Hiding a field

It is bad practice to declare the same field in a class and its subclass. We never do it, and we hope you don't either. We show you with an example that doing so does *not* override the field in the class, it simply hides it.

Overriding is meant for overriding *behavior* (provided by methods), not structure.

In the program shown below, classes C and T declare a field f; class S does not. Execution of the first three statements —assignments to fresh local variables t, s, and c— produce the object shown below to the right as well as the three local variables t, s, and c, all of which contain the same pointer to the object. But the three local variables have different types. Note that we have made the fields public to make it easier to make our point.

Execution of the last three statements, the println statements, produces this output:

```
t.f is 2
s.f is 1
c.f is 1
```

Now, if the overriding/bottom-up rule were used, t.f, s.f, and c.f would all yield the value 2. But the output shown above shows you that this is not the case.

Instead, the type of the pointer —T for t.f, S for s.f, and C for c.f,— determines in which partition of the object to start an upward search for f. That is why both s.f and c.f yield the value 1.

```
public class TestF{
   public static void main(String args[]) {
      T t= new T();
      S s= t;
      C c= s;
      System.out.println("t.f is " + t.f);
      System.out.println("s.f is " + s.f);
      System.out.println("c.f is " + c.f);
   }
}

class C {
   public int f= 1;
}

class S extends C {
}

class T extends S {
   public int f= 2;
}
```