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

Lecture 3: Inheritance and Composition

Reflecting the Whole in the Part

Wholeness of the Lesson

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.

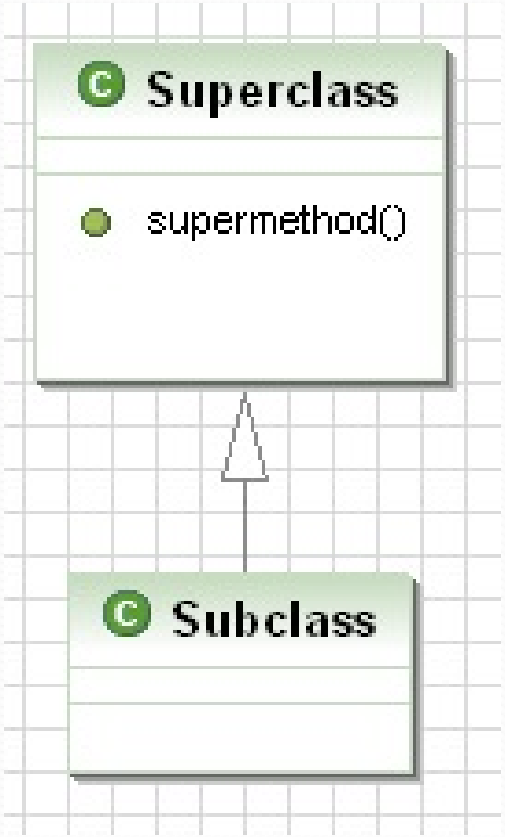
Composition and inheritance are techniques based on the principle of preserving sameness in diversity, silence in dynamism

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

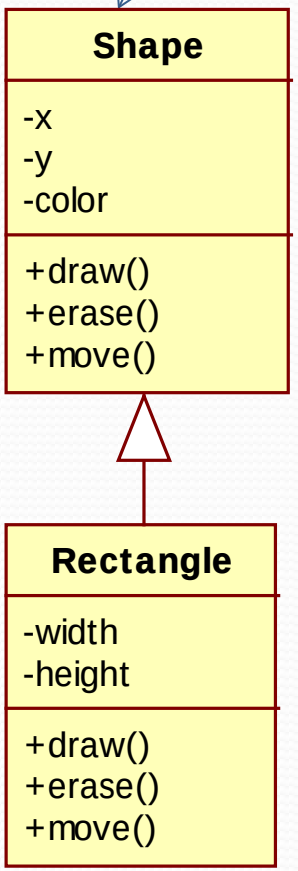
```
class Superclass {
    protected void supermethod() {
        int x = 0;
    }
}
class Subclass extends Superclass {
}
class Main {
    public static void main(String[] args) {
        Superclass sub = new Subclass();
        //subclass has access to non private data
//and non-private methods of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



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 {
    ...
}
```

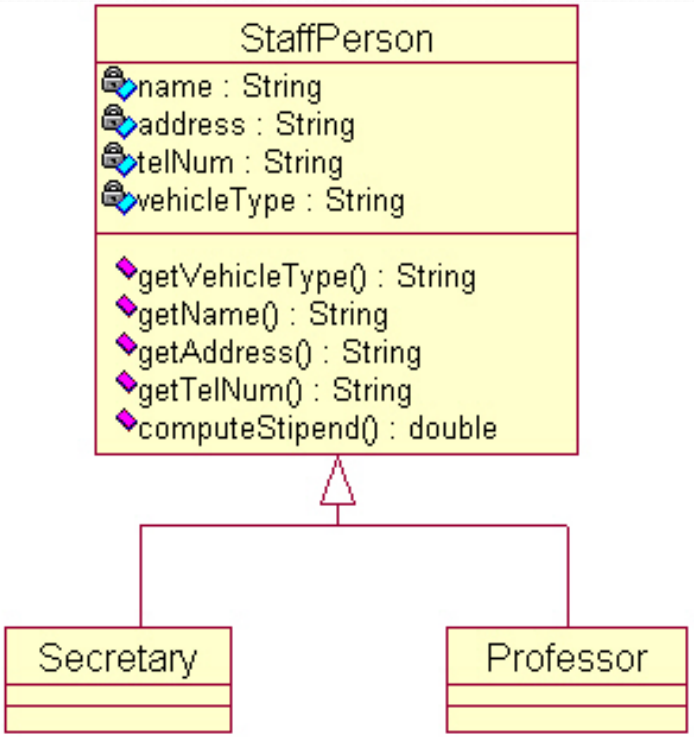
Rectangle inherits all attributes
and methods from Shape that are not private

Inheritance Arises . . .

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

Secretary
properties:
name
address
phone number
drivesVehicle
salary
behavior:
computeSalary()

Professor
properties:
name
address
phone number
drivesVehicle
salary
behavior:
computeSalary()



See demos in `lesson3.lecture.polymorphism1, lesson3.lecture.polymorphism2`

And . . .

```
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;
}
```

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

```
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;
}
```

Rules Concerning Inheritance

- A subclass constructor must make use of one of the superclass constructors (see Manager class), but does not need the same signature as any of these constructors
- A class can have multiple (overloaded) constructors. To call one constructor from another, "this" is used (must be the first line of the constructor).
- Example:

```
public Employee(String name) {
        this(name, 0.00);
 }
```

- A constructor can call a superclass constructor using "super". See Manager class (also notice super is used in another way to call a superclass method).
- To prevent a class from having any subclasses, the class can be declared *final*.
- If A is a subclass of class B, when the constructor of A is invoked, there is a specific sequence of steps by which the static/instance variables are initialized and the bodies of the two constructors are executed.

Order of Execution in a Class

When a class is used for the first time, it needs to be loaded.

1. **Static Initialization** : After a class is loaded to the memory, its static data fields and static initialization block are executed in the order they appear in the class. (Static fields are initialized only once; static blocks executed only once.)

2. **Instance initialization block** : It is initialized immediately after all static initialization has occurred (and before any instance variables are initialized). There are mainly three rules for the instance initializer block. They are as follows:

    1. The instance initializer block is created when instance of the class is created.
    2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
    3. The instance initializer block comes in the order in which they appear.

3. All instance variables are initialized with their default values.
4. If the first line of the constructor calls another constructor, the body of the another constructor is executed, then the body of the constructor is executed.

DEMO: package lesson3.lecture.orderofexec

Using the Default Constructor

A subclass may make use of the implicit (default) constructor *only if* either

- the no-argument constructor of the superclass has been explicitly defined, OR
- no constructor in the superclass is explicitly defined

In either of these cases, the subclass may make use (possibly implicitly) of the superclass' default constructor.

Example

```
//This is ok
class Employee{
            //…//
}
class Manager extends Employee {
            //…//
}
```

Example

```
//This is ok

class Employee{
    Employee(String name, double salary) {
            //…//
    }

     //explicit coding of default constructor
     //since another constructor is present
     Employee() {
          //…//
     }
}

class Manager extends Employee {
    //no explicit constructor call here,
    //so the  superclass default
    //constructor is used implicitly
}
```

Overriding a method

- A subclass can change inherited behavior of the super class by overriding methods
- To override an inherited method, the method in the subclass must have the same signature and return type as the method in the superclass
- 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. [ie : IS-A should be replaced with IS- Substitutable-for]

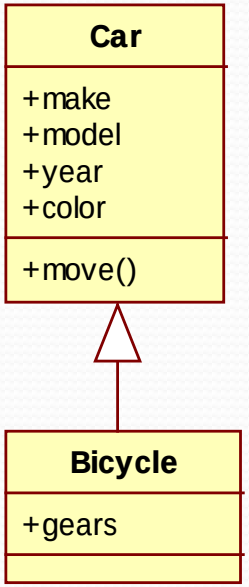
  Example: We may use a Manager instance wherever an Employee instance is expected, so having Manager as a subclass of Employee adheres to LSP.

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

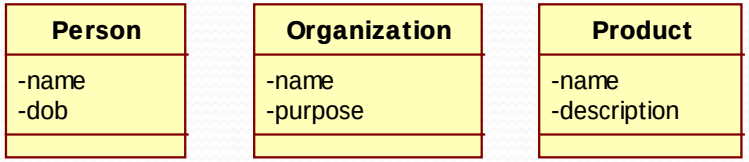
Wrong Use of Inheritance: Convenient Code Re-use

- 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?

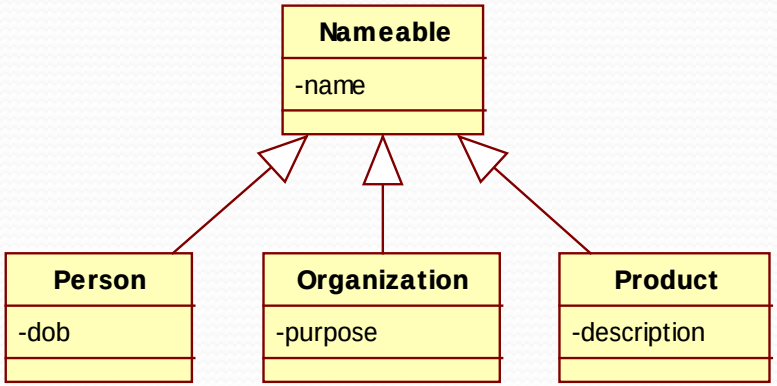


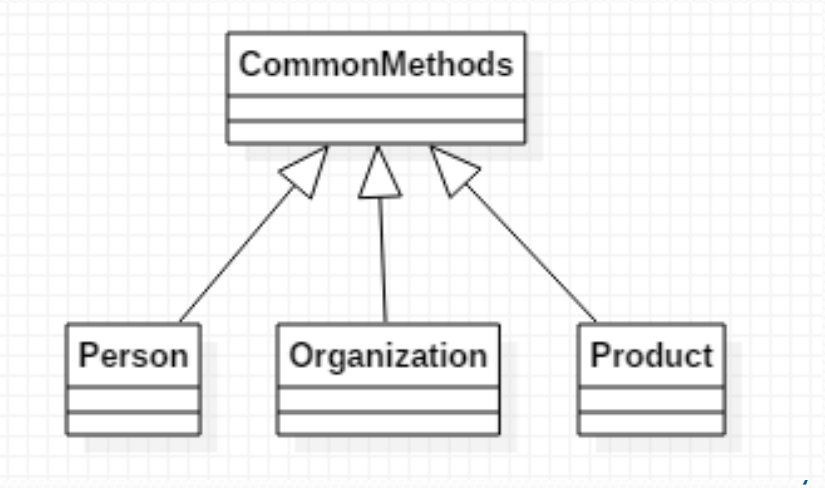
Inheritance Just for Code Reuse

- The following classes all have a name property



- Why do these examples of inheritance represent BAD design decisions?





Inheritance Violating IS-A Principle

What's wrong with the following implementation of a stack?

```
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:
lesson3.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 the *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
- Subclasses are much more succinct (smaller class file). (E.g. Faculty, Secretary classes.)
- You can reuse and extend 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 we don't own the source code for the latter! (from existing class)

  (See demo: `lesson3.lecture.inheritance1`)

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 `lesson3.lecture.inheritance2`
- The Rectangle-Square Problem can be handled by disallowing setters in the Rectangle class. This shows that the reason Square cannot properly inherit from Rectangle is that it violates LSP – LSP implies that we should be able use setSide for Square whenever it would be called for Rectangle. We have seen LSP cannot be satisfied in this example, so it is incorrect for Square to inherit from Rectangle.

Inheritance Violates Encapsulation

- 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: Suppose A overrides all methods in B by first validating input arguments in each method (for security reasons). If a new method is added to B and A is not updated, the new method introduces a security hole.
- Example: Extending HashSet – see lesson3.lecture.inheritance3.
  - Problem: In implementation of HashSet, addAll calls the add method, so we are increment 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, our ExtendedHashSet will break. This is an example of the Ripple Effect; the internal implementation of addCount 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.

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

Forbidding Inheritance

Two ways:

- Make the class final, OR
- Make all constructors private and provide static factory methods to create instances. [Discuss in Lesson-5]

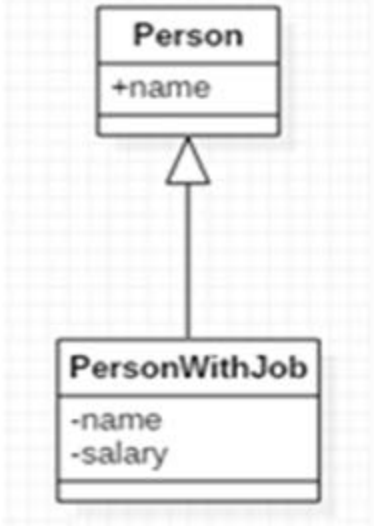
Outline of Topics

- 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

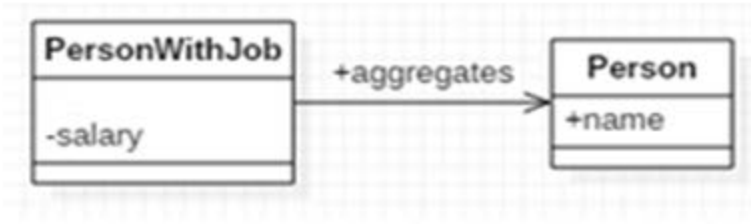
Using “Composition” Instead of Inheritance

- 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.

CHANGE



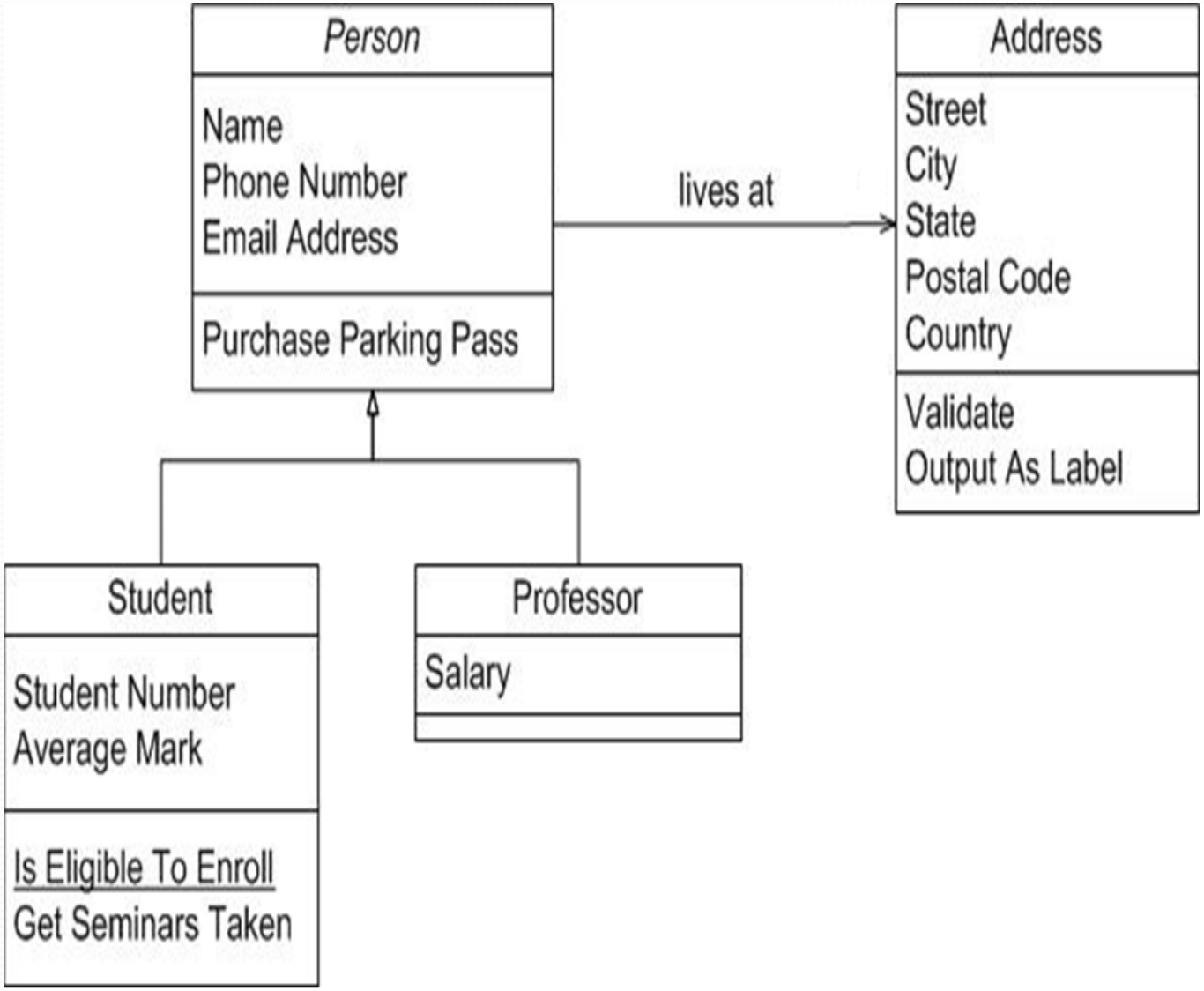
TO



- In Coding :

```
class PersonWithJob{
Person p;
double salary
}
```

Inheritance and Composition

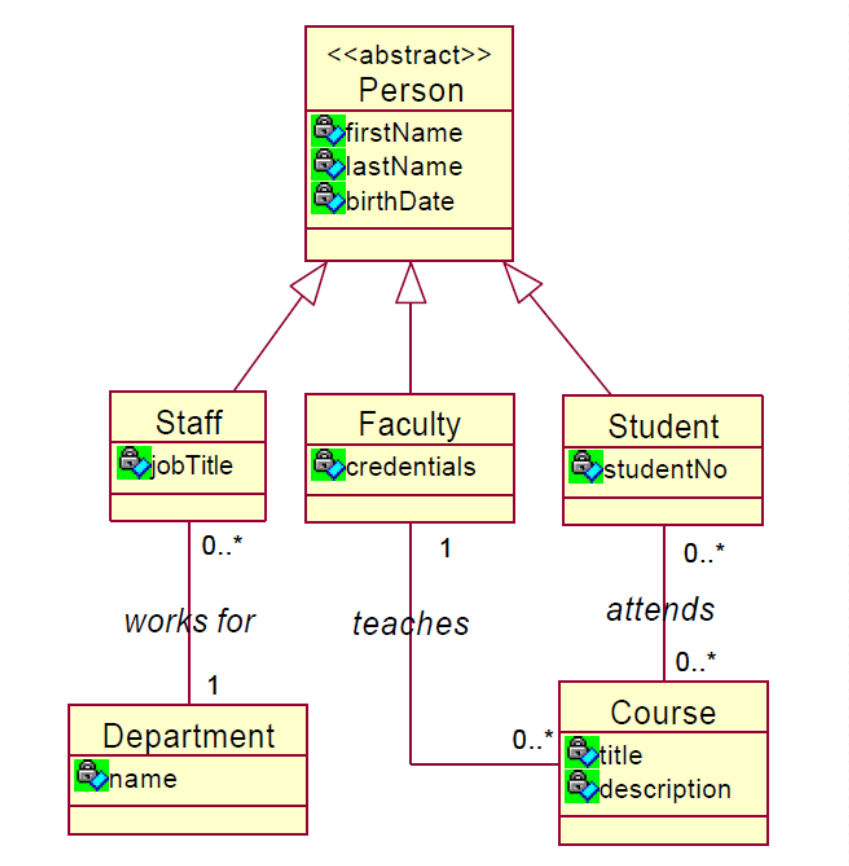


Example: Better Implementation of Stack

See lesson3.lecture.composition2 for an implementation using Composition.

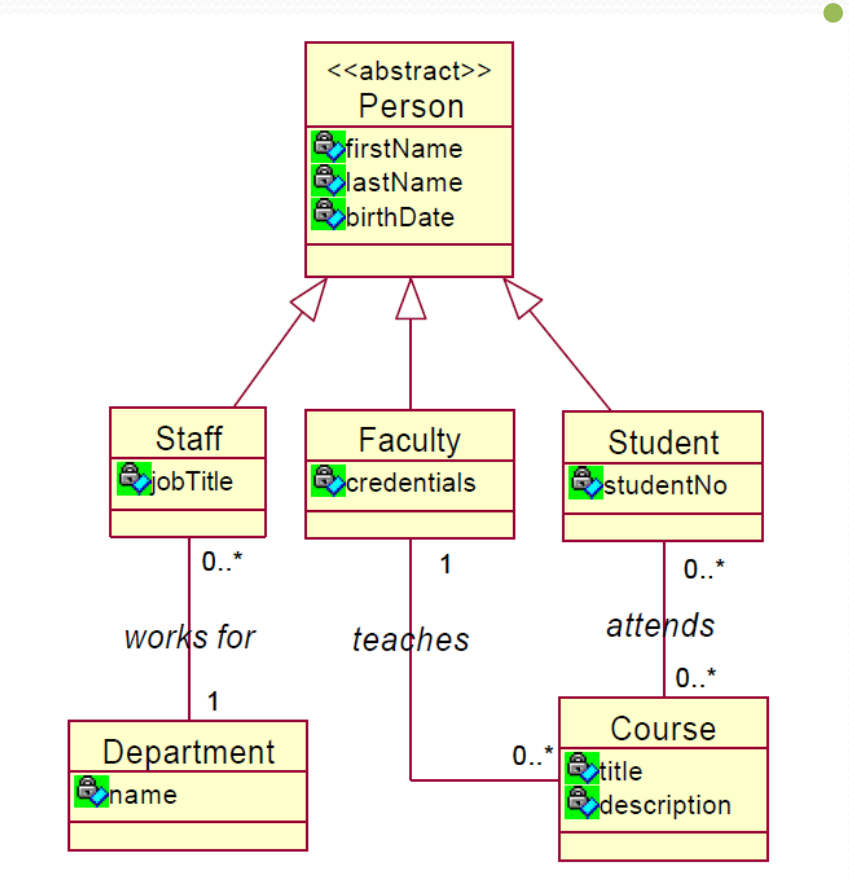
Stack has a LinkedList.

Example
Composition over Inheritance



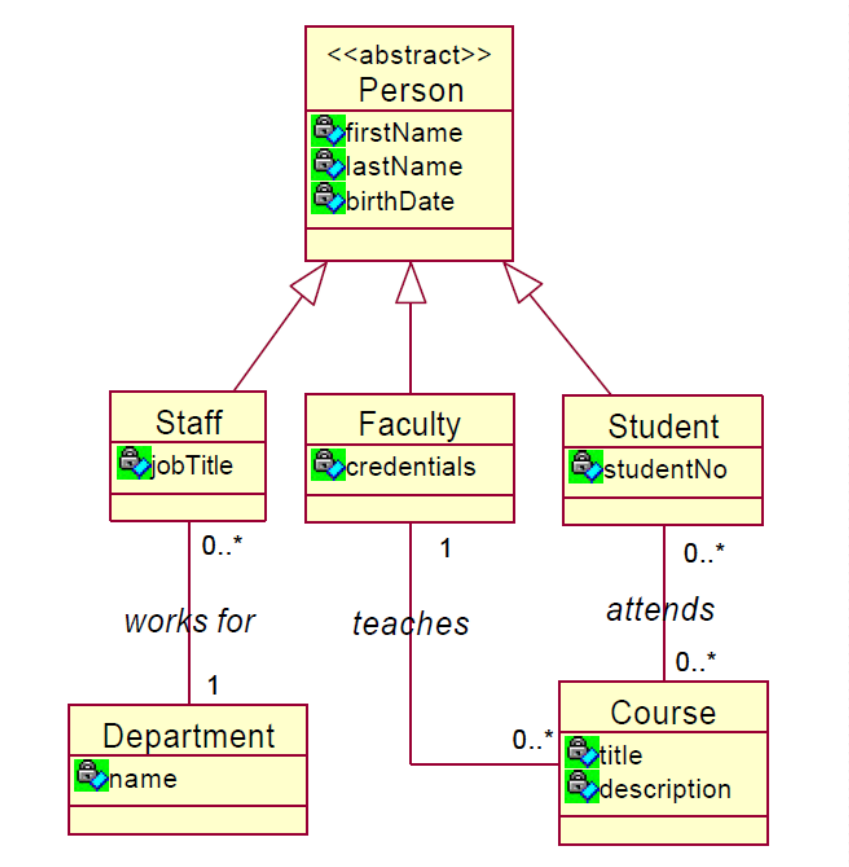
See any problems with this design?

Example
Composition over 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**

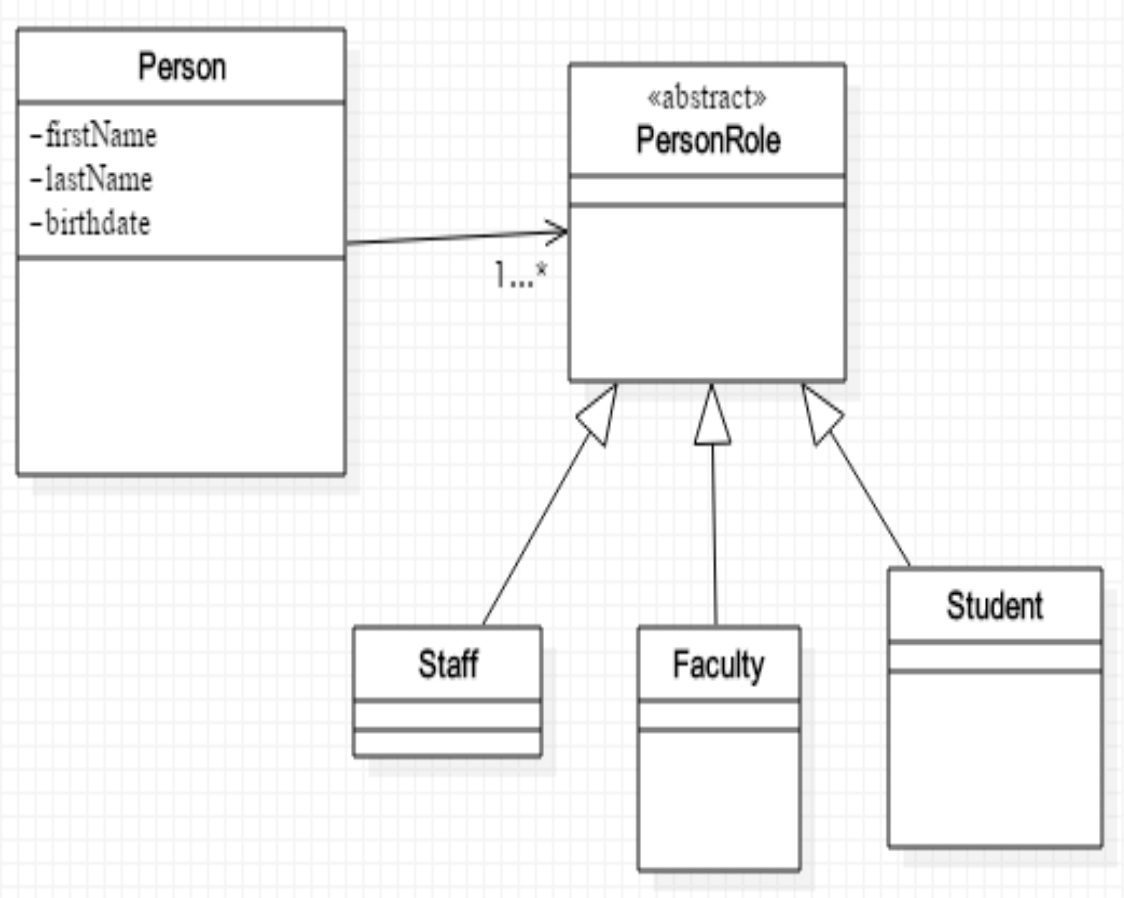
In-class exercise: Composition over Inheritance



- In your small groups try to redesign this class hierarchy using composition. You do not need to eliminate inheritance, but can you use composition to solve the problems mentioned above?

Solution

- Introduce a PersonRole class. This allows a Person to assume 1 or more PersonRoles



In Coding :

```
class Person
{
String fname, lname;
Date birthdate;
List <PersonRole> roles;
// or PersonRole[] roles;
}
```

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".

Exercise

Problem Description: Our rent-a-vehicle business rents cars, trucks, motorcycles, and mopeds. Create an inheritance model that we might use for our rentals.

1) Show a few common attributes and methods for your super-class.

2) Show some unique attributes and at least one unique method for each sub-class.

3) Show one method that will be overridden in all sub-classes.

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.