Whole in the Part

Wholeness Statement

Inheritance and Composition are types of relationships between classes that support reuse of code. Inheritance makes polymorphism possible, but can lock classes into a structure that may not be flexible enough in the face of change. Composition is more flexible but does not support polymorphism.

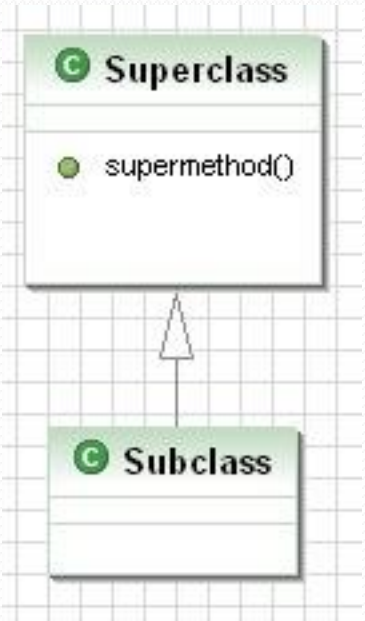
Composition and inheritance are techniques based on the principle of preserving sameness in diversity, silence in dynamism

Outline of Topics

- Review of inheritance concepts and implementation in Java
- Wrong uses of inheritance
- Benefits of inheritance
- Problems with inheritance
  - Fragility
    - Rectangle-Square Problem
  - Violates encapsulation: Ripple effect
    - Enhancing HashSet
- Best Practice (J. Bloch): Design for inheritance or else prevent it
- Using Composition
  - Instead of inheritance – Example: a Stack class
  - In combination with inheritance – Example: Inheriting from a Role

Review of Inheritance

```
class Superclass {
    protected void supermethod() {
        int x = 0;
    }
}
class Subclass extends Superclass {
    public static void main(String[] args) {
        Superclass sub = new Subclass();
        //subclass has access to data and
        // non-private methods of superclass
        sub.supermethod();
    }
}
```

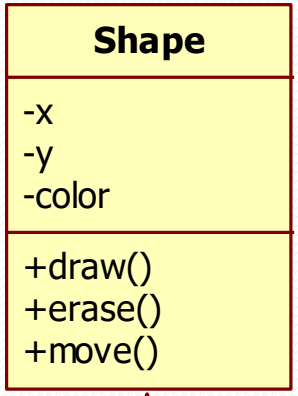


Note: There are some subtle points about the `protected` keyword that we do not explain here (but see the Appendix to these slides at the end for more information).
See demo `lesson03.lecture.inheritance0`

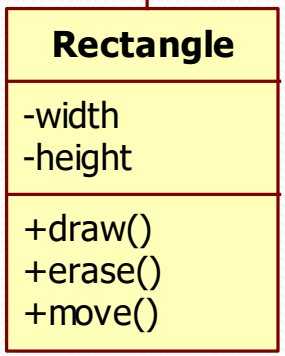
Example

- Relationship between a general and a specific class
  - IS-A relationship
  - no multiplicity

(more general, abstract)
superclass



(more specific, concrete)
subclass



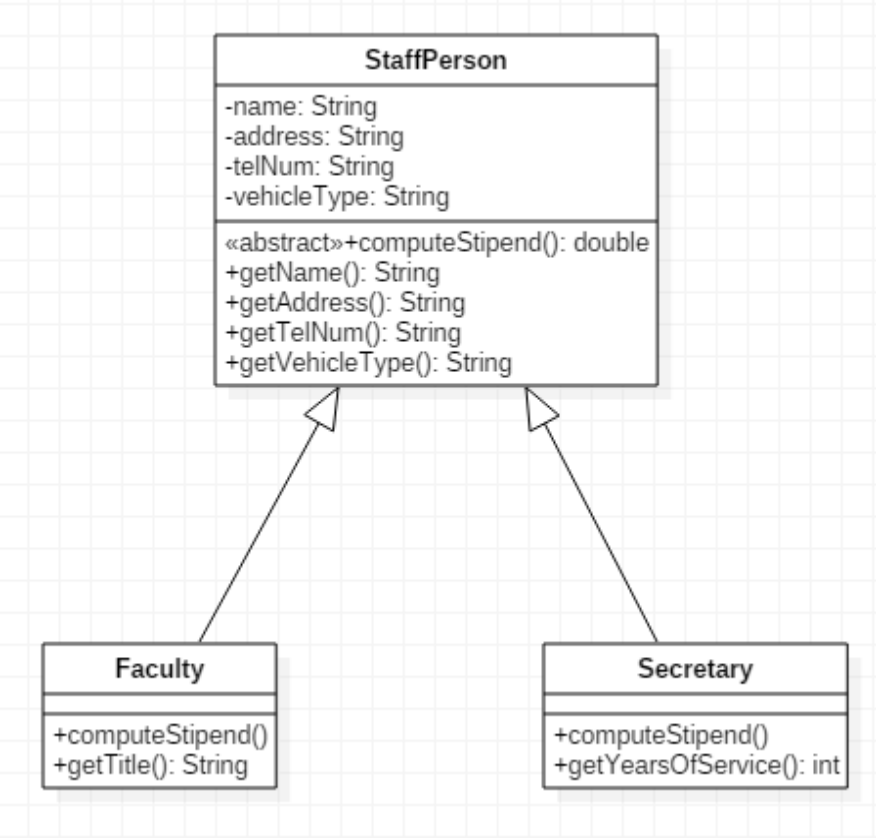

```
public class Shape {
    ...
}

public class Rectangle extends Shape {
    ...
}
```

Rectangle inherits all attributes and methods from Shape that are public or protected. For package-level, depends on if in the same package.

Inheritance Arises . . .

As a way to *generalize* data and behavior of related classes

Faculty	Secretary
properties: name address phoneNumber drivesVehicle title	properties: name address phoneNumber drivesVehicle yearsOfService
behavior: computeStipend()	behavior: computeStipend()



And . . .

As a way to *extend* the behavior of a particular class

```
class Employee {
   //constructor
   Employee(String aName,
            double aSalary) {
         name = aName;
         salary = aSalary;
   }
   public String getName() {
         return name;
   }
   public double getSalary() {
         return salary;
   }
   public void raiseSalary(double byPercent) {
         double raise = salary * byPercent / 100;
         salary += raise;
   }

   private String name;
   private double salary;
}
```

```
class Manager extends Employee {
   public Manager(String name, double salary) {
         super(name,salary);
         bonus = 0;
   }
   @Override
   public double getSalary() {
         //no direct access to private
         //variables of superclass
         double baseSalary = super.getSalary();
         return baseSalary + bonus;
   }
   public void setBonus(double b) {
         bonus = b;
   }
   private double bonus;
}
```

See demos in `lesson03.lecture.polymorphism1, lesson03.lecture.polymorphism2`

Overriding a method

- A subclass can change inherited behavior of the super class by overriding methods
- Java Method Overriding Restrictions & Rules
  - Method must exist in superclass.
    - You're overriding a method, not creating a new one.
  - Method signature must match exactly.
    - Same name, parameters, and order (return type must be compatible).
  - Return type must be the same or covariant.
    - Covariant = subclass of the return type in the parent method.
  - Access level cannot be more restrictive.
    - You can make it more open (e.g., `protected` → `public`), but not more `private`.

Overriding a method (Cont.)

- Cannot override `final` methods
- Cannot override `static` methods
- Cannot override `private` methods
- Throws clause can be more specific
  - You can throw fewer or narrower exceptions than the overridden method.

- Best practice is to also add the `@Override` annotation

```
@Override
public String toString() {
    return "Employee [salary=" + salary + ", getFirstname()="
    + getFirstname() + ", getLastname()=" + getLastname()
    + "]";
}
```

Best Practices for Using Inheritance

- *IS-A Principle* Class C may extend class D if C IS-A D.
  Example: Manager IS-A Employee
  Example: Secretary IS-A StaffPerson

- *Liskov Substitution Principle (LSP)*: C may extend D if an object of type C may be used during execution where an object of type D is expected, without breaking the code.

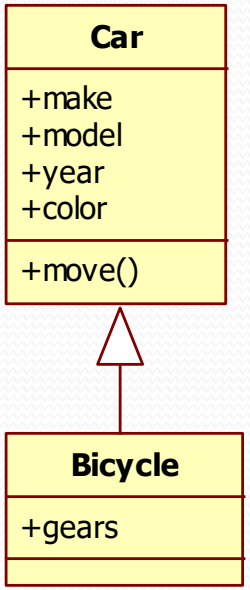
Example of LSP We may use a Manager instance wherever an Employee instance is expected, so having Manager as a subclass of Employee adheres to LSP.

Outline of Topics

- Review of inheritance concepts and implementation in Java
- **Wrong uses of inheritance**
- Benefits of inheritance
- Problems with inheritance
  - Fragility
    - Rectangle-Square Problem
  - Violates encapsulation: Ripple effect
    - Enhancing HashSet
- Best Practice (J. Bloch): Design for inheritance or else prevent it
- Using Composition
  - Instead of inheritance – Example: a Stack class
  - In combination with inheritance – Example: Inheriting from a Role

Wrong Use of Inheritance: Convenient Code Re-use

- We've written the code for move() in our car class, and we want to re-use this code for our bicycle class.
- Why is this a bad design decision?

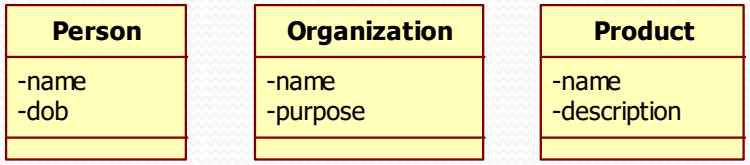


Changes to the Car class would be inherited by Bicycle and such changes may not make sense.

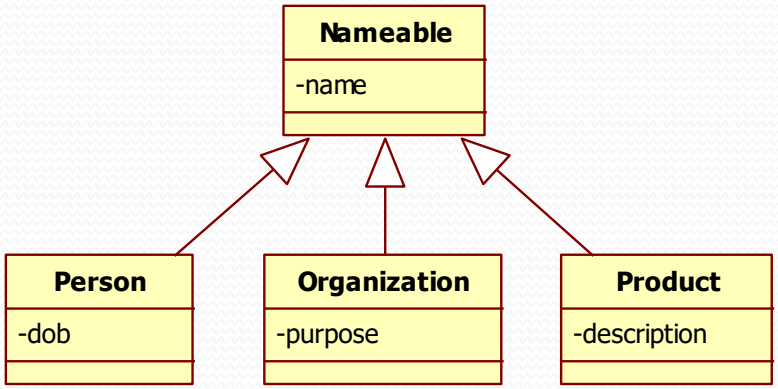
For instance: Car class may be updated to have a choice between automatic and manual transmission, and then be equipped with an abstract method updateTransmission(). Then Bicycle will have to implement it – but it would be meaningless to do so.

Inheritance Just for Code Reuse

- The following classes all have a name property

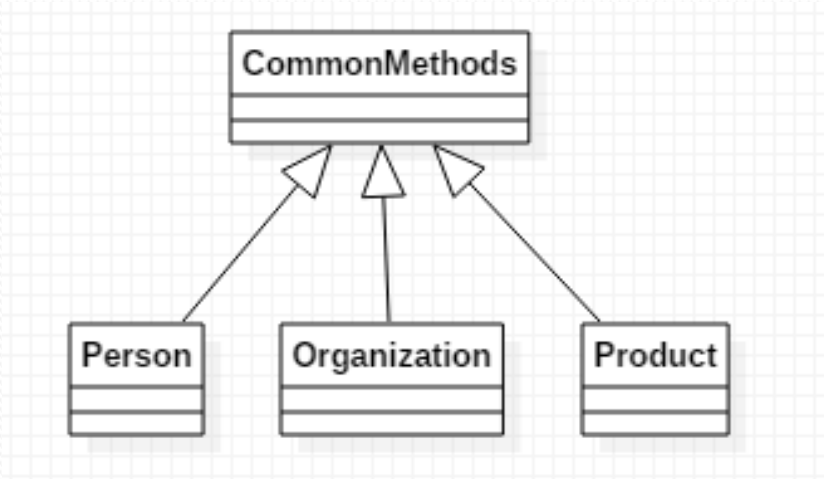


- Why is this use of inheritance a poor design decision?



- **Rigidly binds these classes together**
- **Name may evolve in different ways**
- **Scenario: What if Person is updated to have firstName and lastName?**

- What about using inheritance like this?



Problems:

- **Violates IS-A and LSP**
- **Wastes inheritance opportunity**
- **CommonMethods will evolve into a mess – eventually it will contain methods that may be useful only for one or two of its subclasses.**
- **Use utility class instead**

Exercise 3.1

For which of the pairs A, B of classes shown below is it correct to say that A inherits from B?

- Cat, Animal
- DigitalWatch, Timepiece
- HumanBeing, Being
- CompanyPresident, Employee
- Customer, Account
- Book, Library
- Stack, List
- Circle, Ellipse 

Exercise 3.1 - Solution

For which of the pairs A, B of classes shown below is it correct to say that A inherits from B?

- Cat, Animal - **Good (Cat IS-A Animal)**
- DigitalWatch, Timepiece – **Good (DigitalWatch IS-A Timepiece)**
- HumanBeing, Being – **Not Bad (but maybe not useful)**
- CompanyPresident, Employee - **Good (usually the case)**
- Customer, Account – **Bad (Customer is not an Account)**
- Book, Library – **Bad (Book is not a Library though it may belong to one)**
- Stack, List - **?? See upcoming slides**
- Circle, Ellipse **?? See upcoming slides**

Subtle Mistake Using Inheritance

What's wrong with the following implementation of a stack? (Hint: Problem shows up when you try to apply LSP.)
Note: the Java 1.1 library implementation of Stack made the same mistake – jdk 1.2 fixed it, but IS-A principle is still violated.

```
class Stack<T> extends ArrayList<T> {
    private int stackPointer = 0;

    public void push(T article) {
        int insertPosition = stackPointer++;
        add(insertPosition, article);
    }

    public T pop() {
        return remove(--stackPointer);
    }
}
```

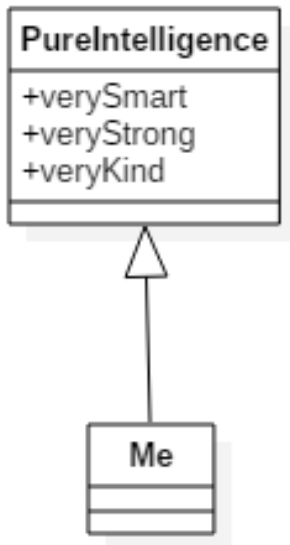
See Demo:
`lesson03.lecture.stacklinkedlist`

Main Point 1

Inheritance is used to model IS-A relationships and must obey the Liskov Substitution Principle.

Although Inheritance offers reuse (the subclass inherits all public and protected methods and attributes), reuse should never be the *sole reason* for creating an inheritance relationship.

The field of pure intelligence is inherited by everyone, and can easily be accessed through the practice of the TM technique.



Outline of Topics

- Review of inheritance concepts and implementation in Java
- Wrong uses of inheritance
- **Benefits of inheritance**
- Problems with inheritance (even when used correctly)
  - Fragility
    - Rectangle-Square Problem
  - Violates encapsulation: Ripple effect
    - Enhancing HashSet
- Best Practice (J. Bloch): Design for inheritance or else prevent it
- Using Composition
  - Instead of inheritance – Example: a Stack class
  - In combination with inheritance – Example: Inheriting from a Role

Benefits of Inheritance

- It reduces code redundancy. (E.g. Faculty, Secretary classes.)
- Subclasses are much more succinct (smaller class file) than they would be without inheritance. (E.g. Faculty, Secretary classes.)
- You are reusing and extending code that has already been thoroughly tested – without modifying it. (E.g. Manager class)
- You can derive a new class from an existing class even if you don't own the source code for the latter! (See demo: `lesson03.lecture.inheritance1.MyStringList`.)

Outline of Topics

- Review of inheritance concepts and implementation in Java
- Wrong uses of inheritance
- Benefits of inheritance
- **Problems with inheritance (even when used correctly)**
  - Fragility
    - Rectangle-Square Problem
  - Violates encapsulation: Ripple effect
    - Enhancing HashSet
- Best Practice (J. Bloch): Design for inheritance or else prevent it
- Using Composition
  - Instead of inheritance – Example: a Stack class
  - In combination with inheritance – Example: Inheriting from a Role

Fragility of Inheritance

Subclasses of a superclass – even when the IS-A criterion is met – may use the superclass in unexpected ways leading to broken code.

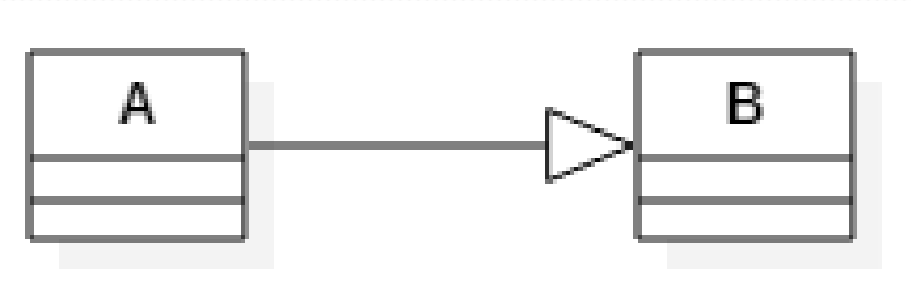
- Example: the Rectangle-Square Problem

  See `lesson03.lecture.inheritance2`
- The Rectangle-Square Problem arises because of the presence of setters.
  - Code shows that Square should not inherit from Rectangle because of LSP
  - If setters are not allowed (which would mean that Square and Rectangle are considered immutable), there is no problem about inheritance.
- Viewing a Circle as a subclass of Ellipse leads to the same set of issues.

Inheritance Violates Encapsulation: The Ripple Effect

If A is a subclass of B, even if A is not modified in any way, a change in B can break A. (This is called the *Ripple Effect*.)

- Example 1: Suppose Super class B changes the method signature, subclass breaks at compile time. (`lesson03.lecture.inheritance5`)



Example 2: Extending `HashSet` – see `lesson03.lecture.inheritance3`.

- *Problem*: In implementation of `HashSet`, `addAll` calls the `add` method, so we are incrementing `addCount` too many times in calls to `addAll`.
- Fix: Don't increment `addCount` in `addAll` operations
- *The real problem*: Now `ExtendedHashSet` depends on an undocumented implementation detail of `HashSet`. If creators of `HashSet` change the implementation of `addAll`, `ExtendedHashSet` could break.

This is an example of the Ripple Effect; the internal implementation of `addCount` in `ExtendedHashSet` can be undermined by a change in implementation in the super class – this is a violation of encapsulation.

Outline of Topics

- Review of inheritance concepts and implementation in Java
- Wrong uses of inheritance
- Benefits of inheritance
- Problems with inheritance (even when used correctly)
  - Fragility
    - Rectangle-Square Problem
  - Violates encapsulation: Ripple effect
    - Enhancing HashSet
- **Best Practice (J. Bloch): Design for inheritance or else prevent it**
- Using Composition
  - Instead of inheritance – Example: a Stack class
  - In combination with inheritance – Example: Inheriting from a Role

Designing for Inheritance

- To support inheritance, a class must document which overridable methods it uses in its own internal operations.

  Example: the remove method in `AbstractCollection`

- More subtle points about use of inheritance may also need to be considered: See Bloch, Effective Java, pp. 88 - 89

Forbidding Inheritance

Three ways:

1. Make the class final, OR
2. Make all constructors private and provide static factory methods to create instances, OR
3. Use sealed classes

What Is a Sealed Class?

- Sealed classes, introduced in Java 15 (as a preview feature) and finalized in Java 17, are a way to restrict which other classes or interfaces may extend or implement them.

Keywords	Meaning
`sealed`	Restricts which classes/interfaces can extend or implement it
`permits`	Declares **exactly** which classes are allowed to extend it
`final`	Prevents further subclassing
`non-sealed`	Allows further subclassing (removes sealing for that subclass)

Example of Sealed Class

```
public sealed abstract class Shape permits Circle, Square {
        public abstract double area();
}

public final class Circle extends Shape {
    private double radius;
    public Circle(double r) { this.radius = r; }
    public double area() { return Math.PI * radius * radius; }
}

public non-sealed class Square extends Shape {
    private double side;
    public Square(double s) { this.side = s; }
    public double area() { return side * side; }
}
```

Rules

- Must Declare Permitted Subtypes
- All permitted subclasses must be in the same module or same package.
- Subclasses must explicitly declare themselves as:
  - `final`
  - `sealed`
  - `non-sealed`
- Enforced at Compile Time
  - If any class not listed in `permits` tries to extend a sealed class, the compiler will throw an error.

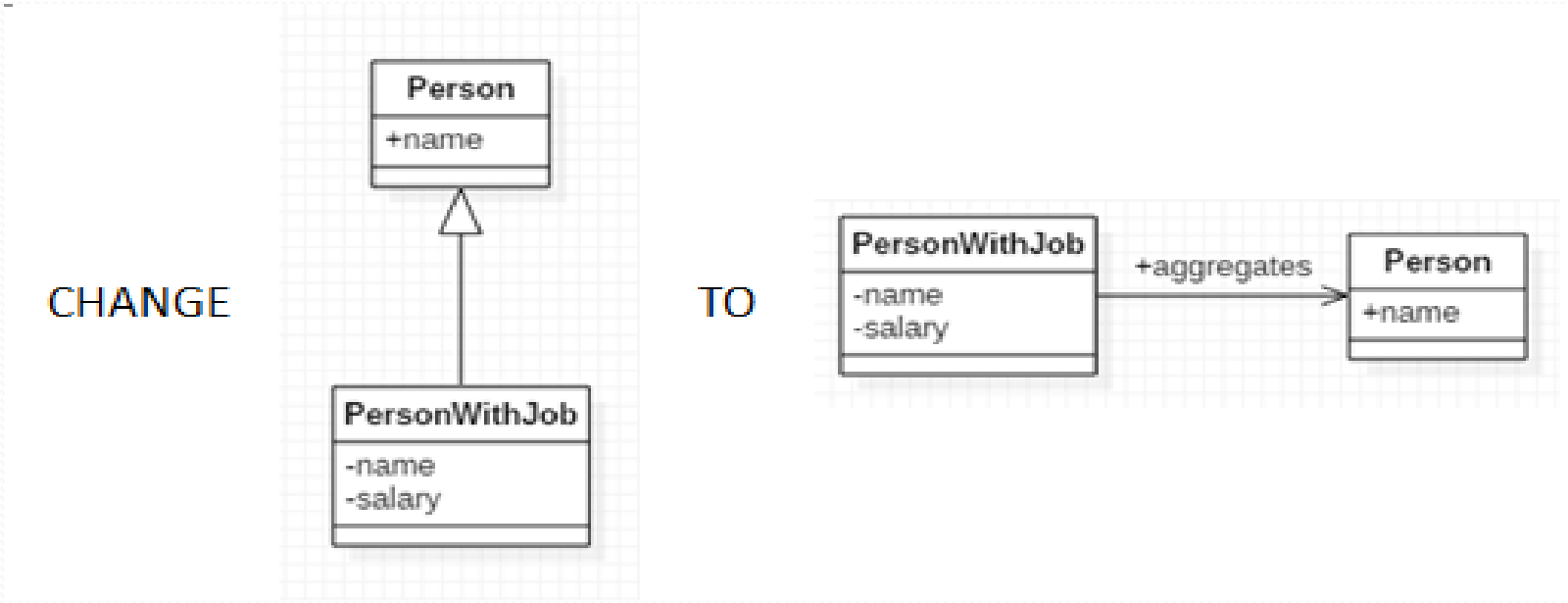
Outline of Topics

- Review of inheritance concepts and implementation in Java
- Wrong uses of inheritance
- Benefits of inheritance
- Problems with inheritance (even when used correctly)
  - Fragility
    - Rectangle-Square Problem
  - Violates encapsulation: Ripple effect
    - Enhancing HashSet
- Best Practice (J. Bloch): Design for inheritance or else prevent it
- **Using Composition**
  - Instead of inheritance – Example: a Stack class
  - In combination with inheritance – Example: Inheriting from a Role

Using "Composition" Instead of Inheritance

Demo: lesson03.lecture.composition1

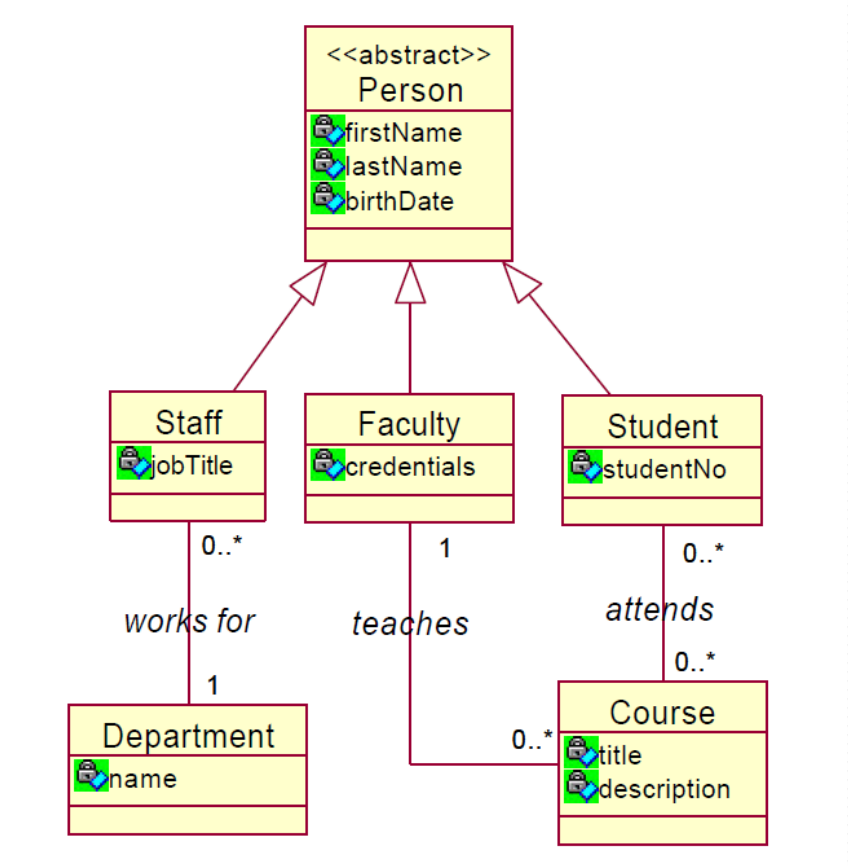
- To avoid the pitfalls of inheritance, it is always possible to use composition instead of inheritance.
- To illustrate the technique, imagine two classes, `Person` and `PersonWithJob`. Instead of asking `PersonWithJob` to inherit from `Person`, you can *compose* `Person` in `PersonWithJob` and forward requests for `Person` functionality to the composed class. We still get the benefit of reusing `Person`.



Example: Better Implementation of Stack

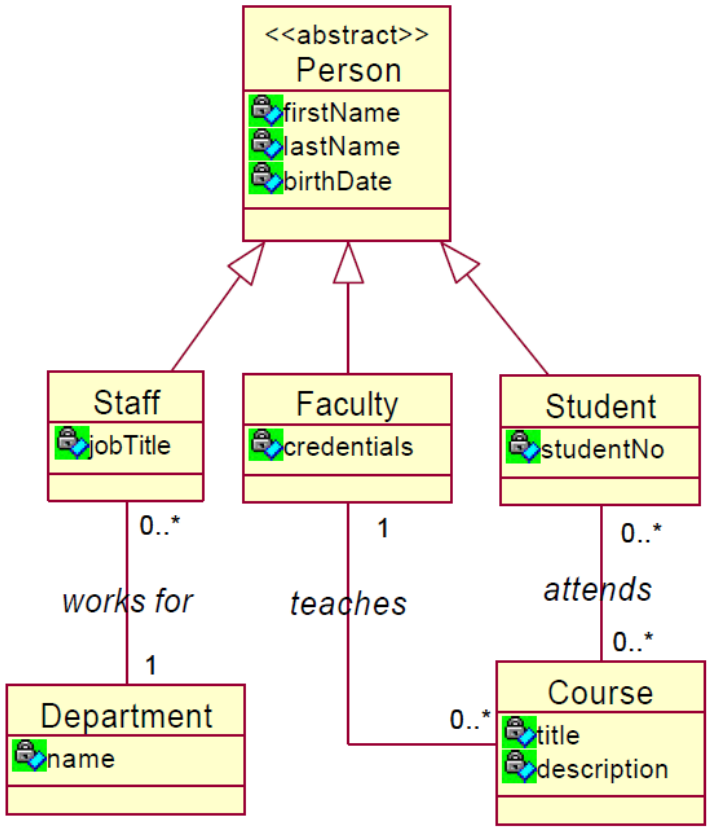
See `lesson03.lecture.composition2` for an implementation using the Composition pattern

Example
Composition and Inheritance



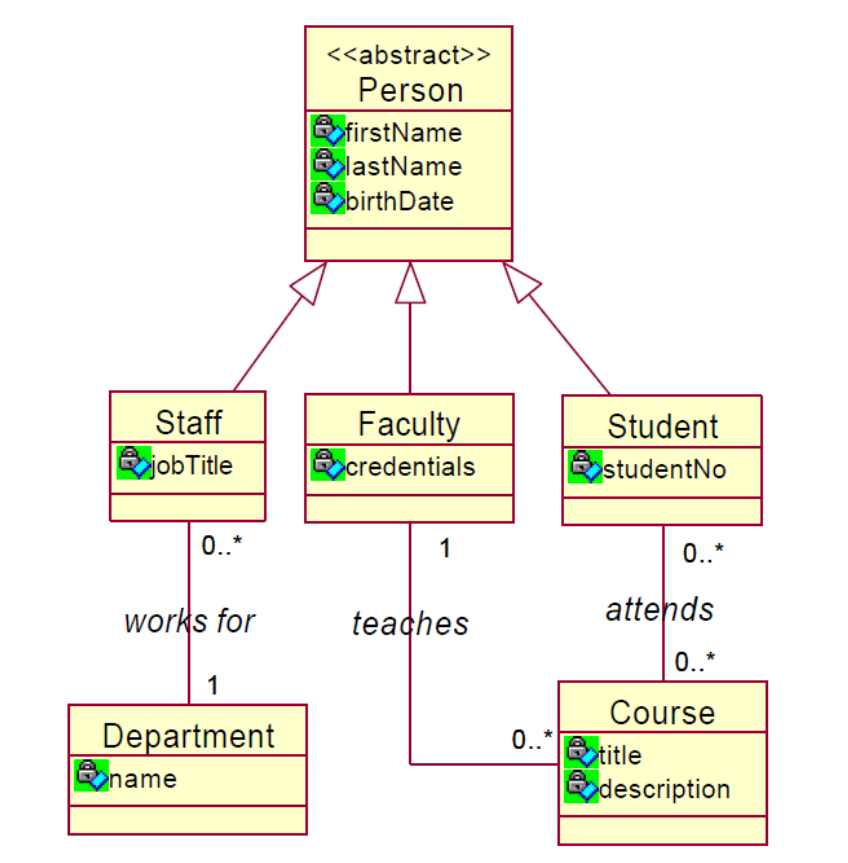
What are some limitations of this design?

Example
Composition and Inheritance



- Problems:
  - Inheritance is a static relationship and it must be decided at object construction time which type of person someone is
  - Once constructed, a person cannot change from being a Student to being Staff or Faculty
    - In the **real world** people change all the time
  - Also a person cannot assume multiple roles of being a Staff member and a Student at the same time
    - Again, not how **it really works**

Exercise 3.2
Composition and Inheritance



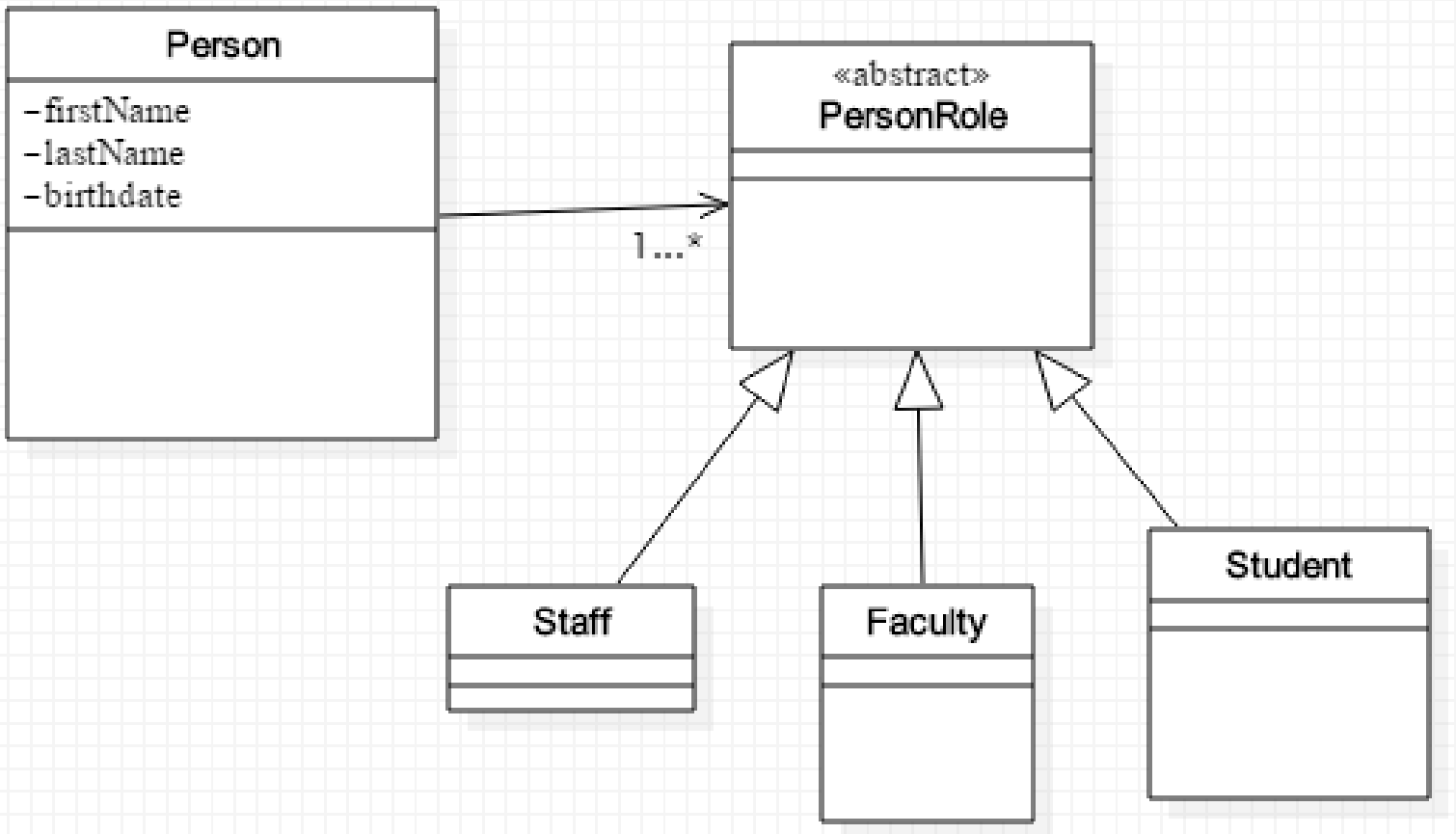
- Think of a way to redesign this class hierarchy using composition. You do not need to eliminate inheritance, but can you use composition to solve the problems mentioned in the previous slide?

Once constructed, a person cannot change from being a Student to being Staff or Faculty

Also a person cannot assume multiple roles of being a Staff member and a Student at the same time

Solution

- Introduce a PersonRole class. This allows a Person to assume one or more PersonRoles



Main Point 2

Inheritance should be used only when you have a clear IS-A relationship and even then, a careful plan for using inheritance should be thought through. Otherwise, it is better to forbid inheritance and use composition.

Even in clear IS-A relationships, inheritance may not be the best choice because of its inflexibility.

Software relationships that reflect the real world are more natural and easier to understand. Likewise, life in accord with natural law tends to go forward without obstacles; life in violation of natural law tends to be “bumpy”.

Summary

Today we considered some of the advantages and disadvantages of using inheritance. We must be cautious when using inheritance because it is a permanent relation for the lifetime of an object. This fact can conflict with our goal to build software that supports change and extensibility.

In general, composition has better support for change so we favor using composition except in cases where we have a clear 'is-a' relationship and anticipate the need for polymorphism.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

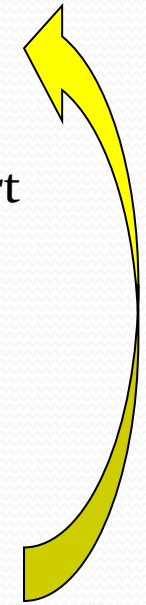
1. When requirements change, you should implement these changes by adding new code, not by changing old code that already works.

2. Inheritance and Composition are Object-Oriented principles that support reuse of implementation.

---

3. **Transcendental Consciousness** is the infinitely adaptable field of pure intelligence that can be 'reused' by every individual in all places, at all times.

4. **Wholeness moving within itself**: In Unity Consciousness, the individual is united with everything else, and inherits the total potential of natural for fulfillment of all desires spontaneously.



Appendix: The Gosling Rules

Whenever we have 3 Java classes, `SuperClass`, `CallingClass`, `ObjectRefClass` so that

- `SuperClass` and `CallingClass` belong to different packages
- `SuperClass` has a `protected` member proMem
- `CallingClass` and `ObjectRefClass` are both subclasses of `SuperClass`
- A method inside `CallingClass` attempts to access `proMem` by way of an object reference to `ObjectRefClass` – for instance

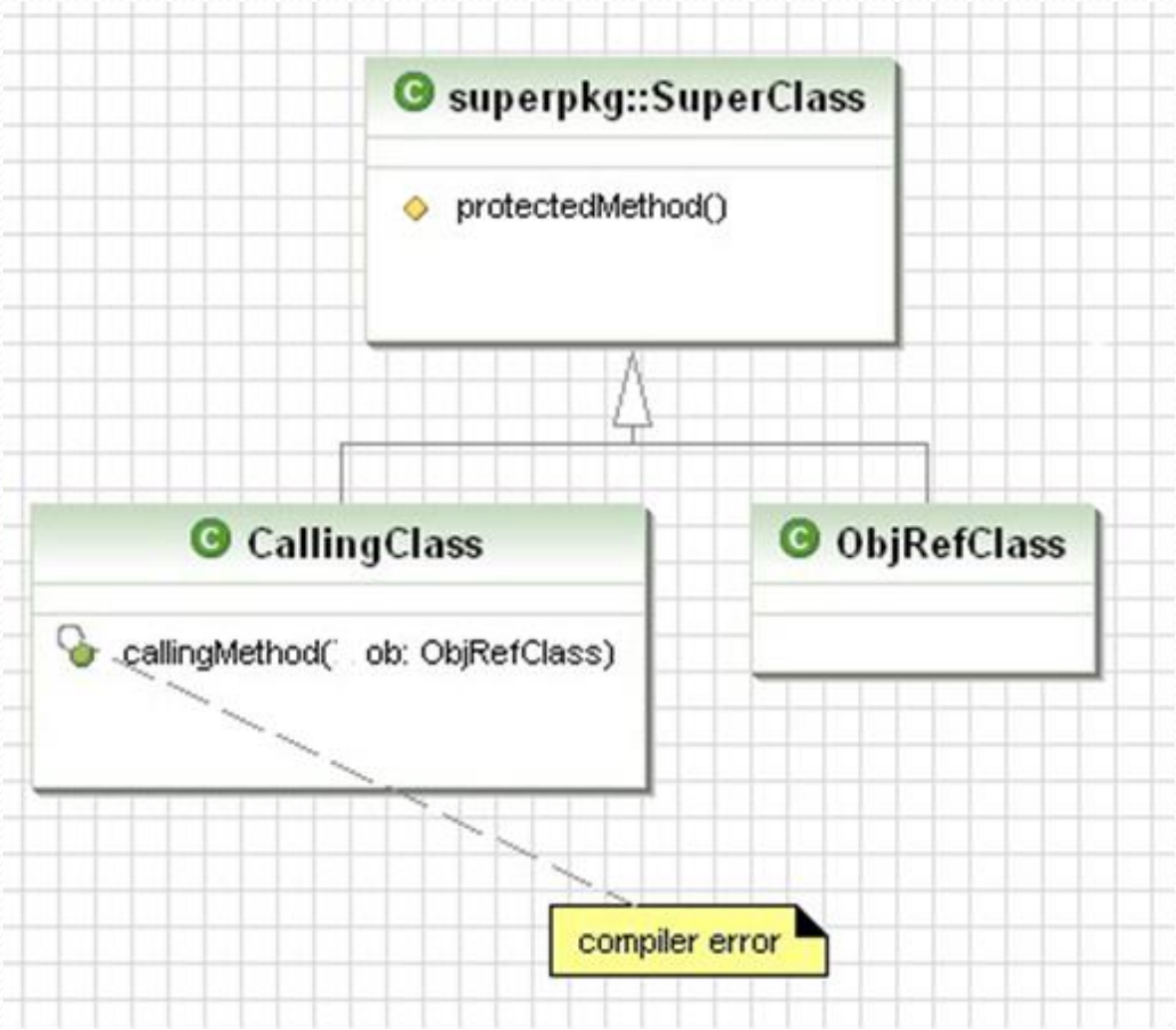
```
void aMethod(ObjectRefClass ref) {
        ref.proMem;
}
```

- AND, `ObjectRefClass` is NOT a subclass of `CallingClass`

THEN

There is a compiler error – `proMem` is not considered to be visible to ref.

Example: For our `clone()` example, the `SuperClass` is `Object` (and the `protected` member `proMem` is `Object's clone()` method), the `CallingClass` is `CallingClass`, and the `ObjRefClass` is `MyDataClass`.


superpkg::SuperClass
protectedMethod()
CallingClass
callingMethod(ob: ObjRefClass)
ObjRefClass
compiler error

Appendix: The Gosling Rules

[From *Gosling, The Java Programming Language*]

A. A `protected` member of a class is always accessible within that class

B. A `protected` member of a class `SuperClass` is always accessible to any method of a calling class that lives in the same package as `SuperClass` (even if the method sends a message to a subclass `ObjRefClass` (of `SuperClass`) object belonging to a different package).

C. A `protected` member of a class `SuperClass` can also be accessed from a another class `CallingClass` in a different package, through an object reference (an instance of `ObjRefClass`) that is a narrower type than, or of the same type as, `CallingClass` – i.e. another instance of `CallingClass` or an instance of a subclass of `CallingClass`. If the type of the object reference fails to be the same as or narrower than `CallingClass`, access to the `protected` member is not allowed.

Appendix: Three Ways to Resolve the Protected Paradox

1. Make `ObjRefClass` a subclass of `CallingClass`
2. Put `CallingClass` in the same package as `SuperClass`
3. Override `SuperClass`'s `protected` method in `ObjRefClass` (but the new version of the method should now be `public`*; this overriding method is then called by `CallingClass`). This 3rd solution is not related to the Gosling rules – it just uses the fact that public methods of an instance can be accessed by other objects.

*`public` is the typical way. It can also be `protected`, but then, in accordance with the Gosling Rules, either `CallingClass` and `ObjRefClass` must belong to the same package, or `ObjRefClass` must be a subclass of `CallingClass`.

Solution 3 is the way `clone()` must be handled. In general, if you are subclassing a third-party class such as `SuperClass` (as we do always in relation to the class `Object`), Solution 3 is our only option. (demo)

Appendix:
Example: Protected Member Accessible If Object Ref Class Is a Subclass of Calling Class

```
public class CallingClass {
    public MyClass tryToClone(MyClass cl)
            throws CloneNotSupportedException {

        //ok because MyClass extends CallingClass
        return (MyClass) cl.clone();
    }

    public static void main(String[] args) {
        CallingClass cc = new CallingClass();
        MyClass cl = new MyClass();
        try {
            //This works
            MyClass result = cc.tryToClone(cl);
        }
        catch (CloneNotSupportedException e) {
        }
    }
}

public class MyClass extends CallingClass implements Cloneable {
    String name = "harry";
}
```