HW0

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

Brief History

- In good, old days… programs had no structures.
  - – One dimensional code.
    - From the first line to the last line on a line-by-line basis.
    - “Go to” statements to control program flows.
      - – Produced a lot of “spaghetti” code
        - » “Go to” statements considered harmful.
  - – No notion of structures (or modularity)
    - Making a chunk of code (module) self-contained and independent from the other code
      - – Improve reusability and maintainability
        - » Higher reusability → higher productivity, less production costs
        - » Higher maintainability → higher productivity and quality, less maintenance costs

Modules in SD and OOD

- Modules in Structured Design (SD)
  - – Structure = a set of variables (data fields)
  - – Function = a block of code
- Modules in OOD
  - – Class = a set of data fields and functions
  - – Interface = a set of abstract functions
- Key design questions/challenges:
  - – how to define modules
  - – how to separate a module from others
  - – how to let modules interact with each other

SD v.s. OOD

- OOD
  - – Intends coarse-grained modularity
    - The size of each code chuck is often bigger.
  - – Extensibility in mind in addition to reusability and maintainability
    - How to add and revise existing modules (classes and interfaces) to accommodate new/modified requirements.
  - – How to gain reusability, maintainability and extensibility in wise ways?

Looking Ahead: AOP, etc.

- OOD does a pretty good job, but it is not perfect
  - Still has some modularity issues
- A solution: Aspect Oriented Programming (AOP)
  - Dependency injection

Encapsulation

What is Encapsulation?

- Hiding each class's internal details from its clients (other classes)
  - To improve its modularity, robustness and ease of understanding
- Things to do:
  - Always make your data fields private or protected.
  - Make your methods private or protected as often as possible.
  - Avoid public accessor (getter/setter) methods whenever possible.
  - Make your classes final as often as possible.

Why Encapsulation?

- Encapsulation makes classes modular (or black box).

```
– final public class Person{
      private int ssn;
      Person(int ssn){ this.ssn = ssn; }
      public int getSSN(){ return this.ssn; } }

– Person person = new Person(123456789);
  int ssn = person.getSSN();
  …
```

- What if you encounter an error about a person's SSN? (e.g., the SSN is wrong or null)… Where is the source of the error, inside or outside Person?
  - You can tell it should be outside Person.
    - A bug(s) should exist before calling Person's constructor or after calling getSSN().
  - You can be more confident about your debugging.
    - You can narrow the scope of your debugging effort.

- However, if the Person class looks like this, you cannot be so sure about where to find a bug.

```
– final public class Person{
      private int ssn;
      Person(int ssn){ this.ssn = ssn; }
      public String getSSN(){ return this.ssn; }
      public setSSN(int ssn){ this.ssn = ssn; } }
```

- However, if the Person class looks like this, you cannot be so sure about where to find a bug.

```
– final public class Person{
      private int ssn;
      Person(int ssn){ this.ssn = ssn; }
      public String getSSN(){ return this.ssn; }
      public setSSN(int ssn){ this.ssn = ssn; } }

– Person person = new Person(123456789);
  int ssn = person.getSSN();
  ……
  person.setSSN(987654321);
```

  - You or your team mates may write this by accident.
    - It looks like a stupid error, but it is common in a large-scale project.
  - Don’t define public setter methods whenever possible.

In a Modern Software Dev Project…

- No single engineer can read, understand and remember the entire code base.

- Every engineer faces time pressure.

- Any smart engineers can make unbelievable errors VERY EASILY under a time pressure.

- Your code should be *preventive* for potential errors.

Scale of Modern Software

- All-in-one copier (printer, copier, fax, etc.)
  - 3M+ lines
- Passenger vehicle
  - 7M+ lines (‘07)
    - 10 CPUs/car in ‘96
    - 20 CPUs/car in ‘99
    - 40 CPUs/car in ‘02
    - 80+ CPUs/car in ‘05
      - Engine control, transmission, light, wipers, audio, power window, door mirror, ABS, etc.
      - Drive-by-wire: replacing the traditional mechanical and hydraulic control systems with electronic control systems
      - Car navigation, automated wipers, built-in iPod support, automatic parking, automatic collision avoidance, etc… hybrid cars! autonomous car!!! (e.g. Google’s)
- Cell phone (not a smart phone)
  - 10M+ lines

- In my experience…
  - 32K, 28K, 25K, 23K, 22K, 20K, 18K, 15K, 12K, 8K, 4K, 3K and 2K lines of Java code for research software
  - 11K and 9K lines of C++ code at an investment bank
  - 7K and 5K lines of C code for research software
- Cannot fully manage (i.e., precisely remember) the entire code base when its size exceeds 10K lines of Java code.
  - What is this class for?
  - Which classes interact with each other to implement that algorithm?
  - Why is this method designed like this?
  - Cannot be fully confident which classes/methods I should modify according to a code revision.
  - Need UML class diagrams for all classes and sequence diagrams for some key methods.
  - Need comments, memos and/or documents about design rationales

Why Encapsulation? (cont'd)

- Assume you are the provider (or API designer) of Person
  - Your team mates will use your class for *their* programming.

  -
    ```
    final public class Person{
        private int ssn;
        Person(int ssn){ this.ssn = ssn; }
        public int getSSN(){ return this.ssn; } }
    ```
- You can be sure/confident that your class will never mess up SSNs.

- However, if you define Person like this,

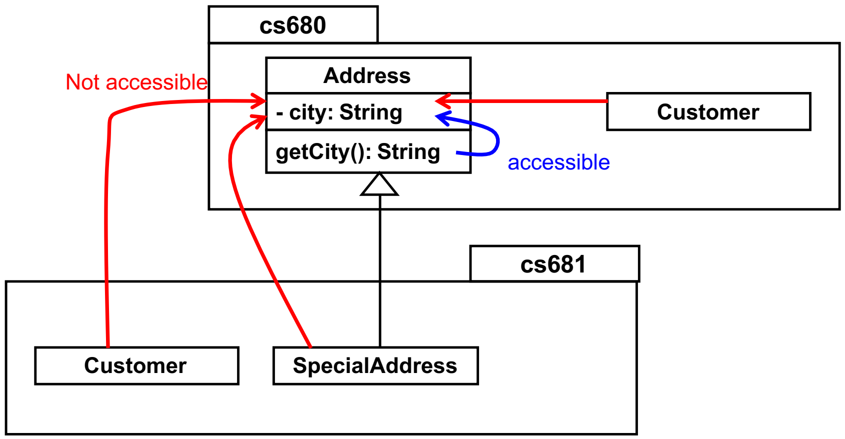
  -
    ```
    public class Person{
        protected int ssn;
        Person(int ssn){ this.ssn = ssn; }
        public int getSSN(){ return this.ssn; } }
    ```
- You cannot be so sure about potential bugs.

- However, if you define Person like this,

  -
    ```
    public class Person{
        protected int ssn;
        Person(int ssn){ this.ssn = ssn; }
        public int getSSN(){ return this.ssn; } }
    ```
- You cannot be so sure about potential bugs.
- Your team mates can define:

  -
    ```
    public class MyPerson extends Person{
        MyPerson(int ssn){ super(ssn); }
        public void setSSN(int ssn){ this.ssn = ssn; } }
    ```
- Your class should be *preventive* for potential misuses.
  - Do not use "protected." Use "private" instead.
  - Turn the class to be "final."

“Private” Visibility



cs680

Address

- city: String

getCity(): String

Customer

Not accessible

accessible

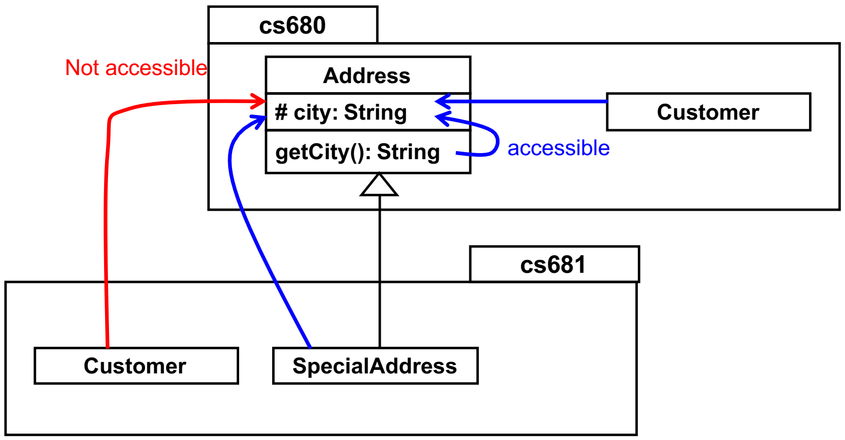
cs681

Customer

SpecialAddress

Encapsulation principle: Use private/protected visibility as often as possible to encapsulate/hide the internal attrs/ops of a class.

“Protected” Visibility



cs680

Address

# city: String

getCity(): String

Customer

Not accessible

accessible

cs681

Customer

SpecialAddress

Be Preventive!

- Encapsulation
  - looks very trivial.
  - is not that important in small-scale (toy) software
    - because you can manage (i.e., read, understand and remember) every aspect of the code base.
  - is very important in large-scale (real-world) software
    - because you cannot manage (i.e., read, understand and remember) every aspect of the code base.

Sounds Trivial?

-
  ```
  public class Person{
      protected int ssn;
      Person(int ssn){ this.ssn = ssn; }
      public int getSSN(){ return this.ssn;  } }
  ```

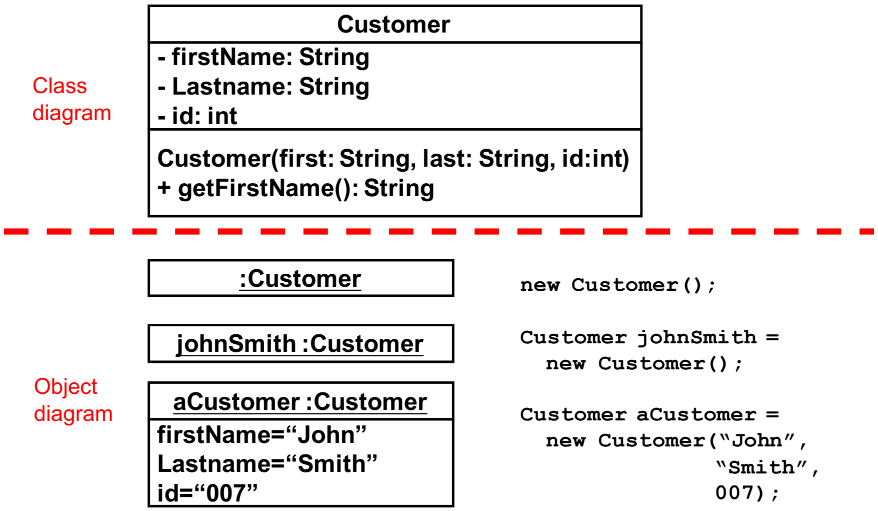
- Once you finish up writing these 4 lines, wouldn’t you define a setter method automatically (i.e. without thinking about it carefully)?
  - “I always define both getter and setter methods for a data field. I can delete unnecessary ones anytime later.”
  - “Well, let’s define a setter just in case.”
  - Think. Fight that temptation.
    - Just define the method you absolutely need.

HW 2

- In “Developing Enterprise Java Applications with J2EE and UML,” by Ahmed et al. Chapter 3 (Intro to the UML)
  - Figure 4-3 has errors/typos. Explain what errors are, and describe how its design should have been to maximize the degree of encapsulation.

Classes and Instances

Class and Object Diagrams



Class diagram

Customer

- firstName: String
- Lastname: String
- id: int

Customer(first: String, last: String, id:int)
+ getFirstName(): String

Object diagram

:Customer

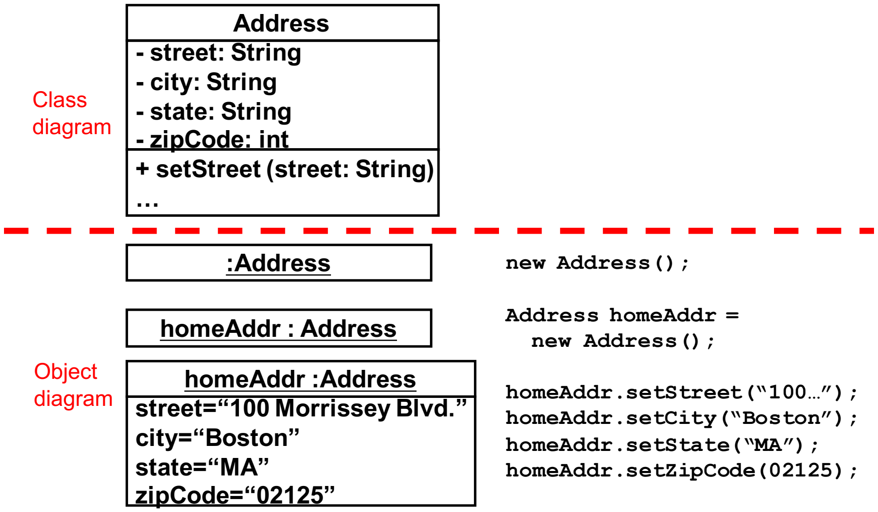
johnSmith :Customer

aCustomer :Customer
firstName=“John”
Lastname=“Smith”
id=“007”

```
new Customer();

Customer johnSmith =
  new Customer();

Customer aCustomer =
  new Customer(“John”,
               “Smith”,
               007);
```



Class diagram

Address

- street: String
- city: String
- state: String
- zipCode: int

+ setStreet (street: String)
…

Object diagram

:Address

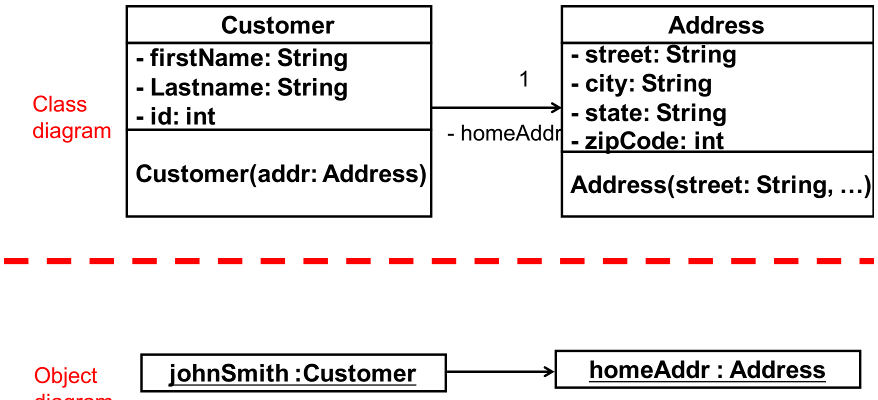
homeAddr : Address

homeAddr :Address
street=“100 Morrissey Blvd.”
city=“Boston”
state=“MA”
zipCode=“02125”

```
new Address();

Address homeAddr =
  new Address();

homeAddr.setStreet(“100…”);
homeAddr.setCity(“Boston”);
homeAddr.setState(“MA”);
homeAddr.setZipCode(02125);
```



Class diagram

Customer
- **- firstName: String**
- **- Lastname: String**
- **- id: int**

Customer(addr: Address)

1

- homeAddr

Address
- **- street: String**
- **- city: String**
- **- state: String**
- **- zipCode: int**

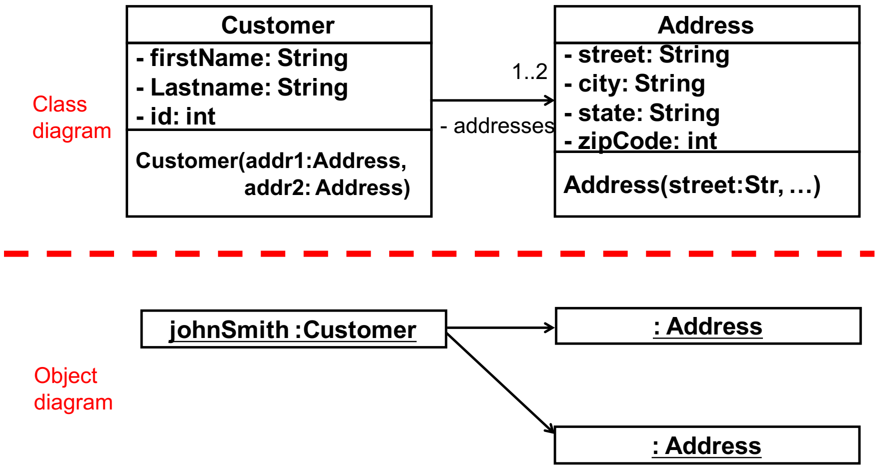
Address(street: String, …)

Object diagram

johnSmith :Customer

homeAddr : Address

```
Customer johnSmith = new Customer( newAddress(…) );
```



Class diagram

Customer
- **- firstName: String**
- **- Lastname: String**
- **- id: int**

**Customer(addr1:Address,
addr2: Address)**

1..2

- addresses

Address
- **- street: String**
- **- city: String**
- **- state: String**
- **- zipCode: int**

Address(street:Str, …)

Object diagram

johnSmith :Customer

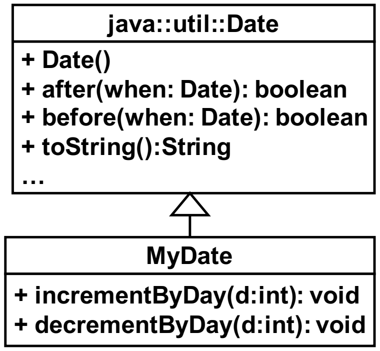
: Address

: Address

```
Customer johnSmith = new Customer(
                        new Address(…), new Address(…));
```

Inheritance (Generalization)

Inheritance



java::util::Date
- **+ Date()**
- **+ after(when: Date): boolean**
- **+ before(when: Date): boolean**
- **+ toString():String**
- **…**

MyDate
- **+ incrementByDay(d:int): void**
- **+ decrementByDay(d:int): void**

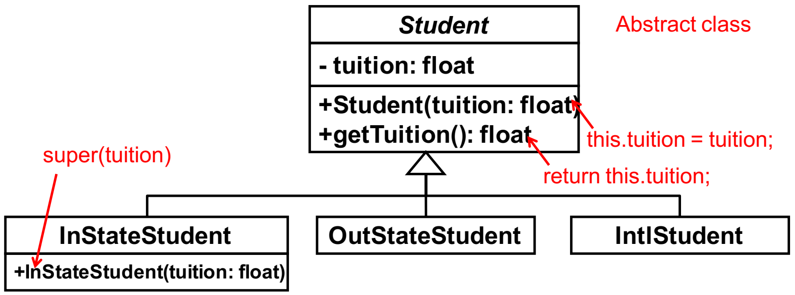
```
Date d = new Date();
d.after( new Date() );

MyDate md = new MyDate();
md.after( new Date() ); ← no need to cast "md"

md.after(d);
d.before(md); ← no need to cast "md"
```

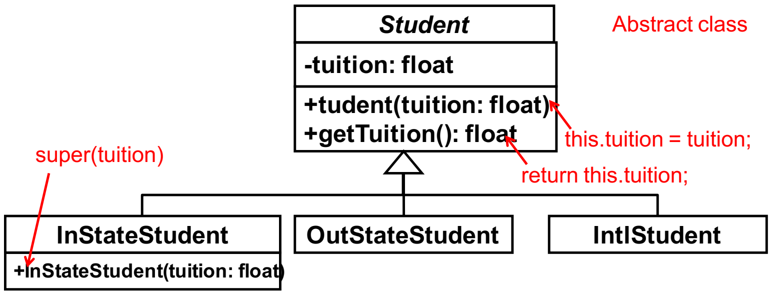
- Generalization-specialization relationship
  - a.k.a. "is-a" relationship
- A subclass can *extend* and reuse a base/super class by adding extra data fields and methods.
  - Constructors are not inherited.

Quiz


Student
Abstract class
- tuition: float
+Student(tuition: float)
+getTuition(): float
this.tuition = tuition;
return this.tuition;
super(tuition)
InStateStudent
+InStateStudent(tuition: float)
OutStateStudent
IntlStudent

```
ArrayList<Student> students = new ArrayList<Student>();
students.add( new OutStateStudent(2000) );
students.add( new InStateStudent(1000) );
students.add( new IntlStudent(3000) );
Iterator<Student> it = students.iterator();
while( it.hasNext() )
  System.out.println( it.next().getTuition() );
```

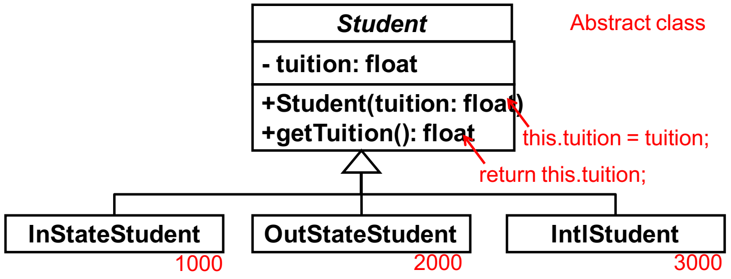
- What are printed out in the standard output?


Student
Abstract class
-tuition: float
+tudent(tuition: float)
+getTuition(): float
this.tuition = tuition;
return this.tuition;
super(tuition)
InStateStudent
+InStateStudent(tuition: float)
OutStateStudent
IntlStudent

```
ArrayList<Student> students = new ArrayList<Student>();
students.add( new OutStateStudent(2000) );
students.add( new InStateStudent(1000) );
students.add( new IntlStudent(3000) );
Iterator<Student> it = students.iterator();
while( it.hasNext() )
  System.out.println( it.next().getTuition() );
```

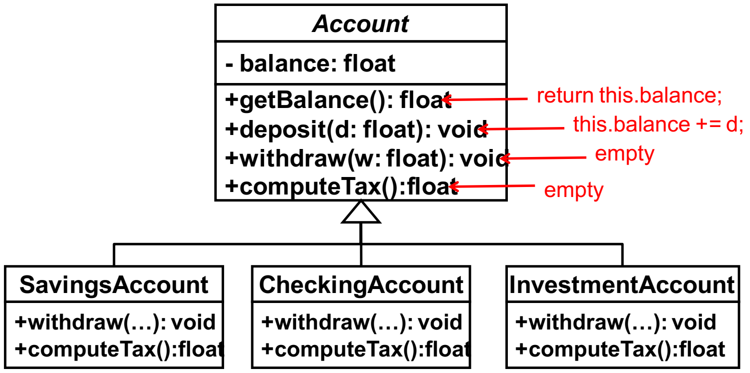
- 2000
  1000
  3000

Polymorphism

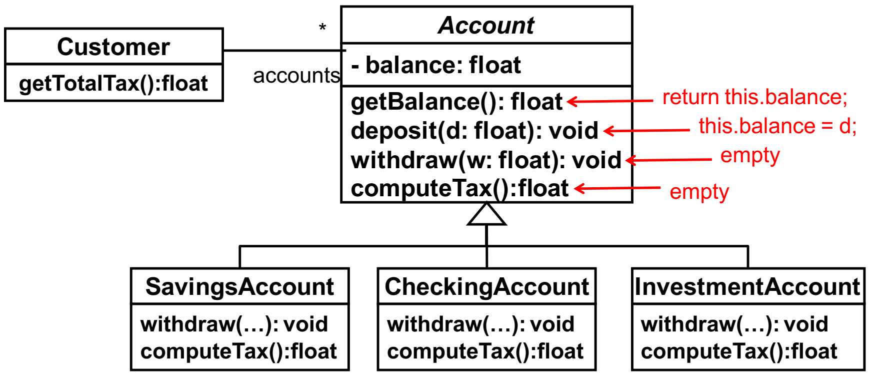

Student
Abstract class
- tuition: float
+Student(tuition: float)
+getTuition(): float
this.tuition = tuition;
return this.tuition;
InStateStudent
1000
OutStateStudent
2000
IntlStudent
3000

```
ArrayList<Student> students = new ArrayList<Student>();
students.add( new OutStateStudent(2000) );
students.add( new InStateStudent(1000) );
students.add( new IntlStateStudent(3000) );
Iterator<Student> it = students.iterator();
while( it.hasNext() )
  System.out.println( it.next().getTuition() );
```

- All slots in “students” (an array list) are typed as Student, which is an abstract class.
- Actual elements in “students” are instances of Student’s subclasses.


Account
- balance: float
+getBalance(): float
+deposit(d: float): void
+withdraw(w: float): void
+computeTax():float
return this.balance;
this.balance += d;
empty
empty
SavingsAccount
+withdraw(…): void
+computeTax():float
CheckingAccount
+withdraw(…): void
+computeTax():float
InvestmentAccount
+withdraw(…): void
+computeTax():float

- Subclasses can redefine (or override) inherited methods.
  - A savings account may allow a negative balance with some penalty charge.
  - A checking account may allow a negative balance if the customer’s savings account maintains enough balance.
  - An investment account may not allow a negative balance.



Customer

getTotalTax():float

accounts

*

Account

- balance: float

getBalance(): float

deposit(d: float): void

withdraw(w: float): void

computeTax():float

return this.balance;

this.balance = d;

empty

empty

SavingsAccount

withdraw(…): void

computeTax():float

CheckingAccount

withdraw(…): void

computeTax():float

InvestmentAccount

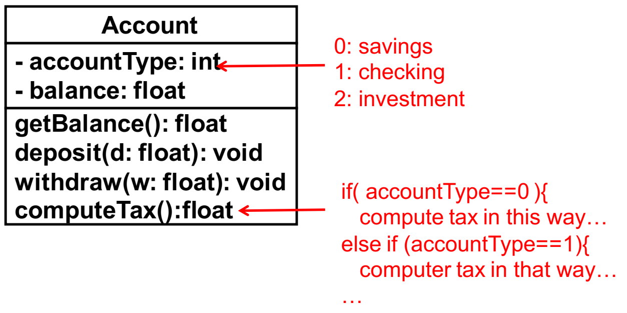
withdraw(…): void

computeTax():float

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

- Polymorphism can effectively eliminate conditional statements.
  - Conditional statements are VERY typical sources of bugs.

If Polymorphism is not available…



Account

- accountType: int

- balance: float

getBalance(): float

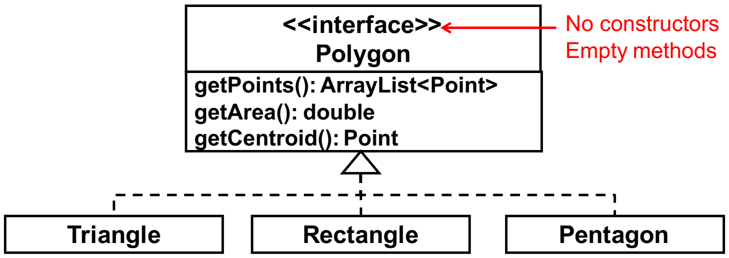
deposit(d: float): void

withdraw(w: float): void

computeTax():float

0: savings
1: checking
2: investment

```
if( accountType==0 ){
   compute tax in this way…
else if (accountType==1){
   computer tax in that way…
…
```



<<interface>>
Polygon

getPoints(): ArrayList<Point>

getArea(): double

getCentroid(): Point

No constructors
Empty methods

Triangle

Rectangle

Pentagon

-
```
ArrayList<Polygon> p = new ArrayList<Polygon>();
p.add( new Triangle( new Point(0,0),
                     new Point(2,2),
                     new Point(1,3) ));
p.add( new Rectangle ( new Point(0,0)... ));
Iterator<Polygon> it = p.iterator();
while( it.hasNext() ){
  Polygon nextP = it.next();
  System.out.println( nextP.getPoints() );
  System.out.println( nextP.getArea() );
  System.out.println( nextP.getCentroid() ); }
```

HW 2-2

- Learn generics in Java (e.g., ArrayList) and understand how to use it.
- Learn how to use java.util.Iterator.

- This code runs.
  -
```
ArrayList<Student> al = new ArrayList<Student>();
al.add( new OutStateStudent(2000) );
System.out.println( al.get(0).getTuition() ); → 2000
```

- This one doesn't due to a compilation error.
  -
```
ArrayList al = new ArrayList();
al.add( new OutStateStudent(2000) );
System.out.println( al.get(0).getTuition() );
```

- Describe what the error is and why you encounter the error.

HW 2-2 (cont'd)

- Write the Polygon interface and its two implementation classes: Triangle and Rectangle.
  - You can reuse Point in Java API or define your own.
- Implement getPoints() and getArea() in the two subclasses.
  - Use Heron's formula to compute a triangle's area.
    - The area of a triangle = Sqrt(s(s-a)(s-b)(s-c))
      - where s=(a+b+c)/2
      - a, b and c are the lengths of the triangle's sides.
- In the main() method, write test code that
  - makes two different triangles and two different rectangles,
  - contains those 4 polygons in a collection (e.g. ArrayList),
    - Use generics and an iterator
  - printouts each polygon's area.
- Keep the encapsulation principle in mind.
  - All data fields must be "private."
  - No setter methods are required.

HW2-3

- Learn general ideas on 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
- Read "Replace Conditional with Polymorphism"
  - http://www.refactoring.com/catalog/replaceConditionalWithPolymorphism.html
  - http://sourcemaking.com/refactoring/replace-conditional-with-polymorphism

Note

- If you are not very familiar with class inheritance and polymorphism, you may want to implement Student and Account examples as well as extra exercise.

Note

- Use Ant (http://ant.apache.org/) to compile/build all of your HW solutions.
  - Learn how to use it, if you don't know that.
  - Turn in *.java and build.xml for every coding HW.
    - Turn in a ***single*** build script (build.xml) that
      - configures all settings (e.g., class paths and a directory to generate binary code),
      - compiles all source code from scratch,
      - generates binary code, and
      - runs compiled code
    - DO NOT include absolute paths in build.xml.
      - You can assume my OS configures a right Java API JAR file (in its env setting).
    - DO NOT turn in byte code (class files).
    - DO NOT use any other ways for configurations and compilation.
      - Setting class paths manually with a GUI (e.g., Eclipse)
      - Setting an output directory manually in a GUI
      - Clicking the "compile" button manually

- I will simply type “ant” (on my shell) in the directory where your build.xml is located and see how your code works.
  - If the “ant” command fails, I will NOT grade your HW code.
- Fully automate configuration and compilation process to
  - speed up your configuration/compilation process.
  - remove potential human-made errors in your configuration/compilation process.
  - Make it easier for other people (e.g., code reviewers, team mates) to understand your code/project.

HW2-4

- J. Spolsky, “The Joel Test: 12 Steps to Better Code,” In Joel on Software, Chapter 3, Apress, 2004.
  - http://www.joelonsoftware.com/articles/fog0000000043.html
- OPTIONAL: M. Chapman, “Apache Ant 101: Make Java builds a snap,” IBM developerWorks, 2003.
  - http://www.ibm.com/developerworks/java/tutorials/j-apant/