University of HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 18: Further Abstraction

Abstract Classes. Interfaces.

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.

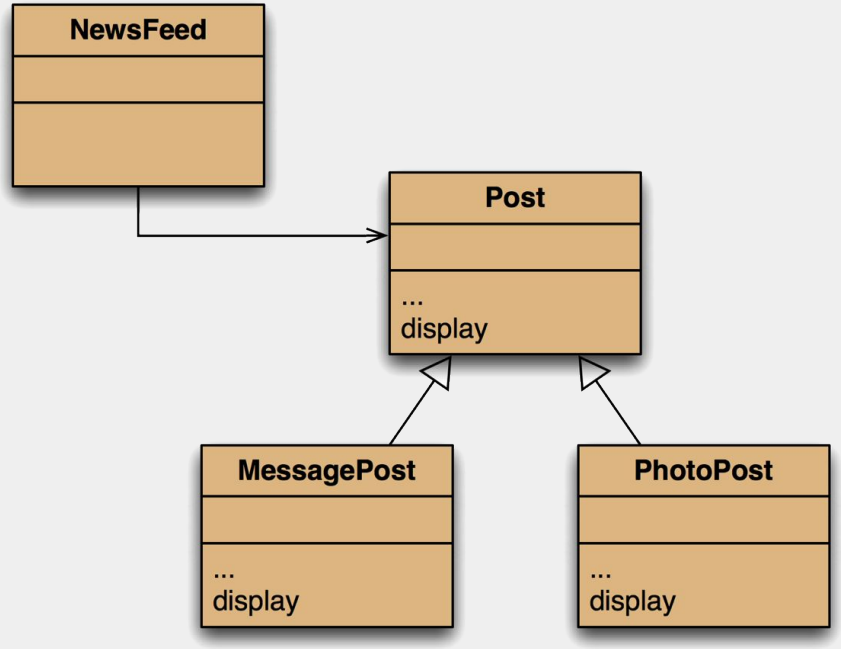
Before We Start: The Assignment

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.


DRIVE BROKEN? NO BACKUP?
GOOD.
imgflip.com


ALL OK.
SOURCE CODE ON GITLAB.
imgflip.com

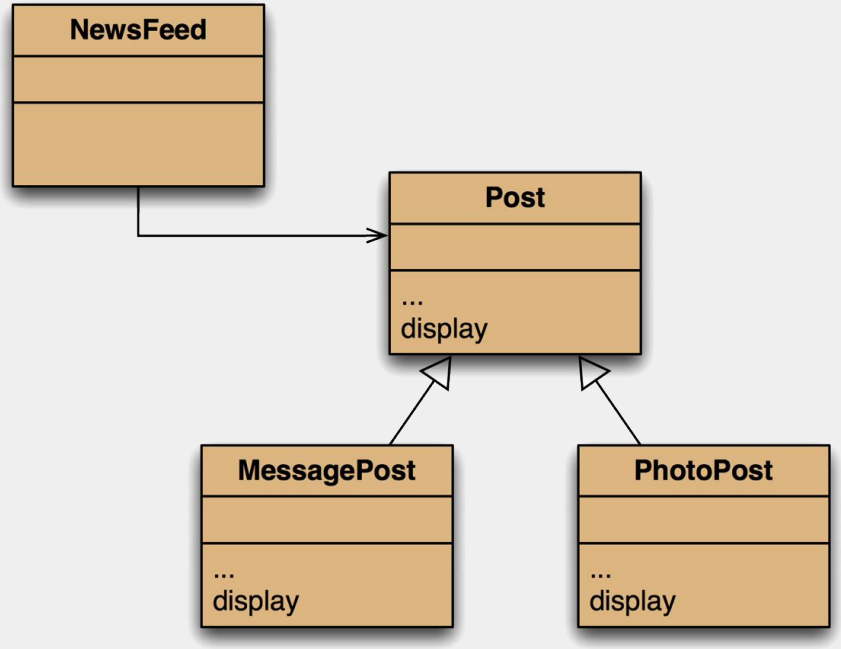
Further Abstraction



We left the NewsFeed project like this.

Today we aim to tidy things up by using some additional abstraction techniques.

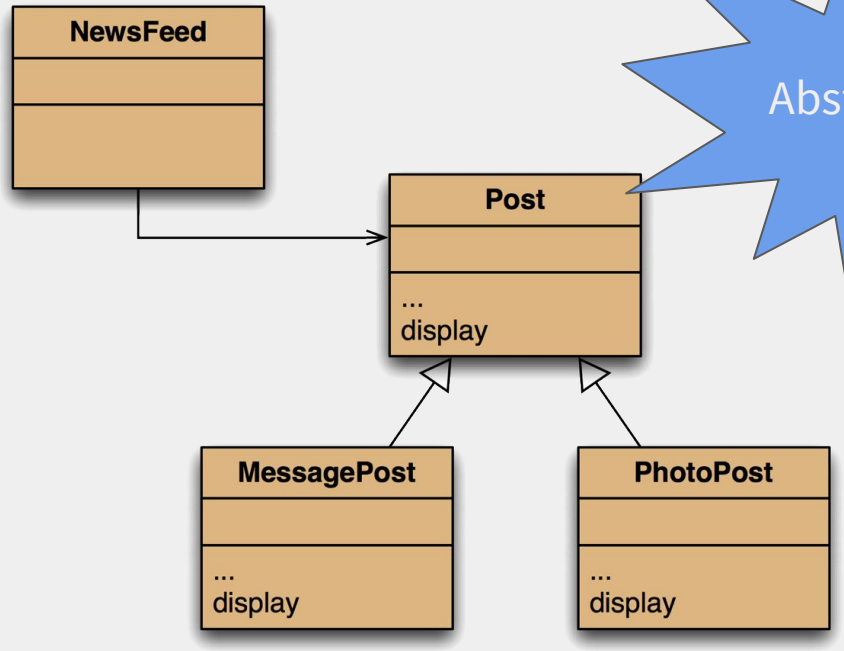
Further Abstraction



Remember that we have agreed that there is no such thing as a `Post`.

So it would be good to stop the creation of any `Post` objects.

Further Abstraction

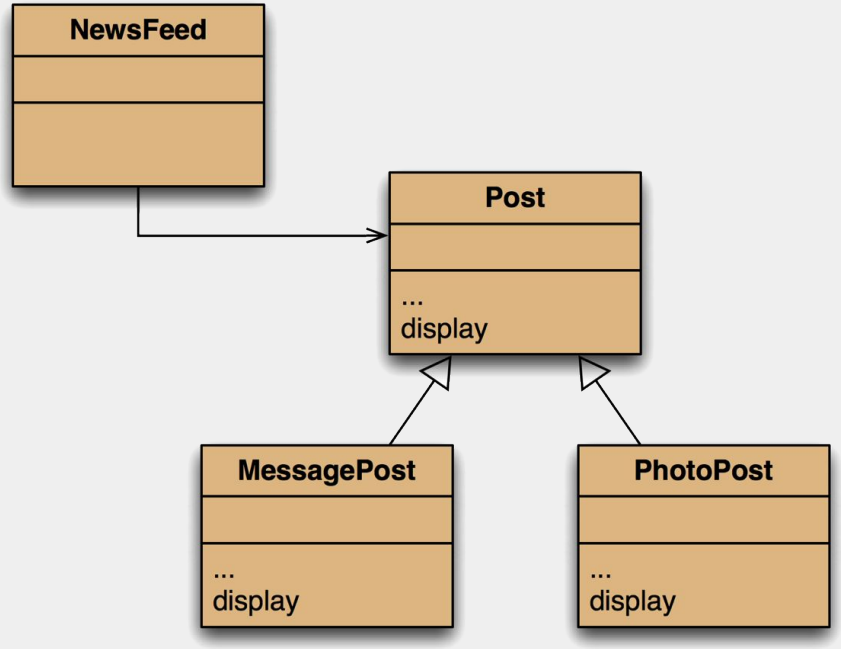


Abstract Class!

ber that we have agreed
at there is no such thing as a `Post`.

So it would be good to stop the creation of any `Post` objects.

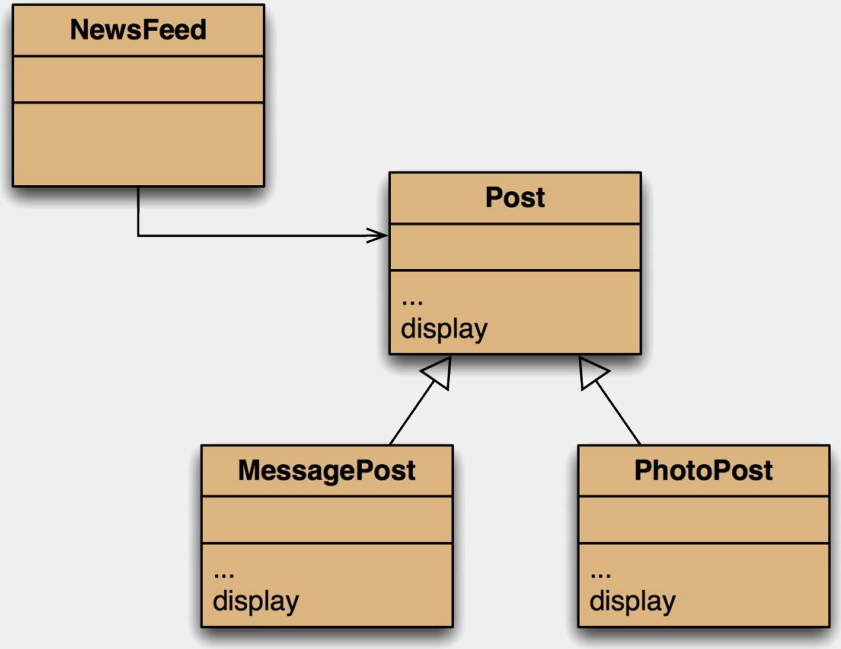
Further Abstraction



We have also agreed that creating additional types of `Post` would be useful.

So it would be neat to have a way to specify what these types should do (that is, what methods they should have).

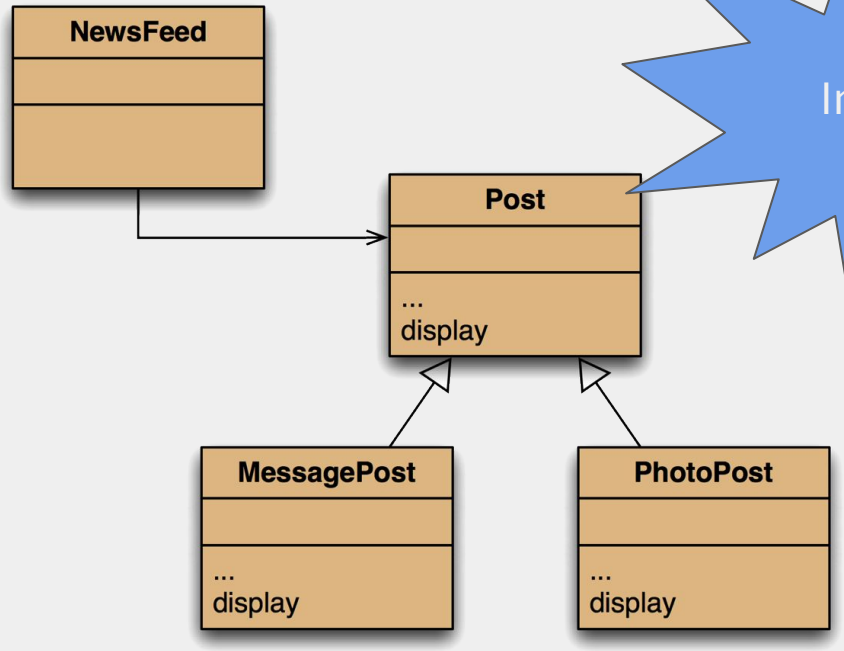
Further Abstraction



We have also agreed that creating additional types of `Post` would be useful.

Example: several types of post will require uploading some media. `MessagePost` does not. So where do we define this?

Further Abstraction

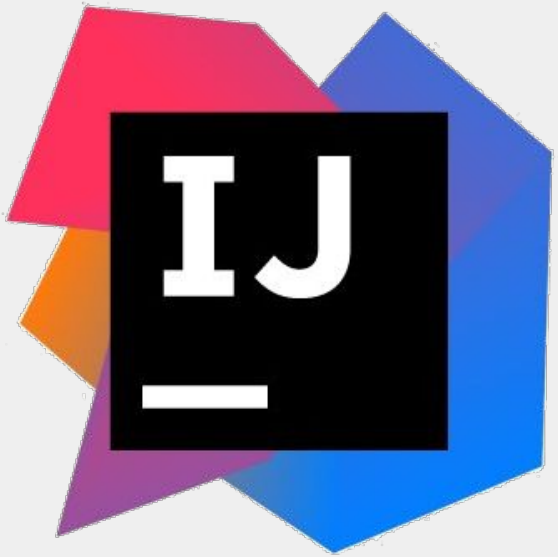


Interface!

We have also agreed that creating additional types of `Post` would be useful.

Example: several types of post will require uploading some media. `MessagePost` does not. So where do we define this?

IntelliJ Demo Time



New Concepts

So today we need to discuss:

- Finding an Object's Dynamic Type.
- Abstract Classes.
- Interfaces.
- Multiple Inheritance.

New Concepts

So today we need to discuss:

- Finding an Object's Dynamic Type.
- Abstract Classes.
- Interfaces.
- Multiple Inheritance.

As usual, these are not really new. We have been using them all along, just without giving them their formal names.

Reminder: Static and Dynamic Type

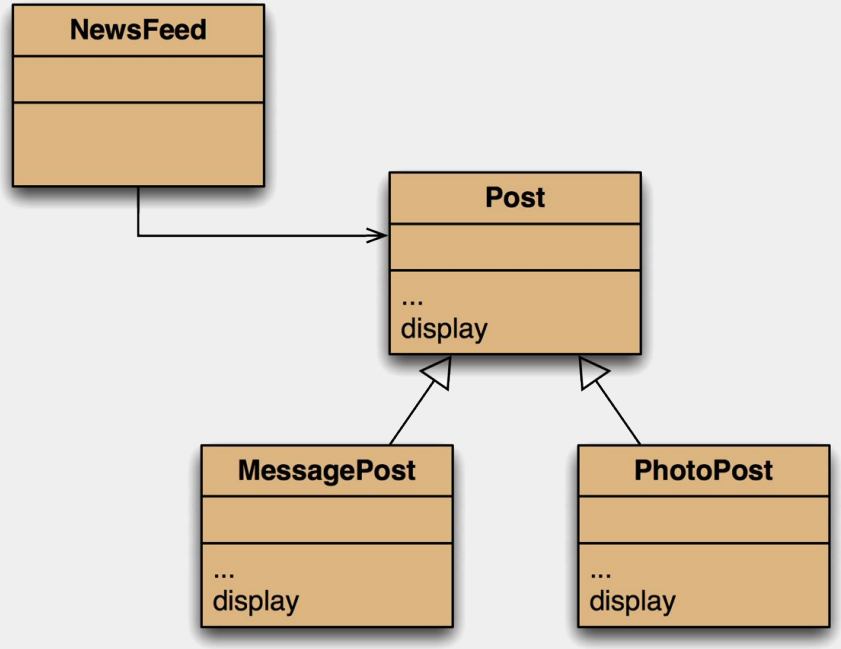
The declared type of a variable is its *static* type.

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 …

Finding Dynamic Type



Suppose we want to display all Posts of a particular subclass.

Finding Object Type

In `NewsFeed` we find an `ArrayList` of `Post` objects.

```
private ArrayList<Post> posts;
```

Finding Object Type

In `NewsFeed` we find an `ArrayList` of `Post` objects.

Along with a method to display all the posts in the feed.

```
private ArrayList<Post> posts;

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

Finding Object Type

In `NewsFeed` we find an `ArrayList` of `Post` objects.

Along with a method to display all the posts in the feed.

As we know, the *dynamic* type of `post` will change as this code executes.

```
private ArrayList<Post> posts;

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

Finding Object Type

In `NewsFeed` we find an `ArrayList` of `Post` objects.

Along with a method to display all the posts in the feed.

As we know, the *dynamic* type of `post` will change as this code executes.

(The static type stays the same.)

```
private ArrayList<Post> posts;

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

Finding Object Type

In `NewsFeed` we find an `ArrayList` of `Post` objects.

Along with a method to display all the posts in the feed.

As we know, the *dynamic* type of `post` will change as this code executes.

We can use this fact to create a method that displays, say, just the `PhotoPost` objects.

```
private ArrayList<Post> posts;

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

Finding Object Type

In `NewsFeed` we find an `ArrayList` of `Post` objects.

Along with a method to display all the posts in the feed.

As we know the *dynamic* type of `post` will change as this code executes.

`instanceof` lets us determine the dynamic type of an object.

```
private ArrayList<Post> posts;

public void showPhotos ()
{
  for (Post post : posts) {
    if (post instanceof PhotoPost) {
      post.display ();
      System.out.println ();
    }
  }
}
```

Finding Object Type

In `NewsFeed` we find an `ArrayList` of `Post` objects.

Along with a method to display all the posts in the feed.

As we know the *dynamic* type of `post` will change as this code executes.

And to make things work, we *may* need to *cast* the object so we get the right `display` method.

```
private ArrayList<Post> posts;

public void showPhotos ()
{
  for (Post post : posts) {
    if (post instanceof PhotoPost) {
      ((PhotoPost) post).display();
```

Subtle Point

`instanceof` tests if an object is of the given type, or *of any subtype of that type*.

This is probably what is required.

Finding Object Type

In `NewsFeed` we find an `ArrayList` of `Post` objects.

Along with a method to display all the posts in the feed.

As we know the *dynamic* type of `post` will change as this code executes.

And to make things work, we *may* need to *cast* the object so we get the right `display` method.

```
private ArrayList<Post> posts;

public void showPhotos ()
{
  for (Post post : posts) {
    if (post instanceof PhotoPost) {
      ((PhotoPost) post).display();
```

If the actual type (excluding subtypes) is required, the code is something like:

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

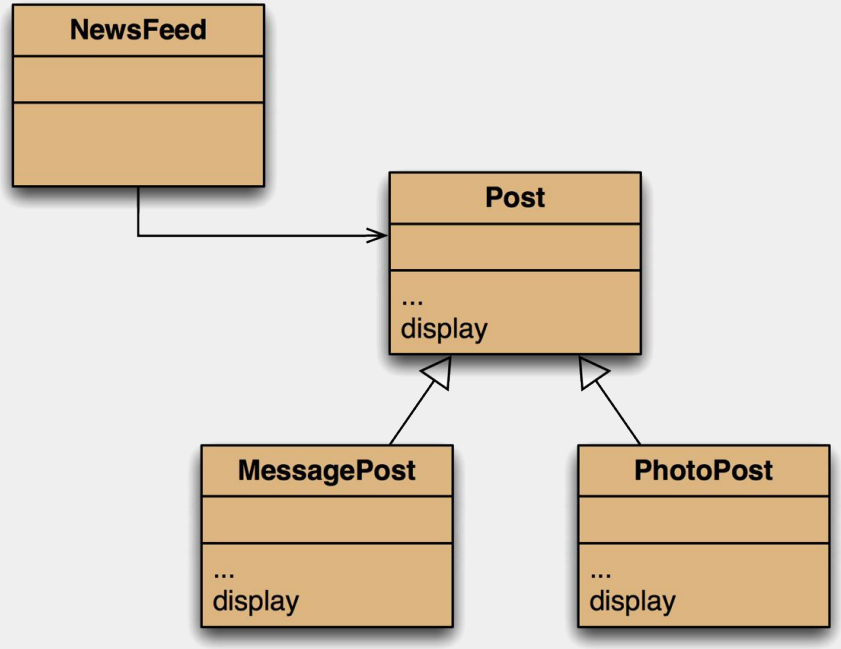
Patterns

Suppose we had:

- Bank accounts, and needed to add interest to some of them.
- Products, and needed to add tax to some of them.
- Cinema snacks, and needed to add tax to some of them.
- Films, and needed to display only the Science Fiction.
- Questions, and needed to choose from only the difficult ones.

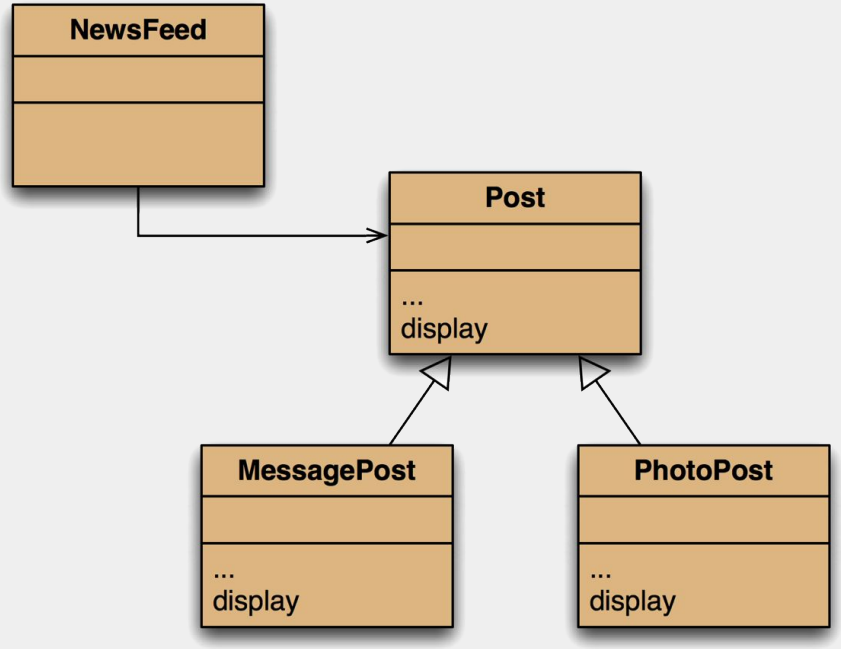
Programming really is all about patterns …

Abstract Class



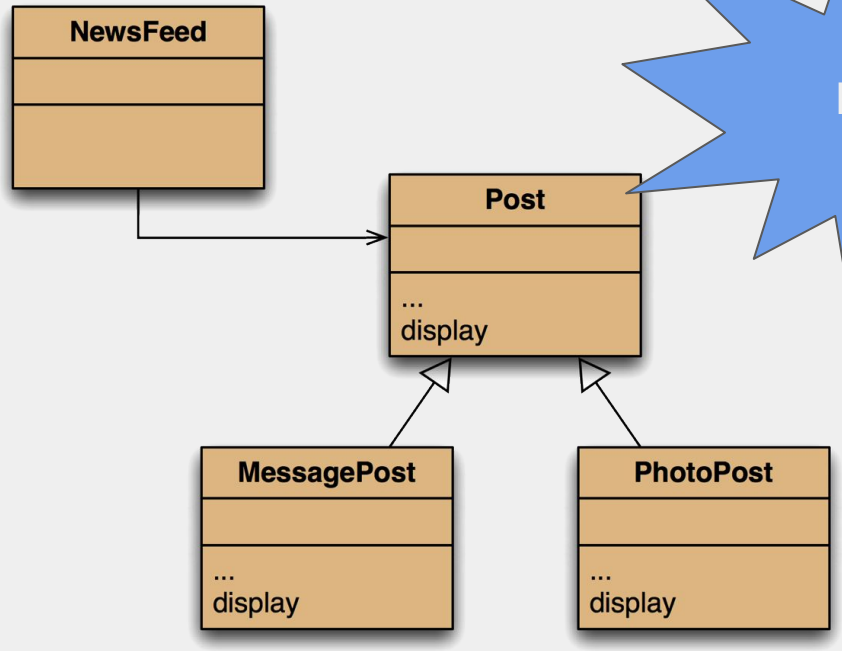
Would we ever want or need an object of class `Post`?

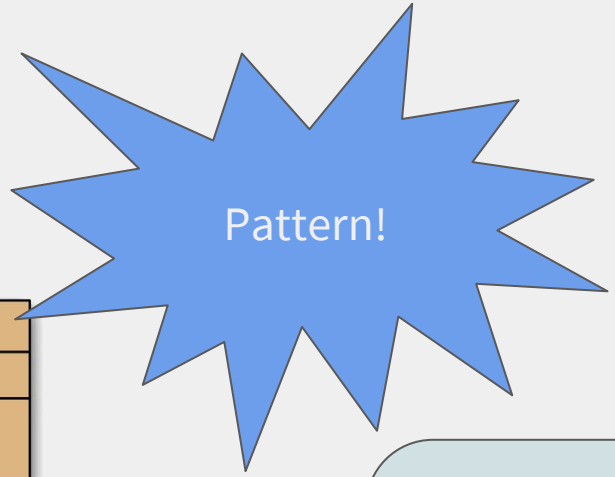
Abstract Class



No.
There is no such thing. Every object will be an instance of one of the subclasses.

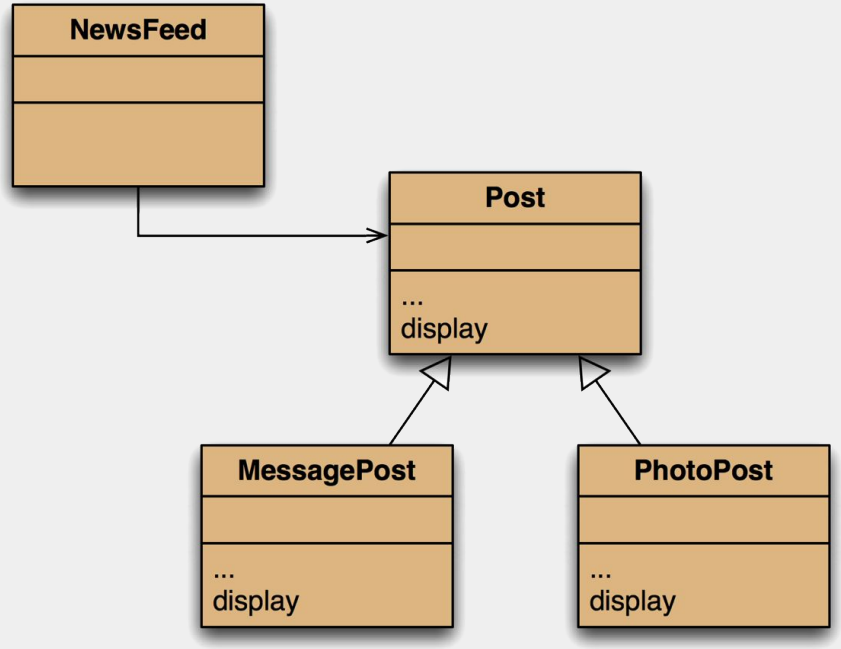
Abstract Class





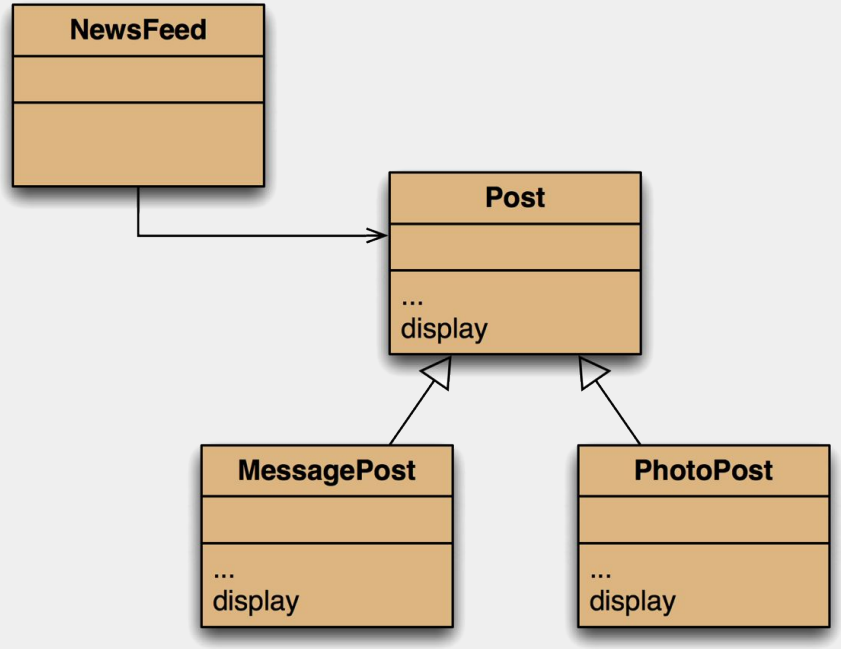
No.
There is no such thing. Every object will be an instance of one of the subclasses.

Abstract Class



And would we ever call the display method in `Post`?

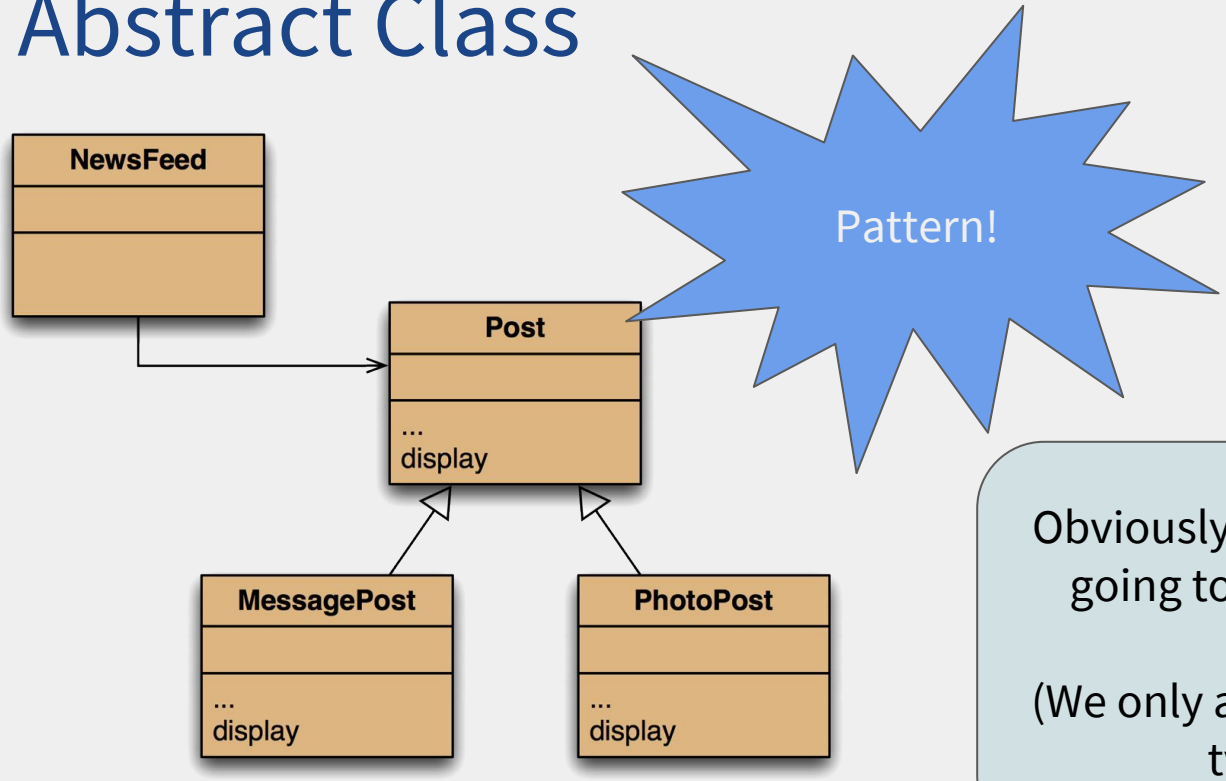
Abstract Class



Obviously not, because there are going to be no `Post` objects!

(We only added it to satisfy static type checking.)

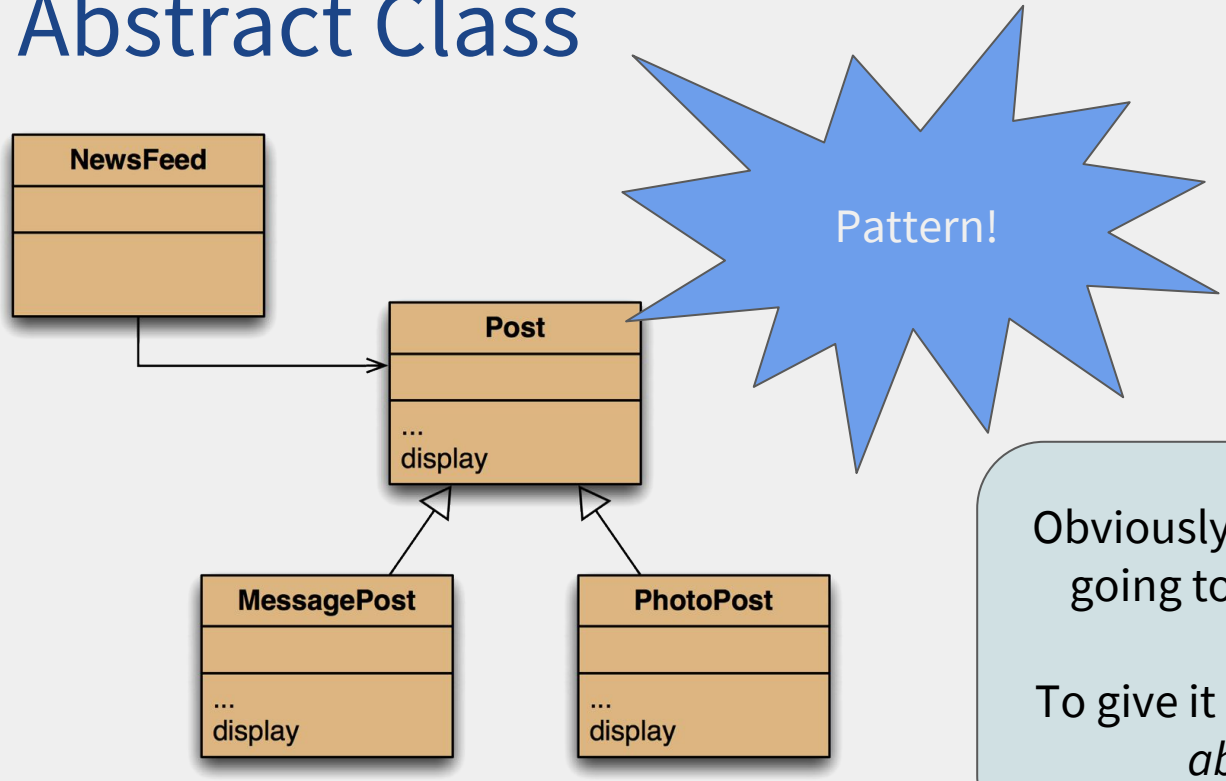
Abstract Class



Obviously not, because there are going to be no `Post` objects!

(We only added it to satisfy static type checking.)

Abstract Class



Obviously not, because there are going to be no `Post` objects!

To give it its proper name, it's an *abstract* 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

So now `Post` is an *abstract* class.

And `display` inside it is an *abstract* method.

An Abstract Class

So now we assert that it should not be possible to create objects of the `Post` class.

```
public abstract class Post
{

  // Define Fields.

  // Define Constructor.

  // Define Methods.

}
```

An Abstract Method

Methods may also be abstract,

Suppose we want to give the user some help when creating a post.

We define a simple method to print a helpful hint.

```
public abstract class Post
{

  // Define Fields.

  // Define Constructor.

  public void help () {

  }

}
```

An Abstract Method

Methods may also be abstract,

Suppose we want to give the user some help when creating a post.

We define a simple method to print a helpful hint.

But there is no point providing help for a `Post` object, because such an object cannot exist.

```
public abstract class Post
{

  // Define Fields.

  // Define Constructor.

  public void help () {

  }

}
```

An Abstract Method

Methods may also be abstract,

Suppose we want to give the user some help when creating a post.

We define a simple method to print a helpful hint.

To put it another way, there is no point providing an implementation here, because such an implementation will never be used.

```
public abstract class Post
{

  // Define Fields.

  // Define Constructor.

  public void help () {

  }

}
```

An Abstract Method

Methods may also be abstract,

Suppose we want to give the user some help when creating a post.

We define a simple method to print a helpful hint.

But, we do want to make sure all subclasses provide an implementation. In fact we want to force them to do so.

```
public abstract class Post
{

  // Define Fields.

  // Define Constructor.

  public void help () {

  }

}
```

An Abstract Method

Methods may also be abstract,

Suppose we want to give the user some help when creating a post.

We define a simple method to print a helpful hint.

So we make this the declaration of an *abstract* method, with no implementation.

```
public abstract class Post
{

  // Define Fields.

  // Define Constructor.

  abstract public void help ();


}
```

An Abstract Method

Methods may also be abstract,

Suppose we want to give the user some help when creating a post.

We define a simple method to print a helpful hint.

Any subclass of this abstract class *must* now provide an implementation of this method. Otherwise it won't compile.

```
public abstract class Post
{

  // Define Fields.

  // Define Constructor.

  abstract public void help ();



}
```

An Abstract Method

Methods may also be abstract,

Suppose we want to give the user some help when creating a post.

We define a simple method to print a helpful hint.

Any subclass of this abstract class *must* now provide an implementation of this method. Otherwise it won't compile.

```
public class PhotoPost
{

  // Define Fields.

  // Define Constructor.

  public void help () {
    System.out.println ("Upload it!");
  }

}
```

Idea: Specifying a Subclass

We have here another *pattern!*

We have an abstract class, which needs to specify the methods that its subclasses will provide.

This is such a common pattern that there is a mechanism for just this case.

The cunning thing is that we can now do *multiple inheritance*.

Idea: Specifying a Subclass

We have here another *pattern!*

We have an abstract class, which needs to specify the methods that its subclasses will provide.

This is such a common pattern that t
this case.

The cunning thing is that we can nov

And to prove this is a pattern, we will now swap examples!

Idea: Specifying a Subclass

We have here another *pattern!*

We have an abstract class, which needs to specify the methods that its subclasses will provide.

This is such a common pattern that f
this case.

The cunning thing is that we can nov

You did spot that the social network (lectures) and the bank (practicals) are the same example, didn't you?

Pattern

Recall our Bank Account example from the practicals.

We have an `Account` class, with various subclasses (`CurrentAccount`, `SavingsAccount`).

We can now see that `Account` is abstract.

We can also see that `Account` might well have abstract methods for deposit and withdraw - to require all other accounts to do this.

Pattern

Recall our Bank Account example from the practicals.

We have an `Account` class, with various subclasses (`CurrentAccount`, `SavingsAccount`).

We can now see that `Account` is Ab

We can also see that `Account` migh s
for deposit and withdraw - to require s.

With this mechanism we are going to have to think carefully about where methods most usefully go, and which should be abstract.

Pattern

Recall our Bank Account example from the practicals.

We have an `Account` class, with various subclasses (`CurrentAccount`, `SavingsAccount`).

Suppose the bank introduces a "Gold" status for accounts. There is a fee, but also attractive additional incentives.

Any account may have "Gold" status.

Multiple Inheritance

What we need now is "multiple inheritance":

- A class inherits directly from multiple superclasses.

Each language has its own rules about this: one difficulty is how to ensure there are no conflicting definitions.

Java forbids this for classes, but permits it for *interfaces*.

Multiple Inheritance

What we need now is "multiple inheritance":

- A class inherits directly from multiple superclasses.

Each language has its own rules about this: one difficulty is how to ensure there are no conflicting de

Java forbids this for classes, but per

Basically, we want `GoldCurrentAccount` to "inherit" from both `GoldenAccount` and `CurrentAccount`.
(It will get different methods from each.)

Interfaces

Interfaces provide a specification, but no implementation.

Interfaces contain no constructor.

Interfaces contain definitions of methods, but no implementation.

Any class implementing the interface must include the methods.

Interfaces

Interfaces provide a specification, but no implementation.

Interfaces contain no constructor.

Interfaces contain definitions of methods, but no implementation.

Any class implementing the interfac

That should all sound very familiar from our discussion of abstract classes and methods.

Interfaces

Interfaces provide a specification, but no implementation.

Interfaces contain no constructor.

Interfaces contain definitions of methods, but no implementation.

Any class implementing the interfac

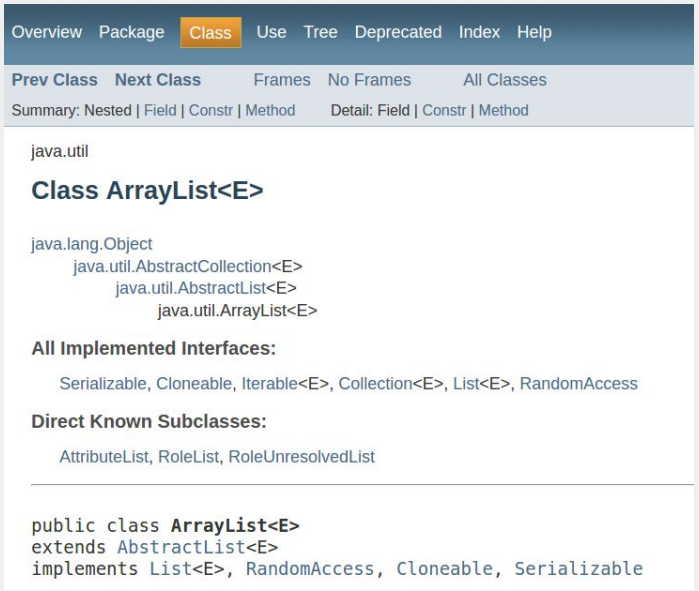
You might even remember that we used interfaces when we were sorting ArrayLists.

Interfaces

Guess what?

We have, in fact, been using interfaces (and inheritance (and abstract classes)) all along.

Look at the docs for `ArrayList`.



Overview Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

An Interface

A "Gold Account" requires the payment of a monthly fee.

In return, a bonus is paid into the account every month. The rules for the bonus vary depending on the type of account.

Any account can have "Gold" features, so simple inheritance cannot be used.

An Interface

A "Gold Account" requires the payment of a monthly fee.

In return, a bonus is paid into the account every month. The rules for the bonus vary depending on the type of account.

Any account can have "Gold" features, so simple inheritance cannot be used.

```
public interface GoldenAccount
{
    boolean takeMonthlyFee ();
    void addMonthlyBonus ();
}
```

An Interface

Any class can now *implement* this interface.

```
public class GoldCurrentAccount
   extends CurrentAccount
   implements GoldenAccount {

    // Instance Variables

    // Constructor

    // Methods

}
```

An Interface

Any class can now *implement* this interface.

And such a class *must* provide implementations of the methods in the interface.

The method signatures are checked for matches by the compiler (and the IDE too).

```
public class GoldCurrentAccount
   extends CurrentAccount
   implements GoldenAccount {

    // Instance Variables

    // Constructor

    public boolean takeMonthlyFee () {
      this.withdraw (10.0);
    }
}
```

An Interface

Any class can now *implement* this interface.

Interfaces may also define instance variables, which are effectively constants.

In this case if the monthly fee is the same in all accounts …

```
public class GoldCurrentAccount
   extends CurrentAccount
   implements GoldenAccount {

    // Instance Variables

    // Constructor

    public boolean takeMonthlyFee () {
      this.withdraw (10.0);
    }
}
```

An Interface

Any class can now *implement* this interface.

Interfaces may also define instance variables, which are effectively constants.

In this case if the monthly fee is the same in all accounts …

We can include that in the interface.

```
public interface GoldAccount
{

    double MONTHLY_FEE = 10.0;

    boolean takeMonthlyFee ();
    void addMonthlyBonus ();
}
```

An Interface

Any class can now *implement* this interface.

Interfaces may also define instance variables, which are effectively constants.

In this case if the monthly fee is the same in all accounts …

And tweak the code.

```
public class GoldCurrentAccount
   extends CurrentAccount
   implements GoldenAccount {

    // Instance Variables

    // Constructor

    public boolean takeMonthlyFee () {
      this.withdraw (MONTHLY_FEE);
    }
}
```

IntelliJ Demo Time