CPS122 Lecture: Encapsulation, Inheritance, and Polymorphism

Last revised January 17, 2023

Objectives:

1. To review the basic concept of inheritance
2. To introduce Polymorphism.
3. To introduce the notions of abstract methods, abstract classes, and interfaces.
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. Projectable of person as employee or student hierarchy class diagram and
2. Project able of Java code for this
3. Projectable of hierarchy with faculty and staff subclasses of employee
4. Projectable of Java code for this
5. Venn diagram showing law of substitution
6. Projectable of basic BankAccount hierarchy
7. Demo and handout of BankAccount hierarchy
8. Projectable of BankAccount hierarchy with HighBalanceSavingsAccount
9. Netbeans OverrideDemo project
10. Projectable of employee class where all employees are paid hourly
11. Projectable of SalariedEmployee as override of Employee paid hourly
12. Employees demo program - Handout and online demo, projectable versions of code snippets
13. Projectable of stages in development of Employee hierarchy
14. Projectable of equivalent ways of declaring a method in an interface
15. Projectable of illegal implementation of a method of an interface
16. Projectable of final on method
17. Projectables of Robot hierarchy in book and different behaviors of each, plus implementation of Butler (Figures 4.29, 4.30 and Table 4.4)
18. Projectable of multiple inheritance examples
19. Projectable of repeated inheritance problem
20. Projectable of containment as an alternative to multiple inheritance

I. Introduction

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 (sometimes called the "OO Pie").

1. Polymorphism

2. Inheritance

3. Encapsulation

   (We'll actually talk about these in reverse order!)

B. Although we have not used the term per se, we have already made use of 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 in a controlled manner, 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. In this series of lectures, we will focus on inheritance and polymorphism.

II. **Inheritance**

A. One of the main uses of inheritance is to model hierarchical structures that exist in the world.

Example: Consider people at Gordon college. Broadly speaking, they fall into two categories: employees and students.

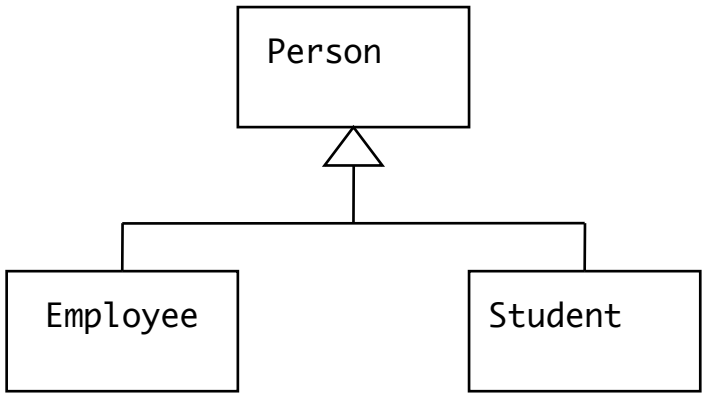
(We will not, for now, consider the possibility that a person might be both a student and an employee. We will assume a person is either one or the other.)

1. There are some features that both employees and students have in common - whether a person is an employee or a student, he or she has a name, address, date of birth, etc.

2. There are also some features that are unique to each kind of person - e.g. an employee has a pay rate, but a student does not; a student has a gpa, but an employee does not, etc.

3. How can we represent this hierarchy as a class structure?

ASK



PROJECT

4. How would we represent this in Java?

   ASK

```
class Person
{
    ...
}

class Employee extends Person
{
    ...
}

class Student extends Person
{
    ...
}
```

   PROJECT

5. With this structure, the classes Employee and Student inherit all the features of the class Person.

6. In addition, each of the classes Employee and Student can have features of its own not shared with the other classes.

B. Basic terminology: If a class B inherits from a class A:

1. We say that B *extends* A or B is a *subclass* of A - So we say Employee extends Person, or Employee is a subclass of Person.

   The term subclass comes from the mathematical notion of subset - the set of all Employees is a subset of the set of all Persons.

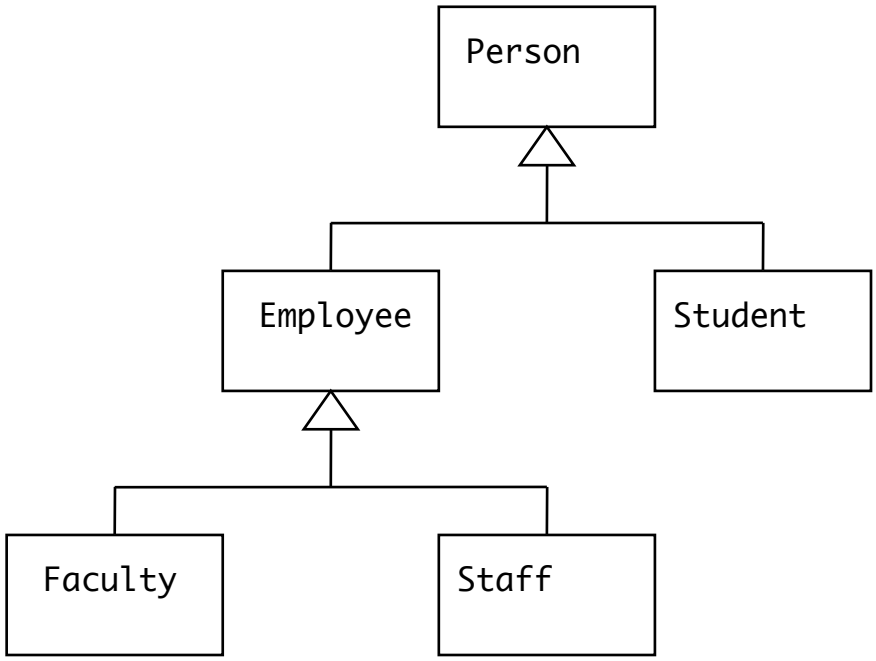
2. We say that A is the *base class* of B or the *superclass* of B - So we say Person is the base class of Employee, or the superclass of Employee.

   The term superclass comes from the mathematical notion of sets as well - the set of a Persons is a superset of the set of all Persons.

3. 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.

Example: suppose we had the following hierarchy



PROJECT

Now we could say that Faculty is direct subclass of Employee, and an indirect subclass of Person, etc.

4. This could be represented in Java by adding the following two classes to the hierarchy we developed earlier.

```
class Faculty extends Employee
{

...
}

class Staff extends Employee
{

...
}
```

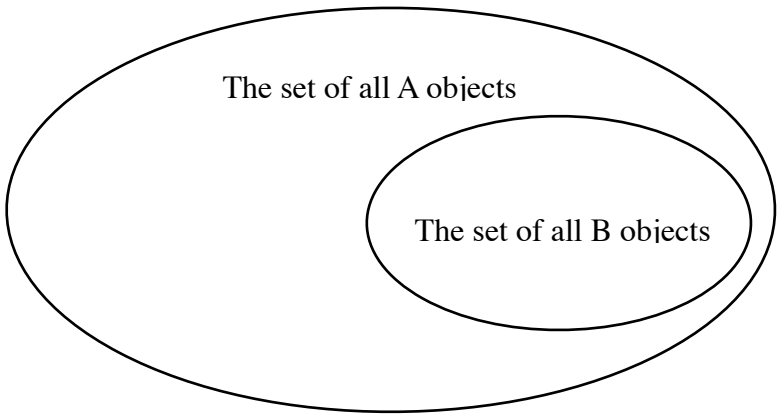
PROJECT

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

1. 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.

   Thus, in the above example, the inheritance structure says that an Employee can always be used anywhere that a Person is needed.

2. 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"

   PROJECT

3. This relationship is sometimes expressed by using the phrase "is a" - we say a B "is a" A.

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

   a) 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.

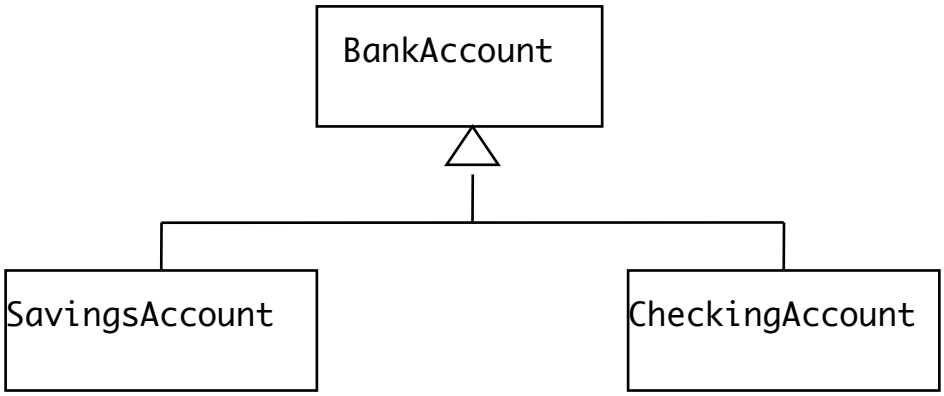
b) Example: suppose we were building a software model of the human body, and we wanted to create a class Person to model a whole person, and a class Arm to model a person's arms. The correct relationship between Arm and Person is a "has a" relationship - a Person "has a" Arm (actually two of them), not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

5. We've seen how inheritance can be used for generalization - e.g. the class Person generalizes the notion of Student and Employee.

6. As used in the OO world, inheritance can also be used for specialization - e.g. in a graphics system we may have a class Square that is a subclass of the class Rectangle - meaning that Square is a specialization of Rectangle, being a Rectangle with equal sides. This is consistent with the law of substitution - anywhere a rectangle is needed, a square can be used.

   Of course, this is a different concept from the way we speak of inheritance in terms of human relationships. For example, I inherited my mother's hair color - but that does mean that I'm a specialization of my mother!

D. A key aspect of inheritance is that a subclass ordinarily <u>inherits</u> all the features of its base class. For example, consider the following example of a class hierarchy for bank accounts (similar to the example we looked at earlier, but modified to incorporate two different kinds of bank account- checking and savings - with a common base class, and with some other changes as well.)



PROJECT

A Java implementation this hierarchy might look like the following:

HANDOUT

Observe the following:

1. The classes SavingsAccount and CheckingAccount inherit the features of BankAccount

   a) Since a BankAccount has an owner and a balance, so does a SavingsAccount or a CheckingAccount.

   b) Since a BankAccount has methods deposit(), reportBalance(), and getAccountNumber(), so does a SavingsAccount or a CheckingAccount.

2. The constructors for SavingsAccount and CheckingAccount must invoke the constructor for BankAccount "passing up" the owner. This is done via super(owner) at the start of each.

3. Savings account adds features that an ordinary BankAccount does not have - e.g. payInterest() and setInterestRate().

4. CheckingAccount overrides the withdraw() method of BankAccount.

a) In the special case where the checking account balance is insufficient for the withdrawal, but the customer has a savings account with enough money in it, the withdrawal is made from savings instead.

b) In all other cases, the inherited behavior is used by invoking super.withdraw(amount).

5. Certain features of BankAccount are declared protected (rather than private). This specifies that the subclasses may access them, though other classes may not.

   a) Note how the payInterest() method of SavingsAccount needs to make use of the inherited feature current balance, and the withdraw() override in CheckingAccount0 needs to make use of both the inherited features currentBalance and owner.

   b) On the other hand, accountNumber remains private in BankAccount, which precludes the subclasses from using it.

E. In designing a class hierarchy, methods should be placed at the appropriate level. For example, in the BankAccountExample:

1. deposit(), reportBalance(), and getAccountNumber() are defined in the base class BankAccount, and so are inherited by the two subclasses.

   If they were defined in the subclasses, we would have to repeat the code twice - extra work and an invitation to inconsistency should we need to make modifications.

2. On the other hand, payInterest() and setInterestRate() are defined in SavingsAccount, because they are not relevant for CheckingAccounts.

3. withdraw() is defined in BankAccount and overridden in CheckingAccount. Why is this better than simply defining separate versions in CheckingAccount and SavingsAcccount?

   ASK

   Although CheckingAccount does override the inherited method, it does make use of it in most cases via the super.withdraw() call. This would not be possible if separate versions were defined in CheckingAccount and SavingsAccount, with no base version in BankAccount.

III. **Polymorphism**

A. The above example also illustrates polymorphism, which we now want to define more formally. 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.

B. Example: Continuing with our BankAccount example

   REFER TO HANDOUT

   1. Suppose we declared a variable as follows:

      BankAccount account;

   2. We could now make it refer to either a CheckingAccount or a SavingsAccount - i.e. (assuming a Customer variable named aardvark exists) either of the following would be legitimate:

      account = new CheckingAccount(aardvark);
      or
      account = new SavingsAccount(aardvark);

   3. If, however, we tried to perform

      account.withdraw(*some amount*);

with an amount that exceeds the balance, the way it would handle the operation would depend on its actual type

a) If it were actually a SavingsAccount, it would reject the operation in all cases

b) If it were actually a CheckingAccount, it would see if its owner had a savings account with sufficient balance.

C. Another example: Show Netbeans project OverrideDemo. Note class declarations for A, B, and C - B extends A and C extends B and therefore A.

D.

1. Show part initially not commented out. Ask class what output should be and then run. Explain as necessary.

2. Repeat for b-g sections by removing /* at start. Handle parts that are illegal by using // at start of line. Explain as necessary.

   Note that some things are rejected by the compiler on the basis of the *declared* types of the variables.

3. We saw earlier that a consequence of inheritance 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. Work through saySomething and speak sections.

   The fact that the overridden method may be used in place of the base class method depending on the actual type of the object is called dynamic binding or dynamic method invocation.

4.

   (BTW: Not all programming languages handle this the same way. For example, in C++ dynamic binding is only used if you explicitly ask for it)

   DEMO using saySomething() peak() methods

5. 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 the saySomething() method defined as `saySomething(short i)`, instead of having a parameter is of type `int`..

```
class C extends B
{
   public void saySomething(short i)
   { System.out.println(-1); }
}
```

   Now what would I get if I tried someC.saySomething(1)?
   What will the output be?

   ASK

   DEMO

   What I actually have in this case is an overload rather than an override, If I wanted to get the C method, I would have to explicitly use a short someC`.saySomething((short) -1)`

   DEMO

6. As we have already seen, 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>)

   DEMO using the speak2() methods

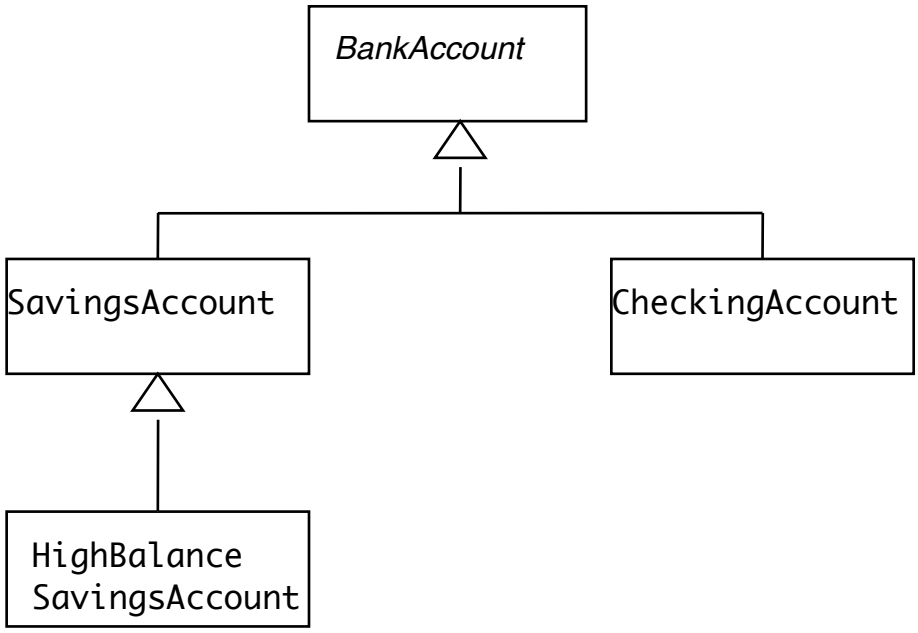
IV. **Abstract Methods, Abstract Classes, and Interfaces**

A. Returning again to our BankAccount example, would it be meaningful - in this case - to have a BankAccount object that is *neither* a CheckingAccount nor a SavingsAccount?

ASK

1. In a case like this, we can declare the base class BankAccount to be abstract. (Note in code). An abstract class cannot have an object that belongs to it, but not to one of its subclasses, which is what we desire in this case.

2. It is not, however, always the case that a base class should be abstract. Suppose our bank created a new kind of savings account called a HighBalanceSavingsAccount which has a minimum balance of $10,000 but pays a higher interest rate. We might picture this as follows:



PROJECT

In this case, though BankAccount would be an abstract class, SavingsAccount would not - it is meaningful to have a SavingsAccount that is not a HighBalanceSavingsAccount.

B. There are other issues involved in creating an abstract class as well.

For 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.

1. 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*.

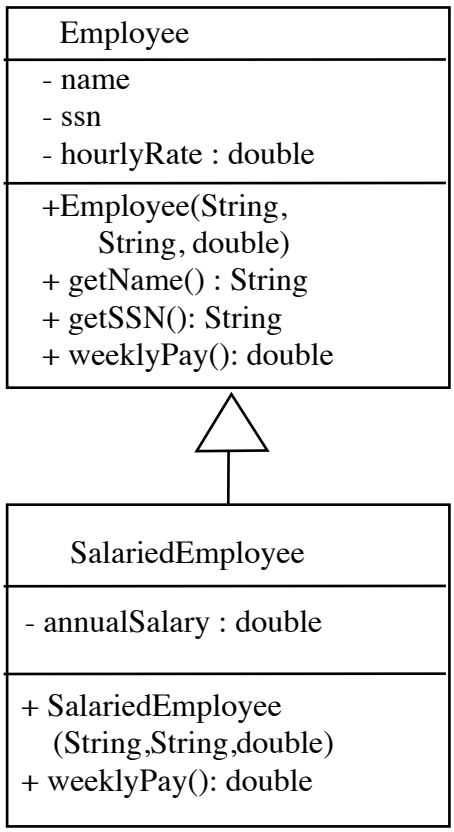
2. But this is not a good solution. Why?

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 UML Class diagram:



PROJECT

(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.

(3)Each field and method is preceded by a visibility specifier. 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. 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.

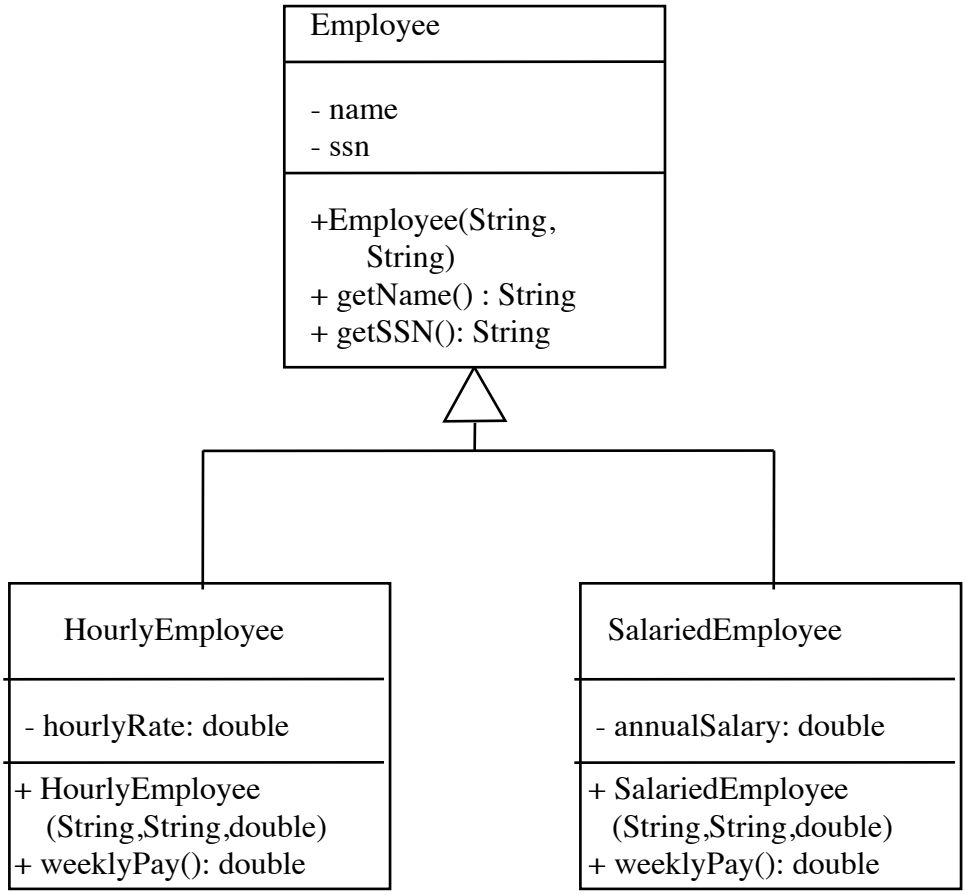
(5)Each method has type specifiers for its parameters (its signature).
A subclass includes all the fields of its superclass (though they may not be accessible in the subclass if they are declared private). Thus, a SalariedEmployee object has four fields - name, ssn, and hourly rate (inherited from Employee) and annualSalary (defined in the class)

b) In this case, each object that belongs to SalariedEmployee has an hourlyRate field, which is not meaningful.

c) 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:



PROJECT

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 an hourly 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.

d) 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.)

e) 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 problems with this

ASK WHY

(1)t is confusing to the reader

(2) 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!

3. To cope with cases like this, Java allows the use of abstract methods.

   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(...)   (PROJECT)
      ```

      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.)

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.

4. Distribute, go over, handout of Employee class hierarchy.

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) Polymorphic calls to weeklyPay()

f) Demo: run class EmployeeTester.

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();
```

PROJECT

And the following is illegal:

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

PROJECT

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 potential 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.

D. The book develops a more complex example of this - a hierarchy of Robots

1. Project hierarchy (Figure 4-29 on page 104) and various ways of implementing perform() (Table 4-4 on page 105)

2. Questions about this (do in groups)

a) Which classes should be abstract?

Robot, Humanoid, DomesticRobot, and AlienRobot

b) Does perform() actually need to be defined as an abstract method in all of them? (The class diagram implies that it is - is this really necessary)

No - it only needs to be defined in the top-level class - Robot - and the bottom level classes. The other three abstract classes do not need to include declarations for perform(), since it is inherited from the base class.

c) The book gives an example of code for ButlerRobot

PROJECT Figure 4.30

The code given sets the instance variables of the superclass directly. Let's rewrite it using the super constructor, and assuming DomesticRobot has constructor

```
DomesticRobot(String meansOfMobility, String language)
```

V. The Use and Misuse of Inheritance

A. Inheritance can be a very powerful and useful tool, saving a great deal of redundant effort.

1. Unfortunately, inheritance can be - and often is - misused. So we will want to consider both how *to* use inheritance and how *not to* use it.

2. A cardinal rule for using inheritance well is *the rule of substitution*.

   ASK

   If a class B inherits from a class A, then it must be legitimate to use a B anywhere an A is expected. That is, it must be legitimately possible to say "a B isa A".

B. Actually, there are a variety of reasons for using inheritance in the design of a software system - including some not so good ones! One writer, Bertrand Meyer, has written a classic article in which he identified twelve! Some of the uses identified in Meyer's article are fairly sophisticated. I will draw on his work here, but in simplified form. Broadly speaking, Meyer classifies places where inheritance can be used as:

1. *Model inheritance* - when the inheritance structure in the software mirrors a hierarchical classification structure in the reality being modeled by the software.

   a) One key feature of human knowledge is that many fields of learning have classification systems:

      (1)The taxonomic system of biology

      (2)The Dewey Decimal and Library of Congress systems used in libraries.

      (3)Other examples?

      ASK

b) When the reality we are working with has such a natural hierarchy, we may want to reflect that hierarchy in our software. However, Meyer warns about what he calls "taxomania" - the tendency to go overboard with classification hierarchies in software. In particular, there is a danger of creating too many levels in a hierarchy, without enough distinctions between classes at a level.

c) In general, we want to reflect a natural hierarchy in our software if the different objects we are working with fall into classes that have enough significant differences in attributes and behavior to make classification worthwhile.

EXAMPLE: In the library problem, the items the library checks out can be categorized as book and DVD. These probably have enough distinctions to warrant two classes inheriting from a common "Item" base class, because the information we need to store about each is different, and their behaviors are a bit different

(1) Books: store call number, title, author. When checked out, a book can be renewed.

(2) DVD: store call number, description, lead actor. Cannot be renewed.

2. A second broad type of inheritance is what Meyer calls *software inheritance*. Here, the inheritance structure reflects a hierarchy that does not exist in the reality being modeled, but is useful nonetheless in the software.

a) Actually, as it turns out, what Meyer calls software inheritance shows up in UML models in two places - here, and under realization. We'll discuss the latter case later.

b) One common motivation for this sort of inheritance is to facilitate *polymorphism*. Suppose we want to create a collection class whose

elements are to be various sorts of objects - e.g. perhaps a home inventory that lists the different items found in our home (useful information in case of a fire or theft.) In order to place these different items in the same polymorphic container, they would need to all derive from a common base class, which is the class of things the collection actually stores. (E.g. in this case, we might create a class HomeAsset and make things like furniture, books, artwork, electronic equipment etc. inherit from it.)

NOTE: In this case, the common base class will most likely be abstract.

EXAMPLE: The Transaction class hierarchy in the ATM system can be regarded as an example of this. The class Session needs to be able to refer polymorphically to different types of Transaction, which are made subclasses of a common abstract base class.

c) Another motivation for using software inheritance is to reuse work already done. When we are designing a new class, it is worth asking the question "is there any already existing class that does most of what this class needs to do, which I can extend?"

(1) However, we need to proceed cautiously when we do this, because this kind of inheritance can easily be abused. When extending an existing class to create a new class, we should ask questions like:

(a) Is the law of substitution satisfied?

If the law of substitution is not satisfied, then we are almost certainly abusing inheritance.

(b) Are we mostly adding new attributes and methods to the existing class, or changing existing methods to do something entirely different? In the latter case, we are

likely abusing inheritance - extension means "adding to" an existing set of capabilities.

(c) Are all (or at least most) of the existing methods of the base class relevant to the new class? If not, it is again likely that we are abusing inheritance.

(2) Note that, in cases like this, we generally do not have to create the base class - instead, we use an existing class to help create a new one.

(a) This is most likely to happen in cases where the base class has been designed from the beginning to facilitate extension. (I.e. we usually consider extending classes whose initial designer created them with the intention that they be extended. Frameworks are often designed this way)

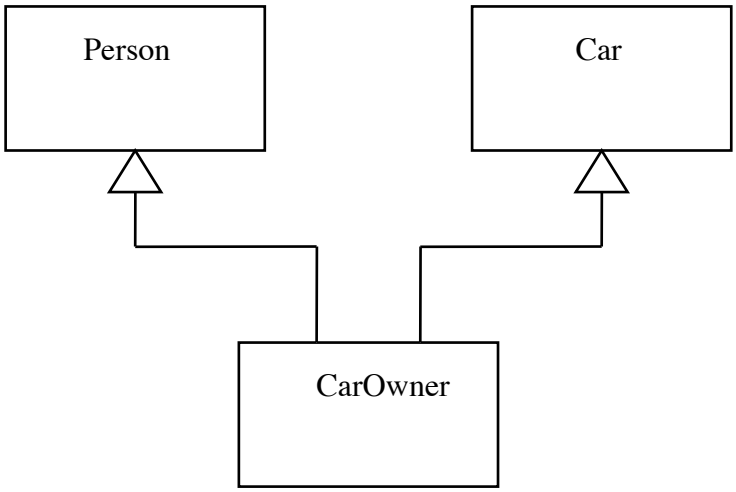
(b) A related idea is that, where appropriate, we should try to design our classes in such a way as to facilitate later extension in other applications. This may mean making a class more general than in needs to be for a specific application, in order to facilitate later reuse.

3. A third broad type of inheritance Meyer identifies is called *variation inheritance*. Here, a class B inherits from a class A because it represents some sort of variation of A. Meyer describes this sort of inheritance this way: "Variation inheritance is applicable when an existing class A, describing a certain abstraction, is already useful by itself, but you discover the need to represent a similar though not identical abstraction, which essentially has the same features, but with different signatures or implementations." (p. 829)

   We will not discuss this type of inheritance further; its applications are a bit more sophisticated than what we're dealing with here.

C. A danger particularly with both software inheritance and variation inheritance (but less so with model inheritance) is letting apparent

convenience lead to misuse of inheritance. For example, Meyer cites a well-known software engineering text that develops the following scenario, using multiple inheritance:

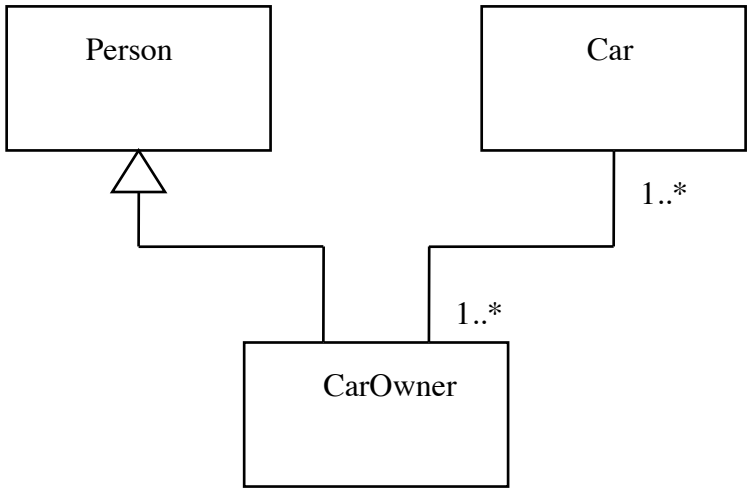


PROJECT

Clearly, having CarOwner inherit from Person makes sense - a car owner is a person - but making CarOwner inherit from Car is another story! The justification is that Car has attributes like registration number and excise taxes due that legitimately apply to a CarOwner as well - but we don't want to saddle a CarOwner with having to have a carburetor, four tires, and brakes!

1. This example, and others like it, typically fail the fundamental law of substitution test. A CarOwner simply cannot be substituted for a car. (Try spending some time in a car wash!)

2. The mistake that is often made is confusing the "has a" relationship (association) with the "isa" relationship (inheritance). A correct way to represent the structure of the problem would be to use inheritance in one case, and association in the other:



PROJECT

(By the way, note that doing it this way lets us allow for the possibility that an owner might have several cars, and that a car might have joint owners.)

D. In Java, inheritance is specified by using the keyword *extends*.

1. The class being extended may be either abstract or concrete.

2. As you know, Java allows a class to only extend one other class - i.e. it does not support multiple inheritance - something which many OO languages do support - but which introduces some interesting complexities we won't get into now.

VI. **Miscellaneous Issues**

A. 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;
}
```

PROJECT

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.

B. 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 ...).

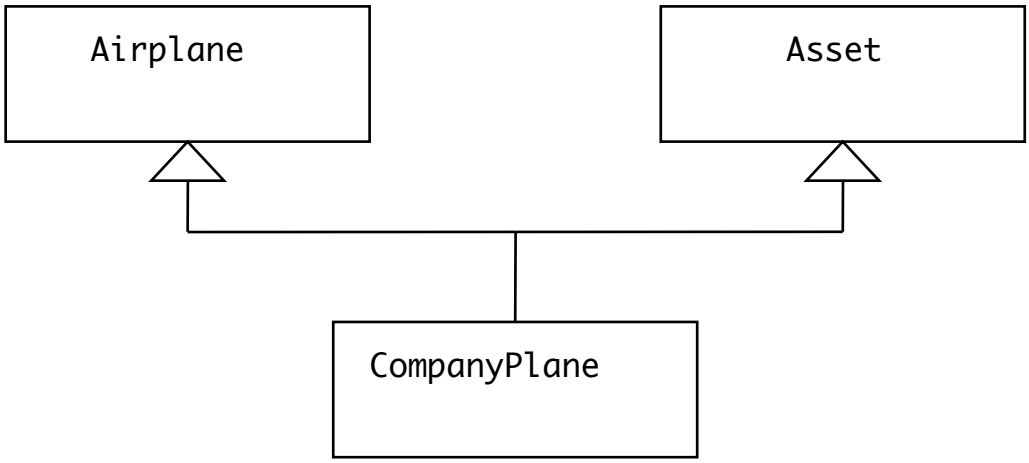
2. A final class cannot be extended. This serves to prevent unwanted extensions to a class - e.g. the class java.lang.System is final.

C. 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 makes sense for a single class to generalize two (or more) bases classes. We call such a situation *multiple inheritance*.

   a) The following example is given by Meyer:



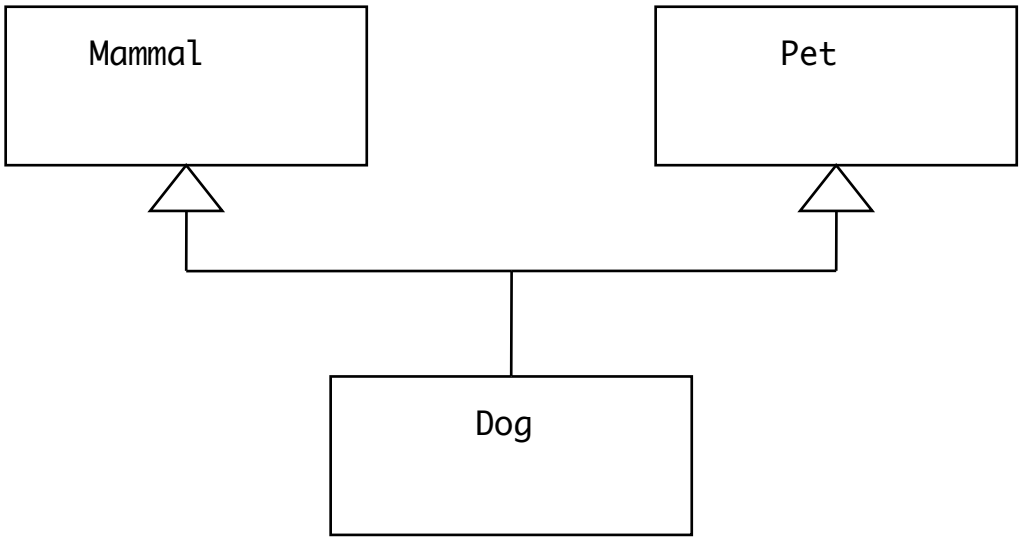
PROJECT

(1)An airplane that is owned by a corporation (a company plane) is, at the same time, both an airplane and a company asset (in terms of bookkeeping)

(2)As an airplane, it has properties like manufacturer, model, range, capacity, runway length needed, etc.

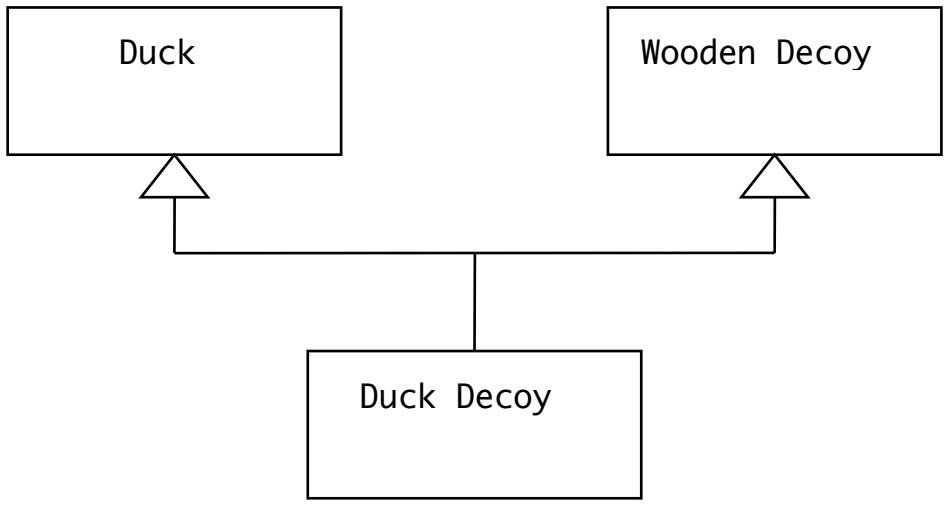
(3)As an asset, it has properties like cost, depreciation rate, current value, book value etc.

b) Here's another example:



PROJECT

c) However, multiple inheritance is easily misused. It is easy to create questionable (or obviously bad) examples. For example, the following is sometimes cited as an example of a place where multiple inheritance is useful, but is really a fairly bad example:



PROJECT

3. Multiple inheritance can give rise to some interesting problems. We will consider two - there are others.

a) Features with the same name in two different base classes.

Example: The company plane example. Suppose that the class airplane had a field called rate (meaning speed), and the class asset had a field called rate (meaning depreciation rate.) If we declared

CompanyPlane p;

what would p.rate mean?

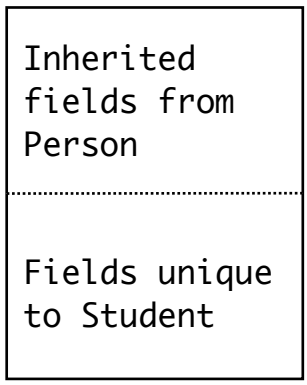
(Arguably, this might not happen in this particular case - but it could. If it did, we could avoid it by changing the name of the field in one of the base classes - if we had access to the source, and if we could then change all the uses of the old name in other software that used this class - a nontrivial task.)

b) Repeated inheritance.

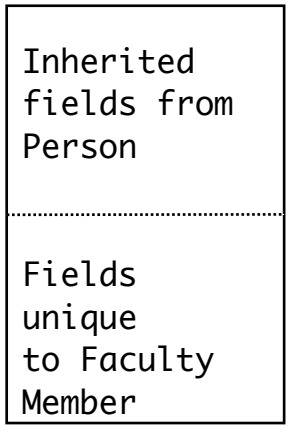
Example: Consider the following situation, which could arise if multiple inheritance is used. (Perhaps in a research university) - and how the objects in question would need to be laid out in memory.

PROJECT

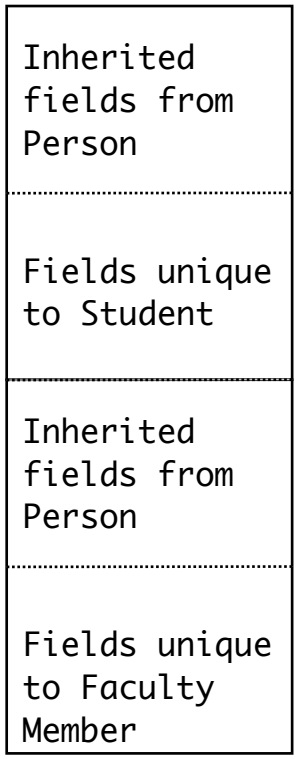
(1)Student



(2)FacultyMember



(3).∴ GraduateStudentInstructor



PROJECT

Note that the straightforward layout of a TA object contains two copies of the Person fields - leading to all sorts of potential ambiguities.

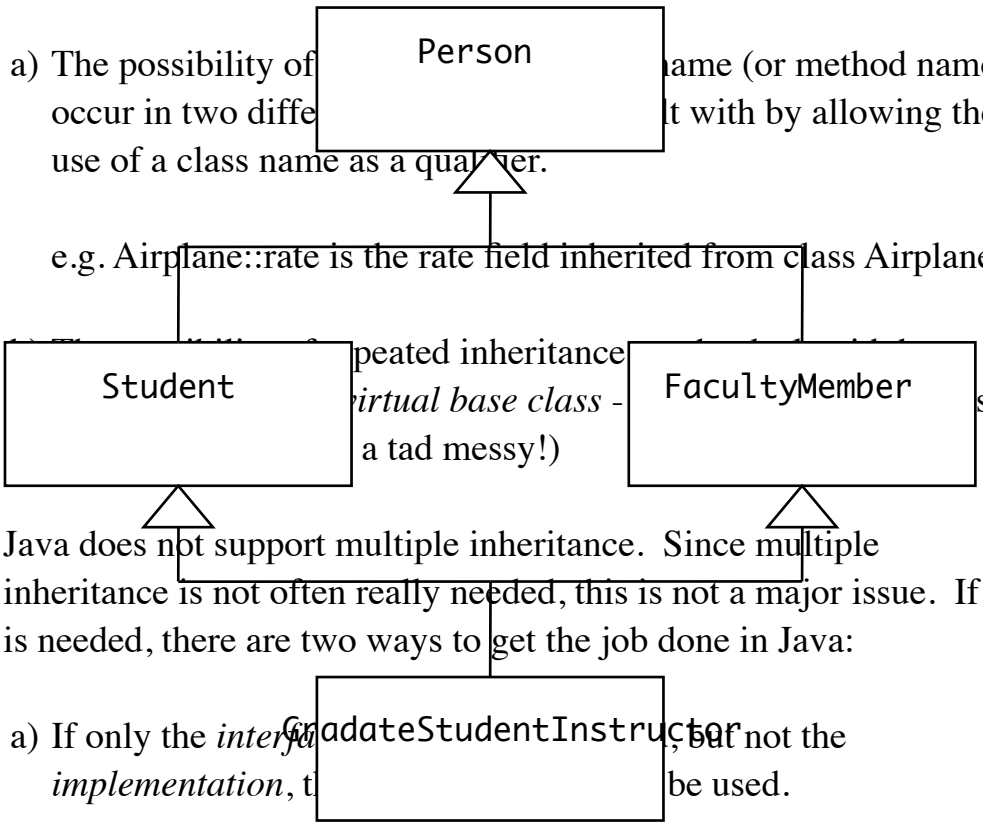
4. Programming languages that support multiple inheritance have to deal with these complexities in some way.

   EXAMPLE: C++

   a) The possibility of [illegible]ame (or method name) occur in two diffe[illegible]lt with by allowing the use of a class name as a qual[illegible]er.

      e.g. Airplane::rate is the rate field inherited from class Airplane.

   [illegible] peated inheritance [illegible]
   irtual base class - [illegible]s!
   a tad messy!)



5. Java does not support multiple inheritance. Since multiple inheritance is not often really needed, this is not a major issue. If it is needed, there are two ways to get the job done in Java:

   a) If only the *interfa[illegible]*, but not the *implementation*, t[illegible] be used.

      (1)A Java class can implement any number of interfaces

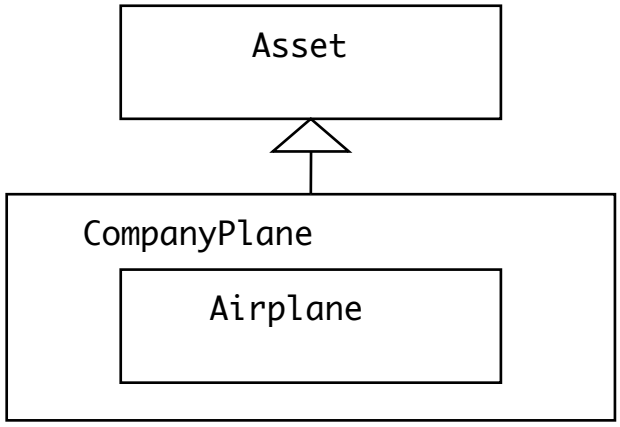
      Example - the main frame in a GUI

```
class ___________ extends Frame
    implements ActionListener, WindowEventListener
{
    ...
```

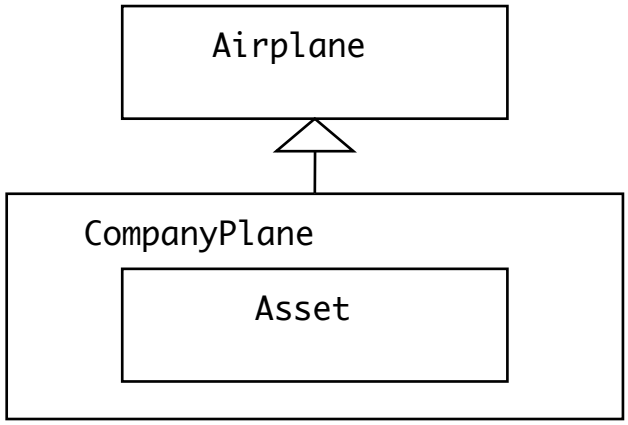
   b) We can use *containment*.

      Example: the CompanyPlane class in Java

(1)implement as



(or)



(2)Then use “forwarding” of methods - example (first case)

```
class CompanyPlane extends Asset
{
    Airplane myInnerPlane;
    public int getCapacity()
    {
          return myInnerPlane.getCapacity();
    }
...
```

PROJECT

VII.Activity

Do Exercise 4.1 in book - break class into groups of 3-4, and assign board space to each.