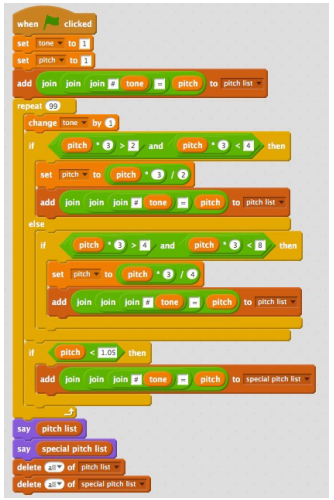
Preliminaries: Road to Object-Oriented Programming and Design (OOPL/D)

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.



- Programming languages had no notion of structures (or modularity)
- As the size and complexity of software increased, languages needed modularity.
  - Module: A chunk of code
  - Modularity: Making modules self-contained and independent from others

- Programming languages had no notion of structures (or modularity)
- As the size and complexity of software increased, languages needed modularity.
  - Module: A chunk of code
  - Modularity: Making modules self-contained and independent from others
  - Goal: Improve productivity and maintainability
    - Higher productivity through higher reusability
      - Lead to less production costs
    - Higher maintainability through clearer separation of concerns
      - Lead to less maintenance costs

Modules in SLs and OOPLs

- Modules in Structured Prog. Languages (SPLs)
  - Structure = a set of variables
  - Function = a block of code
- Modules in Object-Oriented PLs (OOPLs)
  - Class = a set of variables (data fields) and functions (methods)
  - Interface = a set of functions (methods)
- Key design questions/challenges:
  - how to define modules?
  - how to separate a module from others?
  - how to let modules interact with each other?

OOPLs v.s. SPLs

- OOPLs
  - Intend coarse-grained modularity
    - The size of each module is often bigger in OOPLs.
  - Extensibility in mind to enhance productivity and maintainability further.
    - How easy (cost effective) to add and revise existing modules (classes and interfaces) to implement new/modified requirements.
    - How to make software more flexible/robust against changes in the future.
  - How to leverage modularity to address reusability, separation of concerns and extensibility?

Looking Ahead: AOP, etc.

- OOPLs do a pretty good job in terms of modularity, but it is not perfect.
- OOPLs still have 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 (and expressive) code sometimes look redundant (repetitive).
  - Functional programming
    - Makes code less redundant/repetitive.
  - Lambda expressions in Java
    - Intended to make modular OO code less redundant/repetitive.

Encapsulation

What is Encapsulation?

- Hiding each class's internal details from its clients
  - To improve its modularity, robustness and ease of understanding
  - "Clients" ("client code"):
    - A piece of code that uses a class in question.

- Things to do:
  - Always make your data fields private or protected.
  - Make your methods private or protected as often as possible.
  - Avoid public accessor methods (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();
…
```


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 find a runtime 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 narrow the scope of your debugging effort.
    - You can be more confident about your debugging.

Recap

- Specify the modifier for every data field and every method.
- Do not to skip specifying it. (Do not use package-private.)
- It is always important to be 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 members may write this code by accident.
    - It may look stupid, but it is not uncommon in reality.
  - 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), customer IDs, product IDs (e.g., MAC addresses), 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 about which classes/methods I should modify according to a code revision.
  - Need UML diagrams.
  - Need test cases (unit test cases) as example use cases.
  - Need comments, memos and/or documents about design rationales

Why Encapsulation? (cont'd)

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

```
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 these,

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

```
final public class Person{
    private 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 larger-scale (real-world) software
    - because you cannot manage (i.e., read, understand and remember) every aspect of the code base.

Example Cases

-
```
public class Sensor{
    protected int id;
    Sensor(int id){ this.id = id; }
    public int getId(){ return this.id; } }
```
-
```
public class SensorV2 extends Sensor{
    Person(int id){ super(id); }
    public void setId(int id){ this.id=id; } }
```

```
- public class Package{
      protected int id;
      Package(int id){ this.id = id; }
      public int getId(){ return this.id; } }

- public class TrackablePackage extends Package{
      private String trackingId;
      TrackablePackage(int id, String trackingId){
          super(id);
          this.trackingId=trackingId; }
      public String getTrackingId(){ ... }
      public void setTrackingId(String trackingId){ ... } }
```


Sounds Trivial?

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

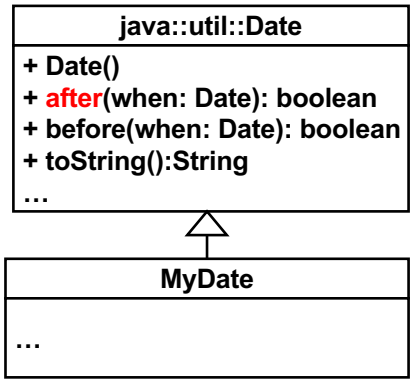
- Once you finish up writing these 4 lines, wouldn't you define a setter method automatically (i.e. without thinking about it carefully)?
  - "I always define both getter and setter methods for a data field. I can delete unnecessary ones anytime later."
  - "Well, let's define a setter just in case."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

Exercise

- Write a program based on a given UML diagram
  - Understand the mapping between UML and Java
  - Understand the concept of visibility
  - Understand other keywords in Java, such as final, static and super.
- An exercise is not a HW. No need to turn in anything for that.

Inheritance (Generalization)

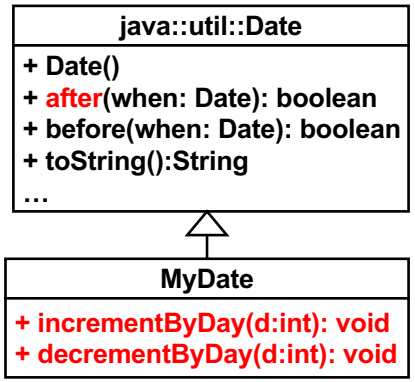
Inheritance




```
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 super class.
  - Exception: Constructors are not inherited.




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

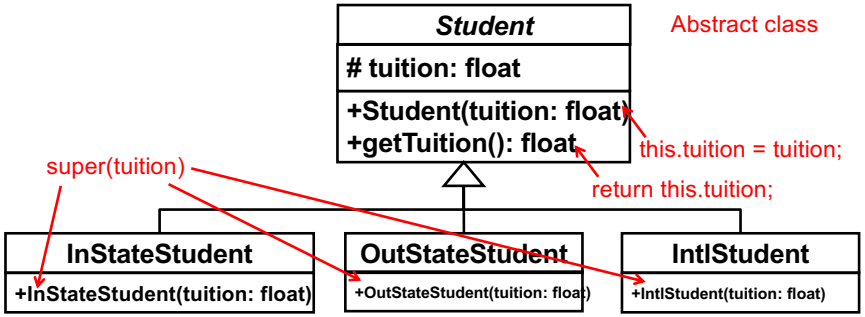
MyDate md = new MyDate();
md.after( new Date() );

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

d = new MyDate();        // OK
d.incrementByDay(1);
// Compilation error
(MyDate)d.incrementByDay(1); // OK
```

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

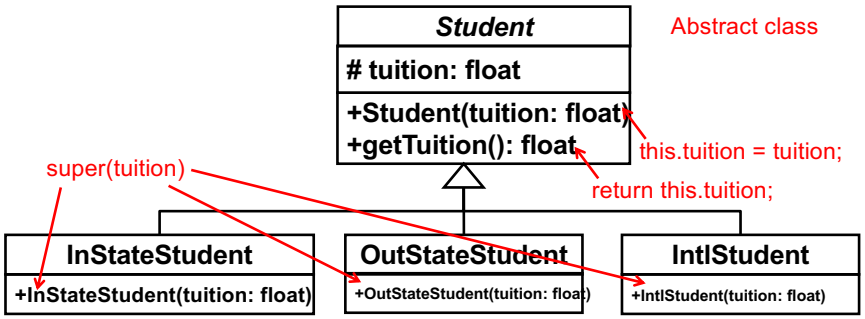
Exercise




```
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) );

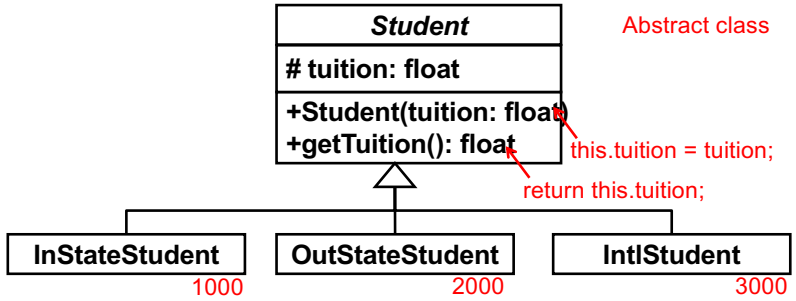
Iterator<Student> it = students.iterator();
while( it.hasNext() )
  System.out.println( it.next().getTuition() );
```

- 2000
  1000
  3000

Exercise

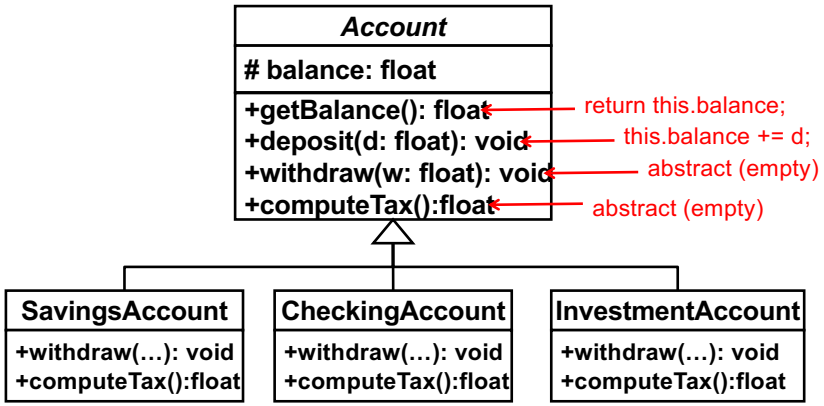
- Learn about collections (e.g. ArrayList) and generics in Java.
- Learn how to use `java.util.Iterator`.
- This code runs.
  - ```
    ArrayList<Student> al = new ArrayList<Student>();
    al.add( new OutStateStudent(2000) );
    System.out.println( al.get(0).getTuition() );
    ```
    → 2000
- This one doesn't due to a compilation error.
  - ```
    ArrayList al = new ArrayList();
    al.add( new OutStateStudent(2000) );
    System.out.println( al.get(0).getTuition() );
    ```
- Understand what the error is and why you encounter the error.

Polymorphism

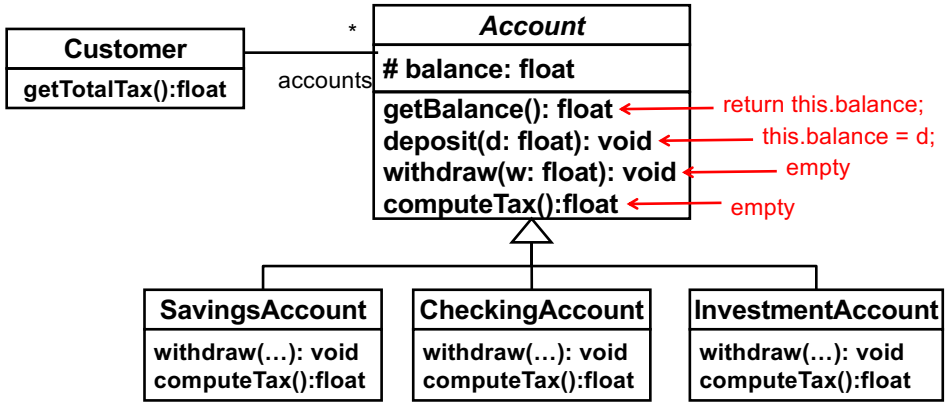



- ```
  ArrayList<Student> students = new ArrayList<Student>();
  students.add( new OutStateStudent(2000) );
  students.add( new InStateStudent(1000) );
  students.add( new IntlStateStudent(3000) );

  Iterator<Student> it = students.iterator();
  while( it.hasNext() )
    System.out.println( it.next().getTuition() );
  ```
- All slots in "students" (an array list) are typed with Student (super class).
- Actual elements in "students" can be 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 withdrawals.




- ```
  public float getTotalTax(){
      Iterator<Account> it = accounts.iterator();
      while( it.hasNext() )
          System.out.println( it.next().computeTax() ); }
  ```
- Polymorphism can effectively eliminate conditionals.
  - Conditional statements are a VERY typical source of bugs.