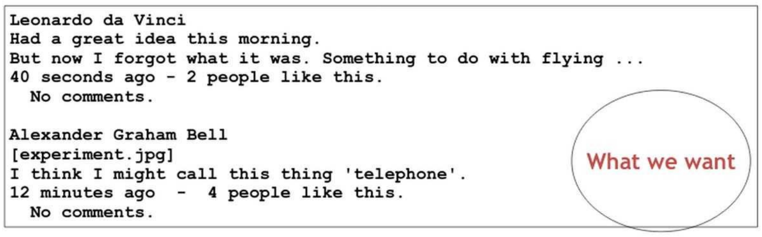
INHERITANCE 2

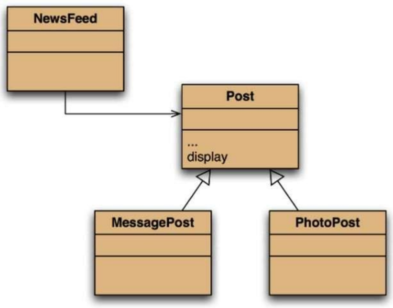
A PROBLEM WITH OUR SOLUTION

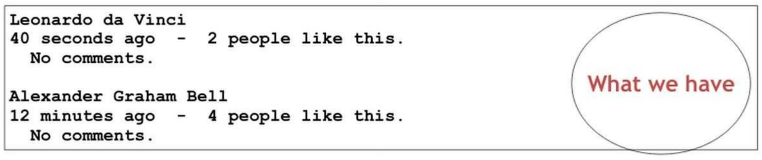
We have a conflicting output...



```
Leonardo da Vinci
Had a great idea this morning.
But now I forgot what it was. Something to do with flying ...
40 seconds ago - 2 people like this.
  No comments.

Alexander Graham Bell
[experiment.jpg]
I think I might call this thing 'telephone'.
12 minutes ago  -  4 people like this.
  No comments.
```





```
Leonardo da Vinci
40 seconds ago  -  2 people like this.
  No comments.

Alexander Graham Bell
12 minutes ago  -  4 people like this.
  No comments.
```

The `display` method in `Post` only prints the common fields.

This is because Inheritance is a one way street.

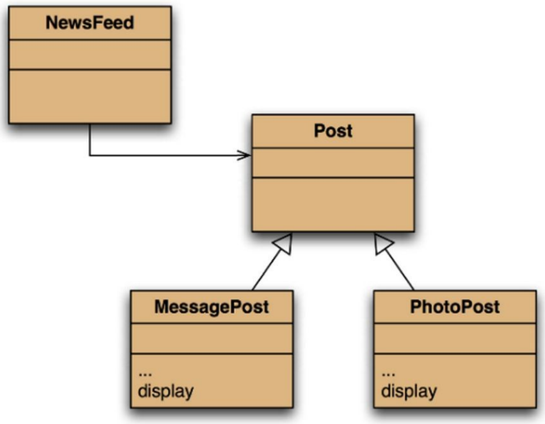
- A subclass inherits the superclass fields.
- The superclass knows nothing about its subclass's fields.

SOLUTION(?)

You could place the `display` method in the subclass...

- Each subclass has its own version
- `Post`'s fields are private
- `NewsFeed` cannot find a display method in `Post`.

  Therefore, it doesn't solve our problem.



STATIC TYPE AND DYNAMIC TYPE

A more complex type hierarchy requires further concepts to describe it.

Some new terminology:

- Static type
- Dynamic type
- Method dispatch/lookup

```
   Car c1 = new Car();
Vehicle v1 = new Car();
```

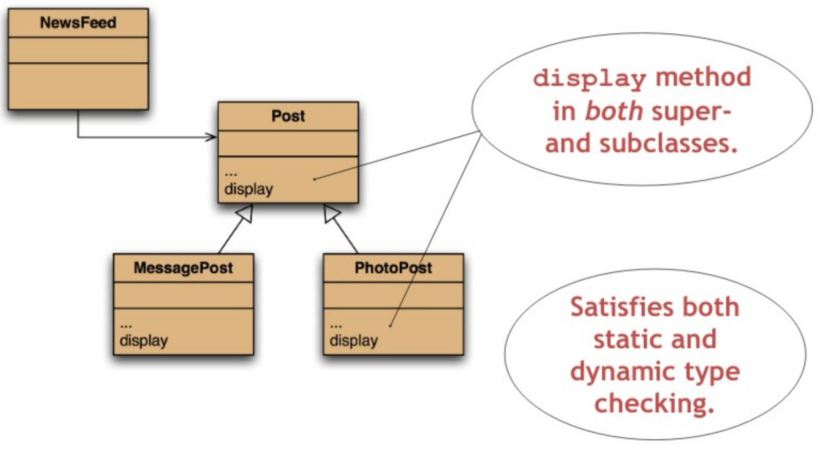
With inheritance, we have a situation where these types aren't the same. Hence, why we have static and dynamic types.

- The declared type of a variable is its static type.
- The type of the object a variable refers to is its dynamic type.
- The compiler's job is to check for static-type violations.

```
for(Post post: posts) {
     post.display();       // compile time error
}
```

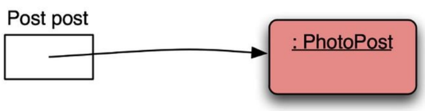
METHOD OVERRIDING

- Superclass and subclass define methods with the same signature.
- Each has access to the fields of its class.
- Superclass satisfies static type check.
- Subclass method is called at runtime – it overrides the superclass version.
- What becomes of the superclass version?

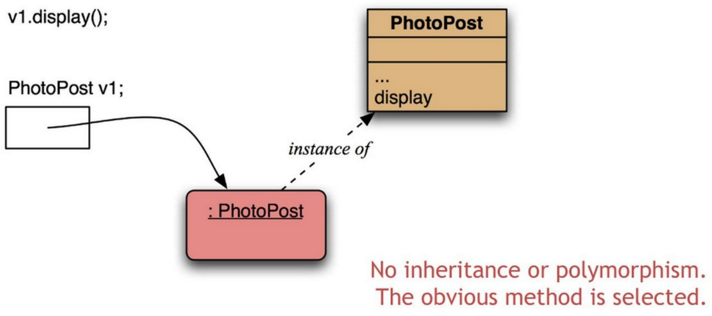

NewsFeed
Post
...
display
MessagePost
...
display
PhotoPost
...
display
display method in both super- and subclasses.
Satisfies both static and dynamic type checking.

DYNAMIC METHOD DISPATCH

What happens if we have this situation and we call a method on our post variable?

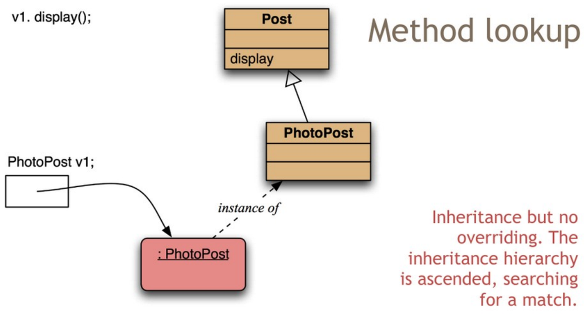


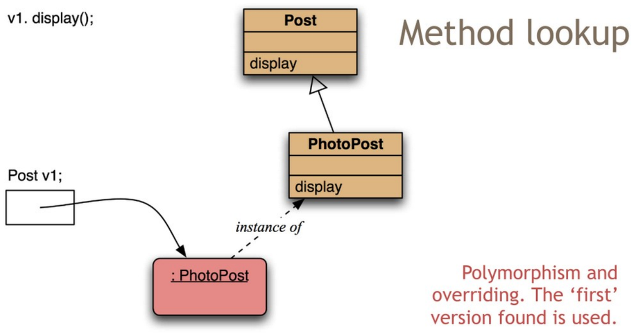
In a situation without inheritance...



1. Access the variable object
2. From the object access the class
3. From the class access the method

In a situation with inheritance…


v1. display();
Post
display
Method lookup
PhotoPost
PhotoPost v1;
instance of
: PhotoPost
Inheritance but no overriding. The inheritance hierarchy is ascended, searching for a match.



1. Access the variable object
2. From the object access the class
3. From the class access the method
4. If the method isn't in the class, it then goes up the hierarchy to the superclass.

CALLING OVERRIDEN METHODS

Overriden methods are hidden, but we often still want to be able to call them.

- An overriden method can be called from the method that overrides it.
    - `super.method(...)`
    - Compare with the use of `super` in constructors.
        - They don’t need to be called first

```
public void display()
{
      super.display();
      System.out.println(" [" + filename +"]");
      System.out.println(" " + caption);
}
```

METHOD POLYMORPHISM

We have been discussing polymorphic method dispatch.

- A polymorphic variable can store objects of varying types.
- Method calls are polymorphic.
    - The actual method called depends on the dynamic object type.

THE INSTANCEOF OPERATOR

- Used to determine the dynamic type.
- Identifies “lost” type information
- In general, it is possible and better to avoid doing this. Is there a better, more extendable way to write your code?
- Usually precedes assignment with a cast to the dynamic type:

```
if (post instanceof MessagePost) {
      MessagePost msg = (MessagePost) post;
      […]
}
```

OBJECT AND TOSTRING

THE OBJECT CLASS’S METHODS

- Methods in `Object` are inherited by all classes.
- Any of these may be overridden.
- The `toString` method is commonly overriden:
    - `public String toString()`
    - Returns a string representation of the object.

Every method in the object class is available to all classes in Java. It is useful to look at what methods are actually in the object class.

OVERRIDING TOSTRING IN POST

```
public String toString()
{
      String text = username + “\n” + timeString(timestamp);
      if (likes > 0) {
           text += “ – “ + likes + “people like this.\n”;
      }
      else {
           text += “\n”;
      }
      if(comments.isEmpty()) {
           return text + “No comments. \n”;
      }
      else {
           return text + “ “ + comments.size() +
           “comment (s). Click here        to view.\n”;
      }
}
```

This creates the string and returns the value rather than printing it out to terminal.

It is better to let the top level class decide how the value is printed.

Calls to println with just an object automatically results in toString being called:

```
System.out.println(post);
```

STRINGBUILDER

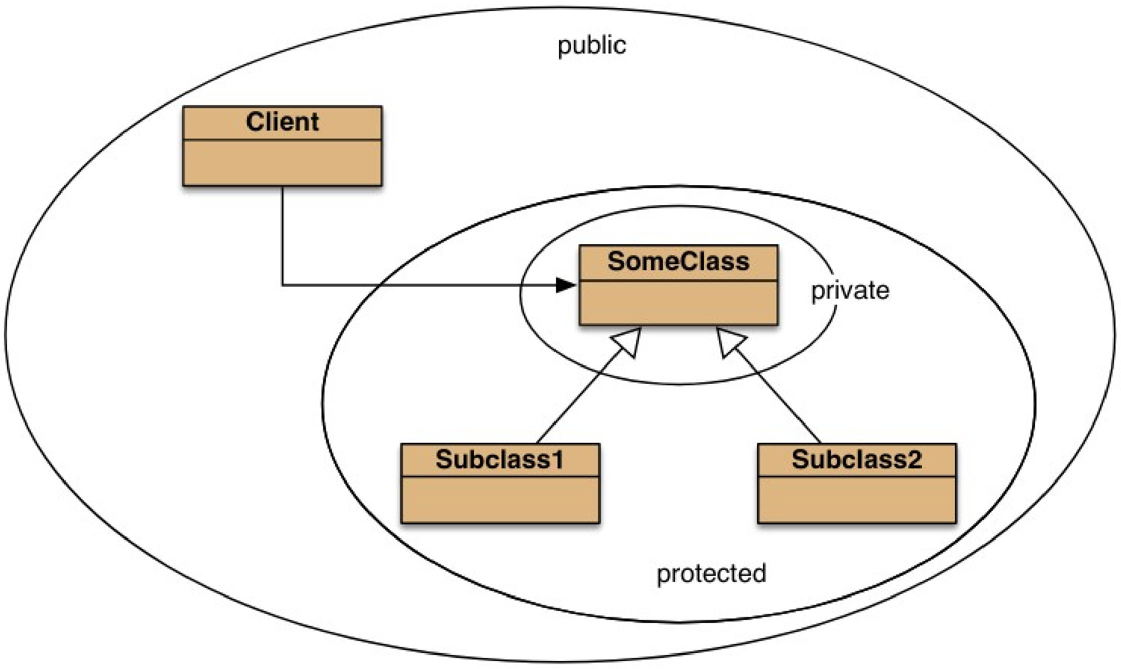
- Consider using StringBuilder as an alternative to concatenation:

```
StringBuilder builder = new StringBuilder();
builder.append(username);
builder.append('\n');
builder.append(timeString(timestamp));
…
return builder.toString();
```

PROTECTED ACCESS

Private access in the superclass may be too restrictive for a subclass.

- The closer inheritance relationship is supported by protected access.
- Protected access is more restricted than public access.
- We still recommend keeping fields private.
    - Define protected accessors and mutators.


public
Client
SomeClass
private
Subclass1
Subclass2
protected