CPS122 Lecture: Encapsulation, Inheritance, and Polymorphism

Last revised January 26, 2012

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. Demo and handout of BankAccount hierarchy
2. Dr. Java for demos + file OverrideDemo.java
3. Employees demo program - Handout and online demo, projectable versions of code snippets

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, 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**

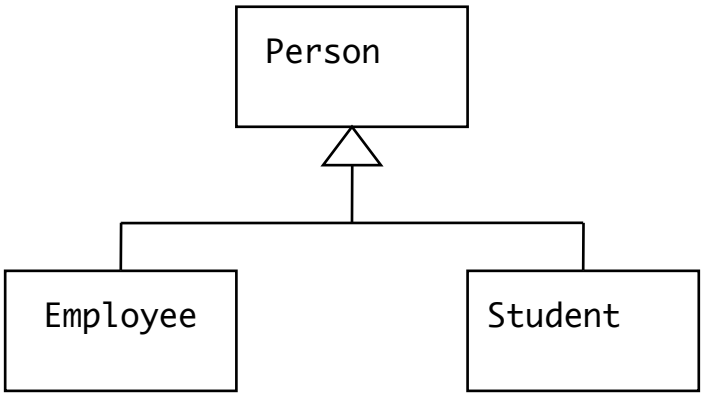
A. One of the main uses of inheritance is to model hierarchical structures that exist in the world.

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

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. We can represent this hierarchical structure this way:



B. In Java, inheritance is specified by the reserved word `extends`. For example, we could declare classes Person, Employee, and Student as follows:

```
class Person {
     ...
}

class Employee extends Person {
     ...
}

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

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

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

C. 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.

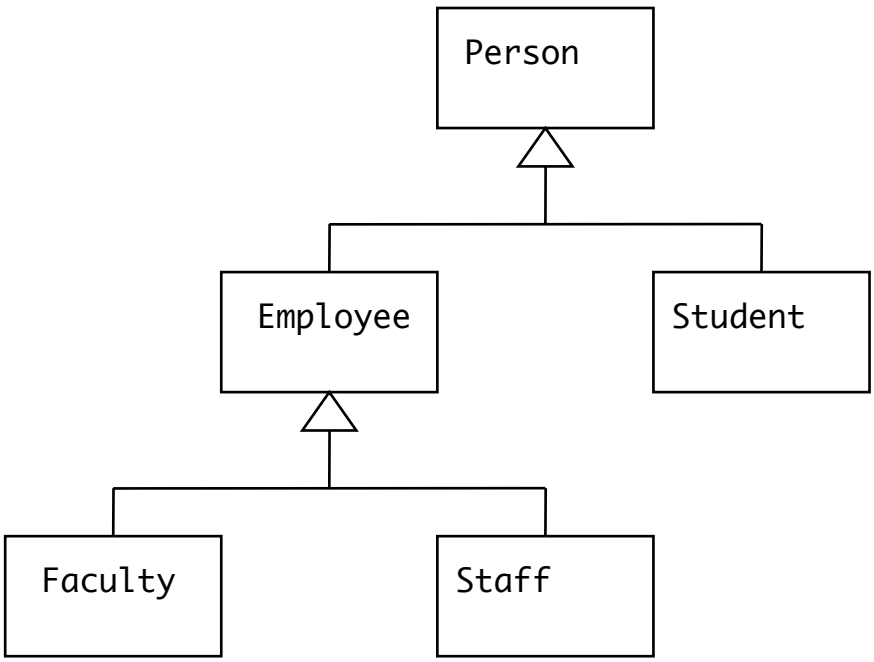
   The term subclass comes from the mathematical notion of subset - the set of all Employees is a subset of the set of all Persons.

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



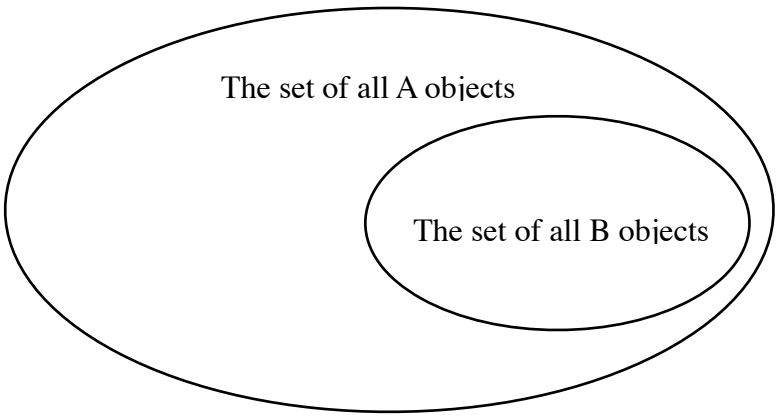
   Now we could say that Faculty is direct subclass of Employee, and an indirect subclass of Person, etc.

D. 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"

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.

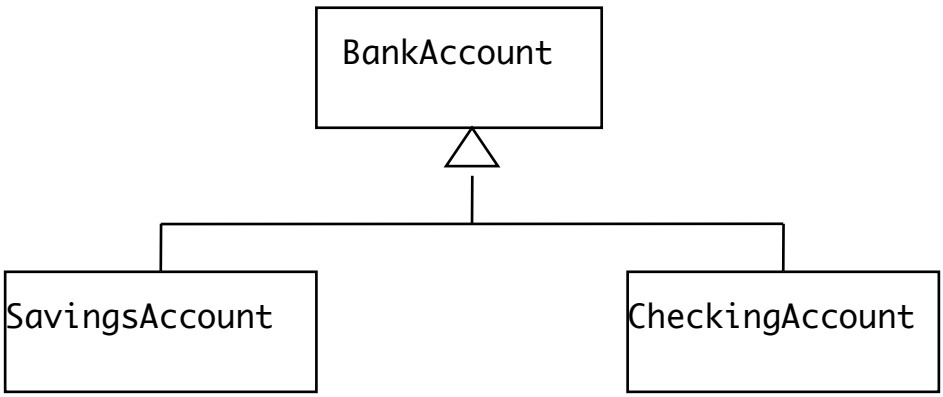
a) The “is a” relationship is similar to another relationship called the containment relationship, or “has a”. Sometimes inheritance is incorrectly used where containment should be used instead.

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. 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!

E. 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.)



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.

F. 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

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: 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); }
}
```

1. Load OverrideDemo.java, compile, then 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 is *not* legal:

```
someB = new A();  // Illegal!
```

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. Let's look at what this distinction means

   a) 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.

      DEMO

   b) 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.

      DEMO

   c) By the way - in both cases the test

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

      would succeed, because a B is an A.

      DEMO

d) likewise, if we did

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

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

DEMO

3. We saw earlier that 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.

4. Recall that, 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*.

   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. E.g., in the previous example the declared type of someA was A, but the actual type was B, so when the saySomething() method was called, the B version was used.

   (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)

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 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,

   a) Now suppose I write

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

      What will the output be?

      ASK

      -1

      DEMO

   b) However, I would get the other method method (hence output of 42) if I used `new C().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>)

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 of the method as

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

DEMO

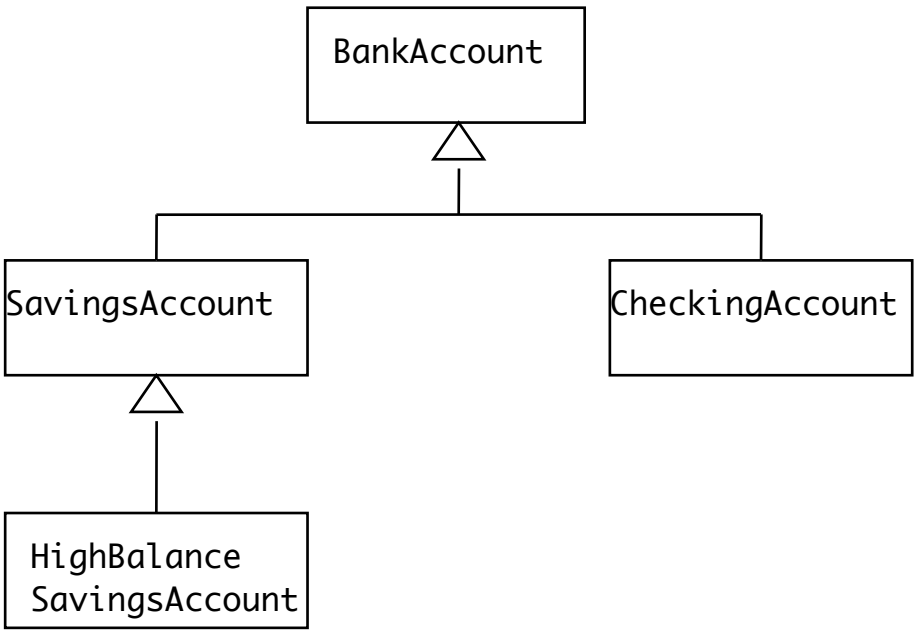
IV. Abstract Methods, Abstract Classes, and Interfaces

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:



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());
```

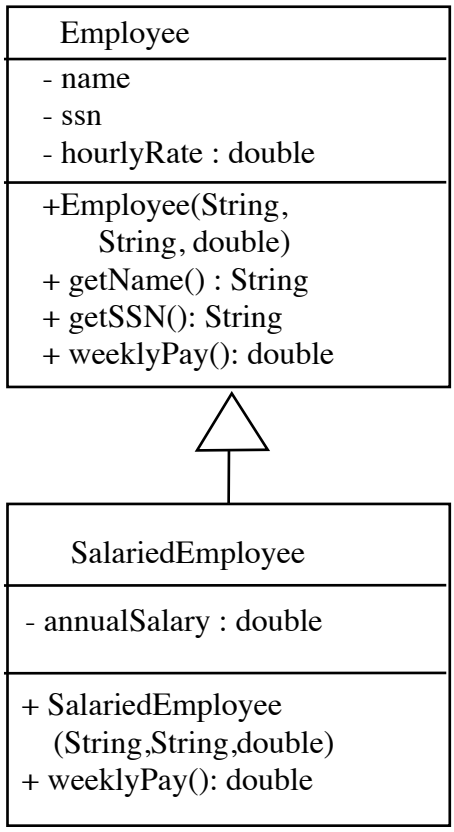
Note that, in each case, the appropriate version of weeklyPay() is called - e.g. for Big Boss, the SalariedEmployee version is called and a check for 1923.08 is printed; for Lowly Peon a dialog is popped up asking for hours worked and the appropriate amount is calculated based on a rate of 4.75 per hour.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



(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 <u>not</u> 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.

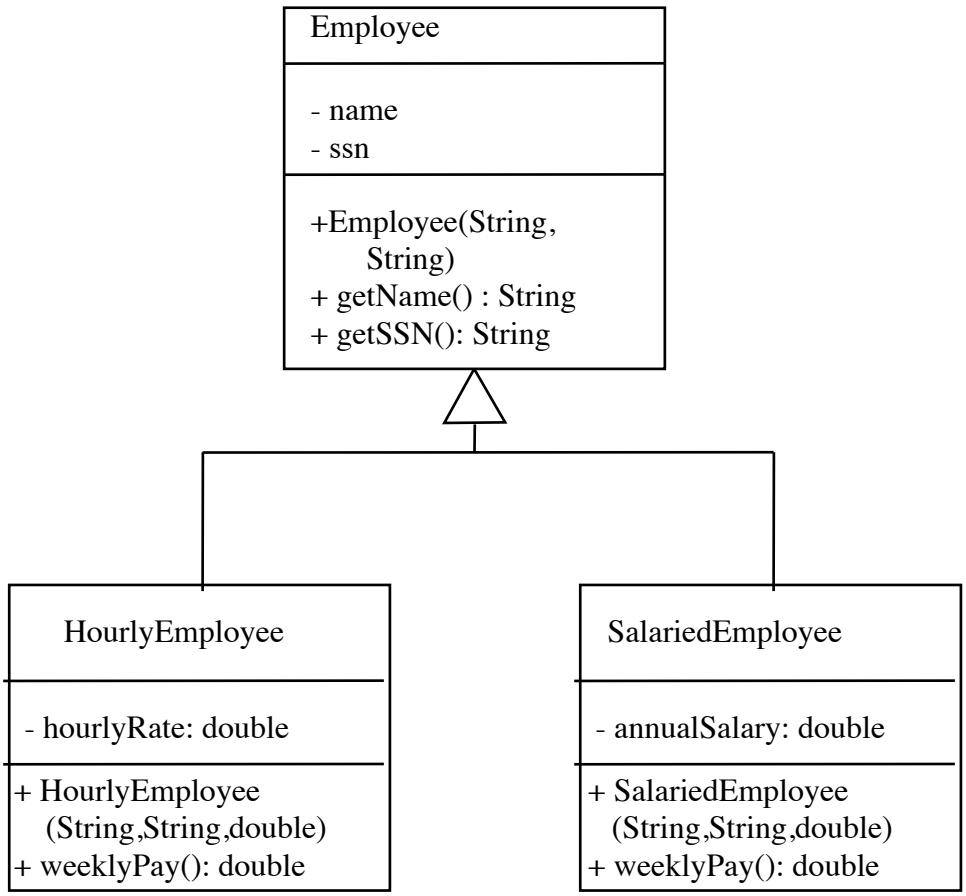
(5) Each method has type specifiers for its parameters (its signature).
A subclass includes all the fields of its superclass (though they may not be accessible in the subclass if they are declared private). Thus, a SalariedEmployee object has four fields - name, ssn, and hourly rate (inherited from Employee)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



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 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!

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(...)
```

   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();
```

      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 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.

V. **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;
   }
   ```

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

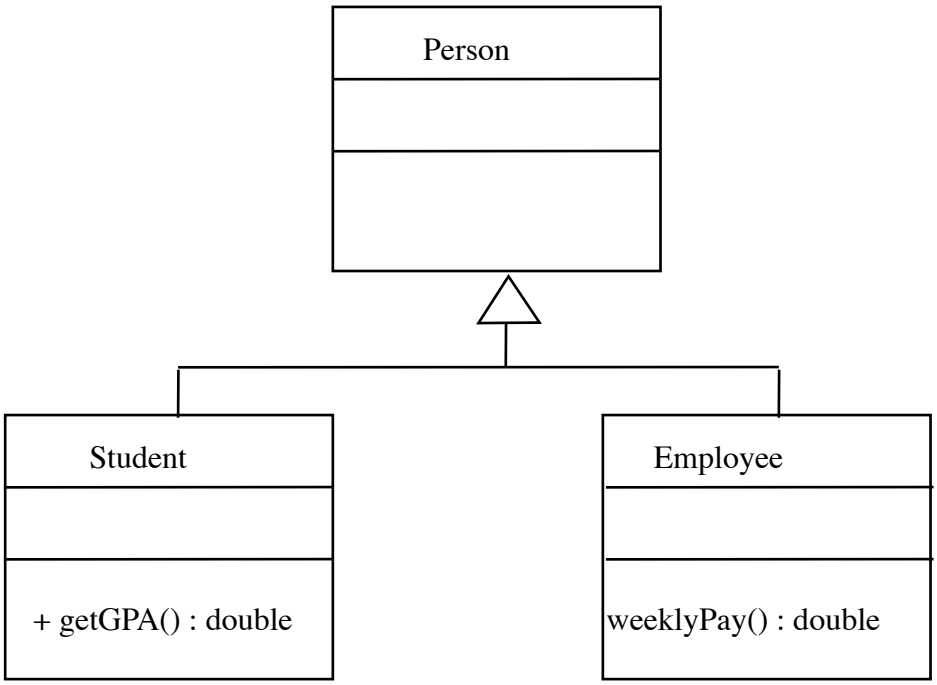
1. Just as an individual method can be declared final, so an entire class can be declared final. (E.g. public final class ...).

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 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.

   a) Does this mean that there are two copies of the Person information stored - one for TA as Student and one for TA as Employee?

   b) Does this mean that a TA can have two names - one as a Student and one as an Employee?

   c) 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.)