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
    - Well-known collection of best practice in refactoring

Example Refactoring Actions

- Encapsulate Field
  - c.f. Lecture note #2
- 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.

What is NOT Refactoring? What is it for?

- Refactoring is NOT about
  - Finding and fixing bugs.
  - Adding/revising new features/functionalities.
- However, refactoring help you turn compilable/runnable code to be more maintainable.
  - Code gets easier to
    - Review and understand.
    - Add/revise new features/functionalities in the future.

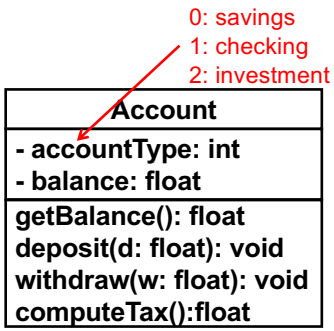
Where/When to Refactor?

- 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 conditionals and simplify 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 Conditionals

- 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

Exercise

- 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

Ant: An Automated Build Tool

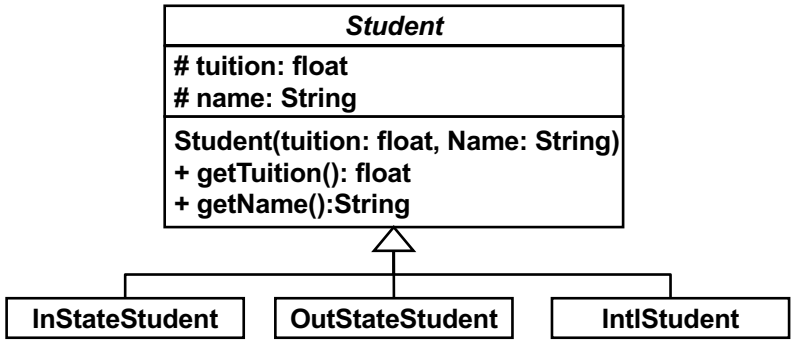
- You will use Ant (http://ant.apache.org/) to compile/build all of your Java programs in every HW.
- Learn how to use it, if you are not familiar with it.
  - No lectures will be given about how to use it.
  - There are a LOT of materials/tutorials online.
    - It's 20+ years old.
- You will turn in source code files (*.java) and a build script (e.g. build.xml).
  - Build script (build.xml) to
    - configure all settings (e.g., class paths, a directory of source code, a directory to generate binary code),
    - compile all source code from scratch,
    - generate binary code (*.class files) to a designated place(s), and
    - run compiled code

Gradle?

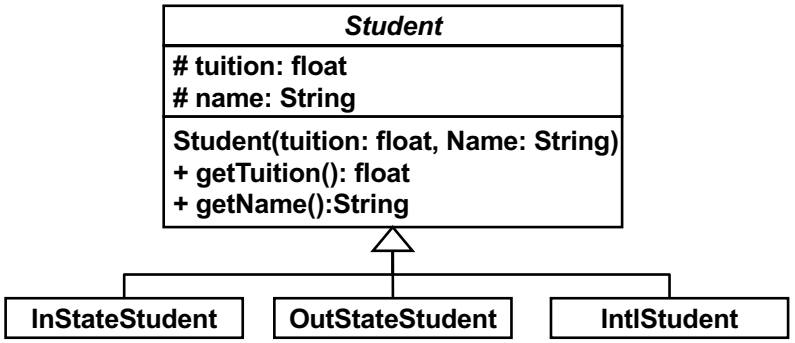
- CS680 ***may*** cover another (newer, increasingly popular) build tool: Gradle.
  - A lecture will be given about how to use it, if it is used in CS680.

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 a visa status change.

Dynamic Class Change

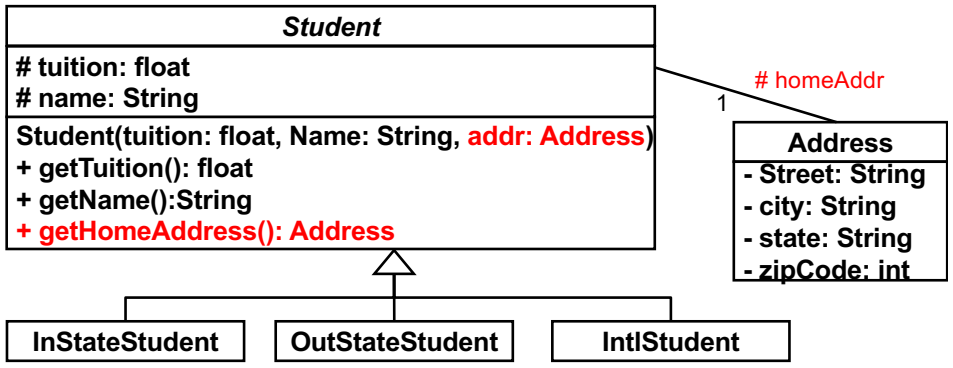
- Most programming languages do not allow/expect this.
  - Exceptions: CLOS and a few scripting languages

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

```
IntlStudent intlStudent = ...;
new OutStateStudent( intlStudent.getFirstName()
                     intStudent.getLastName() );
```

  - 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 class instances.
  - e.g., IntlStudent → Address

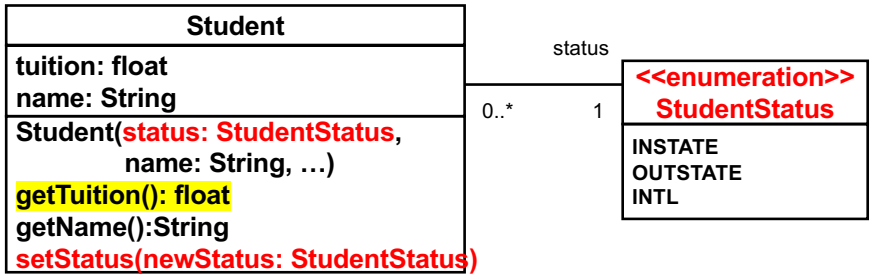



```
IntlStudent intlStudent = ...;
new OutStateStudent( intlStudent.getFirstName(),
                     intlStudent.getLastName()
                     intStudent.getAddress() );
```

When to Use an Inheritance?

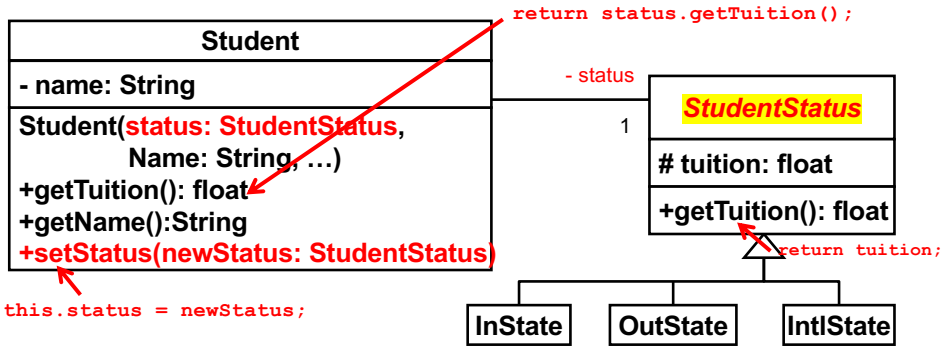
- An “is-a” relationship exists between two classes.
- No instances change their classes dynamically.
- No instances belong to more than one class.

What to Do without Using Class Inheritance



- Dynamic status changes are possible.
- However… Need to have a conditional in `getTuition()`.
  - Two ways to remove the conditionals.
    - With extra classes (*State* design pattern)
    - With extra methods in an enum

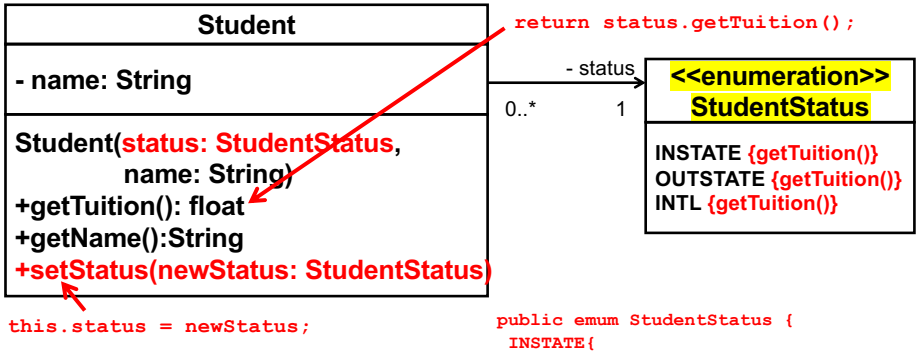
Alternative Design #1




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

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

c.f. “Replace Type Code with State”

Alternative Design #2



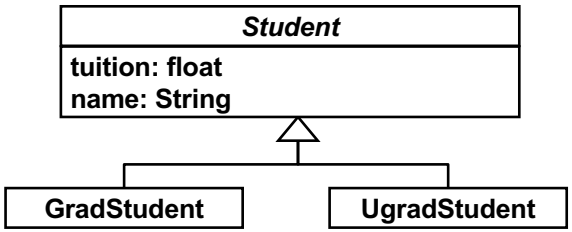

```
Student s1 =
  new Student(StudentStatus.OUTSTATE,
              "John Smith" );
s1.getTuition();

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

```
public emum StudentStatus {
 INSTATE{
  @Override public float getTuition(){
    return 1000;
  }
 }
 OUTSTATE{
  @Override public float getTuition(){
    return 3000;
  }
 }
 ... }
```

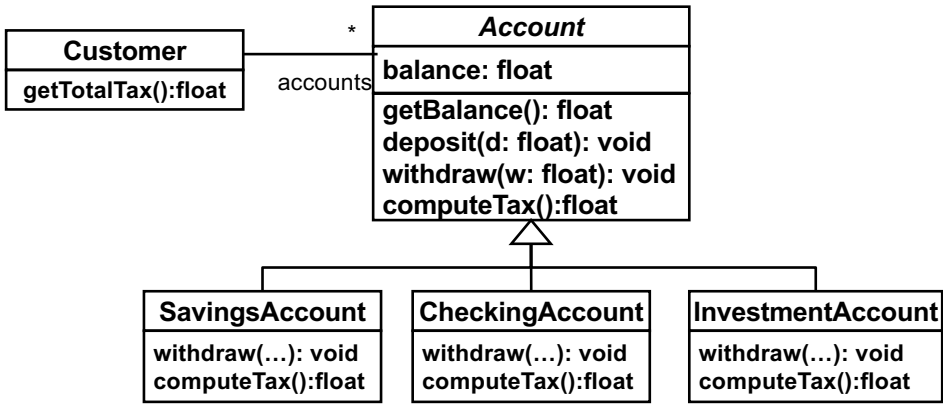
Exercise

- Implement alternative designs #1 and #2

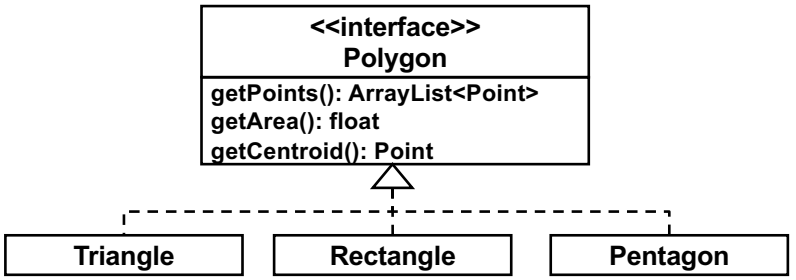
Extra Examples



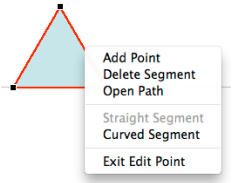
- Grad and u-grad students are students.
  - "Is-a" relationship
  - Conceptually, no problem.
- A class inheritance is NOT reasonable if subclass instances may want to dynamically change their classes in the future.

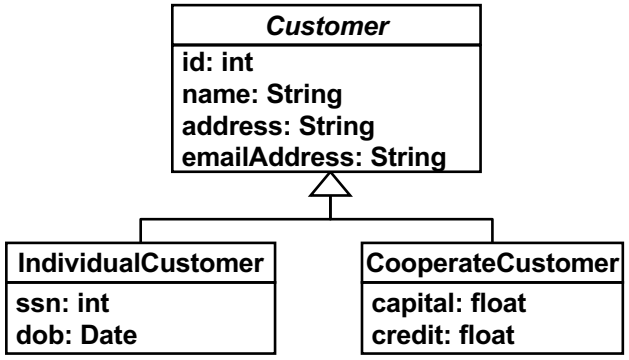


- An Account instance needs to change its type?
  - Savings to checking, for example? No.
  - It is reasonable to use class inheritance.

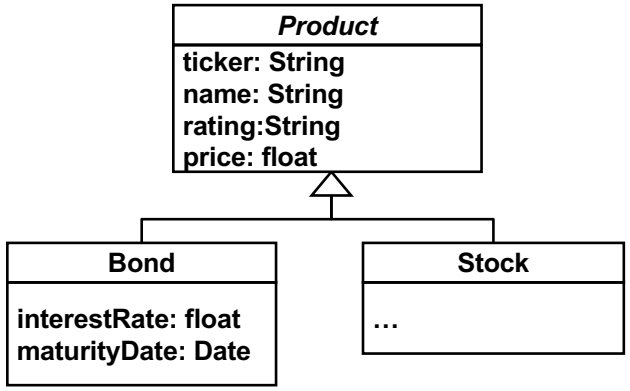


- Can a triangle dynamically change its shape to a rectangle?
- Do we allow that?
  - Maybe, depending on requirements.
  - If yes, it is NOT reasonable to use class inheritance.
  - If no, it is reasonable to use it.

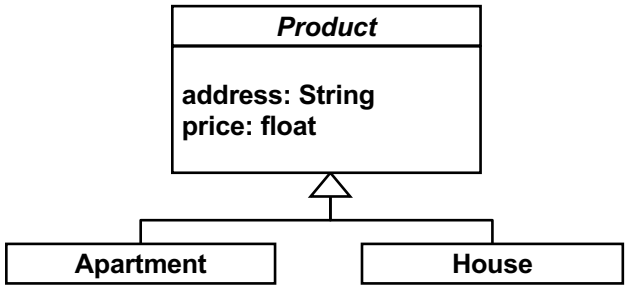




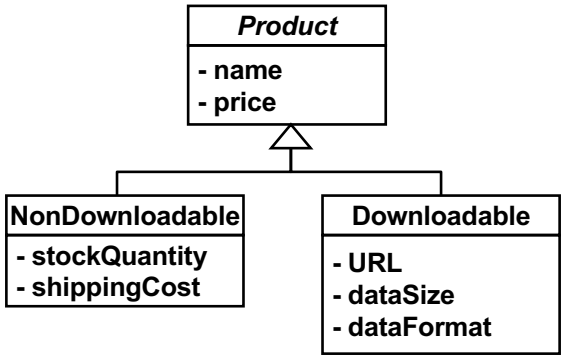
- In most use cases, individual customers do not become corporate customers, and corporate customers do not become individual customers.
- It is reasonable to use class inheritance.



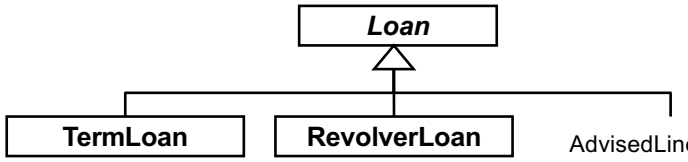
- Bond products never become stock/equity products, and stock/equity products never become bond products.
- It is reasonable to use class inheritance.



- Real estate products usually do not change their product types.
- It is reasonable to use class inheritance.

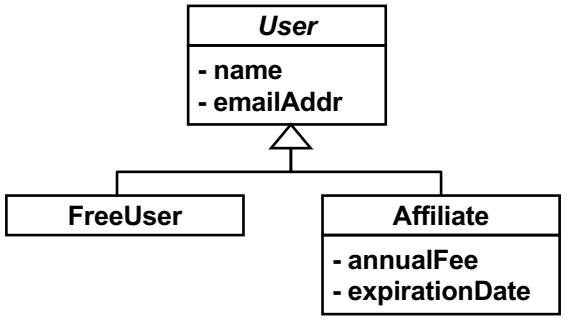


- Downloadables never become non-downloadables, and non-downloadables never become downloadables.
- It is reasonable to use class inheritance.



AdvisedLineLoan
SPLC
RCTL
risk adjusted versions
etc.

- 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.
- It is NOT reasonable to use class inheritance.



- Assume a user management system
  - c.f. Amazon (regular users v.s. Amazon Prime users), Dropbox, Google Drive, etc.
- "Free" users can become affiliate users, and affiliate users can become "free" users.
- It is NOT reasonable to use class inheritance.

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 one class.