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

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:
Object
| Acc
| Cm
| Um
```

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

Why do we keep drawing the overridden `alert`?

```
Sources of apparent and real types

Apparent types come from declarations
real types come from assignment
```

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

```
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.
```

```
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:
Object
| Acct
| Cm
| Um
```

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`?

```
Sources of apparent and real types

Apparent types come from declarations
real types come from assignment
```

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

```
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.
```

```
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:
Object
| Acct
| Cm
| Um
```

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`?

```
Sources of apparent and real types

Apparent types come from declarations
real types come from assignment
```

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

```
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.
```

```
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:
Object
| Acct
| Cm
| Um
```

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`?

```
Sources of apparent and real types

Apparent types come from declarations
real types come from assignment
```

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

```
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.
```

```
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:
Object
| Acct
| Cm
| Um
```

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`?

```
Sources of apparent and real types

Apparent types come from declarations
real types come from assignment
```

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

```
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.)

Workaround: check and cast to the real type.

Null Vector<Acct> v

Apparent type of v.get(0): Acct

Real type of v.get(0): Cm

Try Clip

The correct way to write method equals

Abstract classes and methods (see lab next week)

Example

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!

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(…);
}

Note that method equals should take arbitrary Objects as arguments.

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.