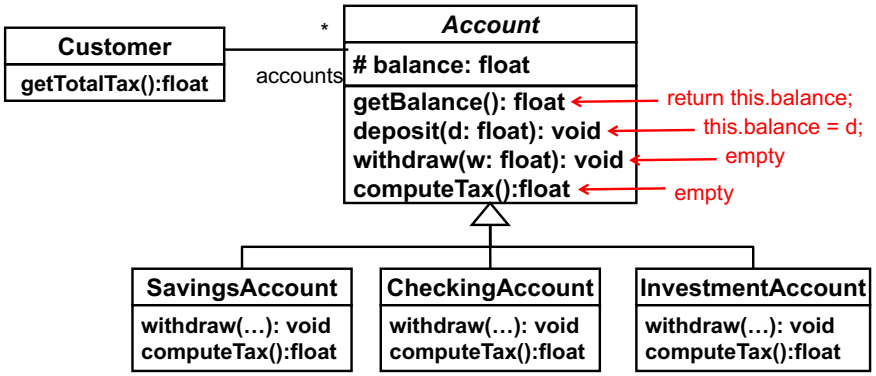
Your Email Address

- Send your (preferred) email address to umasscs680@gmail.com ASAP.
  - I will use that address to email you lecture notes, announcements, etc.

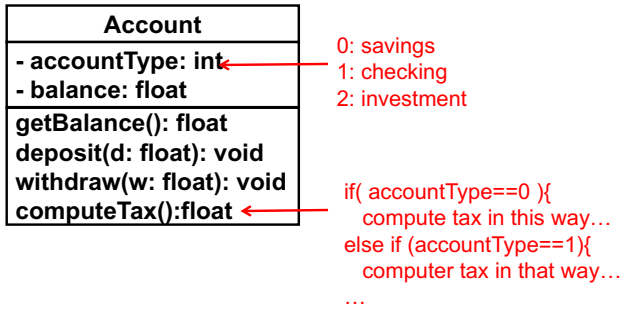
Polymorphism

Polymorphism




```
• public float getTotalTax(){
      Iterator<Account> it = accounts.iterator();
      while( it.hasNext() )
            System.out.println( it.next().computeTax() ); }
```


If Polymorphism is Not Available…

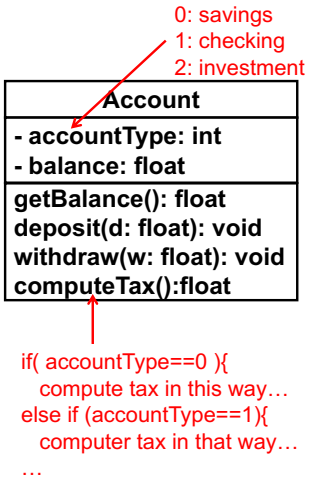



```
• public float getTotalTax(){
      Iterator<Account> it = accounts.iterator();
      while( it.hasNext() )
            System.out.println( it.next().computeTax() ); }
```

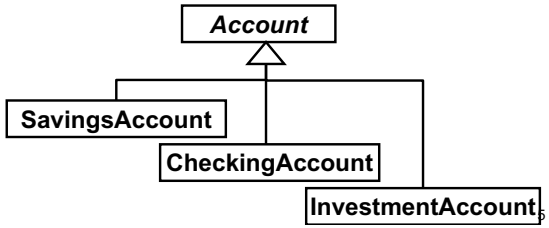
- Issues: ***magic numbers*** and ***conditionals***

What's Wrong with Magic Numbers

- They often scatter in code and get harder to maintain.

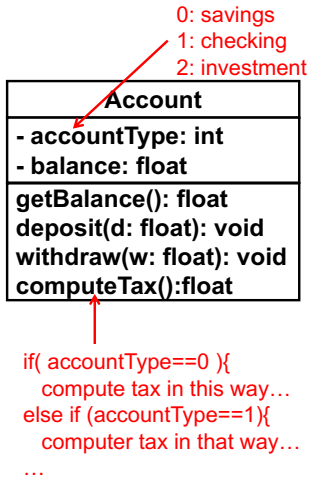


- When you add/remove magic numbers, you have to find all the places where they are used and revise the places.
- Serious maintainability issue if you use many magic numbers and if you often add/remove magic numbers.

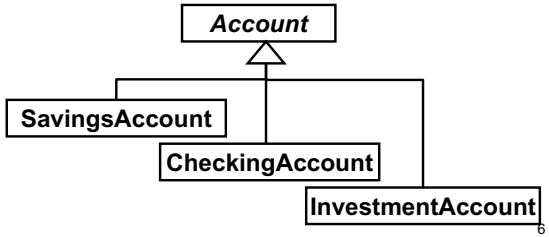


What's Wrong with Conditionals?

- They often scatter in code and get harder to maintain.



- The same conditional statement is duplicated in different places in code.
- When you make a change, you have to find all the conditional statements and change them.



Magic Numbers

If Maintenance Requirement is Limited for Magic Numbers…

- You can use magic numbers if…
  - the number of magic numbers is limited
  - there is no need to change them.
- However, you still need to be careful about how to implement them.

An Example with Magic Numbers

Robot controller code

```
Robot r = new Robot();
r.control(0);
r.control(1);
r.control(2);
```

Robot

+ control(command: int): void

```
if( command == 0 )
  // move forward
else if( command == 1 )
  // stop
else if( command == 2 )
  // move backward
```

- Magic numbers as commands to control a robot.

Try NOT to Use Magic Numbers Directly in Your Code

Robot

+ control(command: int): void

```
if( command == 0 )
  // move forward
else if( command == 1 )
  // stop
else if( command == 2 )
  // move backward
```

- Low readability
  - They do not communicate what they are meant to be.
- Low maintainability
  - An error/typo (e.g. `command==10`) can occur.
  - Need to write error-handling code. In the worst case, errors may not be detected at runtime.

Use *Symbolic Constants*

Robot controller code

```
Robot r = new Robot();
r.control(Robot.CMD_MOVE_FORWARD);
r.control(Robot.CMD_STOP);
r.control(Robot.CMD_MOVE_BACKWARD);
```

Robot

+ CMD_MOVE_FORWARD: int=0 {final}
+ CMD_STOP: int=1 {final}
+ CMD_MOVE_BACKWARD: int=2 {final}

+ control(command: int): void

```
if( command == CMD_MOVE_FORWARD )
  // move forward
else if( command == CMD_STOP )
  // stop
else if( command == CMD_MOVE_BACKWARD )
  // move backward
```

- Use ***symbolic constants*** to improve readability
  - *static final* constants.
    - `public static final int CMD_MOVE_FORWARD = 0;`

Potential Issues

Robot controller code

```
Robot r = new Robot();
r.control(Robot.CMD_MOVE_FORWARD);
r.control(Robot.CMD_STOP);
r.control(Robot.CMD_MOVE_BACKWARD);
```

Robot

+ CMD_MOVE_FORWARD: int=0 {final}
+ CMD_STOP: int=1 {final}
+ CMD_MOVE_BACKWARD: int=2 {final}

+ control(command: int): void

Robot controller code

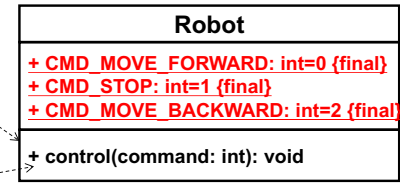
```
Robot r = new Robot();
r.control(0);
r.control(1);
r.control(2);
```

```
if( command == CMD_MOVE_FORWARD )
  // move forward
else if( command == CMD_STOP )
  // stop
else if( command == CMD_MOVE_BACKWARD )
  // move backward
```

- Clients of Robot can still pass integer values (rather than static final constants) to control().

Robot controller code

```
Robot r = new Robot();
r.control(Robot.CMD_MOVE_FORWARD);
r.control(Robot.CMD_STOP);
r.control(Robot.CMD_MOVE_BACKWARD);
```



Robot controller code

```
Robot r = new Robot();
r.control(10);
```

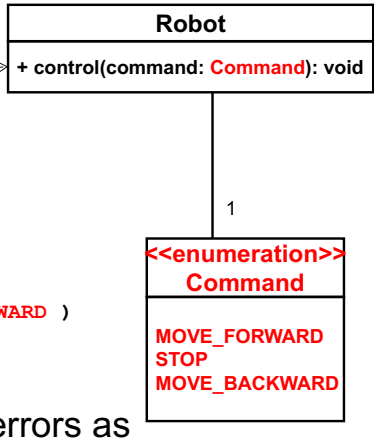
```
if( command == CMD_MOVE_FORWARD )
  // move forward
else if( command == CMD_STOP )
  // stop
else if( command == CMD_MOVE_BACKWARD )
  // move backward
else
  // error handling
```

- Clients of Robot can have typos.
  - They are not detected at compile time (not at runtime)

Use *Enumeration*

Robot controller code

```
Robot r = new Robot();
r.control(Command.MOVE_FORWARD);
r.control(Command.STOP);
r.control(Command.MOVE_BACKWARD);
```




```
if( command == Command.MOVE_FORWARD )
  // move forward
else if( command == Command.STOP )
  // stop
else if( command == Command.MOVE_BACKWARD )
  // move backward
```

- You want to catch as many errors as possible at compile-time.
  - Have your complier work harder!

Again…

- Avoid magic numbers if…
  - You have many of them
  - You need to change them often.
- Try “sub-classing” and polymorphism.

Exercise

- Learn about static final constants.
- Learn about Java’s enumeration type.

Refactoring

Refactoring

- Restructuring existing code by revising its internal structure without changing its external behavior.
  - http://en.wikipedia.org/wiki/Refactoring
  - http://www.refactoring.com/
  - http://sourcemaking.com/refactoring
  - *Refactoring: Improving the Design of Existing Code*
    - by Martin Fowler, Addison-Wesley

Example Refactoring Actions

- Encapsulate Field
  - c.f. Lecture note #1
- Replace Type Code with Subclasses
  - c.f. Lecture note #2
- Replace Conditional with Polymorphism
  - c.f. Lecture note #2
- Replace Magic Number with Symbolic Constant
- Replace Type Code with Class (incl. enumeration)
  - c.f. Lecture note #3
- Replace Type Code with State/Strategy
  - Soon to be covered in CS410.

What is NOT Refactoring? What is it for?

- Refactoring is not about
  - Finding and fixing bugs.
  - Adding/revising new features/functionalities.
- However, refactoring makes it easier to
  - Review and understand (i.e. maintain) code.
  - Add/revise new features/functionalities.

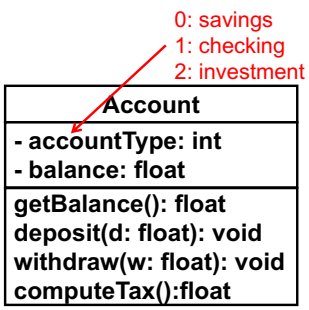
Where/When to Refactor?

- 22 bad smells in code
  - Typical places in code that require refactoring.
  - *Refactoring: Improving the Design of Existing Code*
    - by Martin Fowler, Addison-Wesley
  - http://sourcemaking.com/refactoring/bad-smells-in-code
  - http://en.wikipedia.org/wiki/Code_smell

- Duplicated code
- Long method
- Large class
- Long parameter list
- Divergent change
- Shotgun surgery
- Feature envy
- Data clumps
- Primitive obsession
- Switch statements
- Parallel inheritance hierarchies
- Lazy class
- Speculative generality
- Temporary field
- Message chains
- Middle man
- Inappropriate intimacy
- Alternative classes with different interfaces
- Incomplete library class
- Data class
- Refused bequest
- Comments

Example Bad Smell: Primitive Obsession

- Avoid built-in primitive types. Favor more structured types (e.g. class and enum) and class inheritance.
  - http://sourcemaking.com/refactoring/primitive-obsession



- Replace Magic Number with Symbolic Constant
  - http://sourcemaking.com/refactoring/replace-magic-number-with-symbolic-constant
- Replace Type Code with Class (incl. enumeration)
  - http://sourcemaking.com/refactoring/replace-type-code-with-class
- Replace Type Code with State/Strategy
  - http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy
- Replace Type Code with Subclasses
  - http://sourcemaking.com/refactoring/replace-type-code-with-subclasses
- Replace Conditional with Polymorphism
  - http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism

Example Bad Smell: Switch Statements

- Minimize the usage of conditional statements and simply them.
  - http://sourcemaking.com/refactoring/switch-statements
  - http://sourcemaking.com/refactoring/simplifying-conditional-expressions
  - Replace Type Code with Subclasses
    - http://sourcemaking.com/refactoring/replace-type-code-with-subclasses
  - Replace Conditional with Polymorphism
    - http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism
  - Replace Type Code with State/Strategy
    - http://sourcemaking.com/refactoring/replace-type-code-with-state-strategy

Simplifying Conditional Expressions

- Replace conditional with polymorphism
- Introduce null object
- Consolidate conditional expression
- Consolidate duplicate conditional fragments
- Decompose conditional
- Introduce assertion
- Remove control flag
- Replace nested conditional with guard clauses

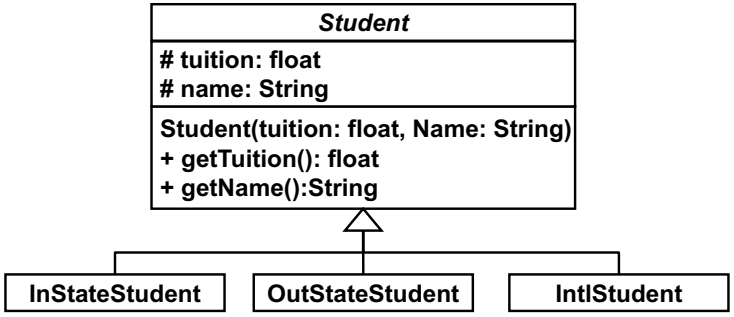
HW 0

- Learn general ideas on refactoring
- Understand code smells
  - http://sourcemaking.com/refactoring/bad-smells-in-code
- Understand the following refactoring actions with
  - http://www.refactoring.com/
  - http://sourcemaking.com/refactoring

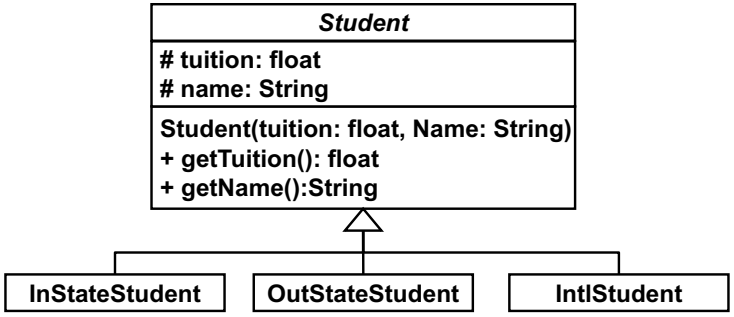
  - Encapsulate Field
  - Replace Type Code with Class (incl. enumeration)
  - Replace Type Code with Subclasses
  - Replace Conditional with Polymorphism
  - Replace Magic Number with Symbolic Constant
  - Replace Type Code with State/Strategy

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 be 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

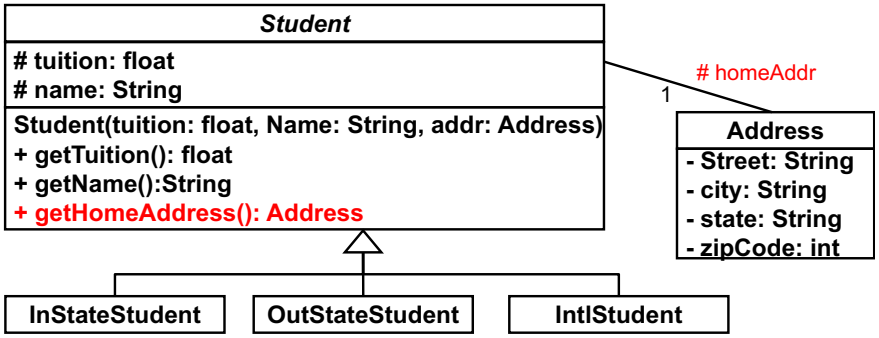
- Most programming languages do not allow this.
  - Exceptions: CLOS and a few scripting languages

– 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

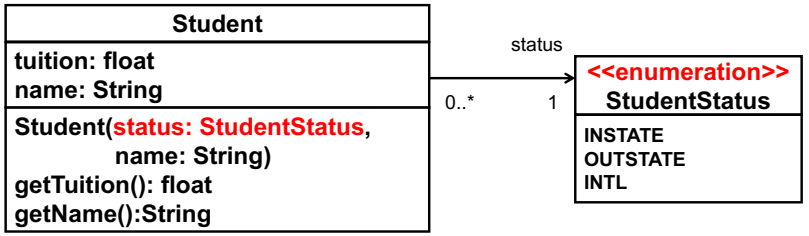



```
IntlStudent intlStudent = new IntlStudent(…);
new OutStateStudent( intlStudent.getTuition(),
                     intlStudent.getName()
                     intStudent.getAddress() );
```


When to Use an Inheritance?

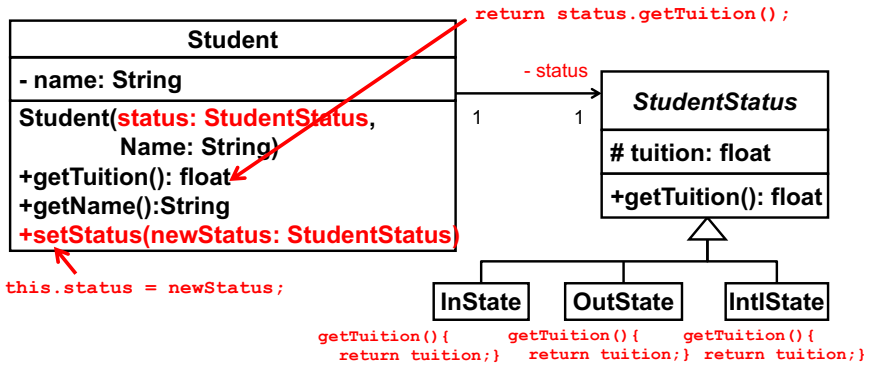
- An "is-a" relationship exists between two classes.
- No instances change their classes dynamically.
- No instances belong to more than two classes.

What to Do without Using Class Inheritance



- Enumeration allows for dynamic status changes.
- However… Need to have conditionalin getTuition().
  – Two ways to remove the conditional statements.
    - With extra classes (*State* design pattern)
    - With extra methods in an enum

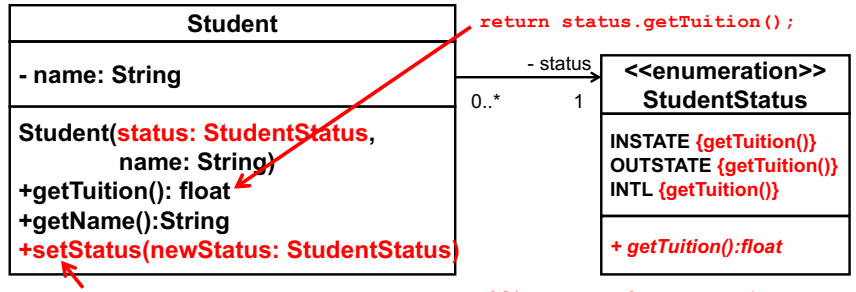
Alternative Design #1




```
Student s1 = new Student( new OutState(3000), "John Smith" );
s1.getTuition();

s1.setStatus( new OutState(1000) );
s1.getTuition();
```

c.f. "Replace Conditional with Polymorphism" and "Replace Type Code with State"

Alternative Design #2



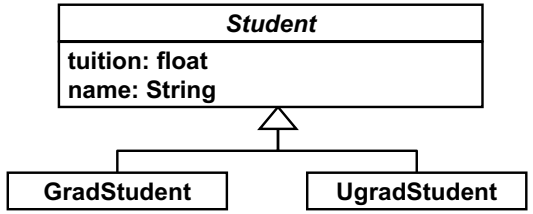

```
Student s1 =
  new Student(StudentStatus.OUTSTATE,
              "John Smith" );
s1.getTuition();

s1.setStatus(StudentStatus.INSTATE);
s1.getTuition();
```

```
public enum StudentCategory{
 INSTATE{
  @Override public void getTuition(){
    return 1000;
  }
 }
 OUTSTATE{
  @Override public void getTuition(){
    return 3000;
  }
 }
 ... }
```


Exercise

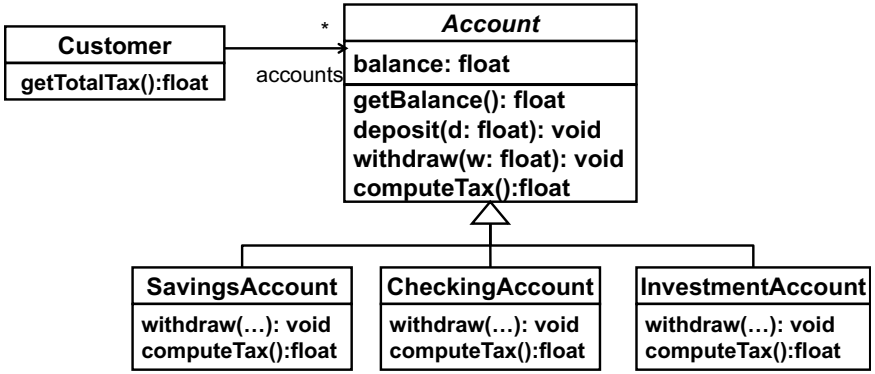
- Implement alternative designs #1 and #2

Another Example

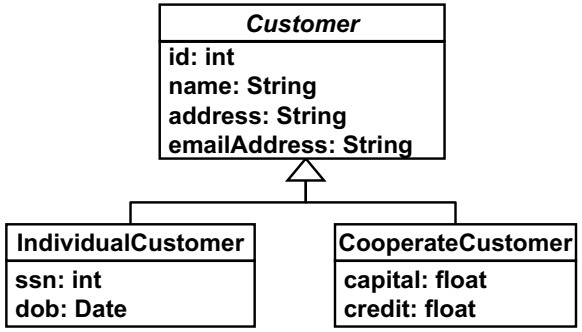


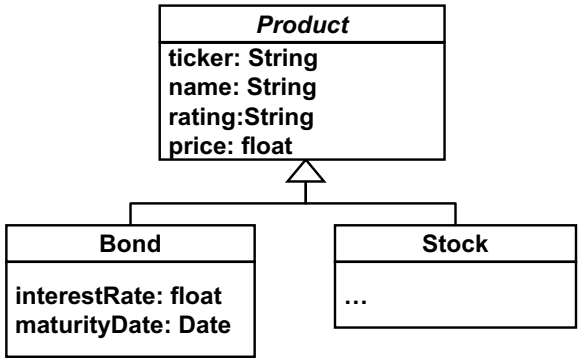
- Grad and u-grad students are students.
  - "Is-a" relationship
  - Conceptually, no problem.
- A class inheritance is NOT reasonable if subclass instances may want to dynamically change their classes in the future.
  - Implementation limitation: Most programming languages do not allow this.

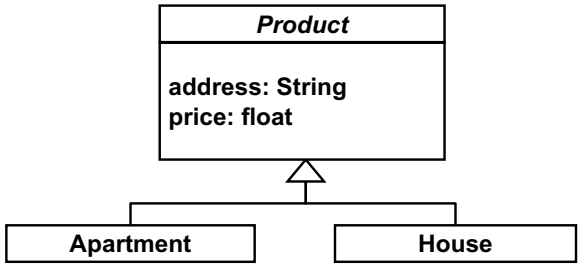
More Examples

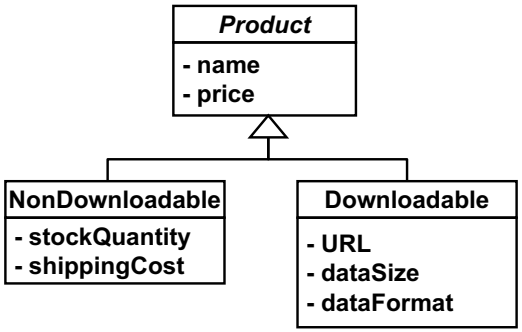


- An Account instance needs to change its type?
  - Savings to checking? No.



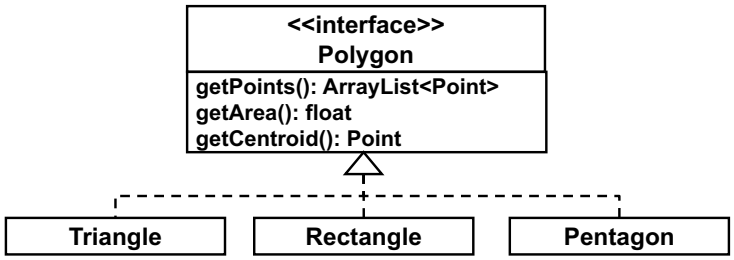




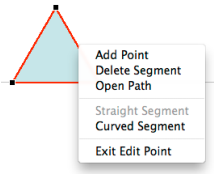


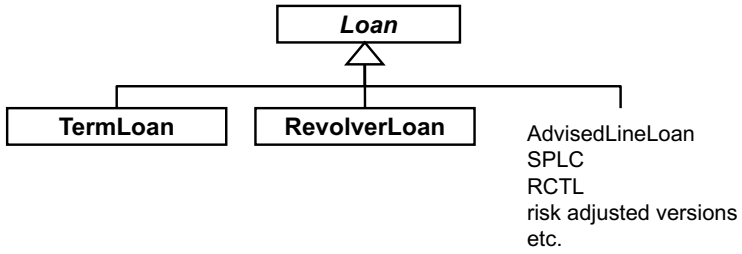
- Assume a product sales app at an online retail store (e.g., Amazon)
- Does this inheritance-based design make sense?
  - An is-a relationship between the super class and a subclass?
  - Does a subclass instance need to change its class?

Some More Examples

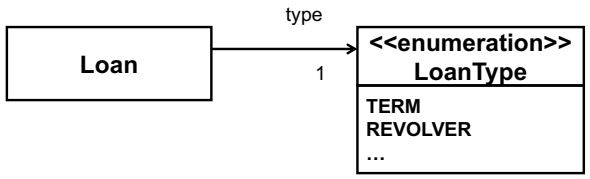


- Can a triangle become a rectangle?
- Do we allow that?
  - Maybe, depending on requirements.

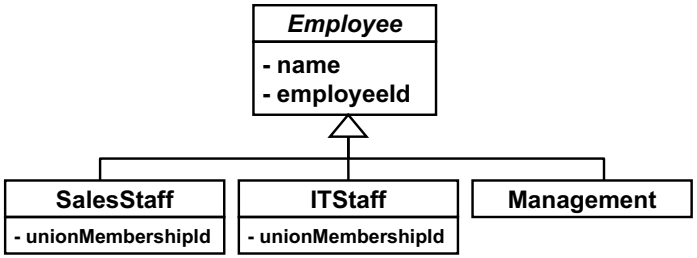




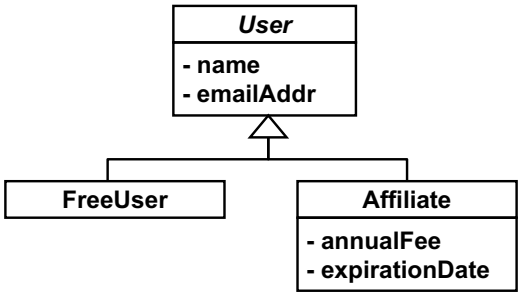
- Term loan
  - Must be fully paid by its maturity date.
- Revolver
  - e.g. credit card
  - With a spending limit and expiration date
- A revolver can transform into a term loan when it expires.



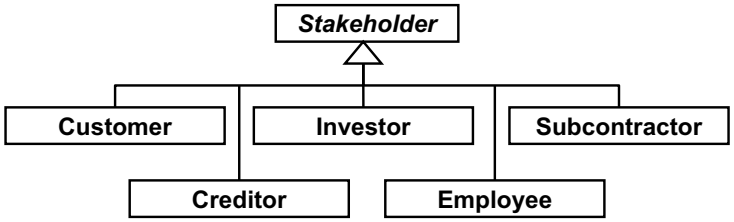
- Use an enumeration
- Use the *State* design pattern



- 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.