CS1110 Lec. 13 **14 Oct 2010**

Another classy lecture: Casting about (secs 4.2, 4.3)

1. the class hierarchy
2. apparent and real classes
3. casting between classes
4. operator **instanceof**
5. function equals
6. abstract methods/classes (section 4.7, labs next week)

Reading for next time: Sec. 2.3.8 and chapter 7 on loops.

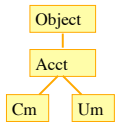
A4 due Saturday; make sure you're using Monday's a4.zip files (see assignments page on the course website for description of updates)

Time management tip #42: schedule deadlines on your calendar; *also schedule the time it will take to do the work*.

Setting: Cmail (Cm) and Umail (Um) accounts.

They have commonalities, like netIDs and an "alert" ability, so we make them subclasses of class Acct.

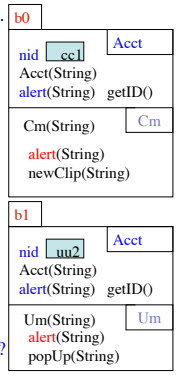
the class hierarchy:



But, Cm and Um override Acct method alert(String), due to system differences.
Cmail shows a "Web clip"; Umail creates a popup.

Why do we keep drawing the overridden alert?

[Note: Acct might best be made an *abstract* class; see last slide and next lab.]



c b0 Cm

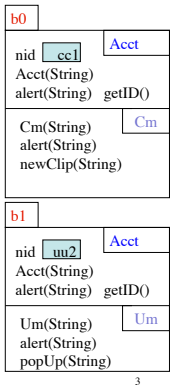
u b1 Um

a b0 Acct

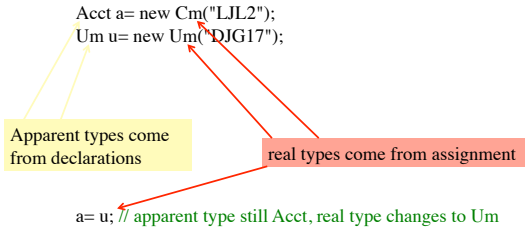
Is a= c; legal? b0 is an Acct, but it's also a Cm.

The **apparent** (declared) type of **a** is Acct, and *will always be* Acct.
This is a syntactic property having to do with compiling.

The **real** type of **a**, the real class of the object whose name is *currently* in **a**, is Cm, but *could change* via assignment: **a**= u;
This is a semantic property having to do with the current value of **a**.



Sources of apparent and real types



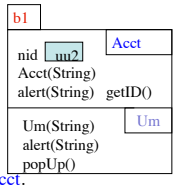
Implicit casting up the class hierarchy (good news)

u b1 Um

Vector<Acct> v

0	1	2 …
b0	b14	…

[Not drawing Vectors as objects to save space]



u has apparent type Um,
but our list v has an apparent type based on Acct.

Does this mean we *must* do an explicit cast to add u to v?

v.add((Acct) u);

Nope; luckily, casts **up** the hierarchy are automatic, allowing this:

v.add(u);

More good news:
Overriding (still) has the correct behavior

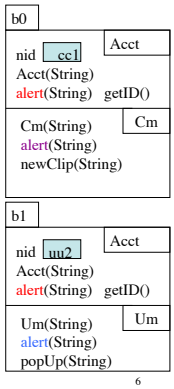
Vector<Acct> v

0	1	2
b0	null	b1

What happens with: v.get(0).alert("flood"); ?

First, the compiler checks that apparent type Acct has an alert method; if that succeeds, **then** the bottom-up rule is applied.

v.get(0).alert("flood") will call the over-riding, Cmail-specific alert("flood") method. ☺



A sensible policy with an embedded "gotcha": The apparent type can rule out some available methods. (Java is conservative).

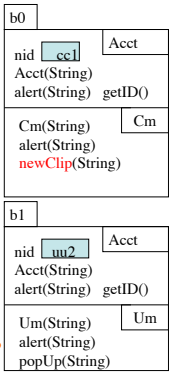
	0	1	2
Vector<Acct> v	b0	null	b1

v.get(0).newClip("FLOOD"); ?

The *apparent* type of v.get(0) does *not* have a newClip() method.

Therefore, the compiler rules the call v.get(0).newClip("FLOOD") **illegal**, even though in practice, the real type of v.get(0) might mean that newClip(…) is available.

Why do we keep drawing the overridden alert? Without it, even v.get(0).alert("FLOOD") is illegal!



Workaround: check and cast to the real type.

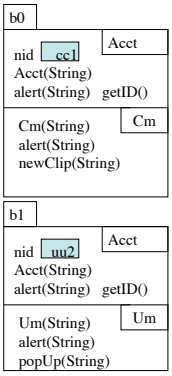
a b0 Acct

To access a subclass's non-overriding methods, we must explicitly downward-cast and/or declare fresh variables of the right apparent type (subclass Cm, not Acct).

To assign correctly to these fresh variables, we first check the real type:

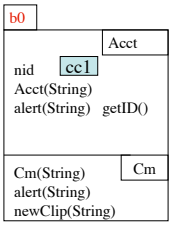
```
if ( a instanceof Cm) {
      Cm newC= (Cm) a;
      newC.newClip(…);
}
```

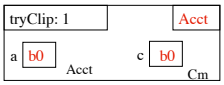
need this downward cast (can't just wedge "big" class into small, despite the "if"!)



Example

```
public class Acct {
    // If Acct is a Cm, apply newClip,
       o.w, do nothing.
   public static void tryClip(Acct a, String msg) {
       if ( ! (a instanceof Cm) )
            return;
       // a is a Cm
       Cm c= (Cm) a ;     // downward cast
       return c.newClip(msg);
}
```





Apparent type of a: Acct
Real type of a: Cm

Here, **(Um) a** would lead to a runtime error.

Don't try to cast an object to something that it is not!

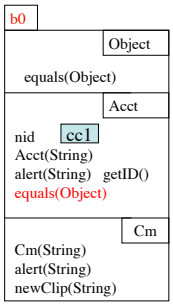
The correct way to write method equals

Method equals helps prevent addition of duplicates to lists, etc.

Note that method equals should take arbitrary Objects as arguments.

```
public class Acct {
   …
   /** = "ob is an Acct with the same
          values in its fields as this Acct" */
   public boolean equals (Object ob) {

     if (!(ob instanceof Acct)) return false;
     Acct a= (Acct) ob;  // why? b/c Objects don't
                         // generally have nids
     return  nid.equals(a.nid); // check field
                                // equality
}
```



Abstract classes and methods (see lab next week)

Make a (super)class abstract if there can only be subclass objects, but you still want default behaviors/info.

Example: Nothing is just a generic Animal (it's a Pig, or a Butterfly, or a Person) --- so, can't create just an "Animal" --- but all breathe oxygen, and live somewhere.

```
public abstract class Animal {
    public boolean breathesOxygen() {return true;}
    public abstract String habitat() ;
}
```

Make a (superclass) method abstract to **force** (non-abstract) subclasses to override it (and hence define it):

Example: In Acct (note stranded semi-colon!):

```
public abstract void alert(String s) ;
```

means every sub-type of email account must have an alert method, but there's no default alert method.