CS112 Lecture: Inheritance and Polymorphism

Last revised 4/17/06

Objectives:

1. To review the basic concept of inheritance
2. To introduce the notions of abstract methods, abstract classes, and interfaces.
3. To introduce Polymorphism.
4. To introduce issues that arise with subclasses - protected visibility, use of the super() constructor
5. To discuss the notion of multiple inheritance and Java's approach to it

Materials:

1. Dr. Java for demos + file OverrideDemo.java
2. Projectable versions of code snippets
3. Employees demo program - Handout and online demo

I. Review of Basic Concepts

A. Throughout this course, we have been talking about a particular kind of computer programming - object-oriented programming (or OO). As an approach to programming, OO is characterized by three key features:

1. Polymorphism

2. Inheritance

3. Encapsulation

B. Although we have not used the term extensively, much of our study so far has centered on encapsulation.

1. In OO systems, the *class* is the basic unit of encapsulation. A class encapsulates data about an object with methods for manipulating the data, while controlling access to the data and methods from outside the class so as to ensure consistent behavior.

2. This is really what the visibility modifier "private" is all about. When we declare something in a class to be private, we are saying that it can only be accessed by methods defined in that class - that is, it is encapsulated by the class and is not accessible from outside without going through the methods that are defined in the class.

C. We have also made considerable use of inheritance.

1. Examples:

a) At the start of the course, when working with Karel, we developed robot classes that inherited from a basic Robot class that provided certain basic capabilities (e.g. turnLeft(), frontIsClear(), etc.). We extended that basic class to provide additional capabilities (e.g. turnRight(), carpet a corridor with beepers, etc.)

b) The Java awt and swing packages which we have just been studying makes extensive use of inheritance - e.g. various swing classes such as JButton, JLabel etc. are all extensions of a common base class called JComponent.

2. Review of basic terminology: If a class B inherits from a class A:

a) We say that B *extends* A or B is a *subclass* of A.

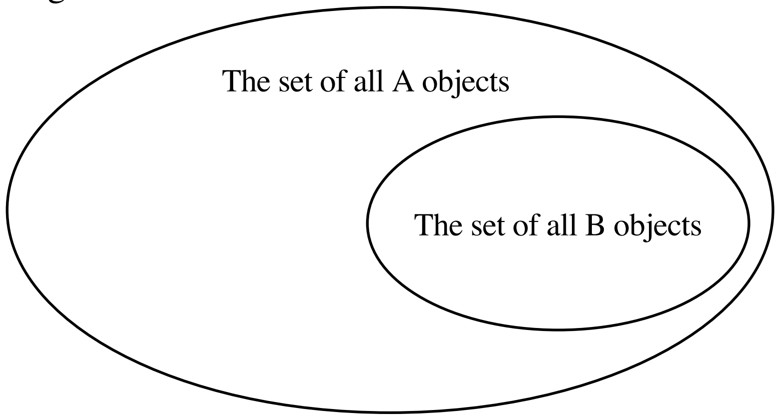
b) We say that A is the *base class* of B or the *superclass* of B.

c) This notion can be extended to multiple levels - e.g. if C extends B and B extends A, then we can say not only that C is a subclass of B, but also that it is a subclass of A. In this case, we sometimes distinguish between *direct* subclasses/base class and *indirect* subclasses/base class.

3. Crucial to inheritance is what is sometimes called *the law of substitution:*

a) If a class B inherits from (extends) a class A, then an object of class B must be able to be used anywhere an object of class A is expected - i.e. you can always substitute a B for an A.

b) This notion is what allows us to call B a subclass of A or A a superclass of B. The set of all "B" objects is a subset of the set of all "A" objects - which potentially includes other "A" objects that are not "B" objects - e.g.



The meaning of "B extends A"

c) This relationship is sometimes expressed by using the phrase “is a” - we say a B “is a” A.

d) Examples:

(1) In the Java awt/swing, the add() method of a Container is defined to take a parameter/first parameter of type Component. However, we never add Components per se to a Container; we always add a particular subclass of Component - e.g. a JButton or a JLabel or a JTextField or whatever. We can do this because a JButton or JLabel or JTextField is a JComponent, which in turn is a Component.

(2) Again, because a Java awt Container is a Component, we can add a Container to another Container, allowing us to have Containers within Containers within Containers ...

e) Remembering the law of substitution will help prevent some common mistakes that arise from misusing inheritance.

(1) The “is a” relationship is similar to another relationship called the containment relationship, or “has a”. Sometimes inheritance is incorrectly used where containment should be used instead.

(2) Example: suppose we were building a software model of the human body, and we wanted to create a class Person to model a whole person, and a class Arm to model a person’s arms. The correct relationship between Arm and Person is a “has a” relationship - a Person “has a” Arm (actually two of them), not “is a” - we cannot say that an Arm is a Person, because we cannot substitute an Arm everywhere a Person is needed.

D. We have also made use of polymorphism, though we have not formally defined the concept.

1. In brief, because of the law of substitution, it is possible for a variable that is *declared* to refer to an object of a base class to *actually refer at run time* to an object of that class or any of its subclasses.

   Example: Given the following declarations (DEMO with Dr. Java - file OverrideDemos.java)

```
class A
{
     public void saySomething(int i)
     { System.out.println(i); }
}

class B extends A
{
     public void saySomething(int i)
     { System.out.println(4); }
}
```

(Load OverrideDemo.java, compile, then issue type the following at the interactions window)

```
A someA;
B someB;
```

all of the following are legal

```
someA = new A();
someA = new B();
someB = new B();
someA = someB;
```

However, the following are *not* legal:

```
someB = new A();   // Illegal!
someB = someA;     // Illegal! (Dr. Java doesn't
                   // catch!)
```

2. Further, when a message is sent to an object, the method used to handle the message depends on the *actual* type of the object, not its declared type.

   Example: To continue the above, suppose that we did the assignment

   ```
   someA = new A();
   ```

   And now performed the the test

   ```
   someA instanceof B
   ```

the instanceof test would fail (An A is *not necessarily* a B, though the reverse is true) and no output would be printed.

However, if we did the assignment

```
someA = new B();
```

and then performed the same test, the test would succeed because instanceof looks at the actual class of the object referred to, which may be the declared class or one of its subclasses.

By the way - in both cases the test

```
if (someA instance of A)
```

would succeed, because a B is an A.

likewise, if we did

```
someB = new B();
```

`someB instanceof A` would succeed since a B is an A.

3. A consequence of this is that a class can *override* a method of its base class, and the method that is used depends on the actual type of the receiver of a message.

   Example

   ```
   someA = new B();
   someA.saySomething(-1);
   ```

   What will the output be?

   ASK

   42 - since `someA` actually belongs to class `B`, the class `B` version of `saySomething()` is the one that is used.

   a) When a class has a method with the *same name and signature* as an inherited method in its base class, we say that the inherited method is *overridden.*

   b) Note that overridden methods must have the *same* signature as the inherited method they override - otherwise we have an overload, not an override.

EXAMPLE: Suppose, in the above, I instead defined subclass C with a method called `saySomething(short i)`, instead of the method whose parameter is of type `int`..

```
class C extends A
{
   public void saySomething(short i)
   { System.out.println(42); }
}
```

What I actually have in this case is an overload rather than an override,

Now suppose I write

```
new C().saySomething(-1);
```

What will the output be?

ASK

-1

However, I would get the other method method (hence output of 42) if I used `new C().saySomething((short) -1)`

c) When a base class method is overridden in a subclass, the base class method becomes invisible unless we use a special syntax to call it:

super.<methodname> (<parameters>)

EXAMPLE: Suppose I include a method in B like the following:

```
   public void speak()
   { saySomething(0); }
```

Then issued the command

```
new B().speak();
```

what will the output be?

ASK

42 - Since we use the B version of `saySomething()`. To get the A version, I could code the body as

```
super.saySomething(0);
```

II. Abstract Methods, Abstract Classes, and Interfaces

A. We have already seen that, when B is a subclass of A, B inherits all the methods of A.

1. Much of the power of inheritance comes from the fact that B ordinarily inherits A's *implementation* of these methods, so B does not have to implement them itself.

   Example: All of the Robot classes we defined early in the course inherited the implementation of the primitive methods such as turnLeft(), so we didn't have to provide an implementation for them ourselves. (Indeed, if we did have to do so, we would have been stuck!)

2. Sometimes, though, B needs to replace A's implementation of some method with a version of its own. In this case, we say that B's method *overrides* A's.

   Example: Suppose we were developing a payroll system for a company where all the employees are paid based on the number of hours worked each week. We might develop an Employee class like the following:

   PROJECT

```
public class Employee
{
     public Employee(String name, String ssn, double hourlyRate)
     {
          ...
          this.hourlyRate = hourlyRate;
     }
     public String getName()
     ...
     public String getSSN()
     ...
     public double weeklyPay()
     {
          // Pop up a dialog box asking for hours worked this week
          return hoursWorked * hourlyRate;
          // Actually should reflect possible overtime in above!
     }
     ...
     private String name;
     private String ssn;
     private double hourlyRate;
}
```

   Now suppose we add a few employees who are paid a fixed salary.

   a) We could create a new class SalariedEmployee that overrides the weeklyPay() method, as follows: (PROJECT)

```
class SalariedEmployee extends Employee
{
    public SalariedEmployee(String name,String ssn,double annualSalary)
   ...
   public double weeklyPay()
   { return annualSalary / 52; }
   ...
   private double annualSalary;
}
```

b) It would now be possible to create an array of Employee objects, some of whom would actually be SalariedEmployees - e.g. (PROJECT)

```
Employee [] employees = new Employee[10];
employees[0]=new SalariedEmployee("Big Boss","999-99-9999",100000.00);
employees[1]=new Employee("Lowly Peon", "111-11-1111", 4.75);
...
```

c) Further, we could iterate through the array and call the weeklyPay() method of each, without regard to which type of employee each represents, and the correct version would be called: (PROJECT)

```
for (int i = 0; i < employees.length; i ++)
   printCheck(employees[i].getName, employees[i].weeklyPay());
```

Note that, in each case, the appropriate version of weeklyPay() is called - e.g. for Big Boss, the SalariedEmployee version is called and a check for 1923.08 is printed; for Lowly Peon a dialog is popped up asking for hours worked and the appropriate amount is calculated based on a rate of 4.75 per hour. This is another example of *polymorphism*.

B. There are times, though, when defining a method in the base class whose implementation is inherited by the subclasses is not at all what we want to do. In such cases, we may want to create an abstract class and method(s).

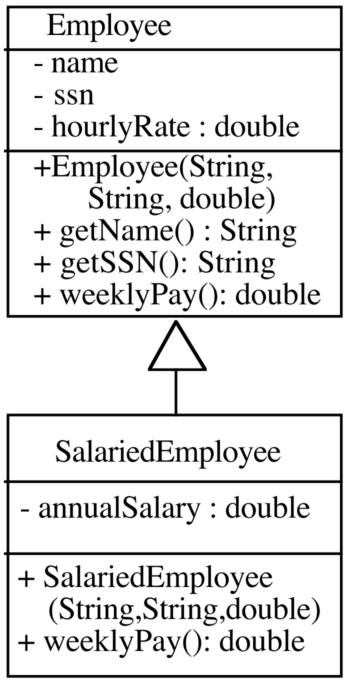
1. To see why this is so, consider the example we just looked at.

a) The solution we just developed is not really a good one.

Why?

ASK

Because SalariedEmployee inherits from Employee, every SalariedEmployee has an hourly rate field, even though it is not used. (The hourlyRate field is private, so it is not inherited in the sense that it is not accessible from within class SalariedEmployee; however, it does exist in the object and is initialized by the constructor - so a value must be supplied to the constructor even though it is not needed!) This can be seen from the following Class diagram, which uses a language-independent notation known as UML (Unified Modelling Language)



(1) Each box stands for a class. The arrow with a triangle at the head connecting them indicates that the class SalariedEmployee extends Employee - i.e. a SalariedEmployee "isa" Employee.

(2) Each box has three compartments. The first contains the name of the class (and potentially certain other information about the class as we shall see later). The second contains the fields of the class (the instance and class variables). The third contains the methods.

(a) A subclass <u>inherits</u> all the fields of its superclass. Thus, a SalariedEmployee object has four fields - name, ssn, and hourly rate (inherited from Employee) and annualSalary (defined in the class)

(b) A subclass also <u>inherits</u> all the instance methods of its superclass, unless it overrides them. Thus, the SalariedEmployee class inherits the getName() and getSSN() methods from is superclass and overrides the weeklyPay() method. It does <u>not</u> inherit the constructor, since this is not an instance method.

(3) Each field and method is preceeded by a <u>visibility specifier.</u> The possible specifiers are:

(a) + - accessible to any object - this corresponds to Java public

(b) - - accessible only to objects of this class - this corresponds to Java private

(c) # - accessible only to objects of this class or its subclasses - which corresponds to Java protected (which we haven't really used yet.). Note that, in this example, name and ssn are not made protected - the subclass has access to them through public methods getName() and getSSN().

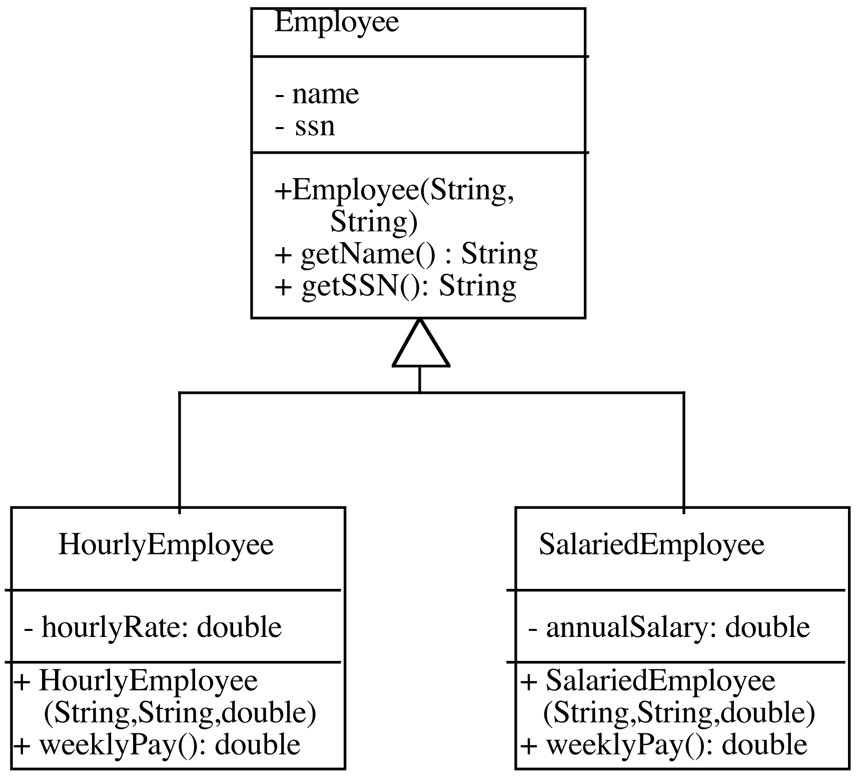
(4) Each field has a type specifier, and each method has a return type specifier.

(5) Each method has type specifiers for its parameters (its signature).

b) What would be a better solution?

ASK

Create a *class hierarchy* consisting of a base class called Employee and *two* subclasses - one called HourlyEmployee and one called SalariedEmployee. Only HourlyEmployees would have an hourlyRate field, while SalariedEmployees would have an annualSalary field. This is expressed by the following diagram:



Notice that what we have done is to leave in the base class only those fields and methods which are common to the two subclasses. We have also eliminated the need for a pay rate parameter in the Employee constructor - we only specify the name and ssn. We likewise have eliminated the weeklyPay() method, since this is different for each subclass, and each implementation uses a field specific to that subclass.

c) However, this solution introduces a new problem. The following code, which we used above, would no longer work: (PROJECT AGAIN)

```
Employee [] employees = new Employee[10];
...
for (int i = 0; i < employees.length; i ++)
      printCheck(employees[i].getName, employees[i].weeklyPay());
```

Why?

ASK

There is no method called weeklyPay() declared in class Employee, though there is such a method in its subclasses. Since the array employees is declared to be of class Employee, the code

```
employees[i].weeklyPay()
```

will not compile. (The compiler is not aware of a class's subclasses when it compiles code referring to it - and, in general, cannot be aware of its subclasses since new ones can be added at any time.)

d) How might we solve this problem? Note that the type of the array has to be Employee, since individual elements can be of either of the subclasses.

ASK

We could solve this problem by adding a weeklyPay() method to the Employee class. But what should its definition be? As it turns out, it doesn't matter, since we know that it will be overridden in the subclasses. So we could use a dummy implementation like: (PROJECT)

```
public double weeklyPay()
{ return 0; }
```

However, there are all kinds of problems with this - it is confusing to the reader, and if we accidentally did create an object directly of class Employee (which we would be allowed to do), we would get in trouble with the minimum wage laws!

2. To cope with cases like this, Java allows the use of abstract methods and classes.

   a) An abstract method uses the modifier abstract as part of the declaration, and has no implementation - the prototype is followed by a semicolon instead. It serves to declare that a given method will be implemented in *every* subclass of the class in which it is declared.

      Example: We could declare an abstract version of weeklyPay in class Employee as:

      ```
      public abstract double weeklyPay();
      ```

   b) A class that contains one or more abstract methods must itself be declared as an abstract class. (The compiler enforces this):

      (1) Example: (PROJECT)

      ```
      public abstract class Employee
      {
           ...
      ```

      (2) An abstract class cannot be instantiated - e.g. the following would now be flagged as an error by the compiler

      ```
      new Employee(...)
      ```

      This is because an abstract class is incomplete - it has methods that have no implementation, so allowing the creation of an object that is an instance of an abstract class could lead to an attempt to invoke a method that cannot be invoked.

      (3) A class that contains abstract methods must be declared as abstract. The reverse is not necessarily true - a class can be declared as abstract without having any abstract methods. (This is done if it doesn't make sense to create a direct instance of the class.)

      EXAMPLE: The Java awt classes Component and Container are both abstract - it does not make sense to create a Component that is not some particular type of Component (Button, etc.) - ditto a Container. However, the swing JComponent class is not abstract - this allows creation of a JComponent to use as a blank canvas for drawing. (awt has a subclass of Component called Canvas that is used for this purpose.)

c) Note that, in general, an abstract class can contain a mixture of ordinary, fully-defined methods and abstract methods.

EXAMPLE: The Employee class we have used for examples might contain methods like getName(), getSSN(), etc. which are common to all kinds of Employees - saving the need to define each twice, once for HourlyEmployee and once for Salaried Employee.

d) Note that a subclass of an abstract class must either:

(1) Provide definitions for all of the abstract methods of its base class.

or

(2) Itself be declared as abstract, too.

e) Sometimes, a non-abstract class is called a *concrete* class.

C. Suppose we take the notion of an abstract class and push it to its limit - i.e. to the point where *all* of the methods are abstract - none are defined in the class itself. Such a class would specify a set of behaviors, without at all defining how they are to be carried out.

1. In Java, such an entity is called an *interface*, rather than a class.

a) Its declaration begins

[public] interface Name ...

An interface is always abstract; the use of the word abstract in the interface declaration is legal, but discouraged.

b) An interface can extend any number of other interfaces, but *cannot* extend a class.

c) All of the methods of an interface are implicitly abstract and public; none can have an implementation. The explicit use of the modifiers abstract and/or public in declaring the methods is optional, but discouraged

EXAMPLE: Inside the declaration of an interface, the following are equivalent

```
public abstract void foo();   // Discouraged style
public void foo();            // Discouraged style
abstract void foo();              // Discouraged
style
void foo();
```

And the following is illegal:

```
void foo()
{ anything .... }
```

d) Interfaces can also declare static constants. Any variable declared in an interface is implicitly public, static, and final, and must be initialized at the point of declaration. The explicit use of the modifiers public, static, and/or final in declaring a constant is legal, but discouraged.

e) Interfaces *cannot* have:

(1) Constructors

(2) Instance variables

(3) Non-final class variables

(4) Class (static) methods

2. A Java class can implement any number of interfaces by including the clause

   implements Interface [, Interface]...

   in its declaration.

   A class that declares that it implements an interface must declare and implement each of the methods specified by the interface - or must be declared as abstract - in which case its concrete subclasses must implement any omitted method(s).

3. Why does Java have interfaces as a separate and distinct kind of entity from classes?

   a) An interfaces is used when one wants to specify that a class inherits a set of behaviors, without inheriting their implementation.

   b) Interfaces provide a way of dealing with the restriction that a class can extend at most one other class. A class is allowed to extend one class and implement any number of interfaces.

4. Where have we already used interfaces?

   ASK

   The various types of awt listener are interfaces. An object that wants to listen for a particular type of event must declare that it implements the appropriate type of listener. There is no restriction on how many types of events a given object can be a listener for, nor does being a listener interfere with an objects ability to extend some class.

   EXAMPLE:

   A main window that handles action events and window events might be declared as

   ```
   class MyWindow extends JFrame implements ActionListener, WindowListener
   {
   ```

III. Miscellaneous Issues

A. Protected visibility.

1. We have already seen that variables and methods declared within a class may have different visibilities:

   a) private - accessible only to the methods of the class in which they are declared

   b) (default - none specified) - accessible only to the methods of the class in which they were declared, and other classes in the same package

   c) public - accessible to the methods of any class

2. The fourth - and last - visibility specifier is protected, which makes the item accessible to the methods of the class in which it is declared, any class in the same package, and any subclass (regardless of whether or not it is in the same package).

B. The super() constructor.

1. The very first thing that any constructor of a subclass must do is call the constructor of the immediate base class. This is done by using the keyword super, followed by a list of arguments that is appropriate to the

superclass constructor.

EXAMPLE: Suppose we have a class HourlyEmployee that extends Employee. Suppose further that the Employee constructor requires parameters name and ssn. Then an HourlyEmployee constructor might look like this: (PROJECT)

```
public HourlyEmployee(String name, String ssn, double hourlyRate)
{
   super(name, ssn);
   this.hourlyRate = hourlyRate;
}
```

2. The explicit call to the superclass constructor can be omitted if and only if the base class has a constructor that takes no arguments. In this case, the compiler automatically inserts

   super();

   as the first statement of the constructor.

C. Use of super to access overridden methods

We have already seen that, when a subclass overrides a method of its base class, it can access the original method via

super.methodName(parameters);

D. The final modifier on methods

1. When a class is going to be extended, it may be that some of its methods should not be subject to being overridden. In this case, they can be declared as final.

   EXAMPLE: If the class Employee has a getName() method for accessing the employee's name that cannot meaningfully be overridden, the method could be declared as

```
public final String getName()
{
    return name;
}
```

2. Declaring a method as final when it cannot be overridden allows the compiler to perform some optimizations that may result in more efficient code, so adding final to a method declaration where appropriate is

worthwhile.

E. The Final Modifier on classes

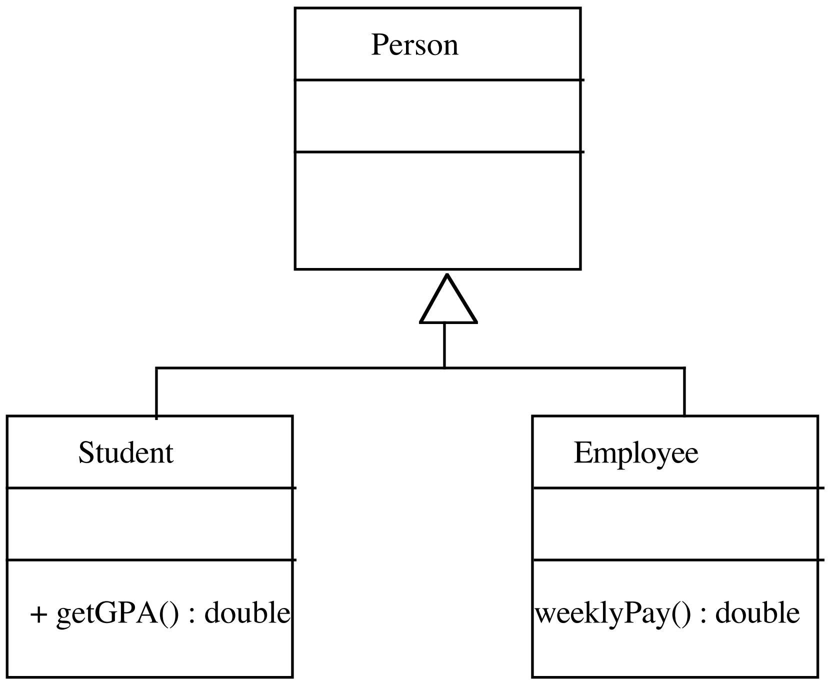
1. Just as an individual method can be declared final, so an entire class can be declared final. (E.g. public final class ...).

2. A final class cannot be extended. This serves to prevent unwanted extensions to a class - e.g. the class java.lang.System is final.

F. Multiple inheritance.

1. We have talked about a lot of things that Java *can* do. We now must consider one capability present in many OO languages that Java does *not* support: multiple inheritance.

2. Sometimes, it is meaningful for a given class to have two (or more) direct base classes. A classic example of this is a system for maintaining information about students at a college or university, which might have the following structure:



Now suppose we wanted to add a new class TA. Such a class would logically inherit from *both* Student and Employee, since a TA is both, and since the methods getGPA and weeklyPay are both applicable. Many OO languages would allow this - Java does not.

3. Multiple inheritance is actually something of a controversial feature in OO. Allowing it introduces all kinds of subtleties. To cite just one example - if we did have TA inherit from both Student and Employee, then a TA is a Person in two different ways. Does this mean that there are two copies of the Person information stored - one for TA as Student and one for TA as Employee? It turns out that dealing with issues like this is non-trivial - one reason why Java opted to not allow multiple inheritance.

4. Note that, although a Java class cannot inherit *implementation* from more than one class, it can inherit *behavior* from more than one class, by means of interfaces. (One reason for including interfaces as a separate construct in Java was to allow this sort of limited multiple inheritance.)

IV. Summary

HANDOUT/DEMO Employee program - note:

A. Abstract class - Employee - and method weeklyPay()

B. final methods - getName(), getSSN() in Employee

C. Call to super() constructor in constructors for HourlyEmployee and SalariedEmployee

D. Overrides of toString() in HourlyEmployee and SalariedEmployee, with explicit use of superclass version in implementation

E. To Demo: run class EmployeeTester.