Welcome to CS680!

Tue Thu 5:30pm - 6:45pm

Y-2-2110

Who am I?

- Academics
  - Associate Professor, UMass Boston (2010–)
  - Assistant Professor, UMass Boston (2004–2010)
    - Distributed systems, software engineering and AI
    - www.cs.umb.edu/~jxs/; dssg.cs.umb.edu
  - Post-doctoral Research Fellow, UC Irvine, CA (2000–2004)
  - Ph.D. in Comp Sci from Keio University, Japan (2001)
- Industrial
  - Consultant, cloud computing platform vendor, supply chain mgt. company
  - Tech Director, Object Management Group Japan
  - Co-founder and CTO, TechAtlas Comm Corp, Austin, TX
  - Programmer Analyst, Goldman Sachs Japan
- Professional
  - Member, ISO SC7/WG 19
  - Specification co-lead, OMG Super Dist. Objects SIG

Course Topics

- Object-oriented design
  - Design patterns
  - Refactoring
- Continuous delivery/integration (CD/CI)
  - Testing (incl. unit testing, static code inspection)
  - Versioning
  - Automated deployment, configuration mgt., integration testing, etc.
- Model-view-controller (MVC) architecture
- Lambda expressions in Java
  - Basics in functional programming

Textbooks

- No official textbooks.
- Recommended textbooks
  - *Object-Oriented Analysis and Design with Applications (3rd edition)*
    - by Grady Booch et al. (Addison Wesley)
    - General intro to OOAD.
  - *Refactoring: Improving the Design of Existing Code*
    - by Martin Fowler
    - Addison-Wesley
  - *Head Start Design Patterns*
    - by Elizabeth Freeman et al.
    - O'Reilly

- The most authoritative and Bible-like book on design patterns:
  - *Design Patterns: Elements of Reusable Object-Oriented Software*
    - By Eric Gamma et al.
    - Addison-Wesley

Course Work

- Lectures
- Homework
  - Paper reading
  - Modeling/design and coding (Java)
- Individual project

Grading

- Grading factors
  - Homework (65%)
  - Project deliverables (30%)
  - Quizzes (5%)
    - Occasionally, at the beginning of a lecture
- No midterm and final exams.

My Email Addresses

- Questions → **jxs@cs.umb.edu**
  - I regularly check this account.
- HW solutions → **umasscs680@gmail.com**
  - I occasionally check this account.
    - Once a week or so.

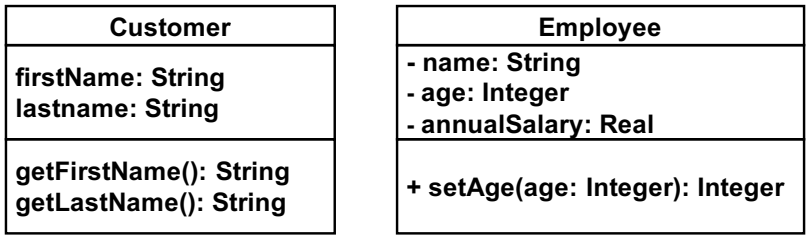
Your Email Addresses

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

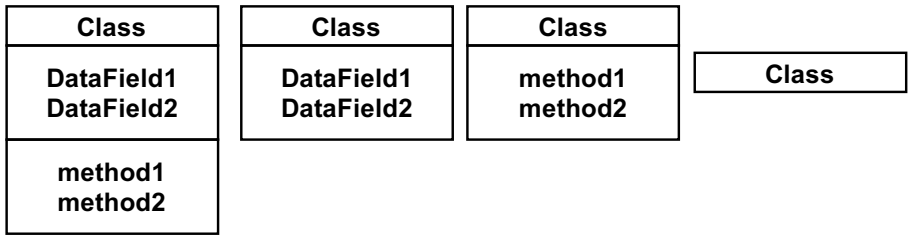
Preliminaries: Unified Modeling Language (UML)

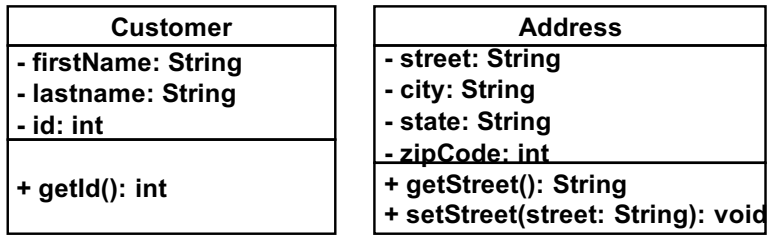
Unified Modeling Language (UML)

- A language to visually ***model*** software
  - Intuitively, it is a set of icons, symbols and diagrams that denote particular elements in software designs.

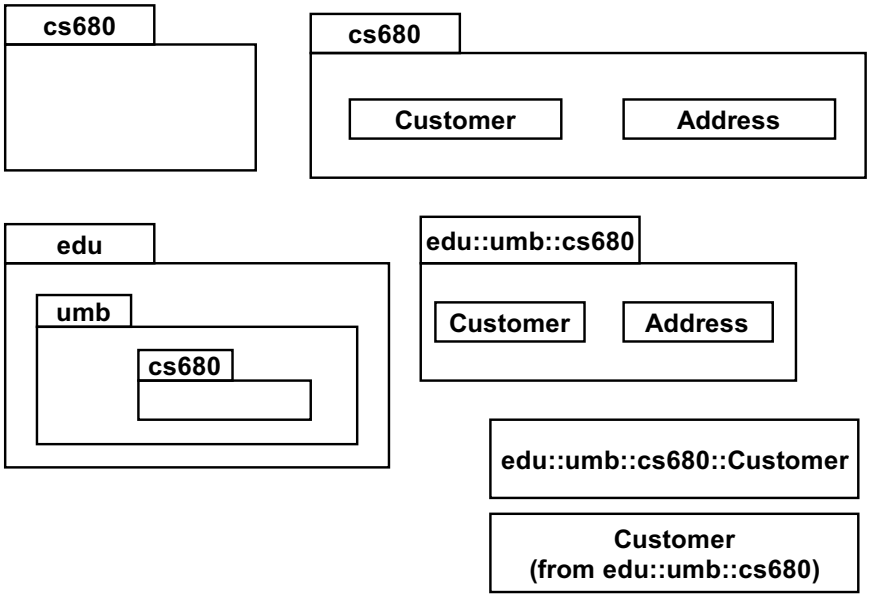


Classes in UML



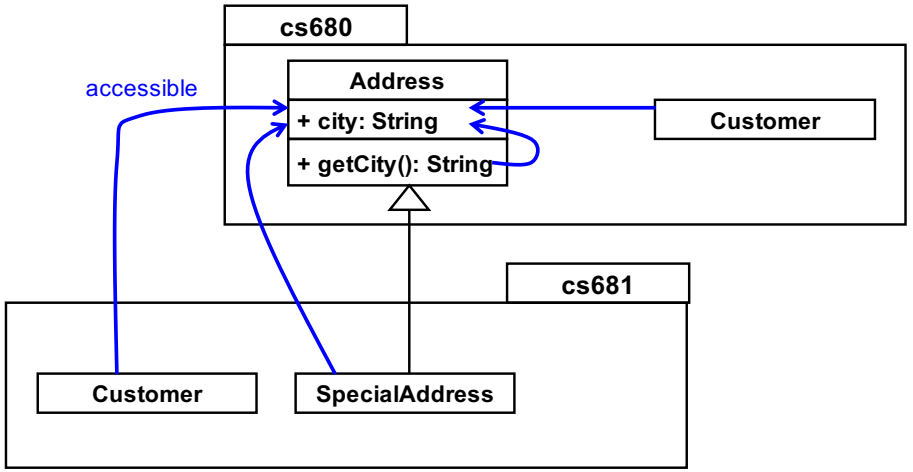


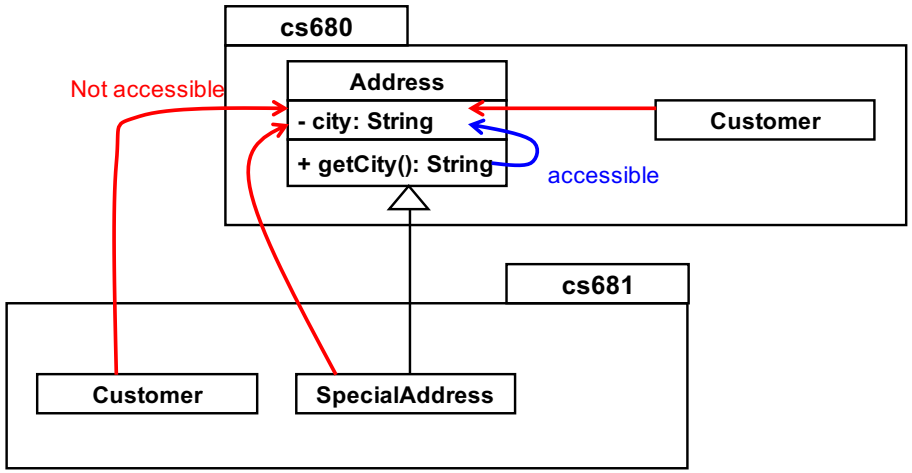
Packages in UML



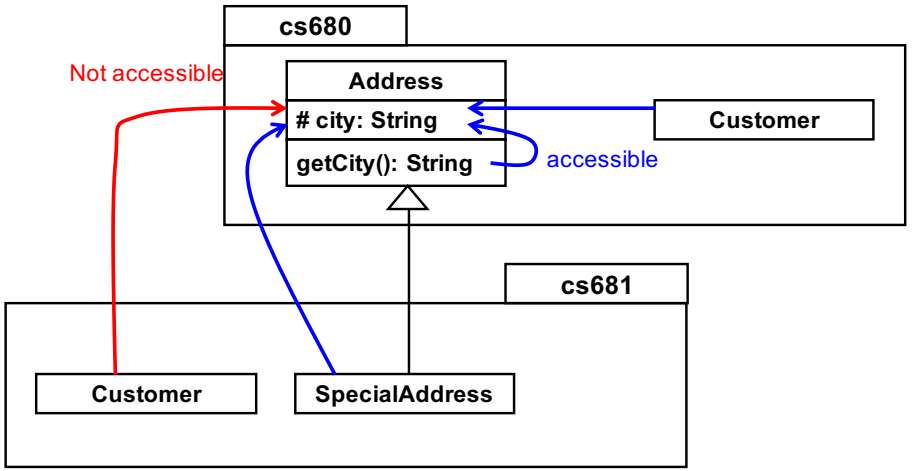
Java's Attribute/Op Visibility in UML

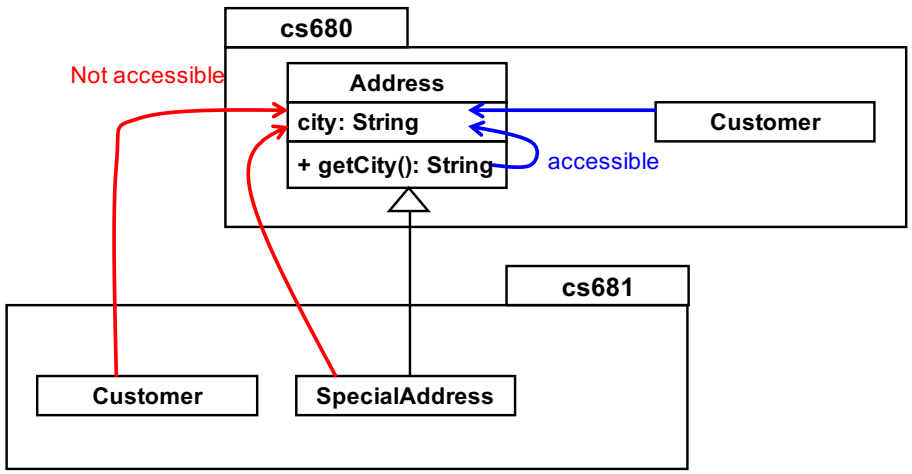
- Defines who can access an attribute or operation.
  - Public (+), private (-) or protected (#)





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

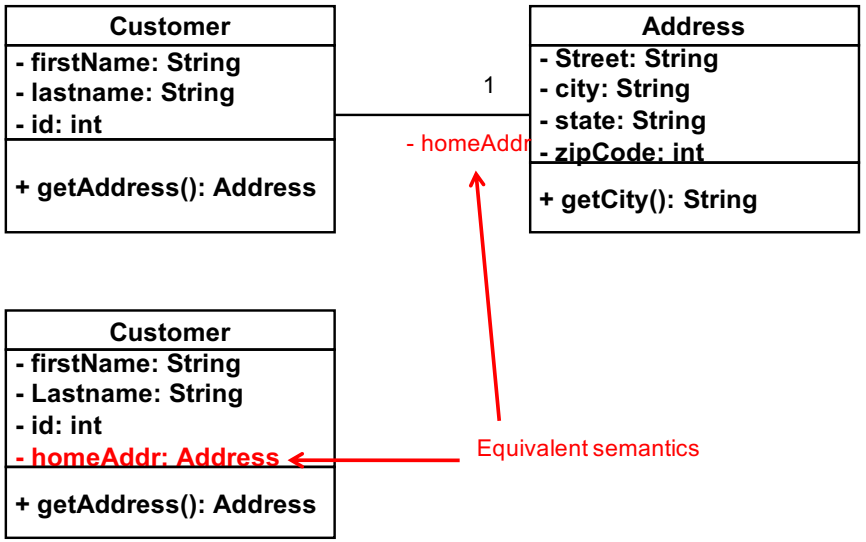


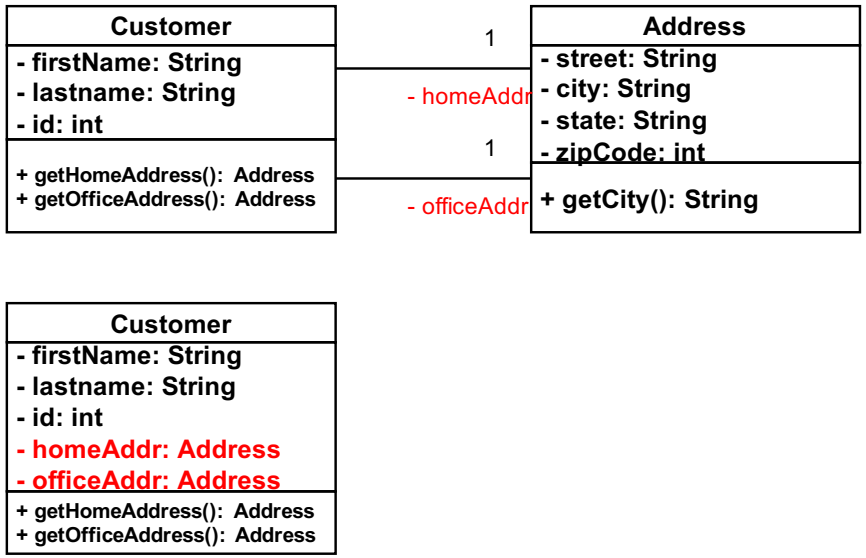


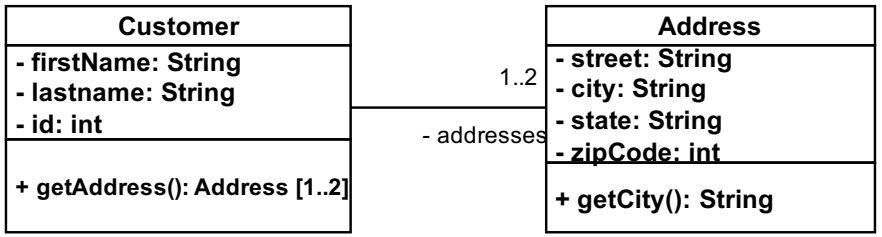
Default visibility (package private) to be used when no modifier is specified.

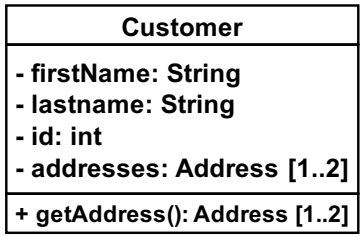
- Specify the modifier for every data field and every method.
- Do not to skip specifying it. (Do not use package-private.)
- It is important to be always aware of the visibility of each data field and method.

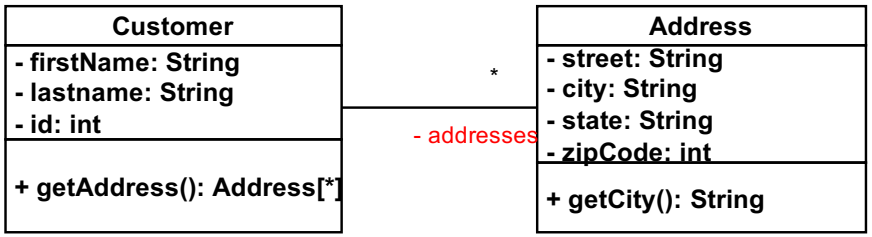
Association

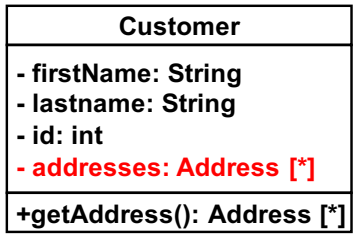


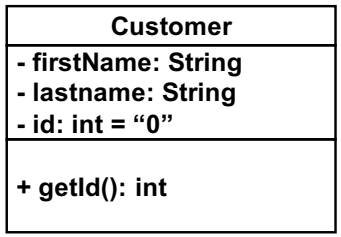












Preliminaries: Road to Object-Oriented Design (OOD)

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 easy (cost effective) to add and revise existing modules (classes and interfaces) to accommodate new/modified requirements.
    - How to make software more flexible/robust against changes in the future.
  - How to gain reusability, maintainability and extensibility?

Looking Ahead: AOP, etc.

- OOD does a pretty good job in terms of modularity, but it is not perfect
- OOD still has some modularity issues
  - Aspect Oriented Programming (AOP)
    - Dependency injection
    - Handles cross-cutting concerns well.
      - e.g. logging, security, DB access, transactional access to a DB
- Highly modular code sometimes look redundant.
  - Functional programming
    - Makes code less redundant.
  - Lambda expressions in Java
    - Intend to make modular code less redundant.

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.

Recap

- Specify the modifier for every data field and every method.
- Do not to skip specifying it. (Do not use package-private.)
- It is important to be always aware of the visibility of each data field and method.

Violation of Encapsulation

- 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.

- There are a good number of data that don't have to be modified once they are generated.
  - e.g., globally-unique IDs (GUIDs), MAC addresses, customer IDs, product IDs, etc.
- Define them as private/protected data fields.
- No need to define setter methods.

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,

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

- You cannot be so sure about potential bugs.

- 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."

Be Preventive!

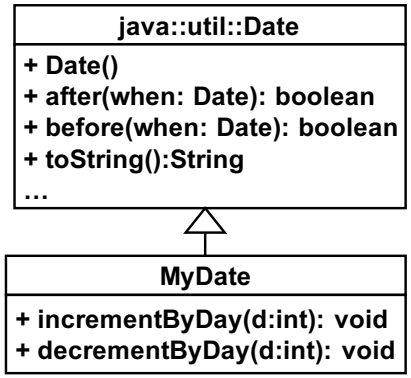
- Encapsulation
  - looks very trivial.
  - is not that important in small-scale (toy) software
    - because you can manage (i.e., read, understand and remember) every aspect of the code base.
  - is very important in large-scale (real-world) software
    - because you cannot manage (i.e., read, understand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

Inheritance (Generalization)

Inheritance

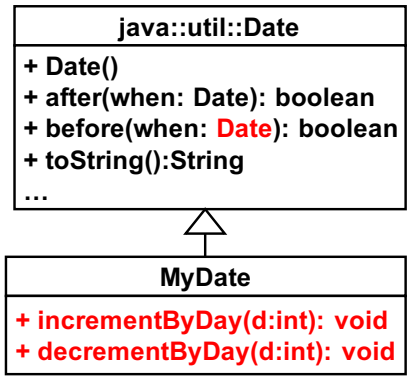



```
Date d = new Date();
d.after( new Date() );

MyDate md = new MyDate();
md.after( d );
// after() is inherited to MyDate
```

- Generalization-specialization relationship
  - a.k.a. "is-a" relationship; MyDate is a (kind of) Date.
- A subclass can *inherit* all public/protected data fields and methods from its base/super class.
  - Exception: Constructors are not inherited.

Inheritance (cont'd)




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

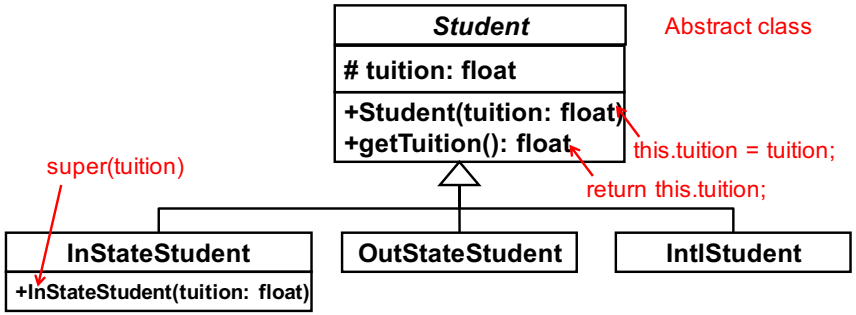
MyDate md = new MyDate();
md.after( new Date() );

d.incrementByDay(1);
// Compilation error
md.incrementByDay(2);  // OK

d.before(md); // OK
```

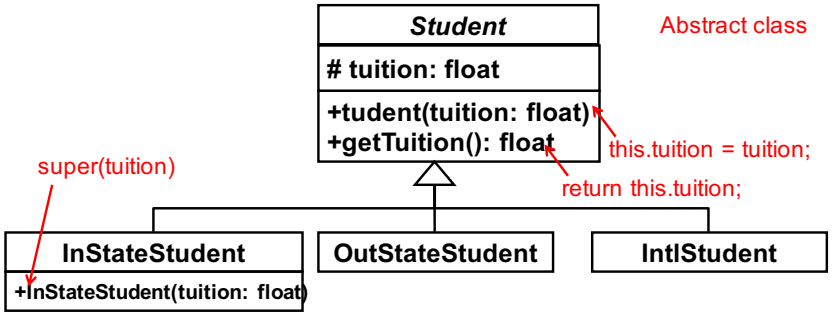
- A subclass can *extend* a base/super class by adding extra data fields and methods.
- An instance of a subclass can be assigned to a variable typed as the class's superclass.

Quiz




```
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() );
```

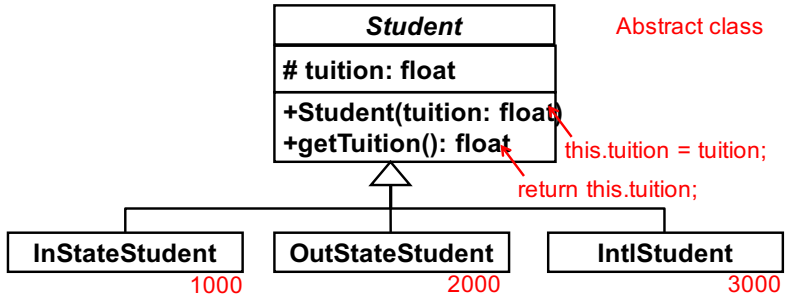
- What are printed out in the standard output?




```
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()  );
```

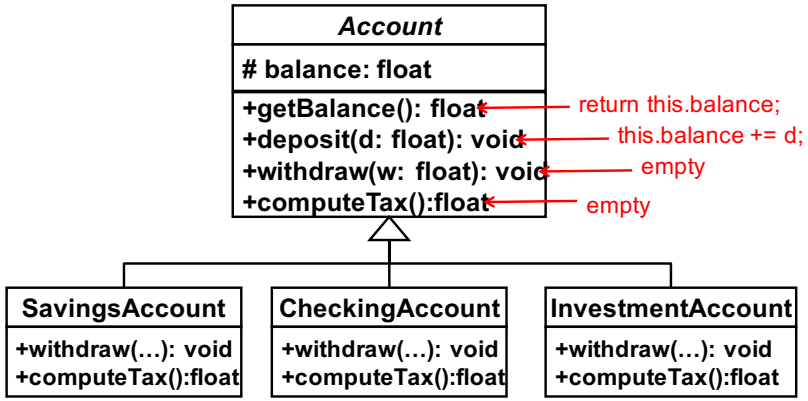
- 2000
  1000
  3000

Polymorphism

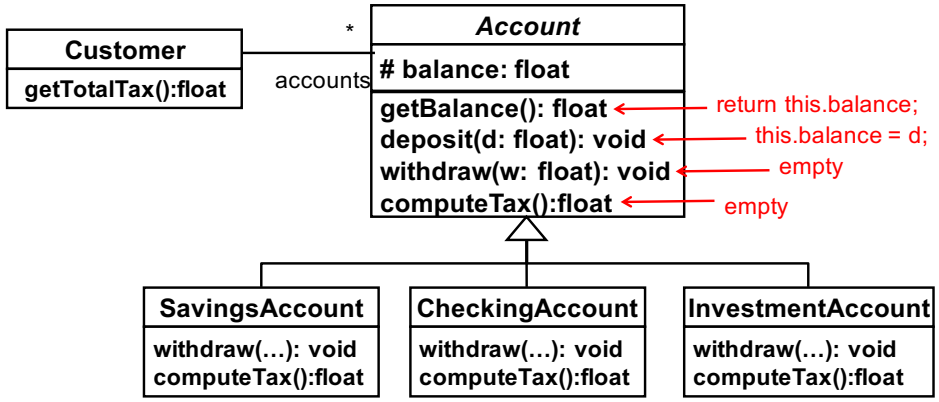



```
ArrayList<Student> students = new ArrayList<Student>();
students.add( new OutStateStudent(2000) );
students.add( new InStateStudent(1000) );
students.add( new IntlStateStudent(3000) );
Iterator<Student> it = students.iterator();
while( it.hasNext() )
  System.out.println( it.next().getTuition() );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

- 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.



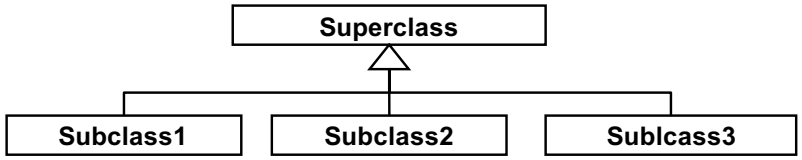
- Subclasses can redefine (or override) inherited methods.
  - A savings account may allow a negative balance with some penalty charge.
  - A checking account may allow a negative balance if the customer's savings account maintains enough balance.
  - An investment account may not allow a negative balance.




```
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.

Why Inheritance?



- Reusability
  - You can define common data fields and methods in a superclass and make them reusable in subclasses.
- Customizability and extensibility
  - You can customize method behaviors in different subclasses by overriding (re-defining) inherited methods.
  - You can add new data fields and methods in subclasses.

HW 1

- 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.

CS682-3

A Year-long Project Course

- Each team works on a project with a customer
  - 3 to 6 students per team
    - 4 on average
- Understand what your customer wants.
  - Requirement gathering
- Deliver a system/product that your customer wants.
  - by the end of May 2017.

Customers

- TBA
- The first customer presentation on this Thu, 5:30pm.
- The remaining ones in the week of Sept 19.

CS682/3

- Each project team will …
  - Meet the customer once or even twice a week to understand his/her/their needs and gather a set of requirements.
  - Regularly meet as a team to prepare questions to the customer and discuss QAs with the customer.
  - Learn required technologies, tools, frameworks, etc.
  - Discuss the structure/architecture of your product
  - Start coding/implementation work as early as possible in Fall.
    - **Your grade will be penalized if you do not start actual coding/impl work in Fall.**
- Progress presentations
  - e.g. mid/beginning Nov, Mid Dec

Grading

- Based on a combination of
  - Individual contribution to your team.
    - You and I will assess the quality/quantity of your individual contribution.
  - Team work
    - Your team mates and I will assess the quality/quantity of your team work.
  - Feedback from your customer
- Notes
  - Do not try to impress me. Impress your customer.
  - Your customer may not be very patient.

FAQs

- How do you assign me to a project?
- How many team mates will I have?
- I want to work with my friends. I do not want to work with that person. Can you redo project assignments?
- How do you grade me?