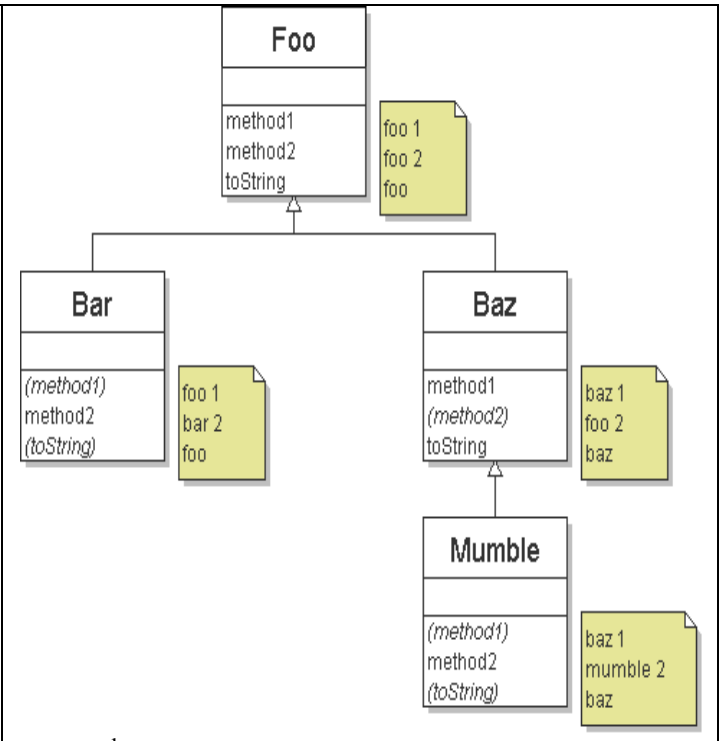
Practice Problems: Inheritance & Polymorphism

```
public class Foo {
   public void method1() {
      System.out.println("foo 1");
   }
   public void method2() {
      System.out.println("foo 2");
   }
   public String toString() {
      return "foo";
   }
}
public class Bar extends Foo {
   public void method2() {
      System.out.println("bar 2");
   }
}

public class Baz extends Foo {
   public void method1() {
      System.out.println("baz 1");
   }
   public String toString() {
      return "baz";
   }
}
public class Mumble extends Baz {
   public void method2() {
      System.out.println("mumble 2");
   }
}

public class Polymorphism{
public static void main(String [] args){
Foo[] pity = { new Baz(), new Bar(),
            new Mumble(), new Foo() };
for (int i = 0; i < pity.length; i++) {
   System.out.println(pity[i]);
   pity[i].method1();
   pity[i].method2();
   System.out.println();
}
}
```




```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1
mumble 2

foo
foo 1
foo 2
```

*( ) method is inherited. Otherwise, method is overridden.

1. **Tracing programs: The above is the program demonstrated in class. Now, what gets printed to the screen when we execute the following classes on the left?**

`` public class A { public int x = 1; public void setX(int a){ x=a; } } public class B extends A { public int getB(){ setX(2); return x;} } public class C { public static void main(String [] args){ A a = new A(); B b = new B(); System.out.println(a.x); System.out.println(b.getB()); } } ```	Result: 1 2 Public instance variable and instance method can be inherited and accessed by subclass (without overriding)
`` public class A { private int x = 1; protected void setX(int a){ x=a; } protected int getX(){ return x;} } public class B extends A { public int getB(){ setX(2); //return x; It does not work because private modifier, so return getX(); } } public class C { public static void main(String [] args){ A a = new A(); B b = new B(); System.out.println(a.getX());//a.x is not allowed, private! System.out.println(b.getB()); } } ```	Result: 1 2 Private instance variable and private instance methods can be inherited but not accessible to subclass! Protected instance variable and protected instance methods can be inherited and accessible to subclass,
`` public class A { protected int x = 1; protected void setX(int a){x=a;} protected int getX(){return x;} } public class B extends A { public int getB(){ setX(2); return x; } } public class C { public static void main(String [] args){ A a = new A(); B b = new B(); System.out.println(a.getX()); System.out.println(b.x); //b.x is protected, then inherited. System.out.println(b.getB()); } } ```	Result 1 1 2 *The difference of B's x is not variable shadowing. It's the expected execution of value resetting (setX(2)).

```
public class A {
protected int x = 1;
protected void setX(int a){
x=a;
}
protected int getX(){
return x;}
}
public class B extends A {
protected int x = 3;
public int getX(){
return x; }
public int getB(){
return x;
}
}
public class C {
public static void main(String [] args){
A a = new A();
B b = new B();
System.out.println(a.getX());
System.out.println(b.getB());//subclass method access own attrib
System.out.println(b.getX());//overriding method, accessing sub
System.out.println(a.x); //protected
System.out.println(b.x); //overriding attribute!
}
}
```

Resutls:
1
3
3
1
3

Do you know which getX of b is called, A's or its own? If you cannot ensure your answer right, please see the comment in the below.

```
public class A {
protected int x = 1;
protected void setX(int a){
x=a;
}
protected int getX(){
return x;}
}
public class B extends A {
protected int x = 3;
public int getX(){
return x; }
public int getB(){
return x;
}
}
public class C {
public static void main(String [] args){
A a = new A();
A b = new B();   //polymorphism, making shadowing possible!
System.out.println(a.getX());
System.out.println(b.getX());//override, access subclass attri.
//System.out.println(b.getB()); not able to load subclass method!
System.out.println(a.x);
System.out.println(b.x);  //variable shadowing!
}
}
```

Results:
1
3
1
1

Subclass variable can be accessed by method, the direct access (without using method) will reach the overridden value from superclass!

b.getB is not permitted because it is out A's signature. b.getX is allowed because it is overridden!

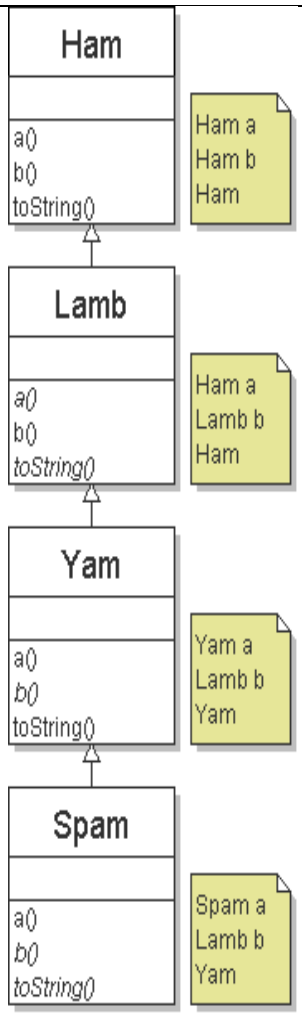
// For your development:
//1) Is it good to block the use of b.getB()?
// ANS>: Good, because methods can be in template. In the security control, no leakage!
//2) Is it good to have the direct access of attribute such as b.x?
// ANS>: Better not, if it is not in your control. See how complicate it is in this program.

```
public class A {
protected int x = 1;
protected void setX(int a){
x=a;
}
protected int getX(){
return x;}
}
public class B extends A {
protected int x = 3;
public int getX(){
setX(2); // call superclass method to set superclass attrib
return x; } //but return attrib of subclass
public int getB(){
return x;
}
}
public class C {
public static void main(String [] args){
A a = new A();
A b = new B();
System.out.println(a.getX());
System.out.println(b.getX());
System.out.println(a.x);
System.out.println(b.x);
}
}
```

Results:
1
3
1
2

b.x is set to 2 because a superclass method is called to change the value of shadowed value.

```
public class Ham {
    public void a() {
        System.out.println("Ham a");
    }
    public void b() {
        System.out.println("Ham b");
    }
    public String toString() {
        return "Ham";
    }
}

public class Lamb extends Ham {
    public void b() {
        System.out.println("Lamb b");
    }
}

public class Yam extends Lamb {
    public void a() {
        System.out.println("Yam a");
    }
    public String toString() {
        return "Yam";
    }
}

public class Spam extends Yam {
    public void a() {
        System.out.println("Spam a");
    }
}



public class Polymorphism2 {
public static void main (String [] args){
Ham[] food = { new Spam(), new Yam(),
               new Ham(), new Lamb() };
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
    }
}
}
```



Yam
Spam a
Lamb b

Yam
Yam a
Lamb b

Ham
Ham a
Ham b

Ham
Ham a
Lamb b

```
public class Ham {
        int a = 0;
             int b = 1;
    public void a() {
        System.out.println("Ham " + a);
    }
    public void b() {
        System.out.println("Ham " + b);
    }
    public String toString() {
        return "Ham " + a + " " + b;
    }
}

public class Spam extends Ham {
        int a = 2;
    public void a() {
        System.out.println("Spam " +a);
    }
}

public class Yam extends Spam {
        int b = 3;
    public void a() {
        System.out.println("Yam " + a);
    }
    public void b() {
        System.out.println("Yam " + b);
    }
}

public class Polymorphism3 {
public static void main (String [] args){
Ham[] food = { new Spam(), new Yam(),
               new Ham()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();

System.out.println(food[i].a);

System.out.println(food[i].b);
    System.out.println();
}
}
}
```

```
Ham 0 1
Spam 2
Ham1
0
1

Ham 0 1
Yam 2
Yam 3
0
1

Ham 0 1
Ham0
Ham1
0
1
```

```
public class A
{
  private String x = "Ax";
  protected String y = "Ay";
  public String z = "Az";

  public String toString() {
    return x + y + z;
  }

public static void main(
String [] args)
{
A a = new A();
System.out.println(a);
}
}

public class B extends A
{
  private String x = "Bx";
public String z = "Bz";

public String toString() {
    return x + y + z;
  }

  public static void main(
String [] args)
  {
    B b = new B();
    System.out.println(b);
  }
}
```

```
public class C extends A
{
private String x = "Cx";

  public static void main(
                        String [] args)
  {
    C c = new C();
    System.out.println(c.x);
    System.out.println(c);
  }
}

public class D extends C
{
  private String x = "Dx";
  public String z = "Dz";

public static void main(
                        String [] args)
  {
D d = new D();
System.out.println(d.x);
    System.out.println(d.y);
    System.out.println(d.z);
    System.out.println(d);

C c = new D();
// Error: System.out.println(c.x);
    System.out.println(c.y);
    System.out.println(c.z);
    System.out.println(c);
  }
}
```

When A is executed, it displays:

```
AxAyAz
```

The println statement implicitly calls a.toString(), which creates a string containing the concatenation of the variables x, y, and z. Once this concatenated String ("AxAyAz") is returned, it gets printed to the screen.

When B is executed, it displays:

```
BxAyBz
```

The println statement implicitly calls b.toString(), which refers to the overriding toString() method defined in subclass B. This toString method says to concatenate x+y+z (like the one defined in class A), but now it is referring to variables x, y, and z in the B class. Two of those variables are defined locally – x and z. These variables hide (or "shadow") the x and z variables defined in class A. The third variable, y, is inherited from A. As a result, the concatenation in the toString method produces a String that looks like "BxAyBz", which then gets printed.

When C is executed, it displays:

```
Cx
AxAyAz
```

When the first println statement executes, it refers directly to c.x, which is defined locally. So that prints out "Cx".
When the second println statement executes, it refers to C's inherited toString method. The inherited toString method is defined in class A, which returns the concatenation of x, y, and z from class A. So that returns "AxAyAz", which gets printed.

When D is executed, it displays:

```
Dx
Ay
Dz
DxAyDz
Ay
Az
DxAyDz
```

The first 3 lines print the values of x, y, and z inside of d. Since x and z are defined inside of class D, those values get printed out. y is inherited from class A, so "Ay" gets printed out.

The next line prints the return value of D's toString method. The toString method is defined locally to override the inherited one (unlike in the example for class C, where the toString method is inherited instead of overridden). Because the toString method is overridden, when it refers to x, y, and z, it refers to the variables inside of class D. So this returns "DxAyDz". Compare this to the inherited toString method in class C, which returns "AxAyAz".

The next two lines use a variable with static type C to refer to an object of dynamic type D. Notice that it is an error to try to print (or refer to) c.x. That's because 'x' is private in class C, and this code is written inside class D. Also notice that c.z is "Az", whereas d.z is "D.z". For fields, Java uses the value of the static type's field (in this case, the value of z from class C, which is inherited from class A and has value "Az").

The last line prints the value of c.toString(). Java uses the value of a the *static* type's *field*, but the *dynamic* type's *methods*. Variable c has dynamic type D, because it refers to an object of type D. So Java uses the toString method defined in class D, which returns the values of x, y, and z within class D (or "DxAyDz"). Notice the difference between how fields get handled, and how methods get handled. The field c.z refers to the field defined in class C (which is inherited from class A). The method c.toString() refers to the method defined in class D, not class C.

I have still not figured out any reason why Java does shadow things this way for fields. It's very confusing, and it can lead to very hard-to-fix bugs. In general, it is HIGHLY RECOMMENDED that you AVOID defining fields with the same name as a superclass's field. Sometimes though (like when you're extending a superclass from the Java API), you may not know what the superclass's fields are called, and in that case, you just have to guess.

2. Program Development

Here's a problem from a previous Final Exam.

Consider the following skeleton for a Robot class, which has private fields for storing the location of aRobot object, its name, and the direction it’s facing (North for a direction parallel to

the positive y axis,South for the negative y axis, East for the positive x axis, or West for the negative x axis). It also hasstub methods for constructing a Robot object, changing the direction, and moving the location of therobot in the direction it's facing.

```
public class Robot
{
      private String name;

      private char direction; //'N','S','E', or 'W'

      privateintxLoc, yLoc; // the (x, y) location of the robot

      // Initialize name, direction, and (x, y) location
      public Robot(String name, char dir, int x, int y) { ... }

      public String toString()
      {
            return name + " is standing at (" + x + "," + y + ") and facing"
            + direction);
      }

      // turn 90 degrees clockwise, e.g. 'N' changes to 'E', 'E' to 'S', ...
      public void turnClockwise() { ... }

      // turn 90 degrees counterclockwise, e.g. 'N' to 'W', 'W' to 'S', ...
      public void turnCounterClockwise() { ... }

      // move numSteps in direction you are facing,
      // e.g. if 'N' 3 steps, then y increases 3
      public void takeSteps(intnumSteps) { ... }
}
```

(a)Assuming the class above is completed correctly, what does the following program display on the screen:

```
public static void main(String args[])
{
      Robot robby = new Robot("Robby", 'N', 10, 12);
      for (inti = 0; i< 5; i++)
      {
            if (i % 2 == 0)
            {
                  robby.turnClockwise();
            }
            else
            {
                  robby.turnCounterClockwise();
            }

            robby.takeSteps(3);
            System.out.println(robby);
      }
}
```

Displayed on screen:

```
Robby is standing at (13, 12) and facing E
Robby is standing at (13, 15) and facing N
Robby is standing at (16, 15) and facing E
Robby is standing at (16, 18) and facing N
Robby is standing at (19, 18) and facing E
```

(b) Complete the constructor, the `turnClockwise`method, and the `takeSteps`method. Make sure your constructor validates its input. You do not need to define `turnCounterClockwise`.

```
public Robot(String name, char dir, int x, int y)
{
this.name = name;
this.direction = dir;
  this.xLoc = x;
  this.yLoc = y;
}

public void turnClockise()
{
  if(direction=='N') { direction = 'E'; }
  else if(direction=='E') { direction = 'S'; }
  else if(direction=='S') { direction = 'W'; }
  else { direction = 'N'; }
}

public void takeSteps(int numSteps)
{
if(direction=='N') { yLoc += numSteps; }
  else if(direction=='E') { xLoc += numSteps; }
  else if(direction=='S') { yLoc -= numSteps; }
  else { xLoc -= numSteps; }
}
```

(c) Write Java code to create an array of 5 robots. Use a for loop to fill in the array so that the n-th robot is named “robot n”, and it starts off life facing east at location (n, n).

```
Robot [] robots = new Robot[5];
for(int i=0; i<robots.length; i++)
{
  robots[i] = new Robot("robot " + i, 'E', i, i);
}
```

Here's another problem from a previous Final Exam. This one is an inheritance/polymorphism question.

```
classSuperClass
{
     protectedint x = 0;

     publicSuperClass(int x)
     {
          this.x = x;
     }

     private void increment() { x++; }

     protected final void add(int y)
     {
          x += y;
     }

     public void display()
     {
          System.out.println(x);
     }
}

public class SubClassextends SuperClass
{
     publicSubClass(int x)
     {
          super(x);
     }

     public void display()
     {
          add(2);
          super.display();
     }

     public static void main(String [] args)
     {
          SuperClasssc = new SuperClass(3);
          sc.display();

          sc = new SubClass(3);
          sc.display();
     }
}
```

(a)List the name of all methods that subclasses of SuperClass inherit.

subclasses inherit all methods: the constructor, increment, add, and display. If you want, you could also list all of the methods that SuperClass implicitly inherits from the Object class (eg, equals, toString, etc.), but that's not required.

(b)List the name of all methods that are visible in subclasses of SuperClass (in other words, methodsthat can be called directly).

add can be called directly just be using the name add(). the constructor SuperClass can be called by using the super() constructor. the display() method from SuperClass is overridden by the display() method in the SubClass, but it can still be called by writing super.display(). In summary, any method that has public or protected access in the superclass can be called directly by the subclass.

(c)List the name of all methods that may NOT be overridden by any subclasses of SuperClass.

*methods that are declared to be **final** in the superclass may not be overridden. So the add() method may not be overridden.*

(d)What gets displayed on the screen when SubClass is executed?

displayed on screen:

```
3
5
```