Lesson 35: Inheritance

Within a new project we will create three classes……*BankAccount*, *SavingsAccount*, and *Tester*. First, the *BankAccount* class:

```
public class BankAccount
{
    public BankAccount(double amt) //Constructor
    { balance = amt; }

    public double getBalance( )
    { // You supply code here that returns the state variable, balance.}

    public void deposit(double d)
    { //You supply code here that adds d to balance. }

    public void withdraw(double d)
    { //You supply code here that subtracts d from balance. }

    private double balance;
}
```

Subclass and Superclass:

This *BankAccount* class will be known as our **Superclass**. We will now create a *SavingsAccount* class that will be known as a **Subclass**. This *SavingsAccount* class is a perfect candidate to use as a subclass of *BankAccount* since it needs all the methods and state variable of the superclass, *BankAccount*. To make the *SavingsAccount* class **inherit** those methods and the state variable, use the key word ***extends*** as follows:

```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double amount, double rate) //Constructor
    {
        super(amount);          //Calls the constructor in
        interestRate = rate;    //BankAccount and sets balance
    }
    public void addInterest( )
    {
        double interest = getBalance( ) * interestRate / 100;
        deposit(interest);
    }
    private double interestRate;
}
```

There are some significant features of the constructor in *SavingsAccount*. In the absence of *super(amount)*, it would have tried to **automatically** call the *BankAccount* constructor and would have failed, since that constructor requires a parameter and we would not have supplied one. By making *super(amount)* **the first line** of code, we are able to supply the needed parameter. When used, it **must** be the first line of code in the constructor.

There is also something interesting in the *addInterest* method above. Notice that we are calling the *getBalance* and *deposit* methods. This is completely legal even though they are not *static* methods and we are not accessing them with a *BankAccount* object. Why is it legal? It is because we have **inherited** these methods from *BankAccount* by virtue of the *extends BankAccount* portion of our class signature.

Testing the subclass and superclass:

And finally, we will create a class called *Tester* that we will use to test the interaction of our superclass and subclass:

```
public class Tester
{
        public static void main(String[] args)
        {
                //This begins a new account in which the initial balance is 200
                // and the interest rate is 5%.
                SavingsAccount myAccount = new SavingsAccount(200, 5);

                //Make a deposit…notice we use an inherited method, deposit
                myAccount.deposit(132.14);

                myAccount.addInterest( );

                //Here, we use another inherited method, getBalance
                System.out.println( "Final balance is: " +
                myAccount.getBalance( ) );
        }
}
```

Important terms and ideas:

Superclass…the "original" class (sometimes called the base class)

Subclass…the one that says "extends" (sometimes called the derived class)

abstract

a. As applied to a **class**…Example, *public abstract class MyClass*… prevents objects from being instantiated from the class. Why would we want to do this? Perhaps the only way we would want our class used is for it to be inherited.

b. As applied to a **method**…Example, *public abstract void chamfer( );*…means that no code is being implemented for this method in this class. This forces the subclass that inherits this class to implement the code there.

Note that the signature of an abstract method is immediately followed by a semicolon and that there can be no body (curly braces) for the method.
If any method in a class is abstract, then that **forces** its class to be abstract too.

final

a. As applied to a **class**…*public final class MyClass*…means no other class can inherit this one.

b. As applied to a **method**…*public final void bisect( )*…means it can't be overridden in a subclass. See the discussion below for the meaning of "overriding".

Overriding: if a method is defined in a superclass and is also defined in a subclass… then when objects are made from the subclass, the redundant method in the subclass will take precedence over the one in the superclass. This is overriding.

There is a way to access a method in a superclass that has been overridden in the subclass. Let's suppose the method's signature in both classes is:

```
public void trans(double x)
```

From within some method of the subclass you can access the method *trans* in the superclass via a command like this:

```
super.trans(15.07);
```

***private* methods not inherited:**

Methods and state variables in the superclass that are designated as *private* are **not** inherited by the subclass.

Shadowing…is when a state variable in a subclass has a name identical to that of a state variable in the superclass. We do not say that the subclass variable overrides the other, rather that it **shadows** it. In such cases, uses within the subclass of the redundant variable give precedence to the subclass variable. It is, however, possible to access the shadowed variable in the superclass by creating a method in the superclass to access it. Suppose that the shadowed public variable in question is *double y*. Then, in the superclass create this method.

```
public double getY( )
{ return y; }
```

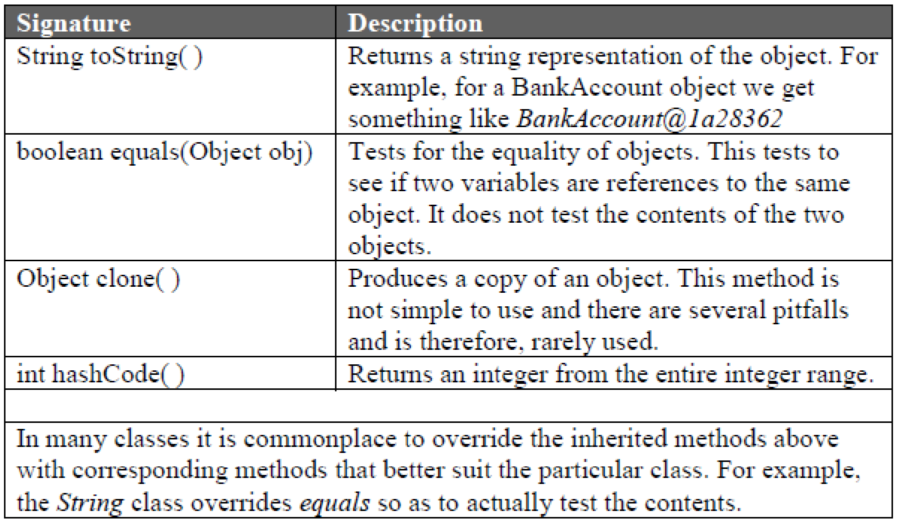
Since this method is inherited by your subclass, use it to obtain the *y* value in the superclass. Assuming that an object created with your subclass is called *myObj*, consider the following code within the subclass:

```
double d = myObj.getY( ); // returns y from the superclass
double p = myObj.y; // returns y from the subclass
```

There is also another type of shadowing. Let's look at a method that brings in the variable *z* as a parameter. Complicating things is a state variable also named *z*.

```
public class MyClass
{
        . . .
        public void aMethod(int z)
        {
                z++;    //This increments the local z which has precedence here
                        // within this method.
                this.z = 19;    //only way to access the state variable z from within
                                //this method.
        } . . .
        public int z;
}
```

Cosmic Superclass: Every class that does not extend another class automatically extends the class *Object* (the cosmic superclass). Following are the signatures and descriptions of four of the most important methods of the *Object* class.



Signature	Description
String toString( )	Returns a string representation of the object. For example, for a BankAccount object we get something like *BankAccount@1a28362*
boolean equals(Object obj)	Tests for the equality of objects. This tests to see if two variables are references to the same object. It does not test the contents of the two objects.
Object clone( )	Produces a copy of an object. This method is not simple to use and there are several pitfalls and is therefore, rarely used.
int hashCode( )	Returns an integer from the entire integer range.
In many classes it is commonplace to override the inherited methods above with corresponding methods that better suit the particular class. For example, the *String* class overrides *equals* so as to actually test the contents.	

Creation of objects:

Suppose we have a superclass called *SprC* and a subclass called *SbC*. Let's look at various ways to create objects:

a. SbC theObj = new SbC( );
SprC anotherObj = theObj;

Since *anotherObj* is of type *SprC* it can **only** access methods and state variables that **belong to *SprC***; however, overridden methods will be **accessed in *SbC***.

b. SprC hallMark = new SbC( );

Since *hallMark* is of type *SprC* it can only access methods and state variables that **belong to *SprC***; however, overridden methods will be **accessed in *SbC***.

c. SbC obj = new SprC( ); //illegal

Expecting a particular object type:

Any time when a parameter is **expecting** to receive an object of a particular type, it is acceptable to send it an object of a subclass, but never of a superclass. This is because the passed subclass object **inherits** all the methods of the object. Otherwise, the expected object may have methods **not** in a superclass object. Consider the following hierarchy of classes where each class is a subclass of the class immediately above it.

Person
Male
Boy

Suppose there is a method with the following signature:

```
public void theMethod(Male ml)
```

The method *theMethod* is clearly **expecting a *Male* object**; therefore, the following calls to this method would be legal since we are either sending a ***Male*** object or an object of a **subclass**:

```
Male m = new Male( );
theMethod(m);                //ok to send m since it's expecting a Male object
Boy b = new Boy( );
theMethod(b);          //ok to send b since b is created from a subclass of Male
```

Since *theMethod* is expecting a *Male* object, we **can't** send an object of a **superclass**.

```
Person p = new Person( );
theMethod(p);                //Illegal
theMethod( (Male)p );        //Legal if we cast p as a Male object
```

Using the same classes from above, the following examples illustrate legal and illegal object creation. Notice when we use a class on the left, the class on the right must be either the **same** class or a **subclass**.

```
Person p = new Male( );      //legal
Person p = new Boy( );       //legal
Male m = new Boy( );         //legal
Boy b = new Male( );         //illegal
Boy b = new Person( );       //illegal
Male m = new Person( );      //illegal
```

instanceof

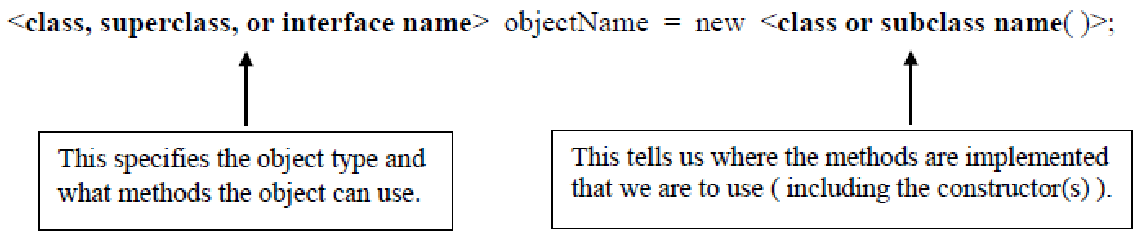
This method tells us if an object was created from a particular class. Suppose *Parent* is a superclass, *Child* is one of its subclasses, *objP* is a *Parent* object, and *objC* is a *Child* object. Also, assume that *Circle* is some unrelated class.

d. (objC instanceof Child) returns a *true*
e. (objC instanceof Parent) returns a *true*
f. (objC instanceof Circle) returns a *false*
g. (objP instanceof Child) returns *false*
h. (objP instanceof Parent) returns *true*
i. (objP instanceof Circle) returns *false*

Notice the syntax of *instanceof* is that an **object precedes** *instanceof* and a **class, subclass or interface follows**.

The big picture:

The following shows the function of each part in the declaration and creation of an object
in which a superclass, subclass, or interface may be involved.



<**class, superclass, or interface name**> objectName = new <**class or subclass name**( )>;

This specifies the object type and what methods the object can use.

This tells us where the methods are implemented that we are to use (including the constructor(s)).

Fig. 35-1 Determining object type, legal methods, and where the methods are implemented. If a method in the subclass overrides that of the superclass, then code in the subclass runs.

Inheritance is considered one of the most important, but, unfortunately, one of the most difficult aspects of Java. See Appendix U for an enrichment activity in which you would be able to participate in electronic communities in the form of message boards (forums). Investigate the questions and answers that other programmers post concerning this topic.