Practice with inheritance and polymorphism.

What is displayed on the screen when we execute classes A, B, C, and D?

<table>
<thead>
<tr>
<th>Class A</th>
<th>Class B</th>
<th>Class C</th>
<th>Class D</th>
</tr>
</thead>
</table>
| ```java
public class A {
    private String x = "Ax";
    protected String y = "Ay";
    public String z = "Az";

    public String toString() {
        return x + y + z;
    }
}
``` | ```java
public class B extends A {
    private String x = "Bx";
    public String z = "Bz";

    public String toString() {
        return x + y + z;
    }
}
``` | ```java
public class C extends A {
    private String x = "Cx";

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

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 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.