University of HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 17: Polymorphism

Pretty polymorphism.

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.

Before We Start: The Assignment

1. Read the Spec.
2. Design first.
3. Start small.
4. Iterate: build gradually.
5. Test as you develop (and test your mate's).
6. Use online sources (StackOverflow is your friend).
7. Think about the demo (test data).
8. Get feedback.

Source Code Control

Your final system will probably consist of many source code files.

And as you program you will generate many *versions* of these files.

You will need to have some sort of *control* over these versions. Learning a little about this has the potential to save a lot of pain.


DRIVE BROKEN? NO BACKUP?
GOOD.
imgflip.com

Git

Long ago, we touched on Git.

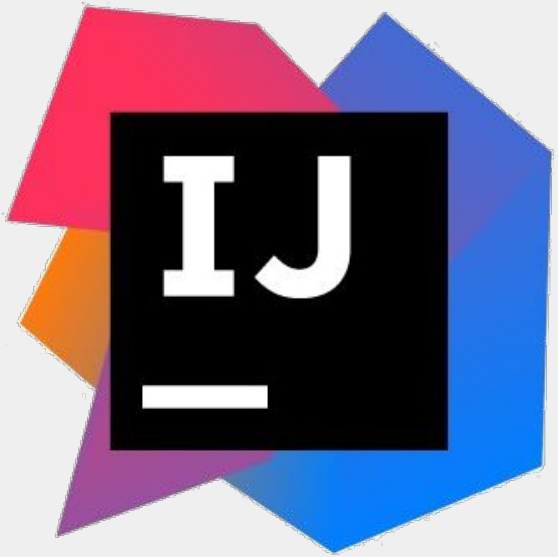
Storing your assignment code in Git (on GitHub, GitLab, etc.) has the potential to save you a lot of pain.

Git can be used from the command-line, or via a GUI.

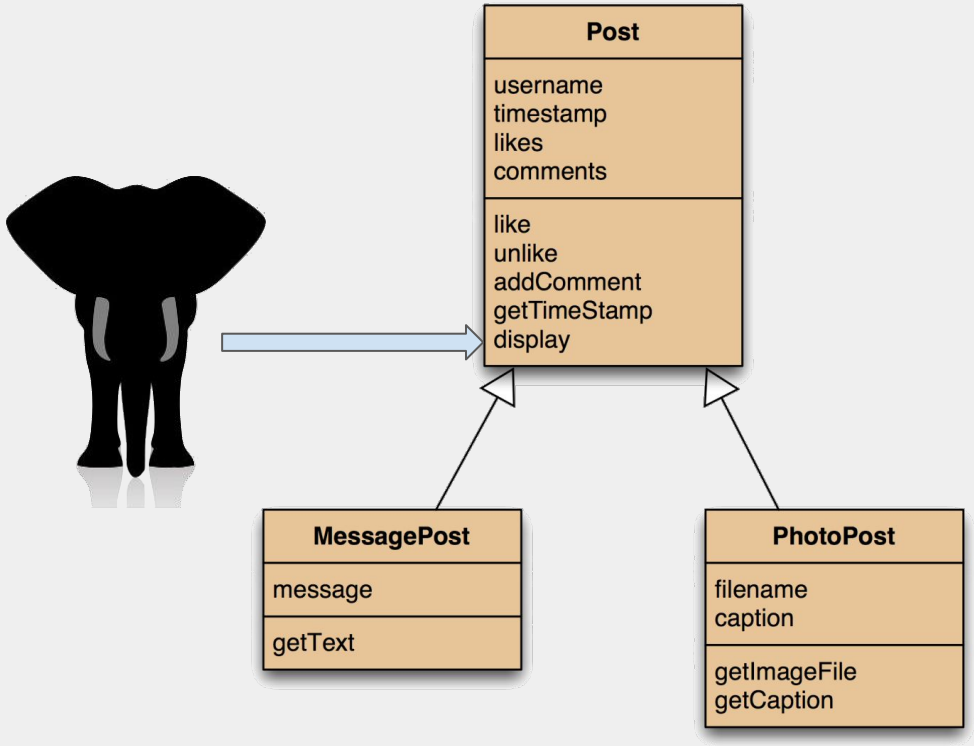
Good editors (Atom) and IDEs (IntelliJ, Eclipse) offer integration with Git.

There are also good standalone GUIs: GitKraken.

IntelliJ Demo Time



Using Inheritance



The Problem

```
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.
```

We want output like this, with slightly different display for the two `Post` types.

Inheritance So Far

Inheritance helps reduce code duplication.

Classes define types. Subclasses define subtypes.

Substitution:

- Objects of subclasses can be used where objects of superclasses are required.

Object variables are polymorphic:

- They can hold objects of more than one type.

Today's Topics

Today we extend our knowledge of inheritance to include:

- Method Polymorphism.
- Static and Dynamic Type.
- Overriding.
- Dynamic Method Lookup.
- Protected Access.

Today's Topics

Today we extend our knowledge of inheritance to include:

- Method Polymorphism.
- Static and Dynamic Type.
- Overriding.
- Dynamic Method Lookup.
- Protected Access.

Actually, we have met most of these over the past few weeks, so we're really just formalising it all.

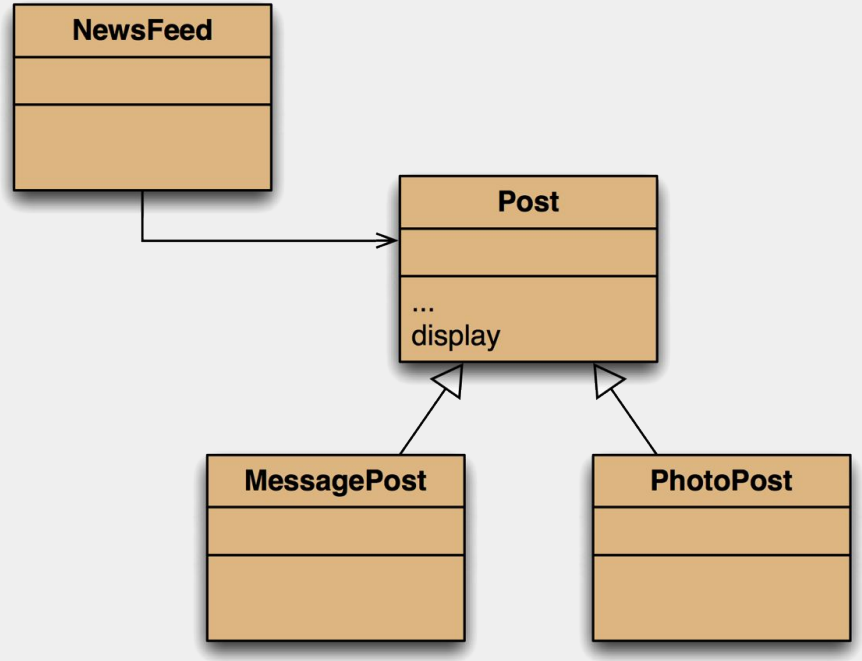
Today's Topics

Today we extend our knowledge of inheritance to include:

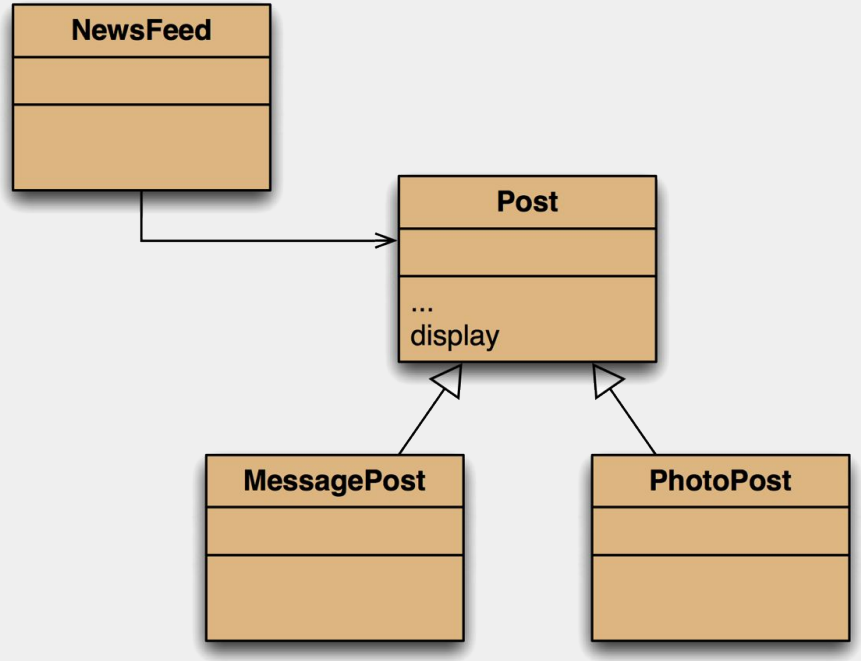
- Method Polymorphism.
- Static and Dynamic Type.
- Overriding.
- Dynamic Method Lookup.
- Protected Access.

We're also going to meet the idea of an *abstract method*, which will give us the final version of the social news feed app.

The Inheritance Hierarchy

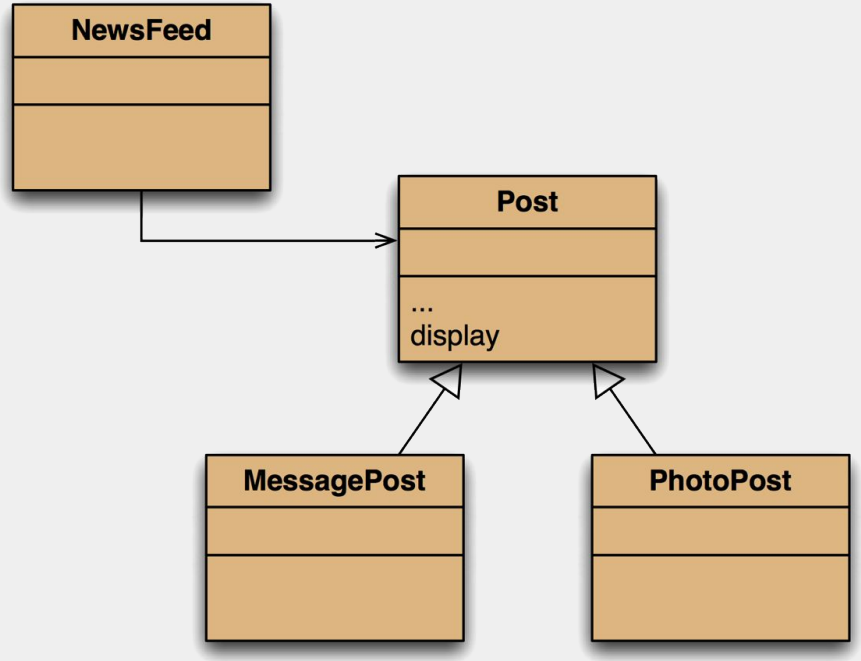


The Inheritance Hierarchy



`MessagePost` and `PhotoPost` inherit all the methods from `Post`.
They inherit all the instance variables too, but they can only access the *public* parts directly.

The Inheritance Hierarchy



Our problem all along has been that the display method needs to be different for the two types of `Post`.

The Problem

`Post` knows nothing of `MessagePost` and `PhotoPost`.

So the `display` method there cannot use any instance variables from the subclasses.

So, with this hierarchy, it can display only the common instance variables: author, likes and comments.

And it displays the same for both types of post.

The Problem

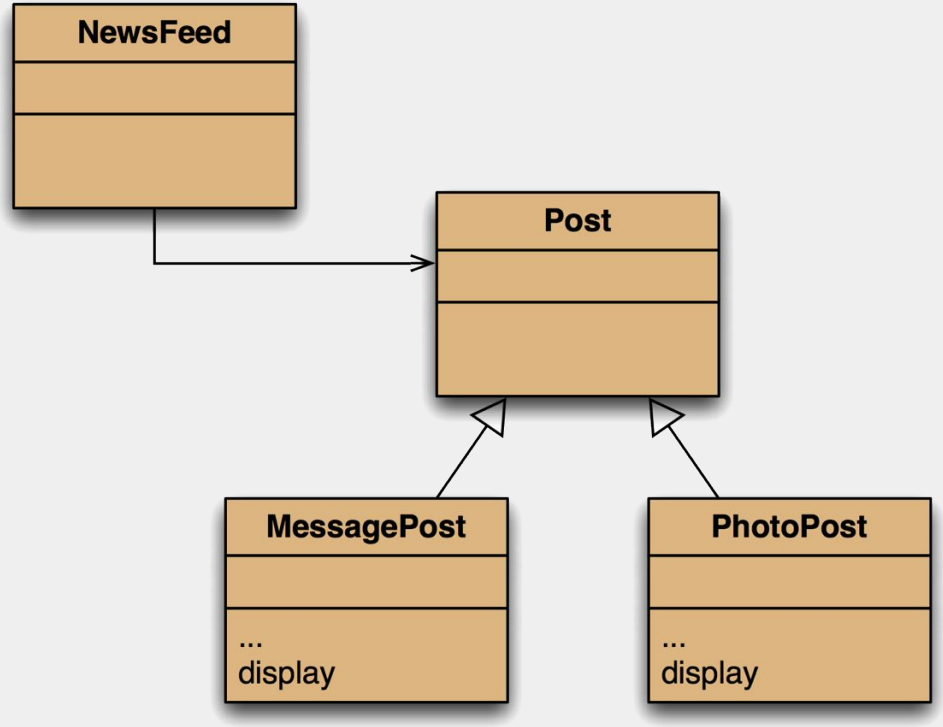
The `display` method in `Post` prints only the common fields.

Inheritance is a one-way street:

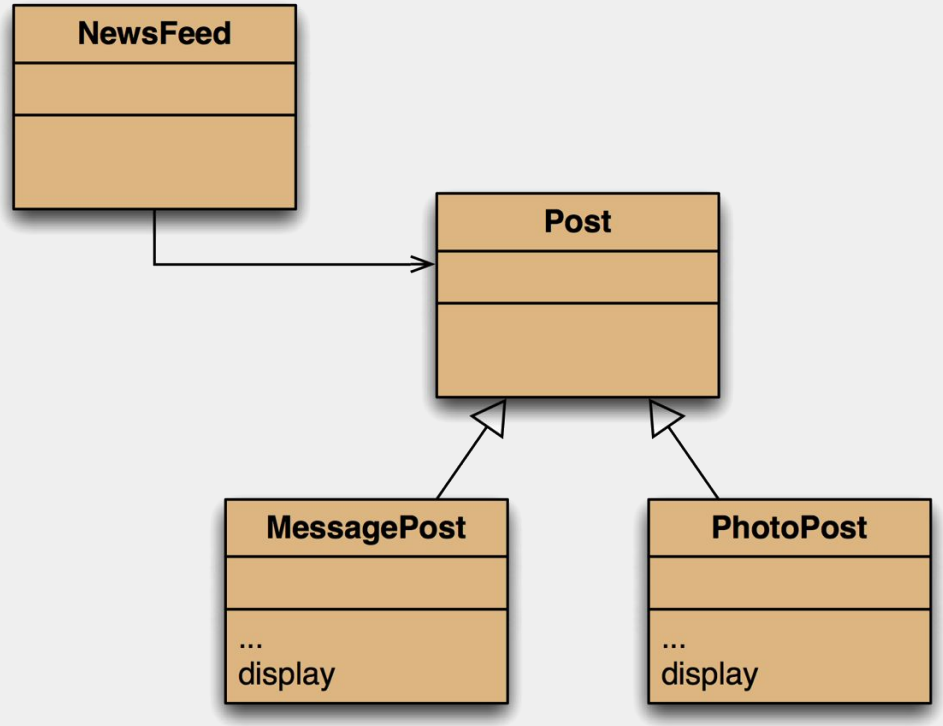
- A subclass inherits the superclass fields.
- The superclass knows nothing about fields in the subclass.

(This is, of course, precisely what we want.)

Attempting a Solution

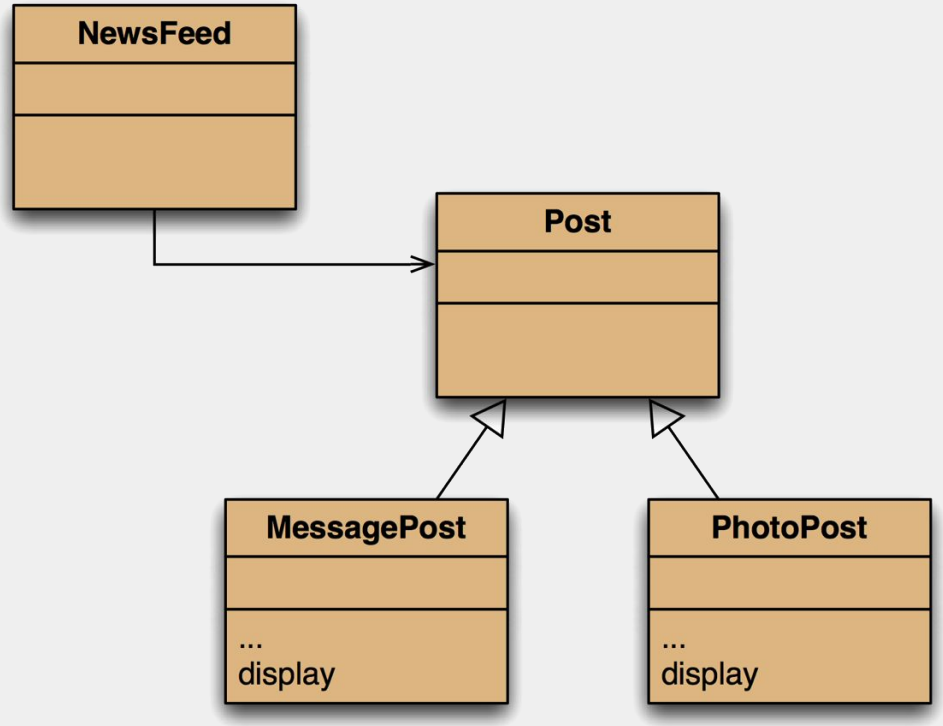


Attempting a Solution



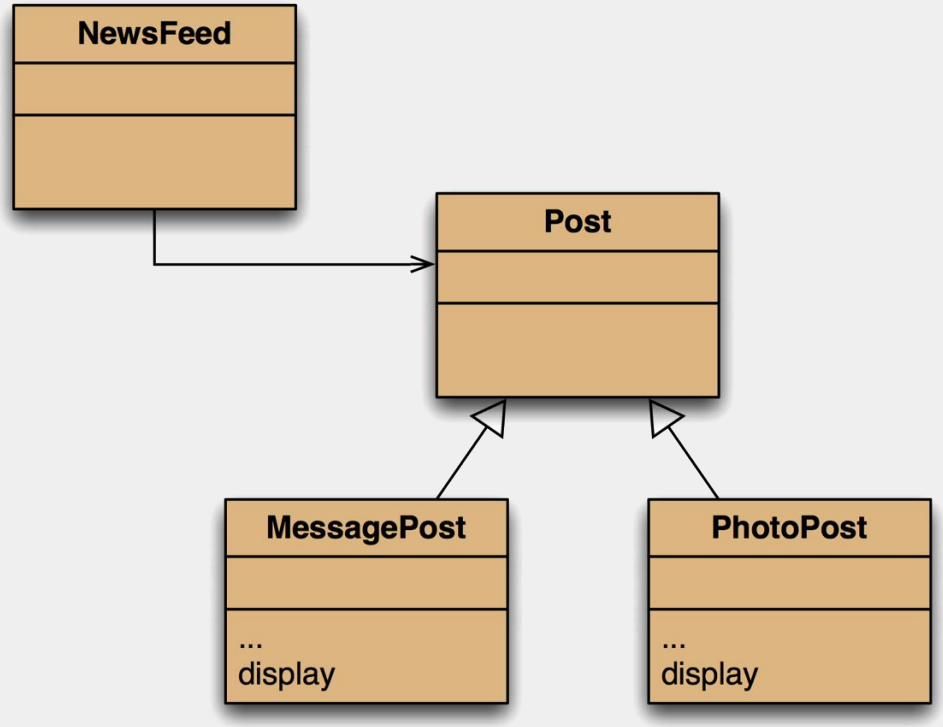
So we moved `display` to a spot where it can see all the instance variables it needed.

Attempting a Solution



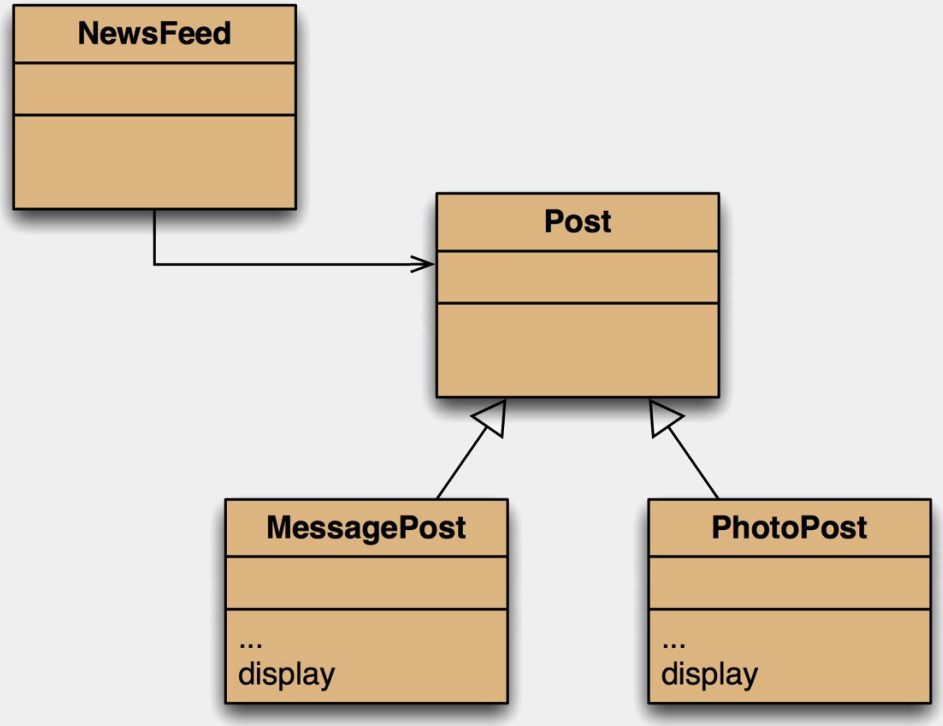
But it turns out that it can't access those in `Post` at all (they're private).
So it has to access them via methods (which are public).

Attempting a Solution



But in any case, this solution breaks because NewsFeed can no longer see a `display` method at all.

Attempting a Solution

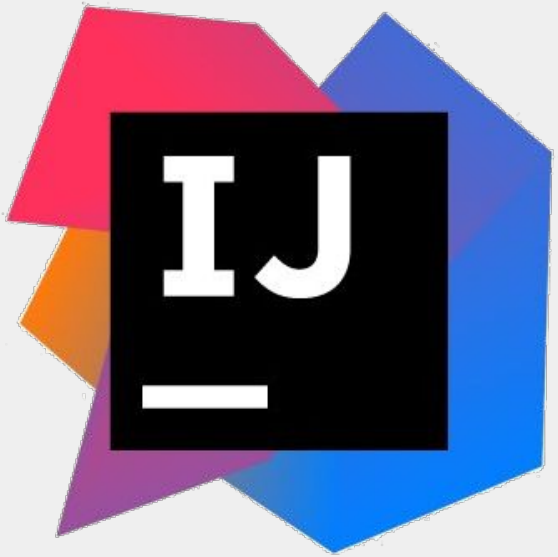


An (ugly) fix was to put a dummy method in `Post`:

```
public void display () {}
```

This does work, but is not the neatest way.

IntelliJ Demo Time



Static and Dynamic Type

To explain what is going on (and how to fix it), we need some new concepts:

- Static Type.
- Dynamic Type.
- Method Despatch / Lookup.
  - That is, how Java decides which method to call, bearing in mind we now have multiple methods with the same signature.

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`.

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.

Or, as we have seen, use a `MessagePost` where a `Post` 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.

But obviously an object has only one type at any point in time!

Static and Dynamic Type

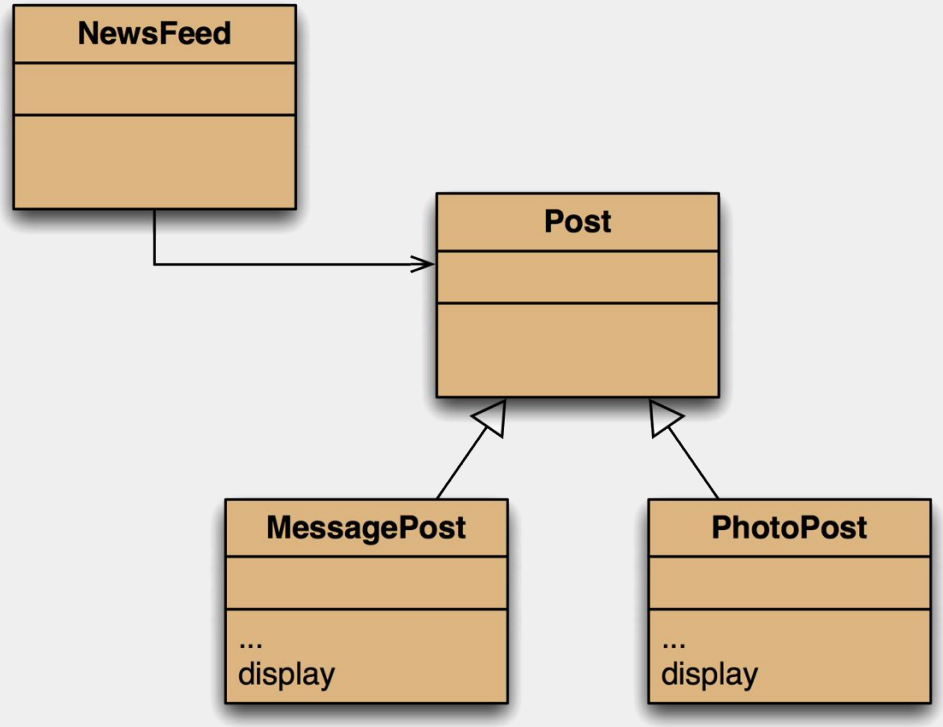
The declared type of a variable is its *static* type.

The type of the object a variable refers at any point during runtime to is its *dynamic* type.

The compiler's job is to check for static-type violations.

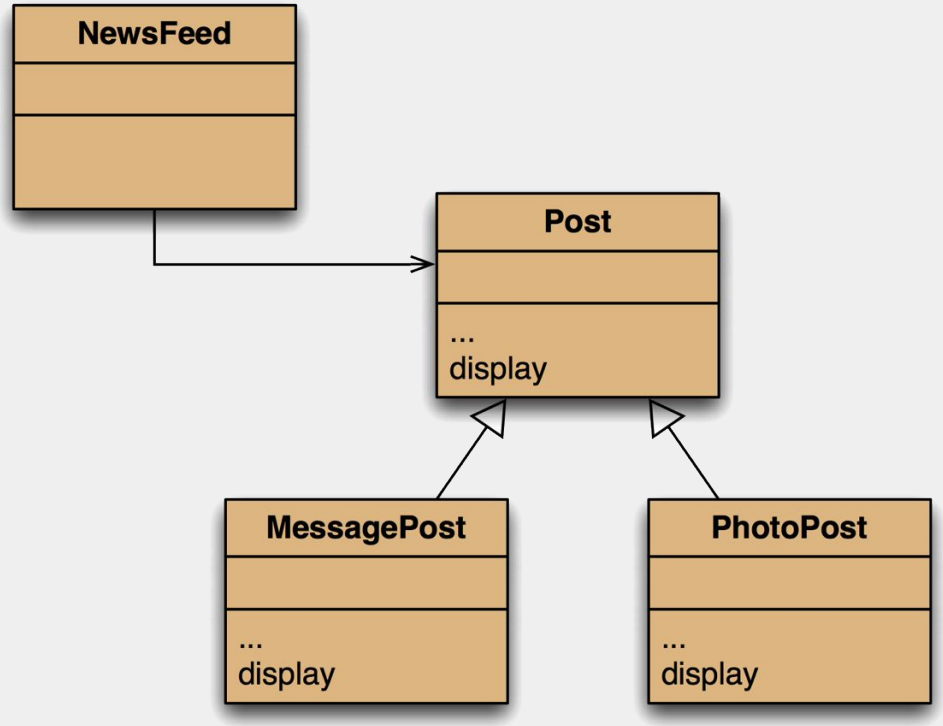
Dynamic-type violations tend to result in program failure …

The Solution



So this version was failing static type checking.
Our hack of the dummy method was just there to satisfy the compiler.

The Solution



But doing just that is actually close to the right solution.
We declare a `display` method in all three classes.

Overriding

The superclass and subclass define methods with the same *signature*.

Each has access to the fields of its class.

The superclass method satisfies the static type check.

The subclass method is called at runtime: it *overrides* the superclass version.

Overriding

The superclass and subclass define methods with the same *signature*.

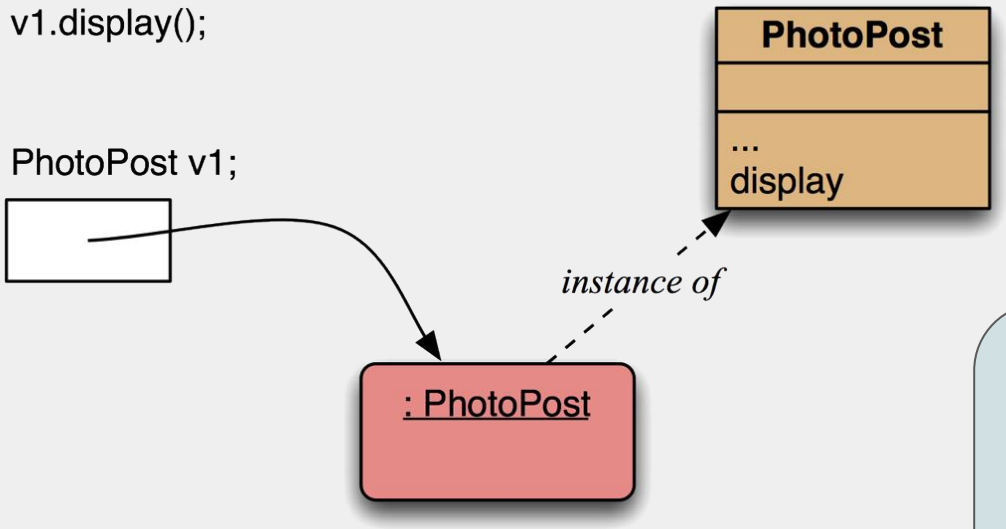
Each has access to the fields of its class.

The superclass method satisfies the

The subclass method is called at run
superclass version.

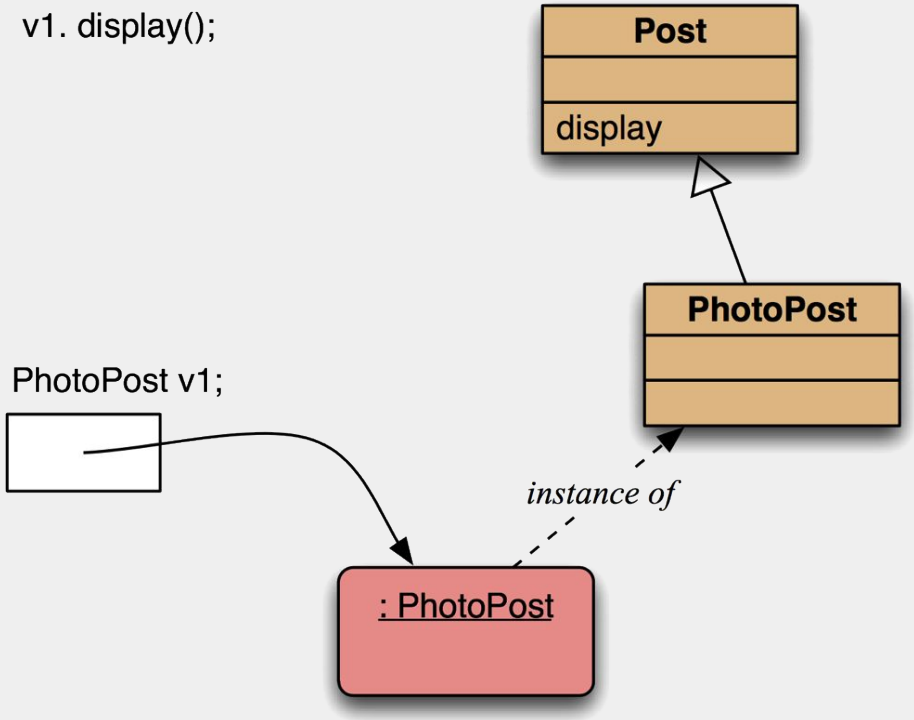
To understand how this works, we need to look out how Java decides which method to invoke on a method call: this is called *method lookup*.

Method Lookup



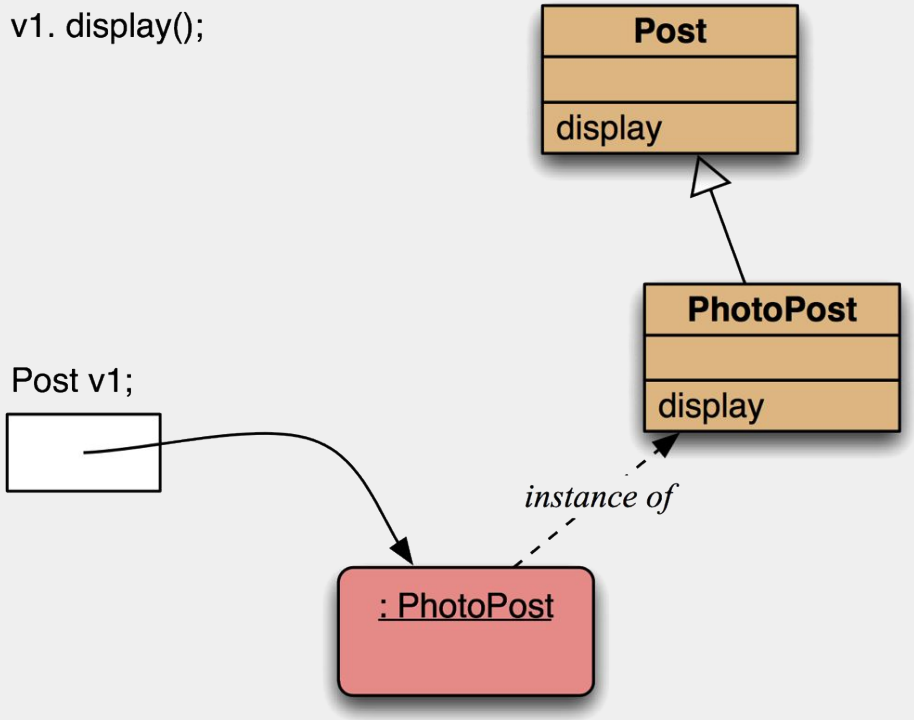
Here the object is a `PhotoPost`, and this class defines the required method.
Nothing new here: no inheritance or polymorphism.

Method Lookup



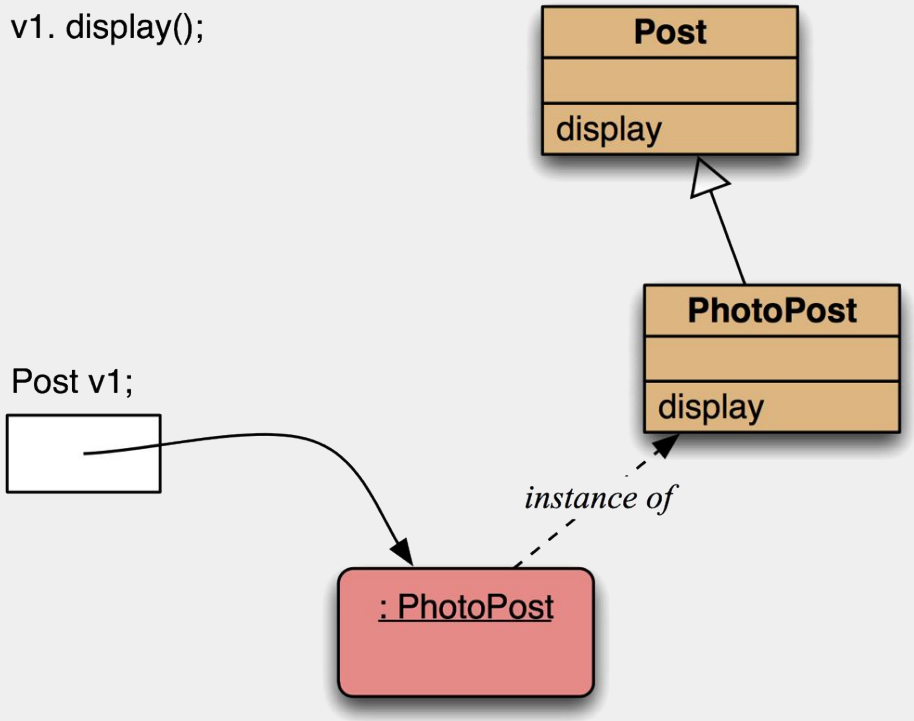
Here the object is a `PhotoPost`, but this class does not define the required method.
But `Post` does, so that method is used.
This is simple inheritance.

Method Lookup



Here the object is a `PhotoPost` which has a `display` method, as does `Post`.
We go up, seeking a match, and use the first one found.
This is polymorphism and overriding.

Method Lookup



Here the object is a `PhotoPost` which has a `display` method, as does `Post`.
We go up, seeking a match, and use the first one found.
(The method in `Post` never gets called.)

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.

Determining Dynamic Type

It is often useful to "ask" an object what type it currently holds (its dynamic type).

This is often used before a cast to determine if the cast will succeed:

```
if (post instanceof MessagePost) {
      MessagePost msg = (MessagePost) post;
```

Determining Dynamic Type

It is often useful to "ask" an object what type it currently holds (its dynamic type).

This is often used before a cast to determine if the cast will succeed:

```
if (post instanceof MessagePost) {
      MessagePost msg = (MessagePo
```

Note that this code will return true if the object is of the type given, or of any subtype.
It respects polymorphism.

Determining Dynamic Type

It is often useful to "ask" an object what type it currently holds (its dynamic type).

This is often used before a cast to determine if the cast will succeed:

```
if (post instanceof MessagePost) {
      MessagePost msg = (MessagePo
```

A hack to subvert this is:

```
post.getClass ().equals (MessagePost.class)
```

Protected Access

In the current solution, the subclasses can display the instance variables from the superclass in two ways:

- Using a `super` call to the `display` method of the superclass.
- Displaying them using the public getter methods.

The subclasses *cannot* access the instance variables directly - they're private.

Protected Access

In the current solution, the subclasses can display the instance variables from the superclass in two ways:

- Using a `super` call to the `display` method of the superclass.
- Displaying them using the pu

The subclasses *cannot* access the ins
they're private.

This is, of course, exactly what we want.
We do not want the subclass to rely on details of the implementation of the superclass.

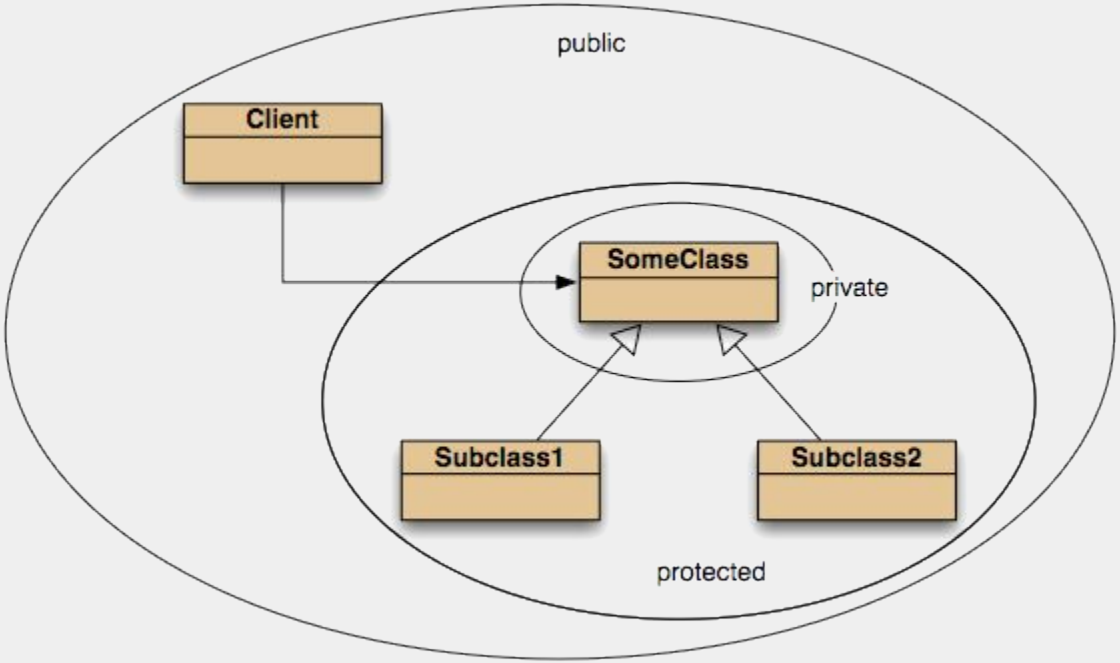
Protected Access

In the current solution, the subclasses can display the instance variables from the superclass in two ways:

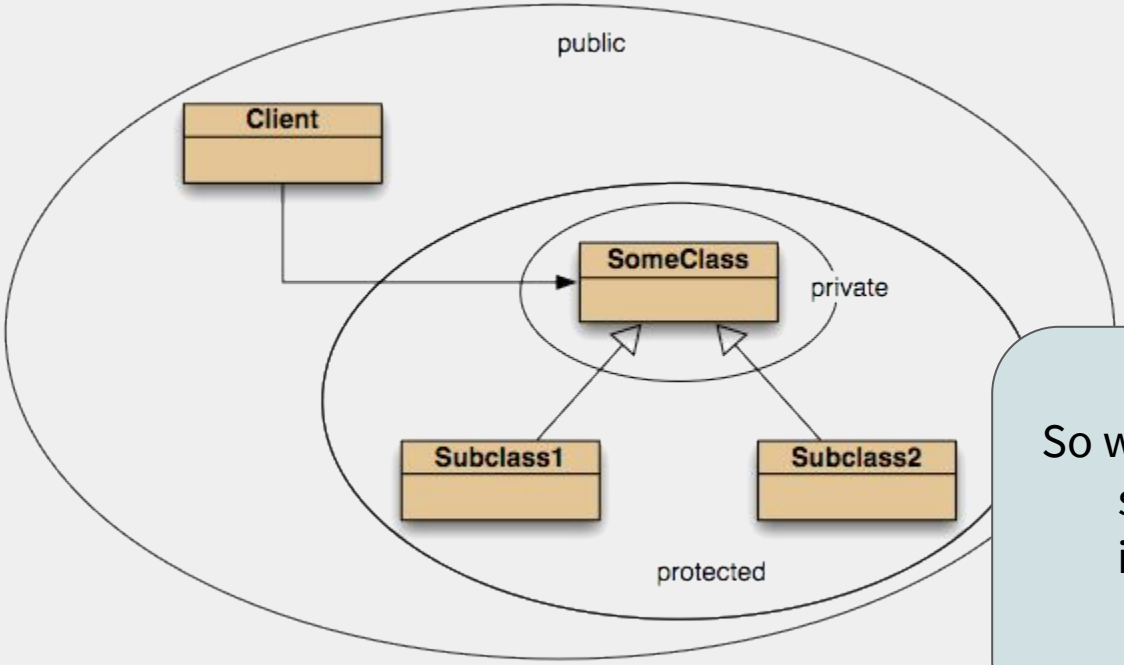
- Using a `super` call to the `display` method of the superclass.
- Displaying them using the public getter methods.

But this can be cumbersome, so if direct access is needed there is the `protected` access modifier.

Protected Access

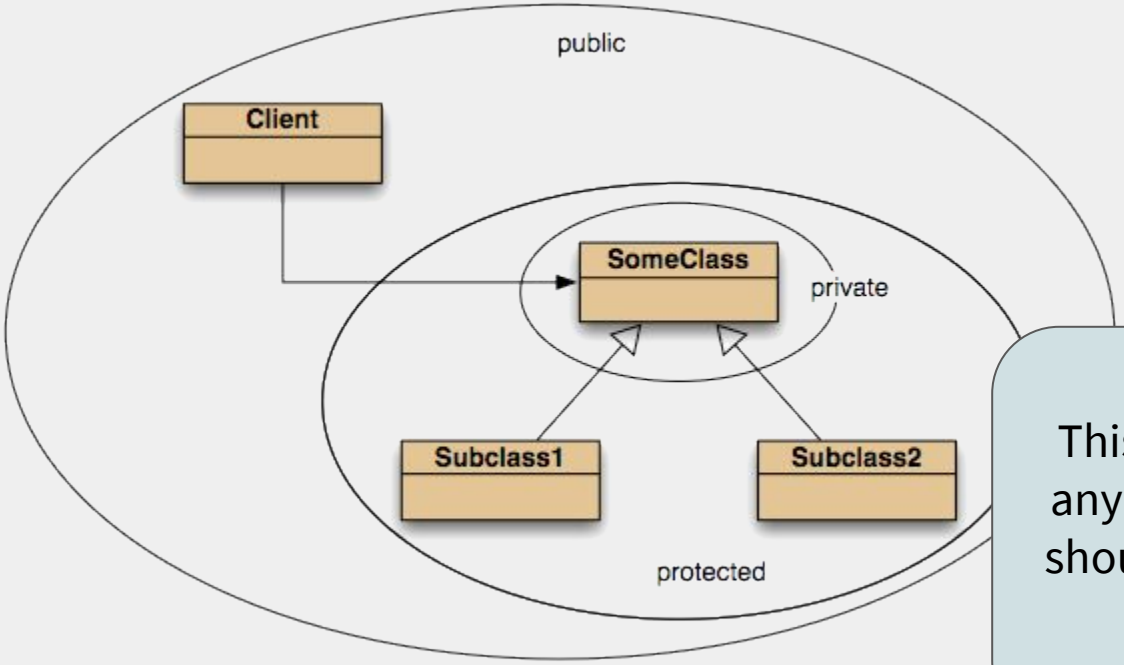


Protected Access



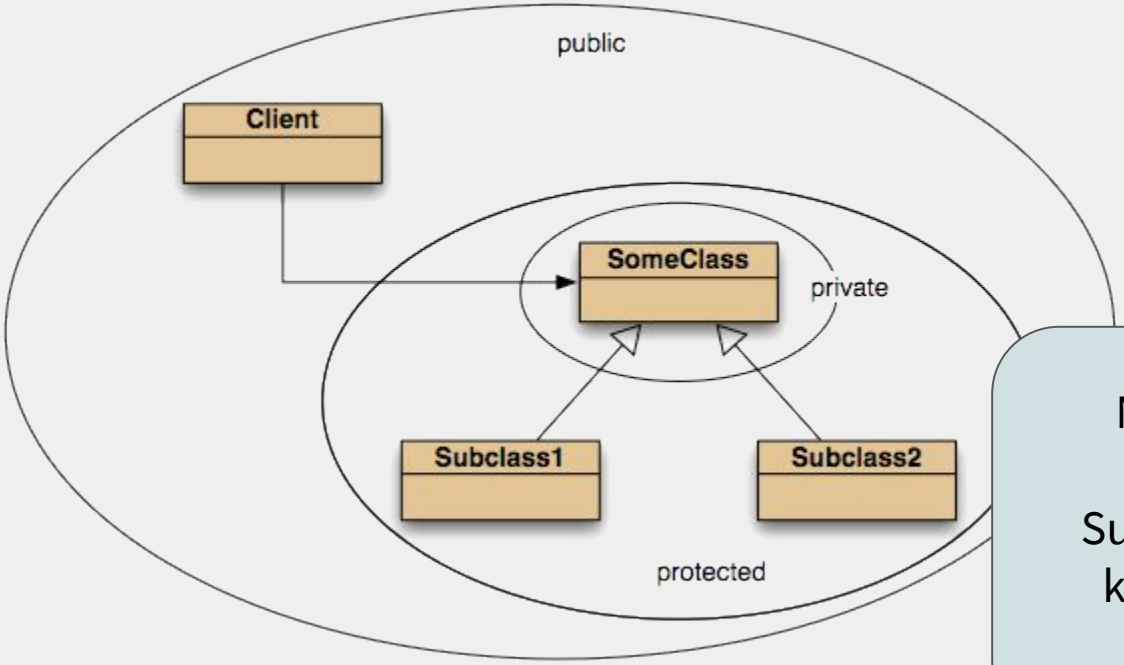
So with `protected` access, the subclasses can access the instance variables of their superclass.

Protected Access



This can be neat, but it subverts any checks in setter methods, so should be used with caution, if at all.

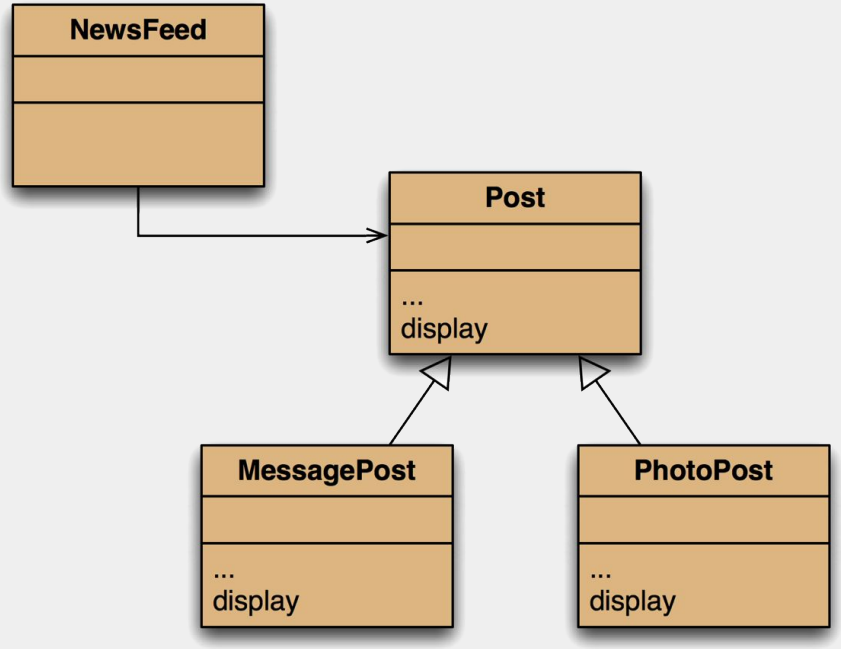
Protected Access



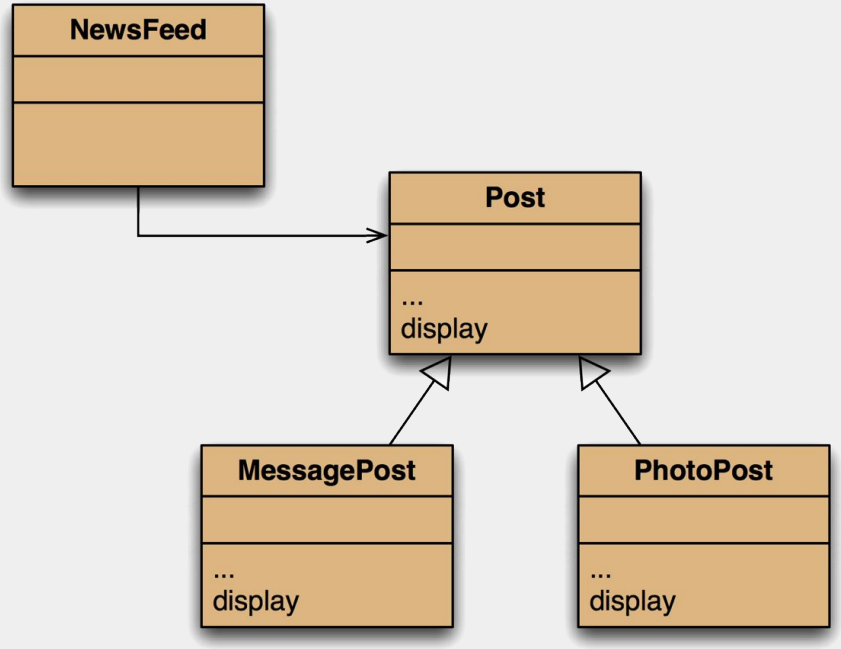
More formally, this breaks *encapsulation*.
Subclasses should ideally not know how the superclass is implemented.

The Solution

The Solution

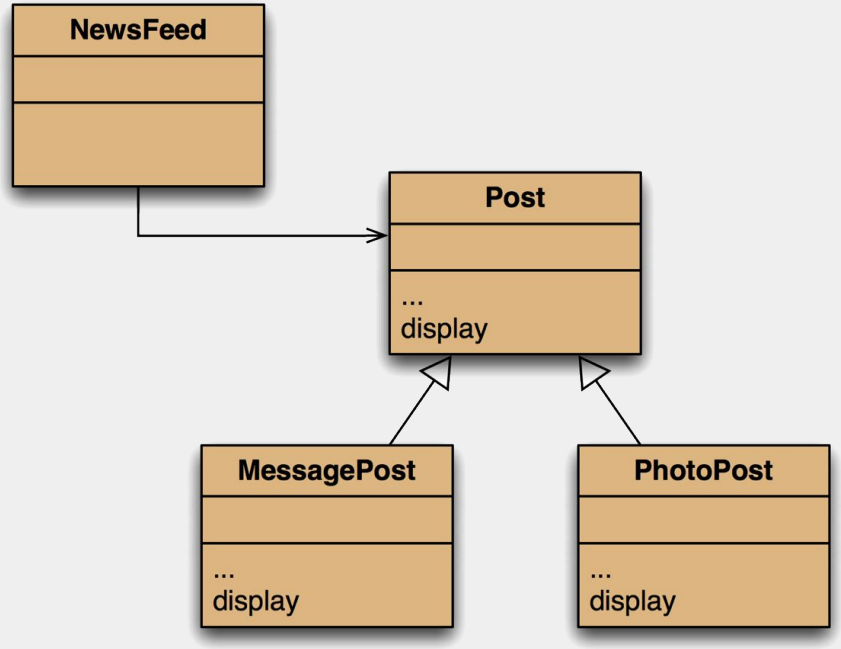


The Solution



This works, but we do have some duplicated code. In this case it can't easily be avoided unless we format the output in an unnatural way.

The Solution



And to minimise the possibly impact of changes in `Post` on its subclasses, we should use its public interface only.

Checking Dynamic Type

To get some neater output, we could remove the display method from `Post`, and check the dynamic type of the objects in the `NewsFeed`.

Checking Dynamic Type

To get some neater output, we could remove the display method from `Post`, and check the dynamic type of the objects in the `NewsFeed`.

The code would look something like this.

```
public void show () {

  for (Post p : posts) {

    if (p instanceof MessagePost) {
      ((MessagePost) p).display ();
    }
    else {
      ((PhotoPost) p).display ();
    }

    System.out.println ("------------");
  }
}
```

Checking Dynamic Type

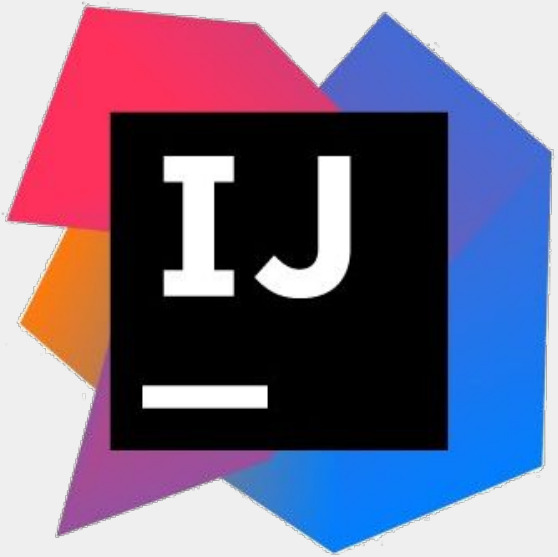
To get some neater output, we could remove the display method from `Post`, and check the dynamic type of the objects in the `NewsFeed`.

The code would look something like this.

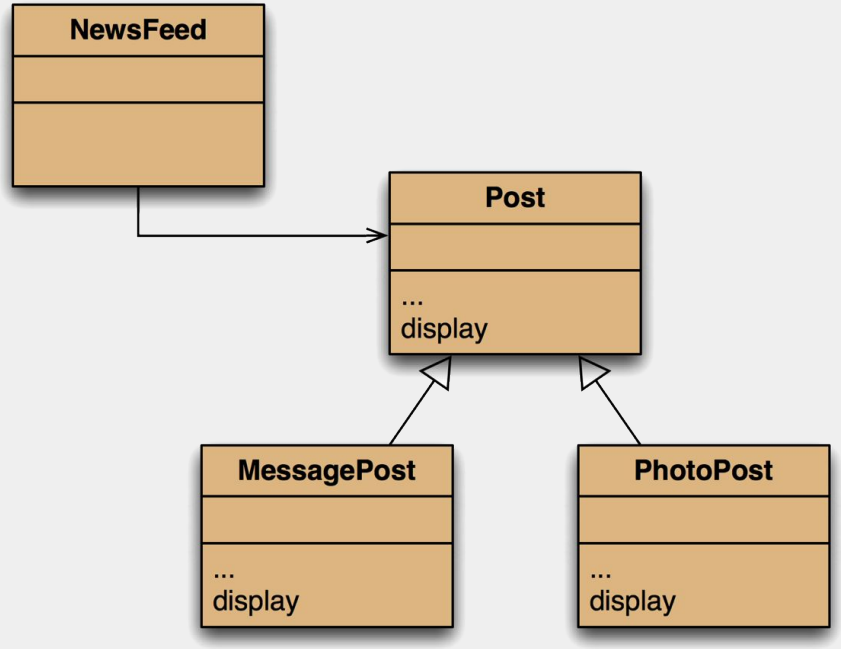
This code is pretty scary (and we won't use it) but it does show how we could print just certain types of `Post`.

```
public void show () {

  for (Post p : posts) {

    if (p instanceof MessagePost) {
      ((MessagePost) p).display ();
    }
    else {
      ((PhotoPost) p).display ();
    }

    System.out.println ("------------");
  }
}
```

IntelliJ Demo Time

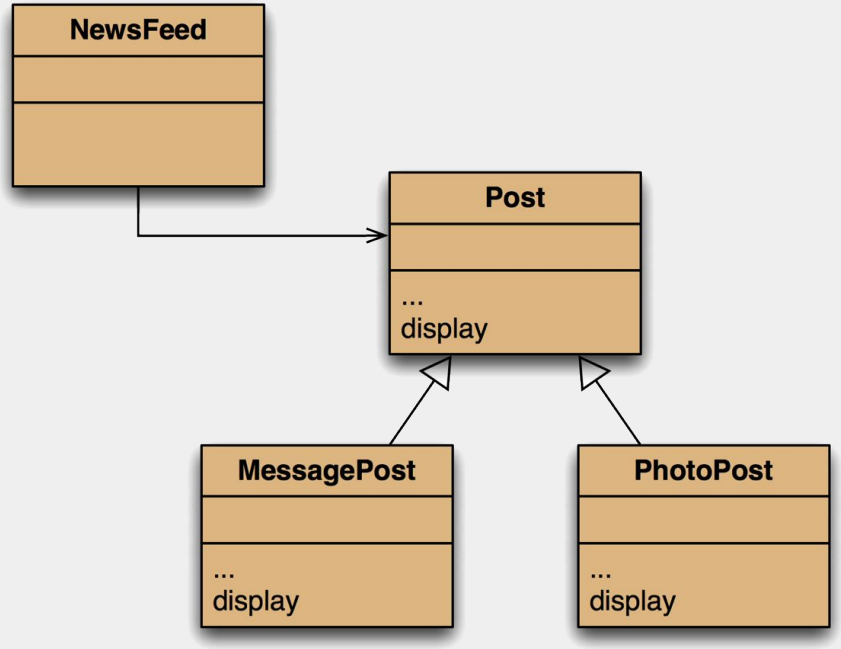


Abstract Class



You might remember that we agreed that the `Post` class was *abstract*.

Abstract Class



Since we can't create a `Post` object, it follows that we will never call that `display` method!

Abstract Classes and Methods

Abstract classes cannot be instantiated.

They are denoted by adding the keyword `abstract`.

Likewise, abstract methods have `abstract` in the signature.

- Abstract methods have no body.
- But they satisfy static type checking.

Concrete subclasses complete the implementation.

Abstract Classes and Methods

Abstract classes cannot be instantiated.

They are denoted by adding the keyword `abstract`.

Likewise, abstract methods have `abstract` in the signature.

- Abstract methods have no body.
- But they satisfy static type checkin

Concrete subclasses complete the ir

Effectively, an abstract method commits every subclass to implement a method with that signature.

Abstract Classes and Methods

Abstract classes cannot be instantiated.

They are denoted by adding the keyword `abstract`.

Likewise, abstract methods have `abstract` in the signature.

- Abstract methods have no body.
- But they satisfy static type checkin

Concrete subclasses complete the ir

It's a concept we'll meet with *interfaces* next week.

IntelliJ Demo Time