University of HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 16: More Inheritance

Getting the most from inheritance.

Tony Jenkins
A.Jenkins@hud.ac.uk

Java

We have now covered the "core" of Java.

We have two remaining things to do:

- Explore the (vast) library of classes available in Java.
- Explore ways to develop more sophisticated object interactions.

Remember that the "trick" in programming is to spot patterns.

Java

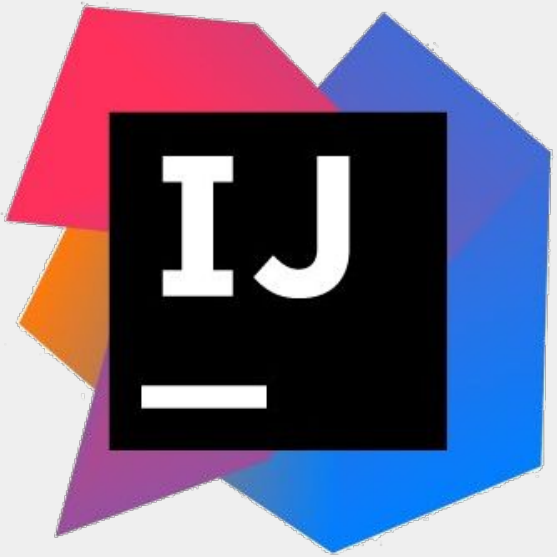
We have now covered the "core" of Java.

We have two remaining things to do:

- Explore the (vast) library of classes available in Java.
- **Explore ways to develop more sophisticated object interactions.**

Remember that the "trick" in programming is to spot patterns.

IntelliJ Demo Time



Assessment

The new solution is better, but not perfect.

We need to untangle how the two subclasses may have different `display` methods.

Trying to do so will also highlight other issues, which we will now try to sort out.

Inheritance …

So far we have seen that inheritance:

- Helps with avoiding code duplication.
- Promotes code reuse.
- Potentially allows for easier maintenance (DRY).
- Makes applications easier to extend.

Inheritance …

So far we have seen that inheritance:

- Helps with avoiding code duplication.
- Promotes code reuse.
- Potentially allows for easier m
- Makes applications easier to

DRY
"Don't Repeat Yourself."

Is really, really, very important.

DRY Programming

A very important principle is DRY:

"**D**on't **R**epeat **Y**ourself"

Usually stated as:

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

DRY Programming

A very important principle is DRY:

"**D**on't **R**epeat **Y**ourself"

Usually stated as:

"Every piece of knowledge must h
authoritative representati

Do it once, and the chances are it will be correct.

DRY Programming

A very important principle is DRY:

"**D**on't **R**epeat **Y**ourself"

Usually stated as:

"Every piece of knowledge must h
authoritative representati

Do it once, and the chances are it will be correct.
Worst case, if it's wrong, you only have to fix it in one place.

DRY Programming

A very important principle is DRY:

"**D**on't **R**epeat **Y**ourself"

Usually stated as:

"Every piece of knowledge must h
authoritative representati

It applies all across Computing, especially in networks, databases, and programming.

DRY Programming

A very important principle is DRY:

"**D**on't **R**epeat **Y**ourself"

Usually stated as:

"Every piece of knowledge must h
authoritative representati

Compare this with **WET**:
Write
Everything
Twice.

DRY Programming

A very important principle is DRY:

"**D**on't **R**epeat **Y**ourself"

Usually stated as:

"Every piece of knowledge must h
authoritative representati

Compare this with **WET**:
We
Enjoy
Typing.

DRY Programming

A very important principle is DRY:

"**D**on't **R**epeat **Y**ourself"

Usually stated as:

"Every piece of knowledge must h
authoritative representati

Compare this with **WET**:
Waste
Everyone's
Time.

Maximise Work

Yet another important principle:

"Maximise the amount of work not done."

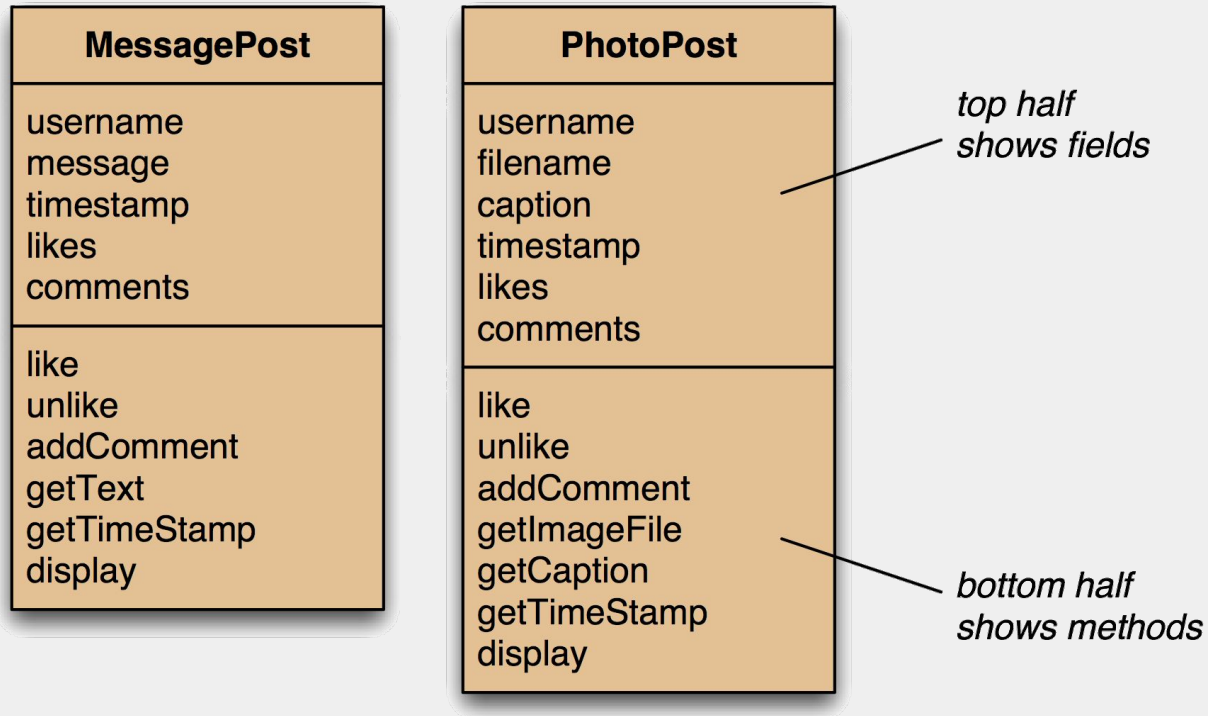
- Reuse code.
- Spot patterns.
- Be DRY.
- Design with change in mind.
- Be "Agile".

More Inheritance

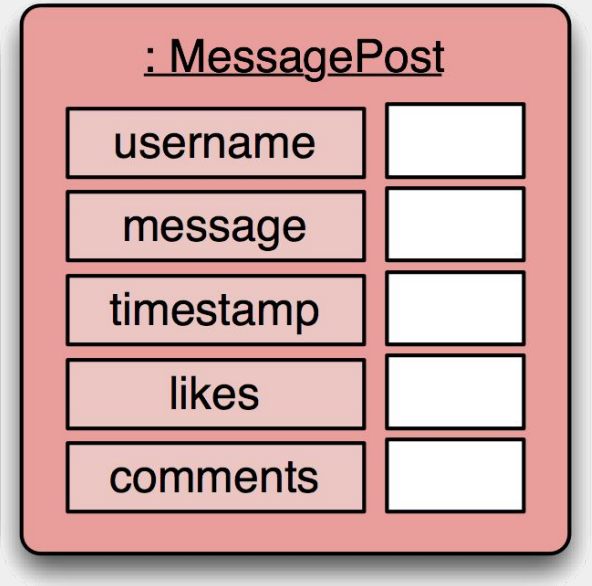
So, today we extend our knowledge of inheritance by looking at:

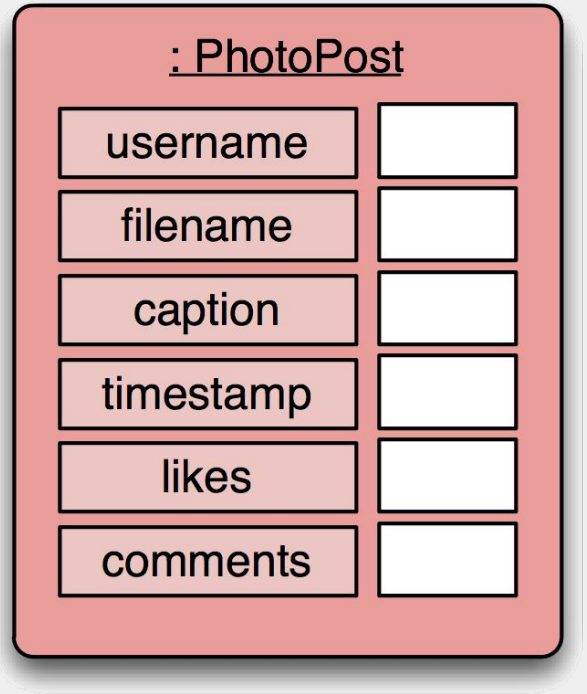
- Subtyping.
- Substitution.
- Polymorphic variables.
- Casting.
- `toString` in inheritance.

Network Classes

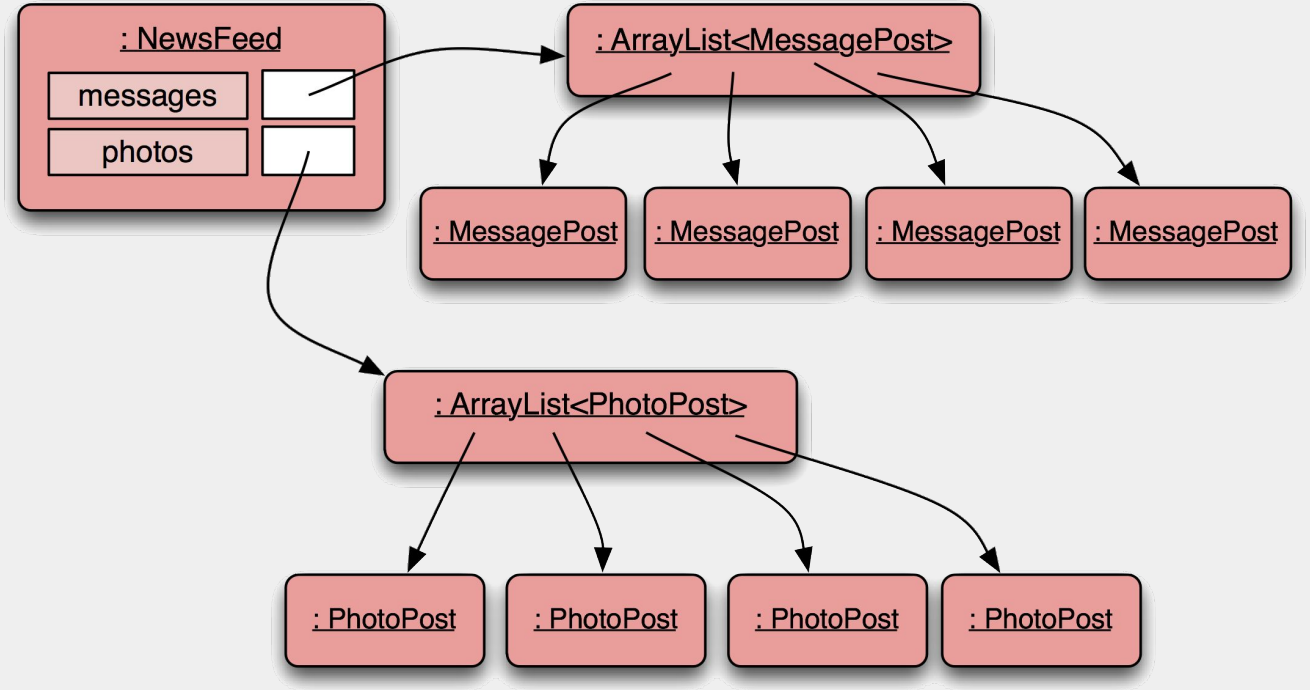


Network Objects

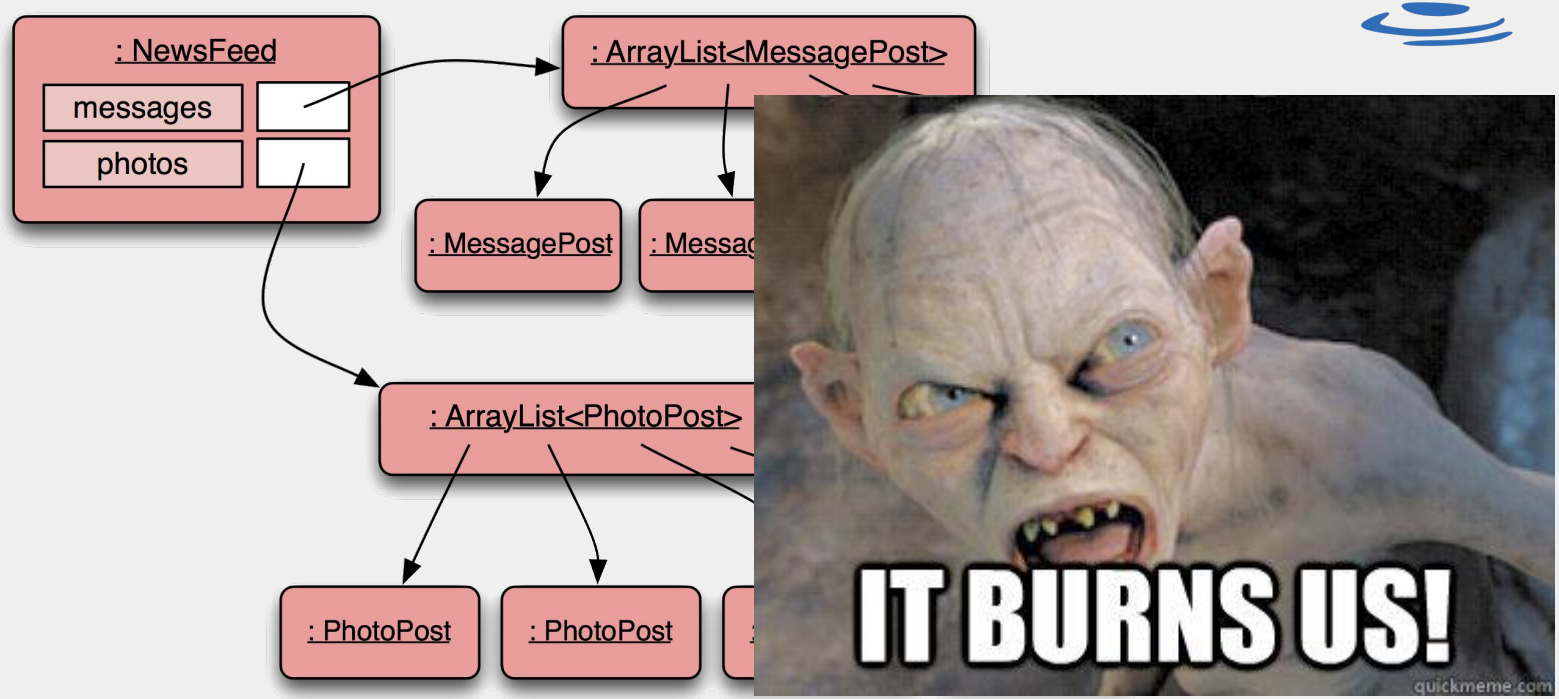




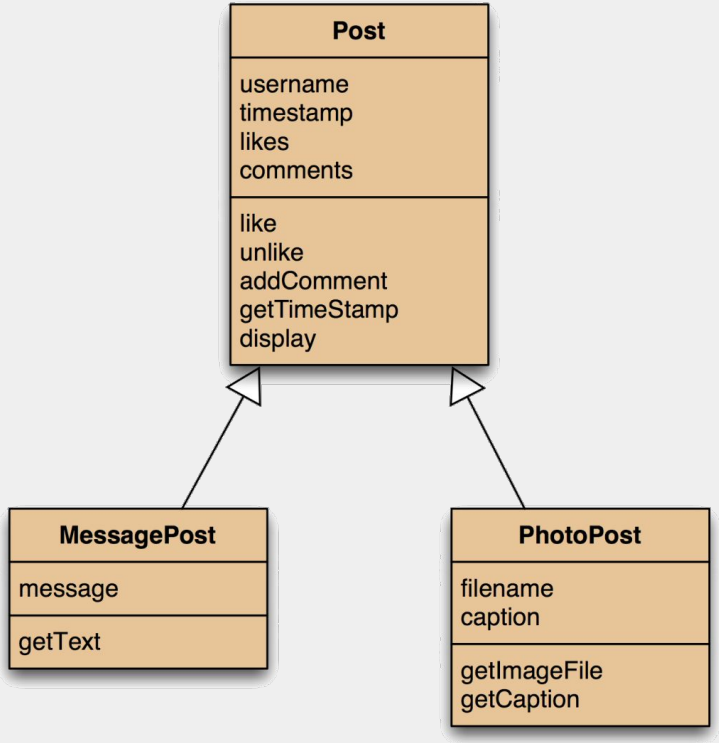
Network Object Model



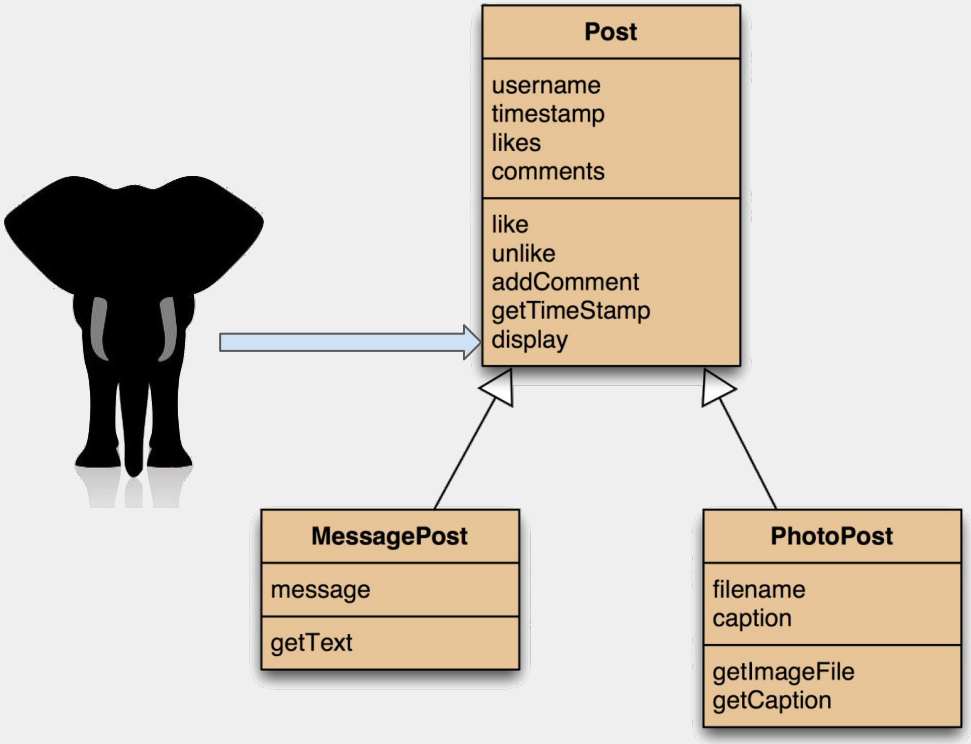
Network Object Model



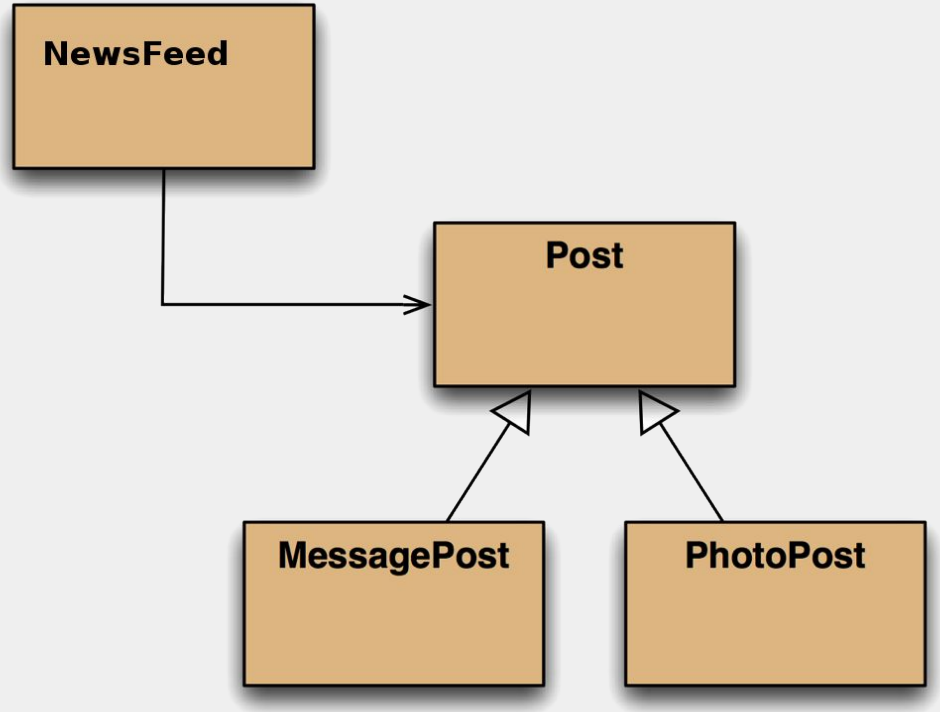
Using Inheritance



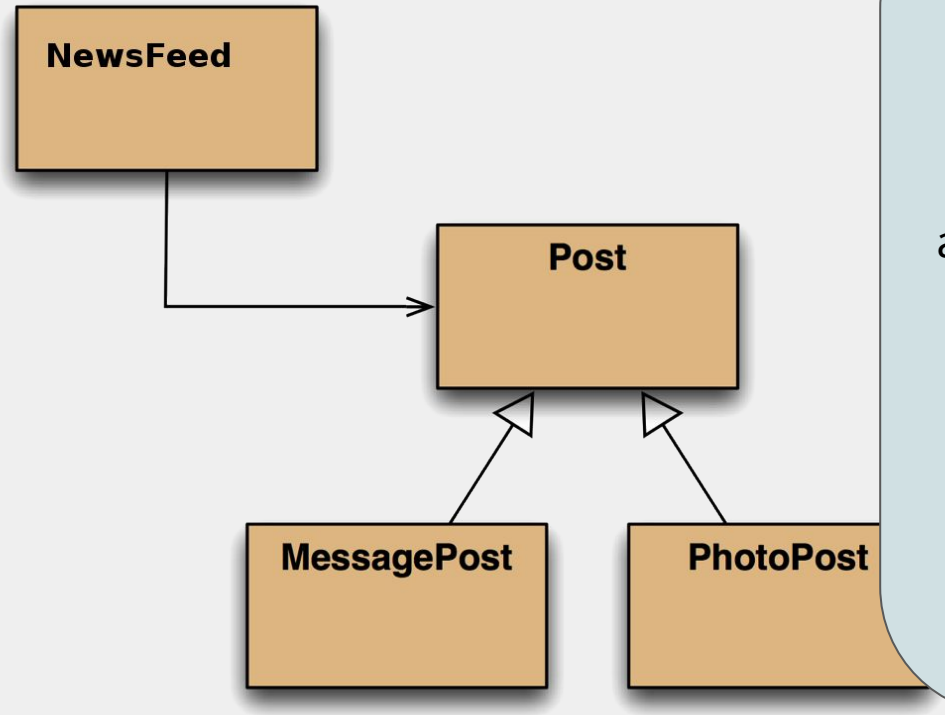
Using Inheritance



Class Diagram

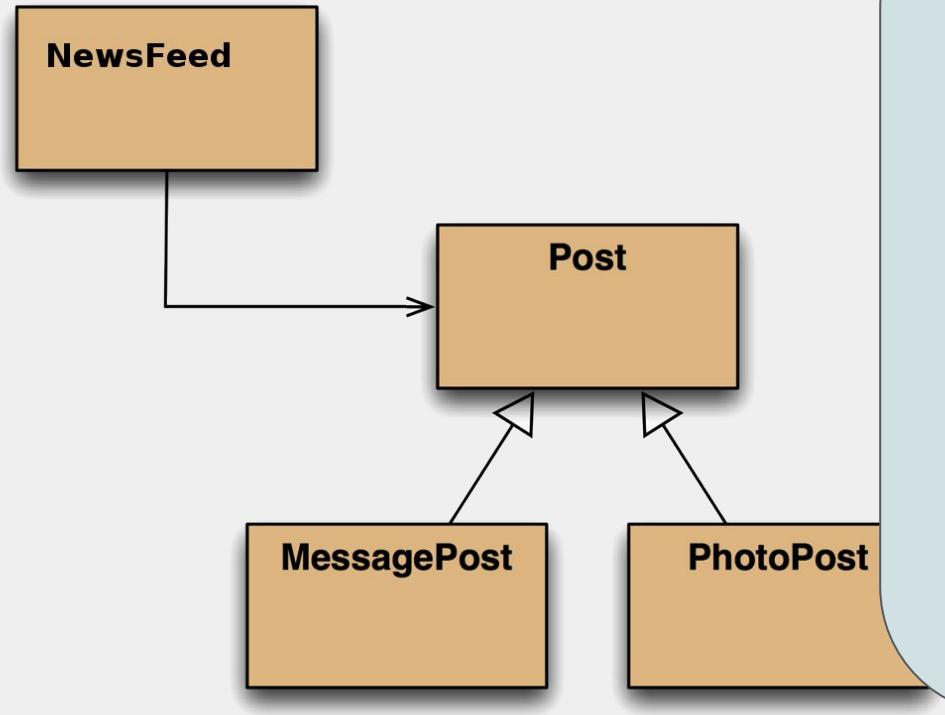


Class Diagram



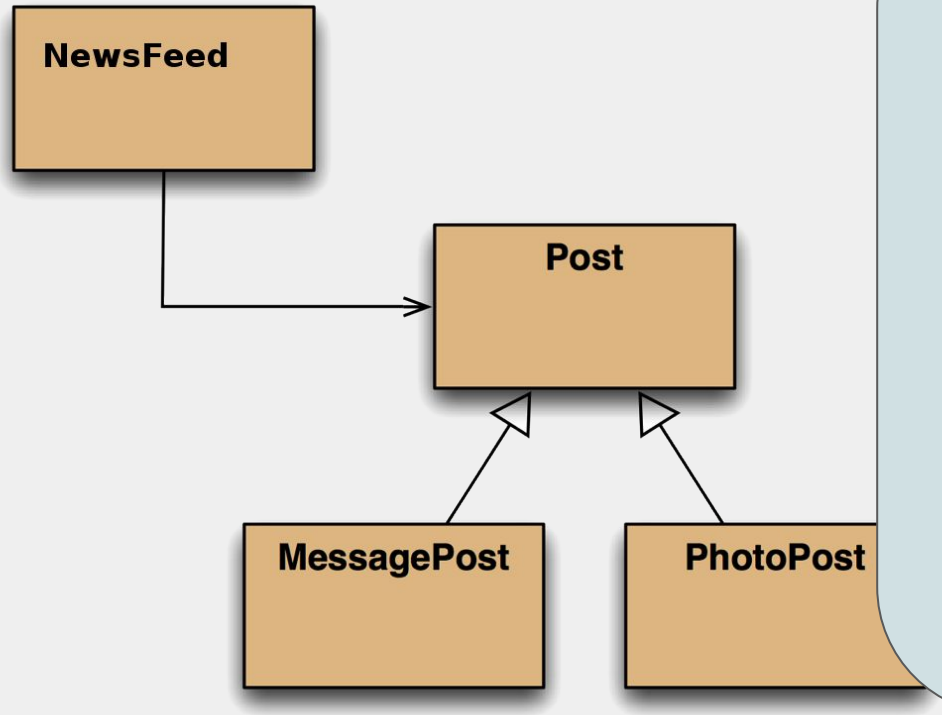
Note that `NewsFeed` is only aware of `Post`: it knows nothing of `MessagePost` and `PhotoPost`.

Class Diagram



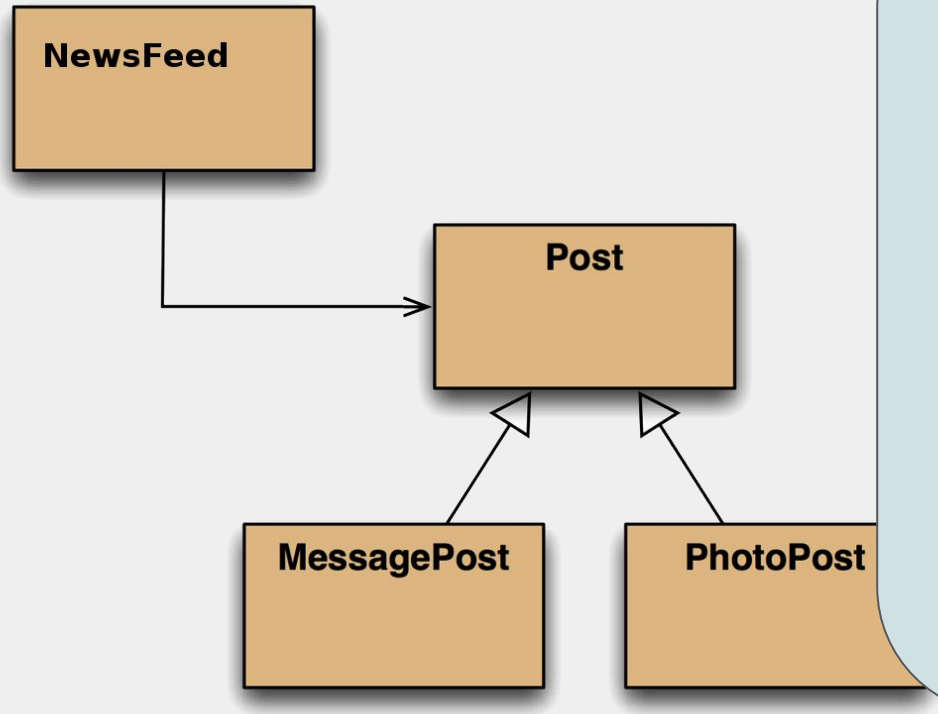
So if `NewsFeed` encounters a display method call it will look for it in `Post`: it cannot look lower down because it knows nothing of `MessagePost` and `PhotoPost`.

Class Diagram



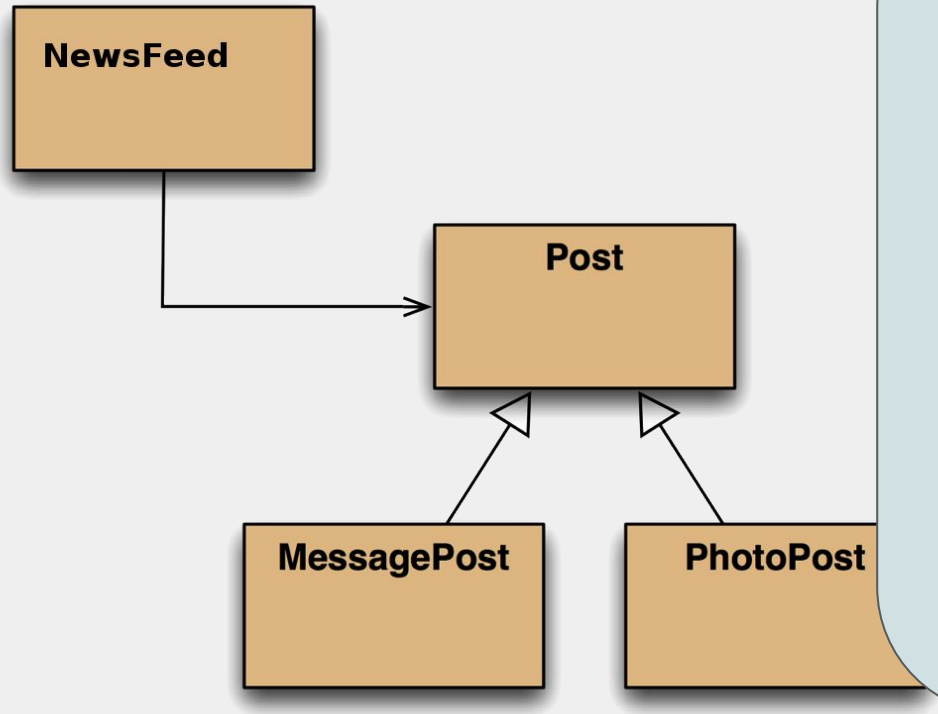
Likewise `MessagePost` and `PhotoPost` have no idea they are being used in a `NewsFeed`.

Class Diagram



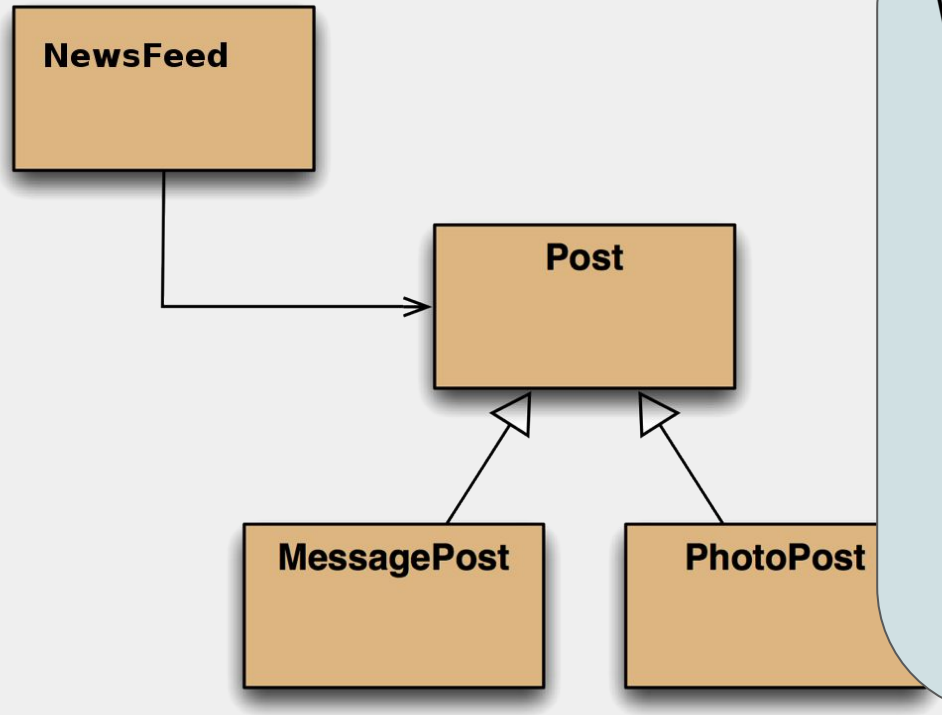
And `Post` is totally oblivious of the fact that there are other classes inheriting from it.

Class Diagram



All these things are, of course, *exactly* what we want.

Class Diagram

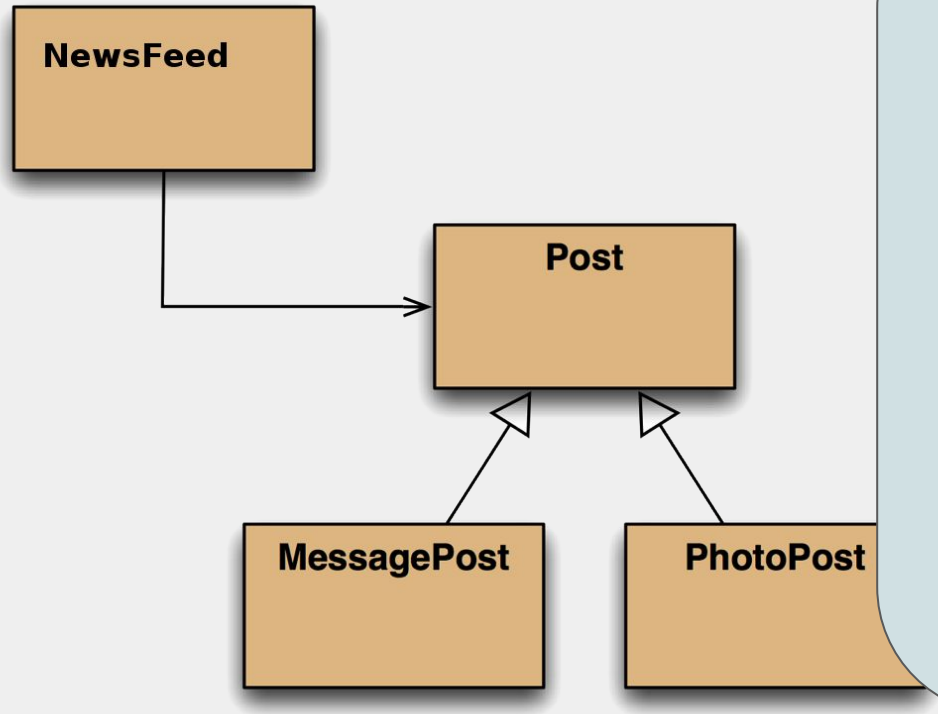


We noted last week that there is, in fact, no such thing as a `Post`.

To be formal, it's an *abstract* class.

In practical terms, this just means that we can't create an instance of the class (just its subclasses).

Class Diagram



We see that we can add new types of `Post` without changing *any* of the existing classes.

Cool Beans!

NewsFeed

```
public class NewsFeed
{
  private ArrayList <Post> posts;

  public NewsFeed ()
  {
    posts = new ArrayList <Post> ();
  }

  public void addPost (Post newPost)
  {
    posts.add (newPost);
  }
}
```

So NewsFeed deals only in Post objects.

It is totally unaware that there are different sorts.

NewsFeed

```
public class NewsFeed
{
  private ArrayList <Post> posts;

  public NewsFeed ()
  {
    posts = new ArrayList <Post> ();
  }

  public void addPost (Post newPost)
  {
    posts.add (newPost);
  }
}
```

But the Post object used here as a parameter could be a MessagePost or a PhotoPost.

NewsFeed

```
public class NewsFeed
{
  private ArrayList <Post> posts;

  public NewsFeed ()
  {
    posts = new ArrayList <Post> ();
  }

  public void addPost (Post newPost)
  {
    posts.add (newPost);
  }
}
```

But the Post object used here as a parameter could be a MessagePost or a PhotoPost.

And, as we know, it could never be an actual Post object.

NewsFeed

```
public class NewsFeed
{
  private ArrayList <Post> posts;

  public NewsFeed ()
  {
    posts = new ArrayList <Post> ();
  }

  public void addPost (Post newPost)
  {
    posts.add (newPost);
  }
}
```

But the `Post` object used here as a parameter could be a `MessagePost` or a `PhotoPost`.

Something clever is clearly going on here, but what?

NewsFeed

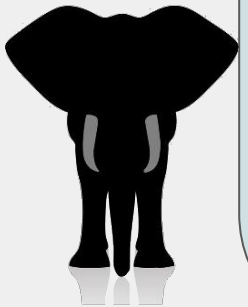
```
public void show ()
{
  for (Post p: posts) {
    p.display ();
    System.out.println ();
  }
}
```

This loop works through all the `Post` objects.

Some will be `MessagePost` and some will be `PhotoPost`, which are (ideally) displayed in different ways.

NewsFeed

```
public void show ()
{
  for (Post p: posts) {
    p.display ();
    System.out.println ();
  }
}
```



This loop works through all the `Post` objects.

Some will be `MessagePost` and some will be `PhotoPost`, which are (ideally) displayed in different ways.

Adding a Post

```
public void addMessagePost (MessagePost m)
public void addPhotoPost (PhotoPost p)
```

In the beginning there were two methods to add posts to the feed.

Adding a Post

```
public void addMessagePost (MessagePost m)
public void addPhotoPost (PhotoPost p)
```

In the beginning there were two methods to add posts to the feed.

Adding a Post

~~public void addMessagePost (MessagePost m)~~
~~public void addPhotoPost (PhotoPost p)~~

```
public void addPost (Post p)
```

These have now gone, replaced with a single method.

Adding a Post

~~public void addMessagePost (MessagePost m)~~
~~public void addPhotoPost (PhotoPost p)~~

```
public void addPost (Post p)

PhotoPost p = new PhotoPost (...);
feed.add (p);
MessagePost m = new MessagePost (...);
feed.add (m);
```

And this new method works with either `MessagePost` or `PhotoPost`.

Adding a Post

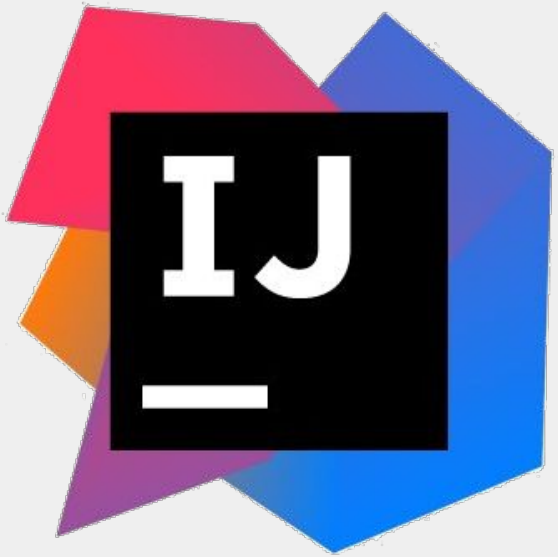
~~public void addMessagePost (MessagePost m)~~
~~public void addPhotoPost (PhotoPost p)~~

```
public void addPost (Post p)

PhotoPost p = new PhotoPost (...);
feed.add (p);
MessagePost m = new MessagePost (...);
feed.add (m);
```

And this new method works with either `MessagePost` or `PhotoPost`.

IntelliJ Demo Time



Subclasses and Subtyping

So:

- Classes define types.
- Subclasses define subtypes.
- Objects of subclasses can be used wherever objects of their superclasses are required.

This is called *substitution*.

Adding a Post

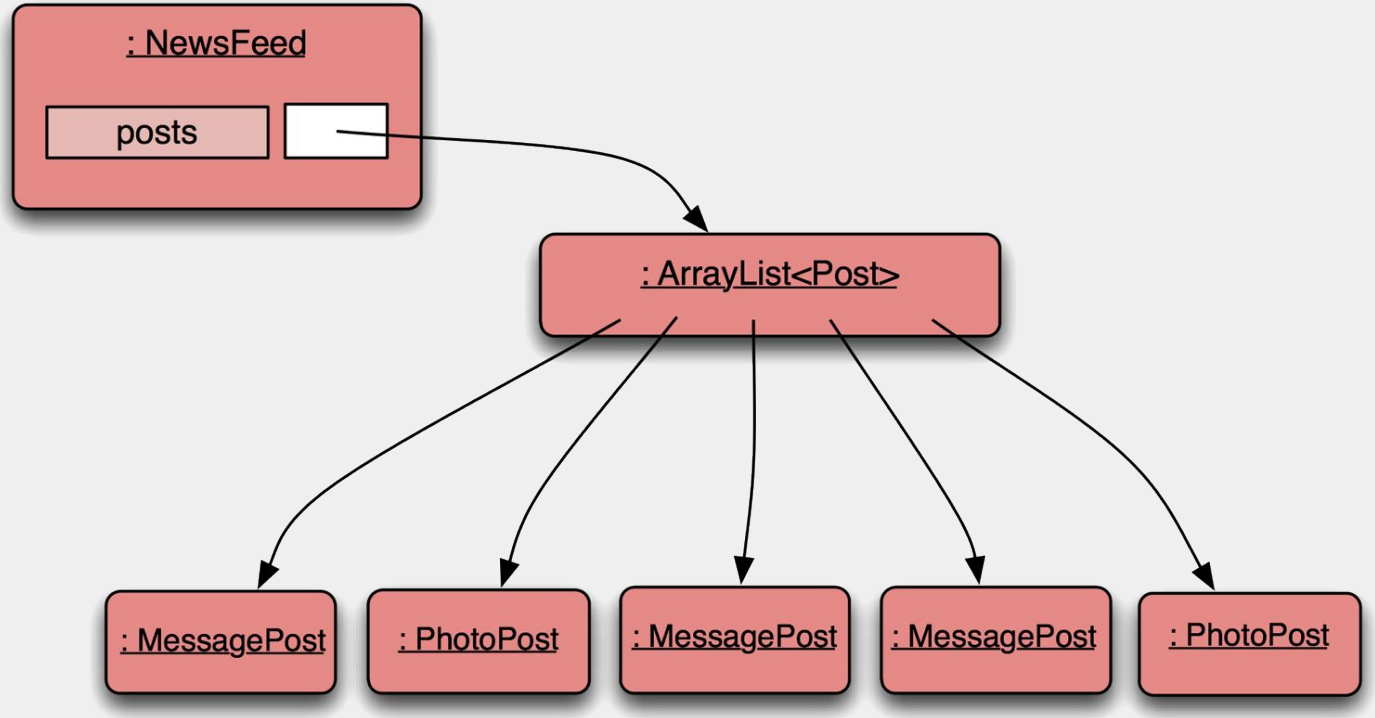
~~public void addMessagePost (MessagePost m)~~
~~public void addPhotoPost (PhotoPost p)~~

```
public void addPost (Post p)

PhotoPost p = new PhotoPost (...);
feed.add (p);
MessagePost m = new MessagePost (...);
feed.add (m);
```

So this is substitution. The code uses subclass objects (`PhotoPost` or `MessagePost`) where the parameter declares that a supertype object (`Post`) is required.

Network Object Model



Working with Types

Subclass objects may be assigned to superclass variables.

```
Post p1 = new Post (...);
Post p2 = new MessagePost (...);
Post p3 = new PhotoPost (...);
```

These can all be used wherever the superclass is required.

Working with Types

Subclass objects may be assigned to superclass variables.

```
Post p1 = new Post (...);
Post p2 = new MessagePost (...);
Post p3 = new PhotoPost (...);
```

These can all be used wherever the superclass is required.

Polymorphic Variables

Object variables in Java are *polymorphic*.

This means they can hold objects of more than one type:

- The originally declared type.
- Any subtype of the declared type.

So it would be possible to declare a `Post`, and later turn it into a `MessagePost`, and later into a `PhotoPost`.

Casting

We can assign subtype to supertype …

… but not the other way round.

```
Post p;
PhotoPost pp = new PhotoPost (...);
p = pp;     // Fine.
pp = p;     // Compile-time error.
```

Casting

We can assign subtype to supertype

… but not the other way round.

```
Post p;
PhotoPost pp = new PhotoP
p = pp;     // Fine.
pp = p;     // Compile-tim
```

If this is necessary, and p really is a `PhotoPost`, then *casting* fixes it:

```
pp = (PhotoPost) p;
```

Casting

The required object type is specified in brackets.

The object is not changed in any way.

A runtime check is made to ensure that the object really is of that type (an exception is thrown if not).

Use sparingly. But sometimes it is essential so as to avoid very ugly code.

Casting

The required object type is specified in brackets.

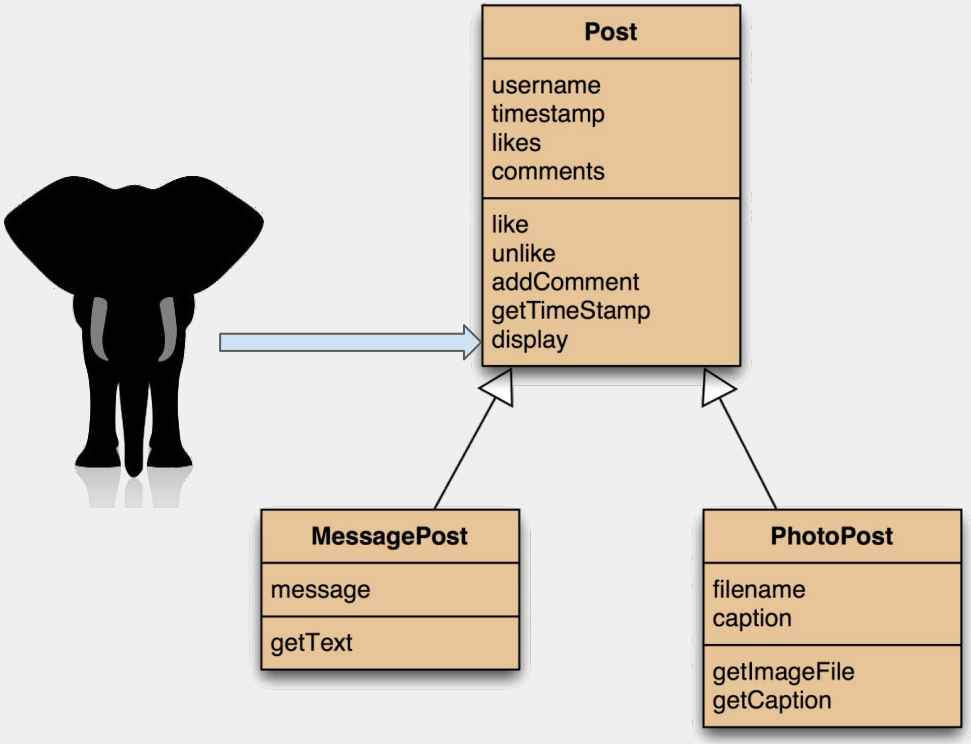
The object is not changed in any way.

A runtime check is made to ensure that the object really is of that type (an exception is thrown if not).

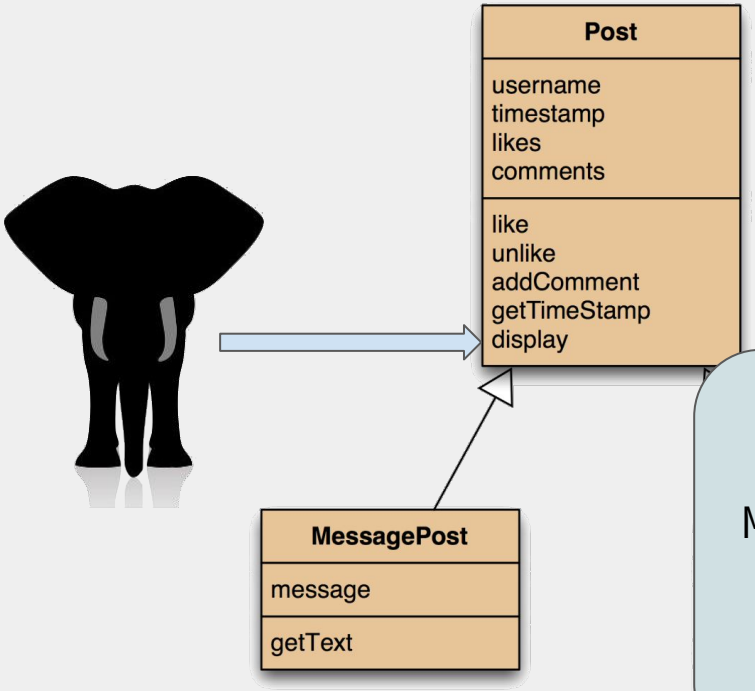
Use sparingly. But sometimes it is es
ugly code.

As an aside, it follows that we sometimes need a way to find out what type an object currently is …

Using Inheritance

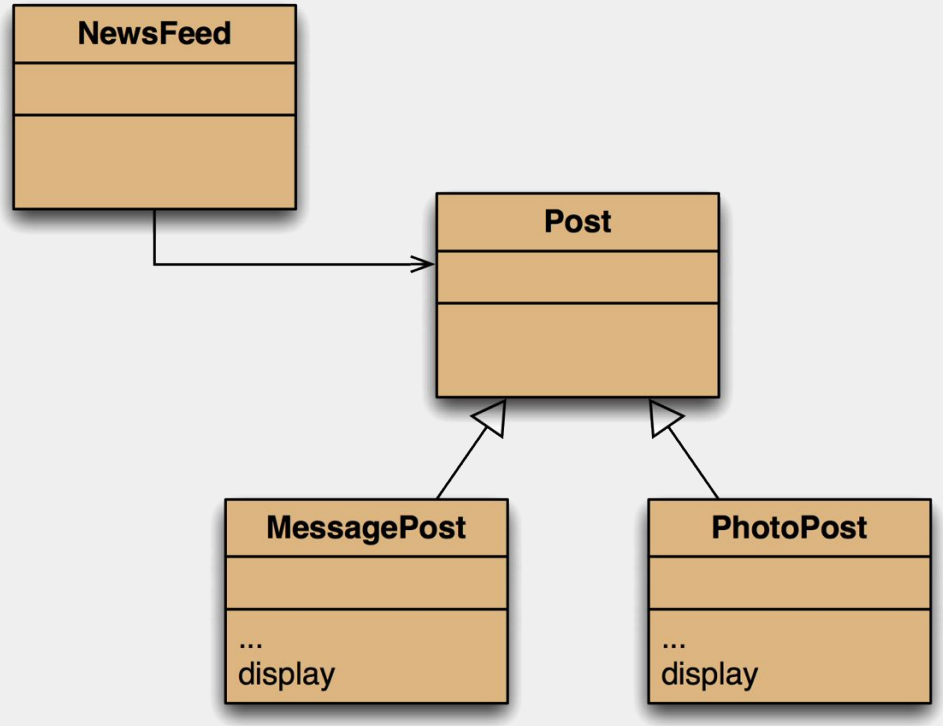


Using Inheritance



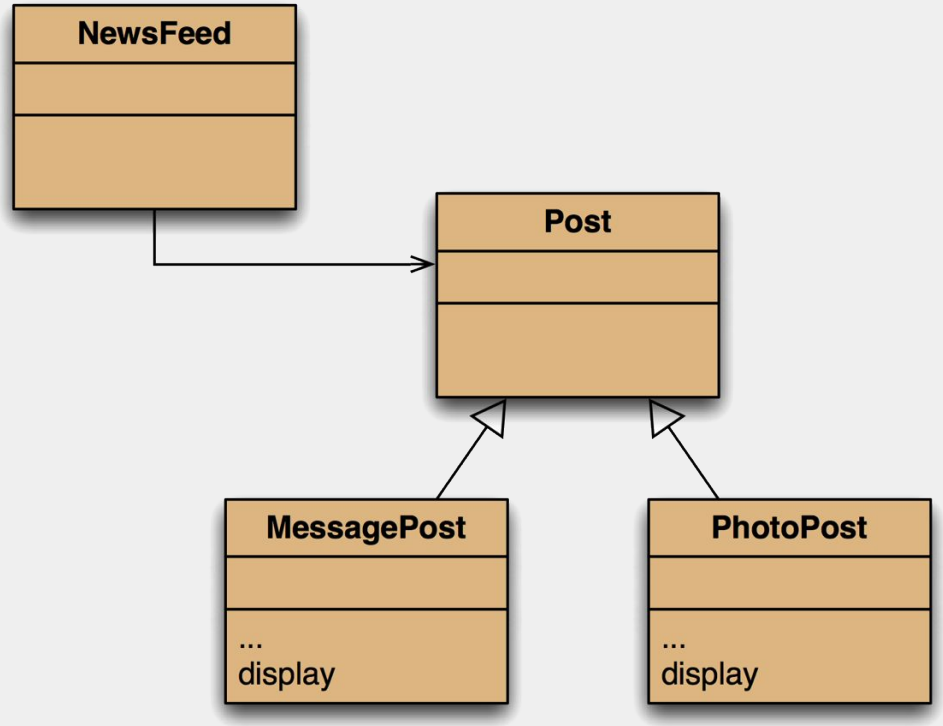
Our problem is that both `MessagePost` and `PhotoPost` have a `display` method, but at present it is the *same* method.

Using Inheritance



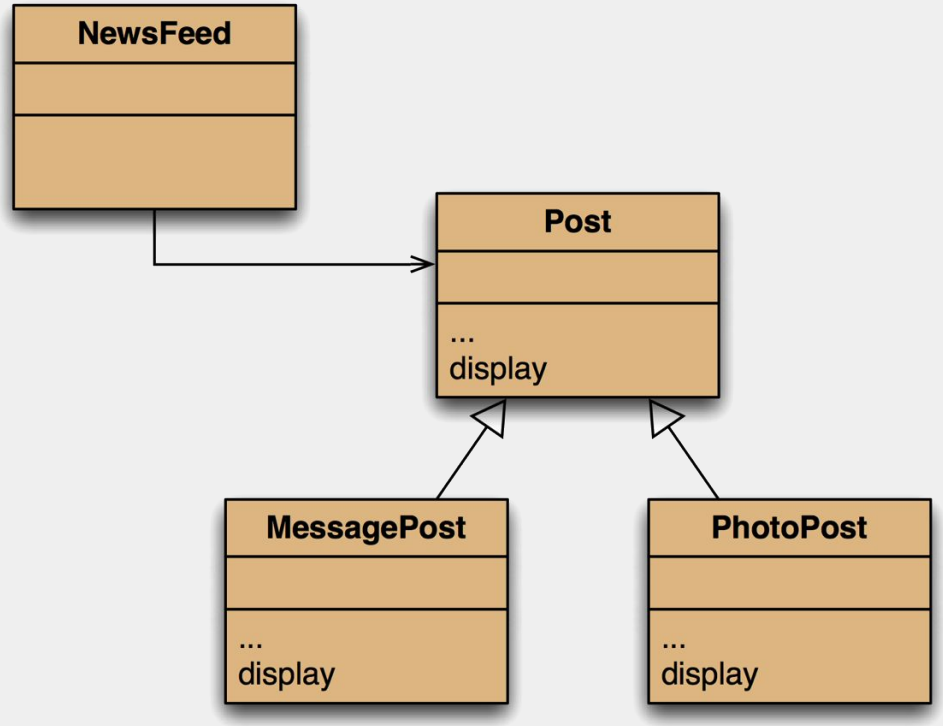
An obvious solution is to move the display methods into the subclasses, like this.

Using Inheritance



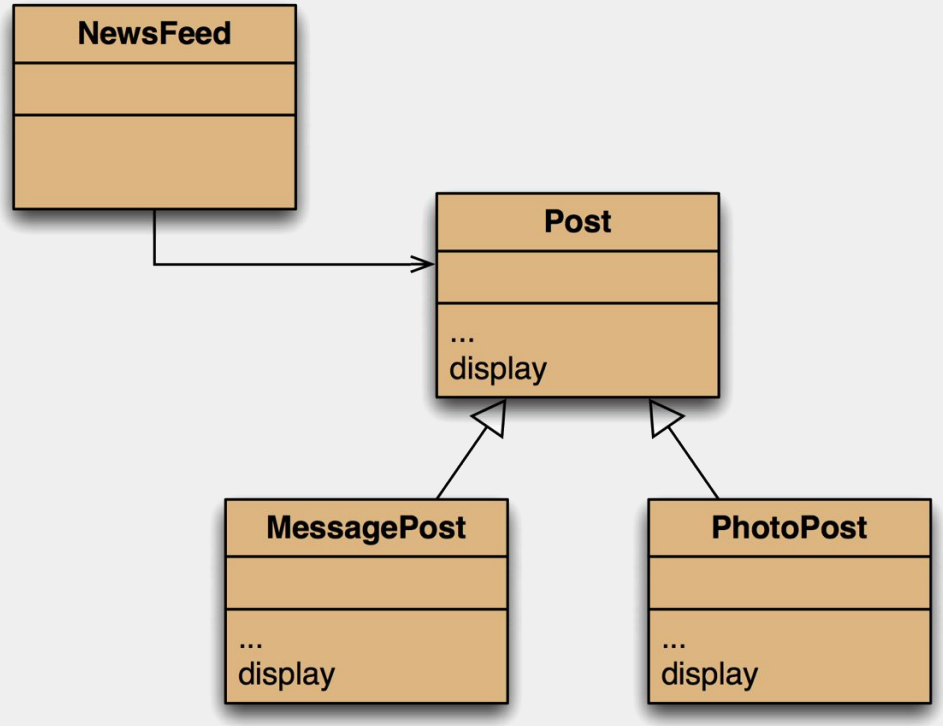
Unfortunately, this breaks the `NewsFeed` code because it deals only in `Post` objects, which no longer have a `display` method.

Using Inheritance



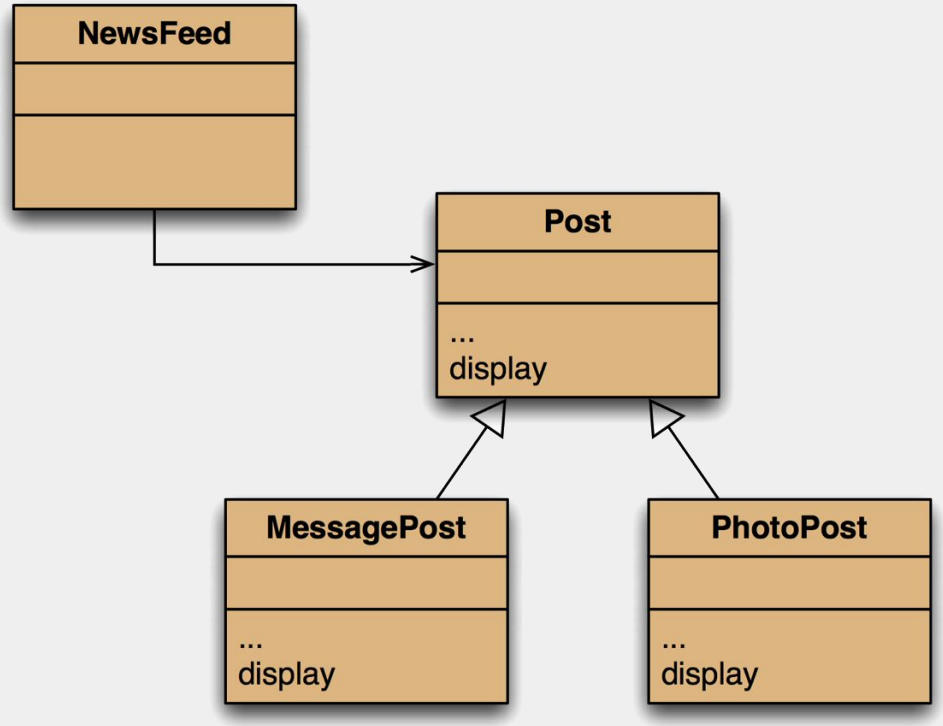
So, this is a solution that will work.
`Post` has a `display` method, which we *override* in the subclasses.

Super



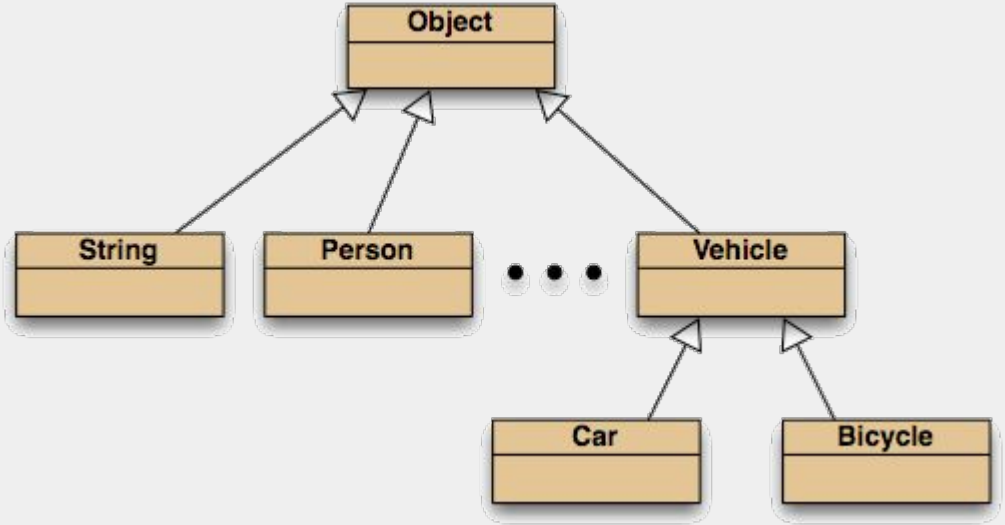
If needed, the subclasses can call any method in the superclass using a `super` call.

Super



What we have here is an example of overriding (which we will look at properly next time).

Overriding



But we have been using this idea all along.

All classes inherit (eventually) from `Object`.

`Object` defines (among a few other things), `toString`.

IntelliJ Demo Time