GitHub, Java IDE, and …

- I have asked you to start learning about GitHub and IDE
  - If you are not familiar with them.
- You need to get familiar with one more tool: Ant.

Ant: An Automated Build Tool

- You will use Ant (http://ant.apache.org/) to build every single homework.
- 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, and
    - generate binary code (i.e., *.class files) to a designated place(s).

IDE and Ant

- Most modern IDEs have integrated Ant.
  - Many offer GUI for editing and running build scripts.
- Learn how to use Ant on your IDE.
- In addition, learn how to use Ant on a shell (command-line window).
- Make sure to learn BOTH ways.

When You Turn in your Homework…

- I will run your build script on a shell (not on an IDE).
  - I will use the most recent stable version of the "ant" command, which is available at http://ant.apache.org/
- If your build script fails, I will **NOT** grade your work.
  - If you want, ask me if your build script runs properly on my machine.
    - You can correct any errors if you consult with me early enough against the deadline.

Magic Numbers

You can Use Magic Numbers if…

- Their variety is limited.
- There is no need to change them in the near future.
- However, you still need to be careful about how to implement them.

Example Magic Numbers

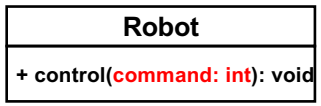
Robot controller code

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


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

- Magic numbers as commands to control a robot.

DO NOT Use Magic Numbers Directly



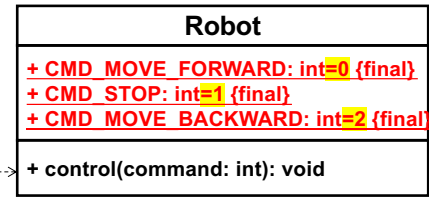

```
if( command == 0 )
  // move forward
else if( command == 1 )
  // stop
else if( command == 2 )
  // move backward
```

- 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);
```




```
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;`

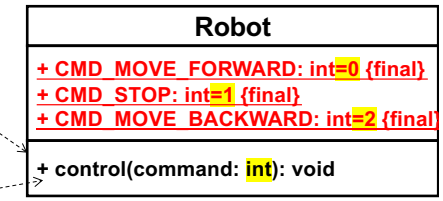
Benefits of Symbolic Constants

- Readability
  - Symbolic constants can communicate what they are meant to be.
    - `CMD_MOVE_FORWARD`
    - `CMD_MOVE_BACKWARD`
    - `CMD_STOP`
- Maintainability
  - Symbolic constants eliminate undefined commands (e.g. `CMD_JUMP`) to be passed to `control()`.

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 controller code

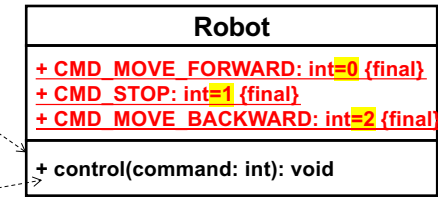
```
Robot r = new Robot();
r.control(0);
r.control(1);
r.control(2);
```

```
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);
```

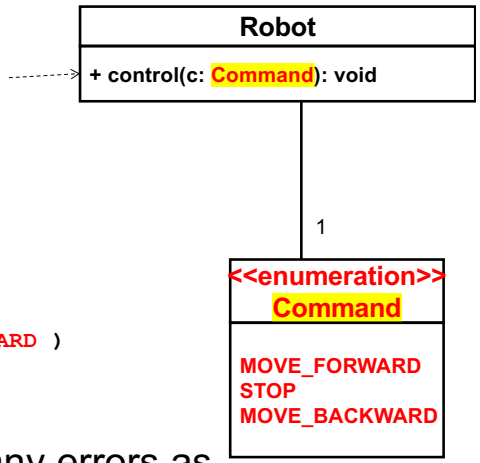
```
if( command == CMD_MOVE_FORWARD )
  // move forward
else if( command == CMD_STOP )
  // stop
else if( command == CMD_MOVE_BACKWARD )
  // move backward
else
  // error handling
```

- Clients of `Robot` can still make typos.
  - They are not detected at compile time.

Use *Enumeration*

Robot controller code

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



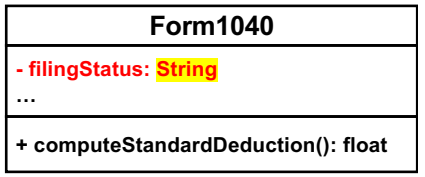

```
if( c == Command.MOVE_FORWARD )
  // move forward
else if( c == Command.STOP )
  // stop
else if( c == Command.MOVE_BACKWARD )
  // move backward
```

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

Type Code

Client code

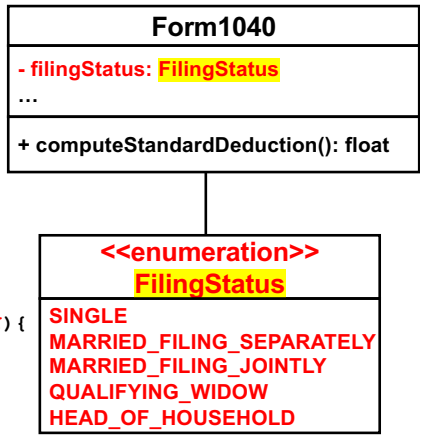
```
Form1040 f = new Form1040("single");
f.computeStandardDeduction();
```




```
if( your/your spouse's DOB < 01/02/53 OR
you/your spouse are/is blind ){
  // use a special algorithm
}
else if(filingStatus == "single"){
  return ...;
}
else if(filingStatus == "marriedFilingJointly"){
  return ...;
}
else if( ... ){
  ...
}
...
else
  // error handling
```


Client code

```
Form1040 f =
  new Form1040(FilingStatus.SINGLE);
f.computeStandardDeduction();
```




```
if(filingStatus==FilingStatus.SINGLE){
  return 6350;
}
else if(filingStatus==
      FilingStatus.MARRIED_FILING_JOINTLY){
  return 12700;
}
else if( ... ){
  ...
}
...
```

You want to catch as many errors as possible at compile-time.

- Have your complier work harder!

Exercise

- Write and run some code with static final constants and enumerations
  - if you are not familiar with them.

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.

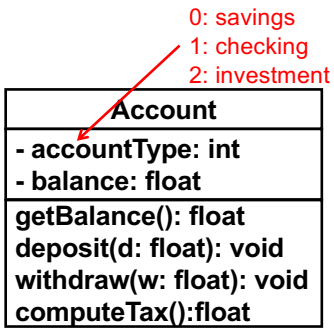
Where/When to Refactor?

- 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

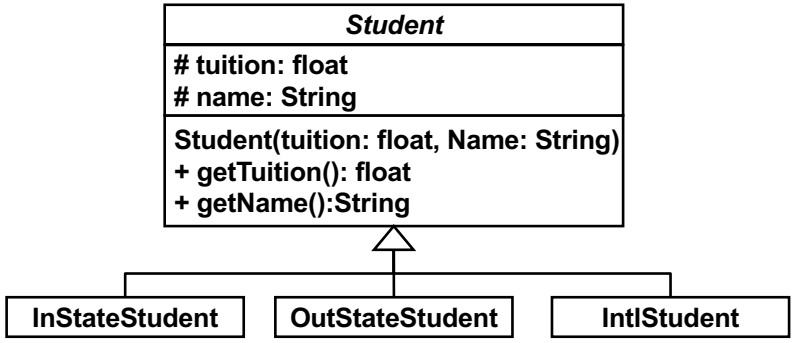
- Replace conditional with polymorphism
- Introduce null object
- Consolidate conditional expression
- Consolidate duplicate conditional fragments
- Decompose conditional
- Introduce assertion
- Remove control flag
- Replace nested conditional with guard clauses

Exercise

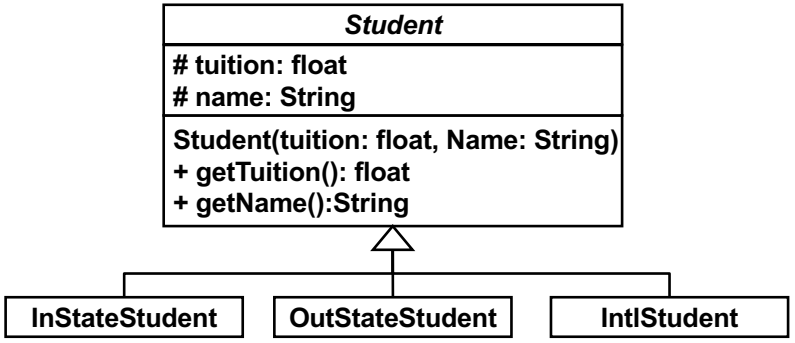
- Learn general ideas on refactoring
- Understand code smells
  - http://sourcemaking.com/refactoring/bad-smells-in-code
- Understand the following refactoring actions with
  - http://www.refactoring.com/
  - http://sourcemaking.com/refactoring
  - Encapsulate Field
  - Replace Type Code with Class (incl. enumeration)
  - Replace Type Code with Subclasses
  - Replace Conditional with Polymorphism
  - Replace Magic Number with Symbolic Constant
  - Replace Type Code with State/Strategy

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 change their classes (i.e. student status) dynamically.



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

Dynamic Class Change

- This is not implemented directly in most languages.
  - If you try to do that, you have to do and maintain a lot of housekeeping operations as a workaround.
  - It's not worth doing that.
    - It will be a headache very soon.

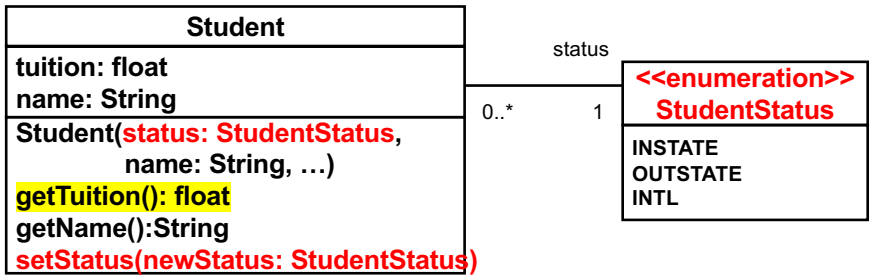
– Exceptions: CLOS and a few scripting languages.
  - Dynamic class changes are doable, technically.
  - It is not encouraged to use it actually in serious (particularly large-scale) projects.

– Do NOT use class inheritance if your class instances may need to change their classes.

When to Use an Inheritance?

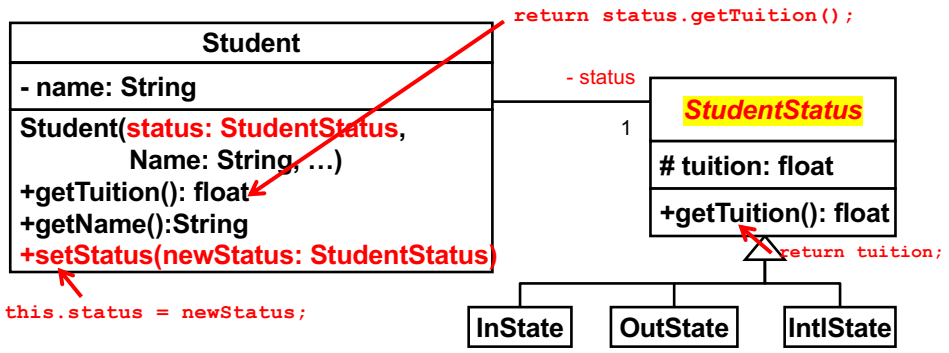
- An "is-a" relationship exists between two classes.
- No instances change their classes dynamically.
- No instances belong to more than one class.

What to Do without Using Class Inheritance



- Dynamic status changes are possible.
- However… still need to have a conditional in `getTuition()`.
  – You can remove the conditionals with State design pattern (with extra classes).
    - To be covered.

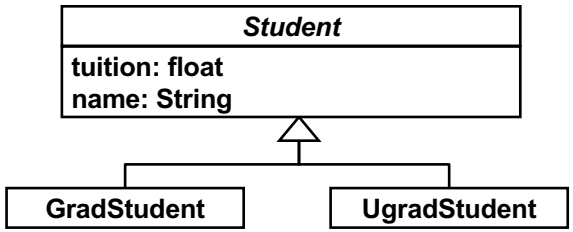
Revised Design with *State*



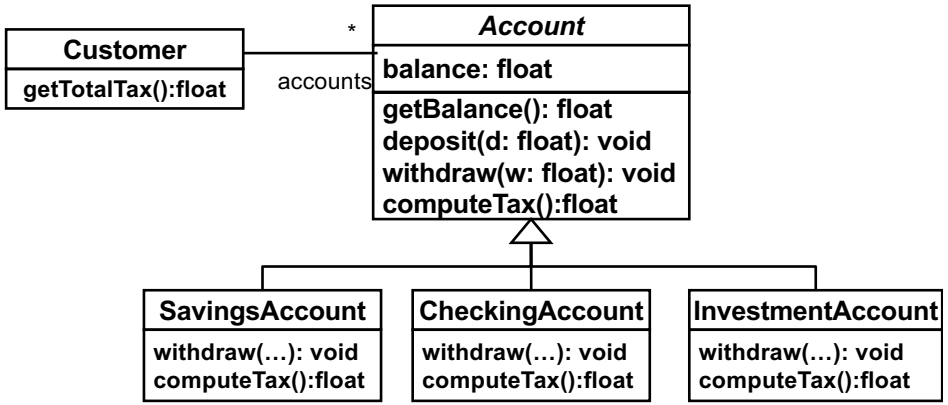

```
Student s1 = new Student( new OutState(3000), "John Smith", ... );
s1.getTuition();

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

c.f. "Replace Type Code with State"

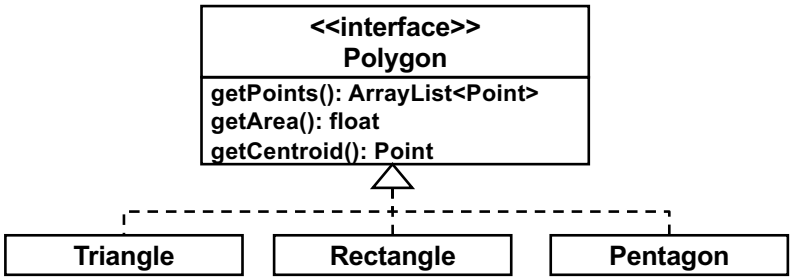
Extra Examples



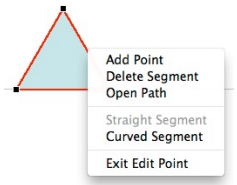
- Grad and u-grad students are students.
  - "Is-a" relationship
  - Conceptually, no problem.
- A class inheritance is NOT reasonable if subclass instances may want to dynamically change their classes in the future.

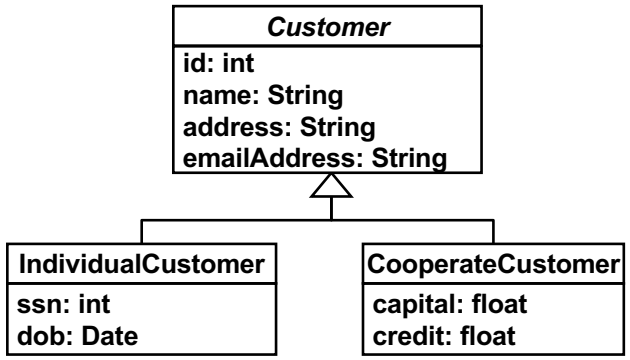


- An Account instance needs to change its type?
  - Savings to checking, for example? No.
  - It is reasonable to use class inheritance.

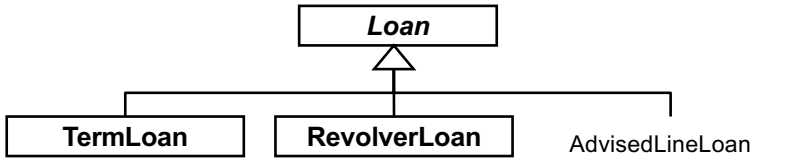


- Can a triangle dynamically change its shape to a rectangle?
- Do we allow that?
  - Maybe, depending on requirements.
  - If yes, it is NOT reasonable to use class inheritance.
  - If no, it is reasonable to use it.

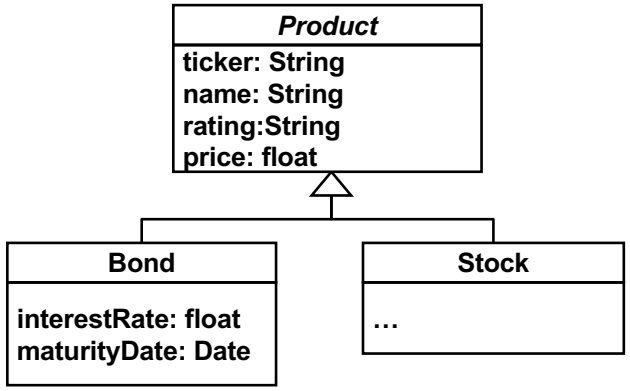




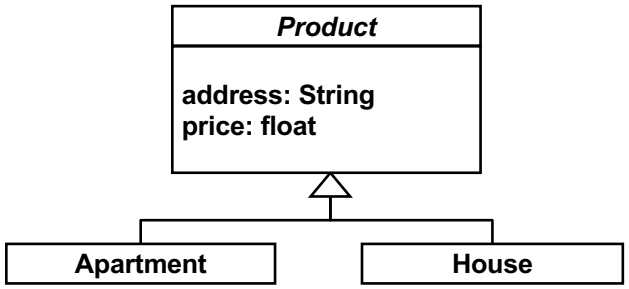
- In most use cases, individual customers do not become corporate customers, and corporate customers do not become individual customers.
- It is reasonable to use class inheritance.



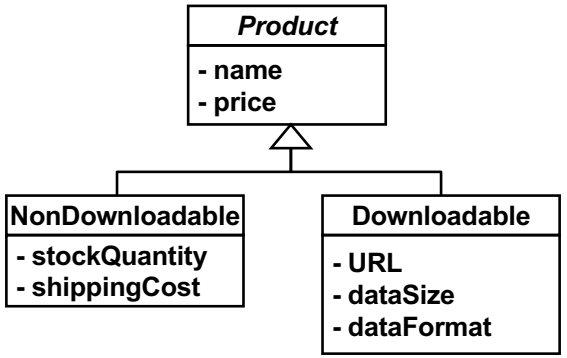
- Term loan
  - Must be fully paid by its maturity date.
- Revolver
  - e.g. credit card
  - With a spending limit and expiration date
- A revolver can transform into a term loan when it expires.
- It is NOT reasonable to use class inheritance.



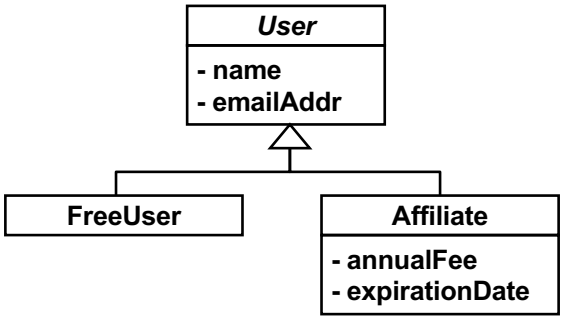
- Bond products never become stock/equity products, and stock/equity products never become bond products.
- It is reasonable to use class inheritance.



- 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.



- 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.