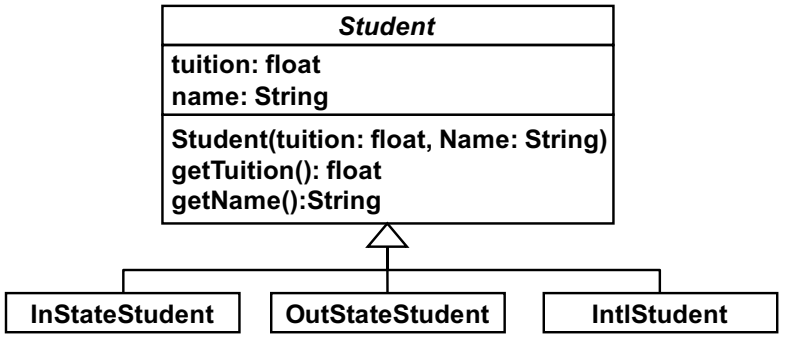
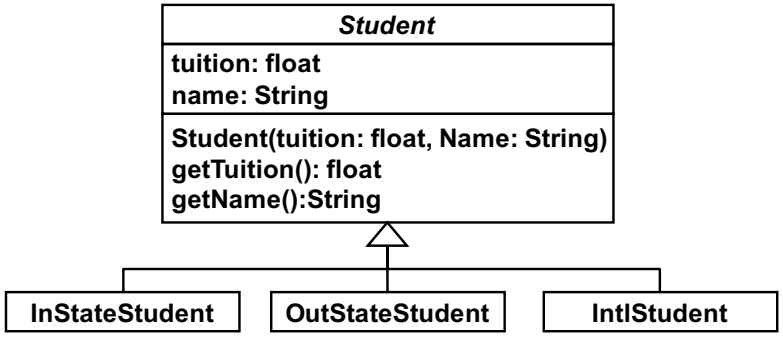
When to Use Inheritance and When not to Use it

An Inheritance Example



- In-state, out-state and int'l students are students.
  - "Is-a" relationship
  - Conceptually, there are no problems.
- A class inheritance is NOT reasonable if subclass instances may want to dynamically change their classes (i.e. student status) in the future.



- An out-state student can be eligible to become an in-state student after living in MA for some years.
- An int'l student can become an in/out-state student through some visa status change.

Dynamic Class Change

- Most programming languages do not allow this.
  - Exceptions: CLOS and a few scripting languages

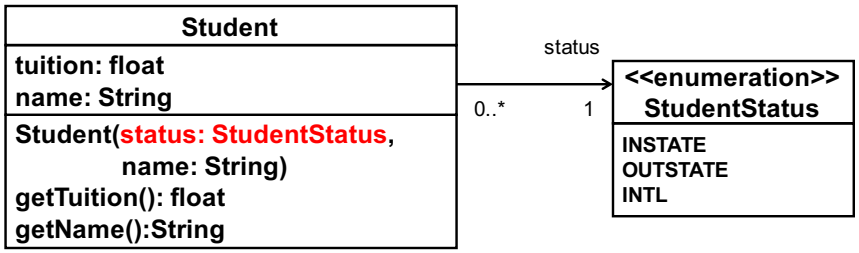
– Need to create a new class instance and copy "some" existing data field values to it.

```
IntlStudent intlStudent = new IntlStudent(…);
new OutStateStudent( intlStudent.getTuition(),
                     intlStudent.getName() );
```

  - Not all existing data field values may go to a new instance.
    - e.g. Data specific to int'l students such as I-20 number and visa #

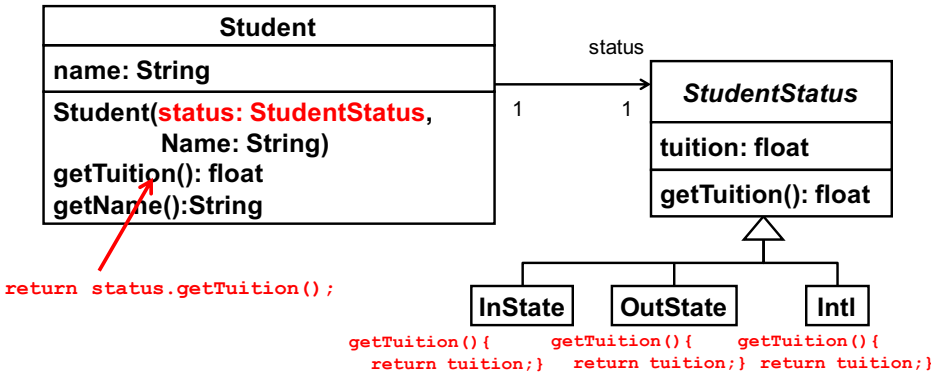
– Need a "deep" copy if an instance in question is connected with other instances.
  - e.g., IntlStudent → Address

Design Tradeoff



- Can allow student status changes
- Need to have conditional statements in getTuition()
  - c.f. lecture note #2 (slide #34)
  - There is a way to remove the conditional statements.
    - *State* design pattern
      - With classes
      - With methods in an enum

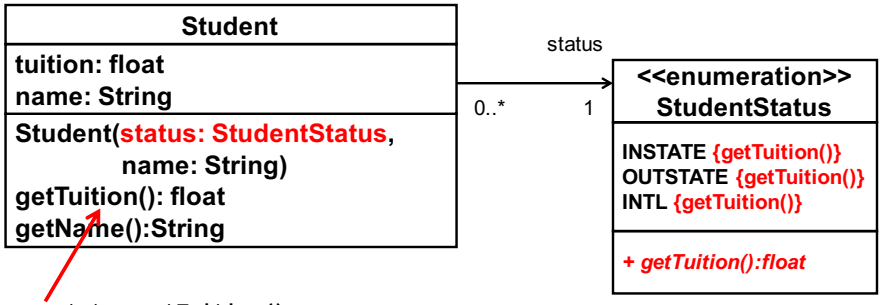
Alternative #1




```
return status.getTuition();
```

```
Student s1 = new Student( new InState(1000), "John Smith" );
s1.getTuition();
```


Alternative #2




```
return status.getTuition();
```

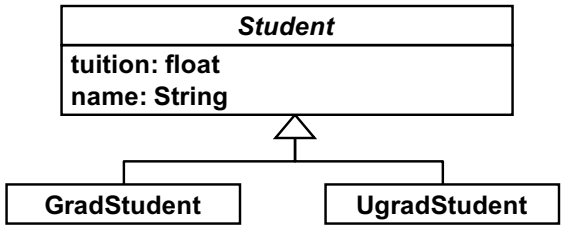
```
Student s1 =
  new Student(StudentStatus.INSTATE,
              "John Smith" );
s1.getTuition();
```

```
public emum StudentCategory{
 INSTATE{
  @Override public void getTuition(){
    return 1000;
  }
 }
 OUTSTATE{
  @Override public void getTuition(){
    return 2000;
  }
 }
 ...
}
```


HW 4-1

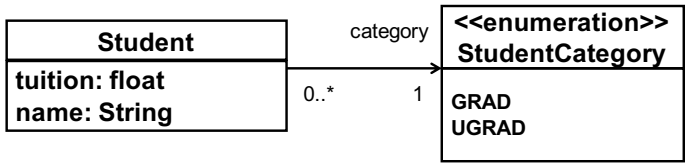
- Implement either of the two alternative designs (#1 or #2)
- [OPTIONAL] If interested, implement both. You will get extra points.

Another Example



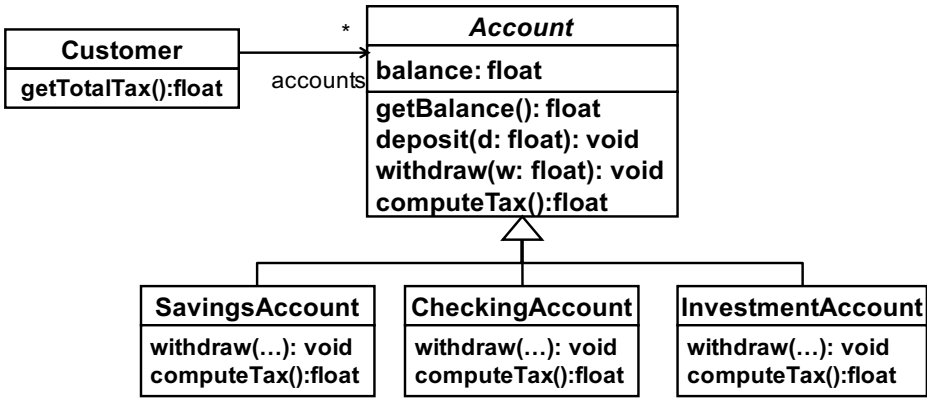
- Grad and u-grad students are students.
  - "Is-a" relationship
  - Conceptually, no problem.
- A class inheritance is NOT reasonable if subclass instances may want to dynamically change their classes in the future.
  - Implementation limitation: Most programming languages do not allow this.

Design Tradeoff

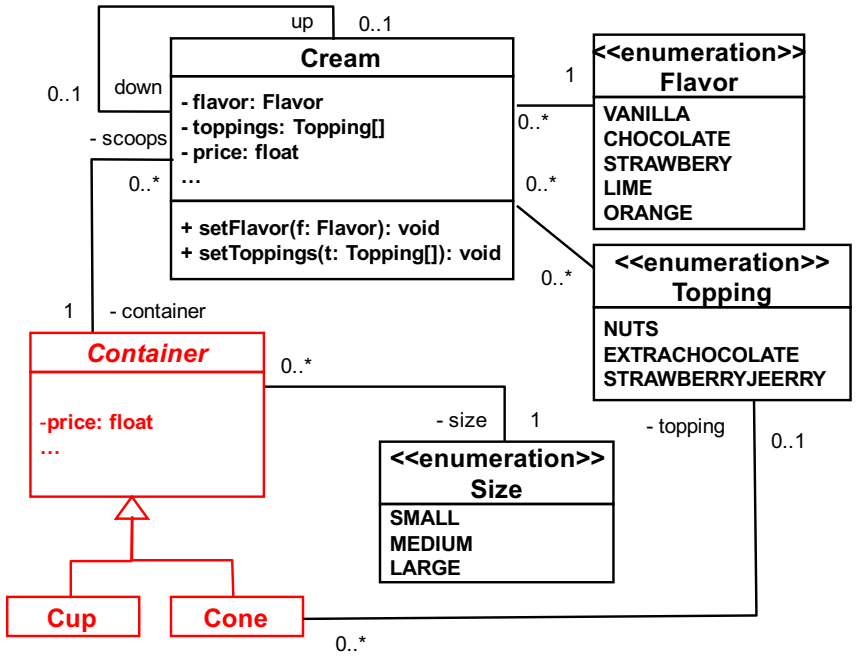


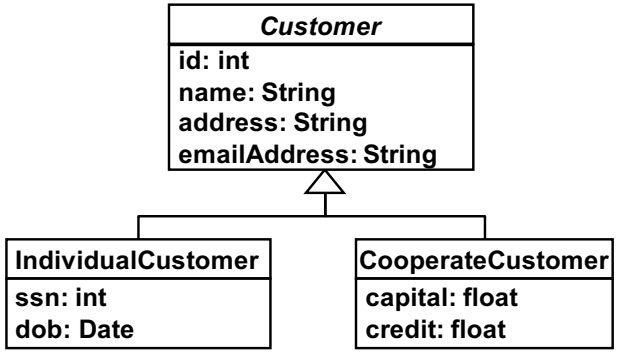
- Can allow student category changes
- Need conditional statements in getTuition()
- Use the *State* design pattern

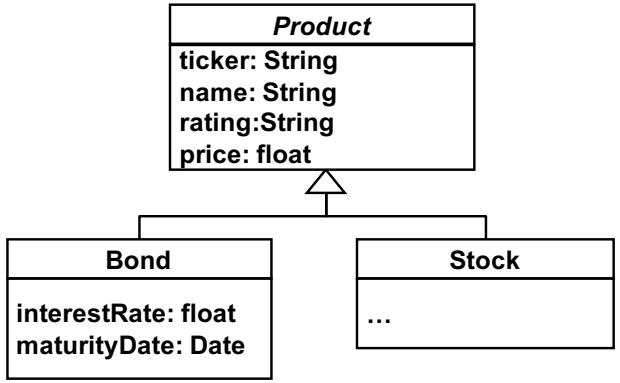
More Examples

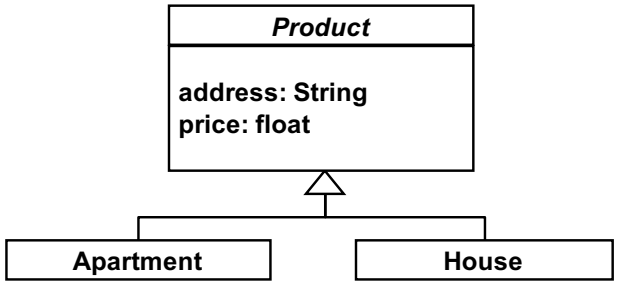


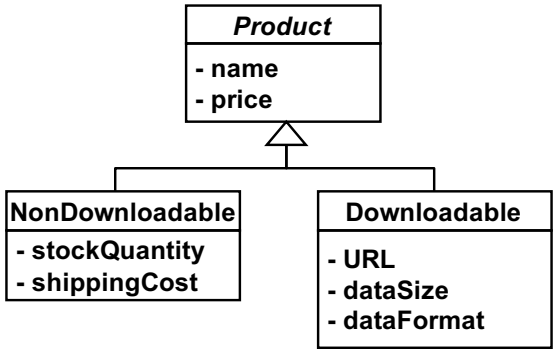
- An Account instance needs to change its type?
  - Savings to checking? No.





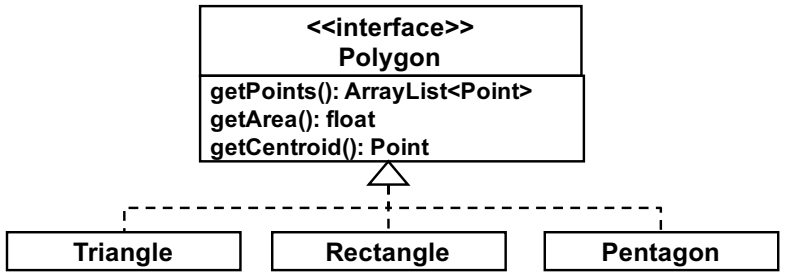




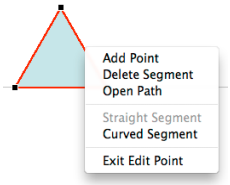


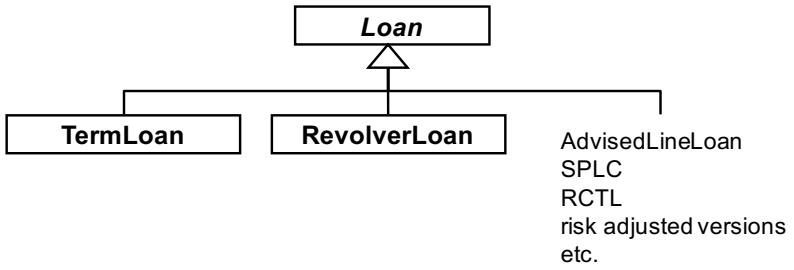
- Assume a product sales app at an online retail store (e.g., Amazon)
- Does this inheritance-based design make sense?
  - An is-a relationship between the super class and a subclass?
  - Does a subclass instance need to change its class?

Some More Examples

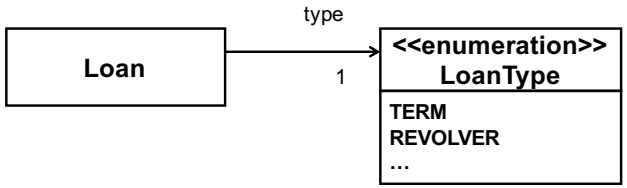


- Can a triangle become a rectangle?
- Do we allow that?
  - Maybe, depending on requirements.

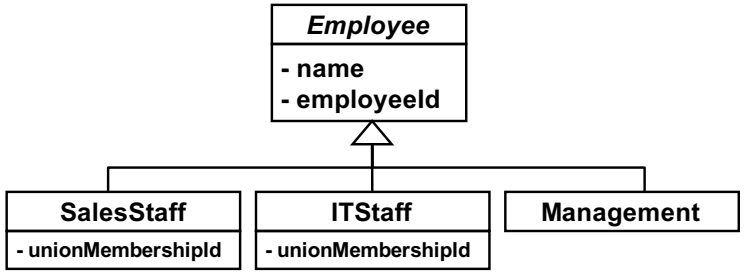




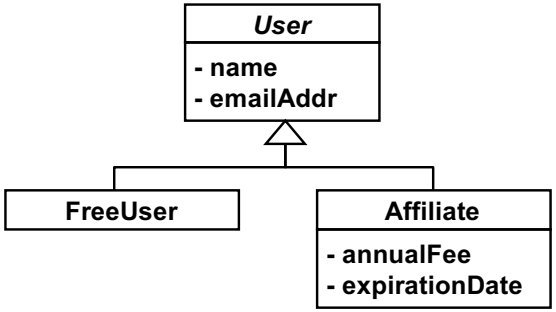
- Term loan
  - Must be fully paid by its maturity date.
- Revolver
  - e.g. credit card
  - With a spending limit and expiration date
- A revolver can transform into a term loan when it expires.



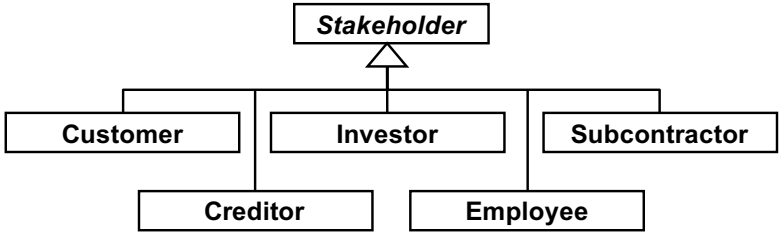
- Use an enumeration
- Use the *State* design pattern



- How about this?
- Assume an employee management system.



- How about this?
- Assume a user management system
  - c.f. Amazon (regular users v.s. Amazon Prime users), Dropbox, Google Drive, etc.

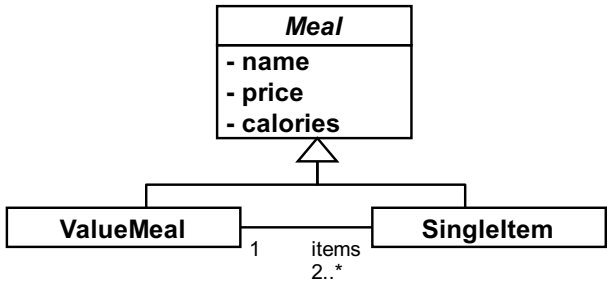


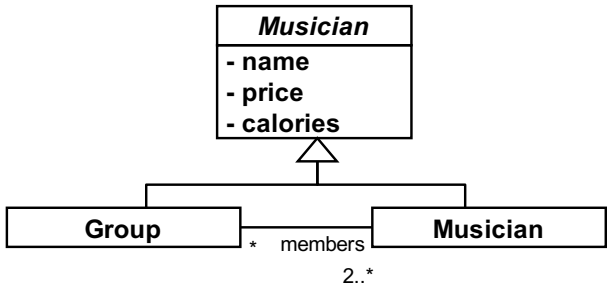
- An employee can be a customer and/or an investor.
- A subcontractor can be a customer.
- If an instance belongs to two or more classes, do not use inheritance relationships.

When to Use an Inheritance?

- An “is-a” relationship exists between two classes.
- No instances change their classes dynamically.
- No instances belong to more than two classes.

Whole-Part Structure





HW4-2

- Replace Type Code with Class
  - http://www.refactoring.com/catalog/replaceTypeCodeWithClass.html
  - http://sourcemaking.com/refactoring/replace-type-code-with-class
- Replace Type Code with State/Strategy
  - http://www.refactoring.com/catalog/replaceTypeCodeWithStateStrategy.html
  - http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy
- Replace Type Code with Subclasses
  - http://www.refactoring.com/catalog/replaceTypeCodeWithSubclasses.html
  - http://sourcemaking.com/refactoring/replace-type-code-with-subclasses