University of HUDDERSFIELD

CFS2160: Software Design and Development

Lecture 15: Inheritance

Improving structure with 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.

Code Reuse

"Code reuse" is an often-claimed benefit of object-oriented programming.

The idea goes that we develop a class, and that class can then be used in many applications.

Often this means that the base class is rather generic, and that we specialise the class to fit the application.

Code Reuse

"Code reuse" is an often-claimed benefit of object-oriented programming.

The idea goes that we develop a class, and that class can then be used in many applications.

Often this means that the *base class* is rather *generic*, and that we specialise the class to fit the application.

Code Reuse

"Code reuse" is an often-claimed benefit of object-oriented programming.

The idea goes that we develop a class, and that class can then be used in many applications.

Often this means that the *base class* is rather *generic*, and that we *specialise* the class to fit the application.

Code Reuse

"Code reuse" is an often-claimed benefit of object-oriented programming.

The idea goes that we develop a class, and that class can then be used in many applications.

Often this means that the *base class* … *specialise* the class to fit the applicat…

We *inherit* some characteristics from our parents.
Other characteristics are specific to us.
Siblings all inherit from their parents.

Java Class Libraries

This concept is not new to us.

Check the top of the `ArrayList` docs:

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

An `ArrayList` is an `AbstractList` is an `AbstractCollection` is an `Object`.

"Everything" in Java is eventually a `java.lang.Object`.

Java Class Libraries

This concept is not new to us.

Check the top of the `ArrayList` docs:

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

An `ArrayList` is an `AbstractLi`
`AbstractCollection` is an `Obj`

"Everything" in Java is eventually a

So this idea of *inheritance* is at the core of Java.
Now we are going to see how to use it to make our programming lives easier.

Bank Accounts

Last week in the practical you developed a collection of classes that could be used in a "Bank" application.

You probably noticed that several of the classes you wrote contained identical code.

You might have found all the copying-and-pasting to be tedious and error-prone.

Bank Accounts

Last week in the practical you developed a collection of classes that could be used in a "Bank" application.

You probably noticed that several of the classes you wrote contained identical code.

You might have found all the copyin
and error-prone.

Inheritance will ease this pain for us.
Using it, we would define a *generic* bank account, and then *specialise* it for different specific types of account.

Bank Accounts

All bank accounts have a balance, account number, account holder.

So these all go in a **base class** - call it `BankAccount` - with constructor, getters, setters, `toString` and so on.

Bank Accounts

All bank accounts have a balance, account number, account holder.

So these all go in a **base class** - call it `BankAccount` - with constructor, getters, setters, `toString` and so on.

Current accounts additionally handle overdrafts, so must be a different class (`CurrentAccount`). But we write *only* what is specific to Current accounts.

Bank Accounts

All bank accounts have a balance, account number, account holder.

So these all go in a **base class** - call it `BankAccount` - with constructor, getters, setters, `toString` and so on.

Current accounts additionally handle overdrafts, so must be a **derived** class (`CurrentAccount`). But we write *only* what is specific to Current accounts.

Bank Accounts

All bank accounts have a balance, account number, account holder.

So these all go in a base class - call it `BankAccount` - with constructor, getters, setters, `toStri`

Current accounts additionally handl
different class (`CurrentAccount`)
specific to Current accounts.

We say that `CurrentAccount` specialises (or just "inherits from") `BankAccount`.

Bank Accounts

All bank accounts have a balance, account number, account holder.

So these all go in a base class - call it `BankAccount` - with constructor, getters, setters, `toString` and so on.

Both current accounts and deposit accounts allow deposits (the rules are the same), so this logic can go in `BankAccount`.

Bank Accounts

All bank accounts have a balance, account number, account holder.

So these all go in a base class - call it `BankAccount` - with constructor, getters, setters, `toString` and so on.

But the two account types have different rules for withdrawals, so need their own separate logic.

Bank Accounts

All bank accounts have a balance, account number, account holder.

So these all go in a base class - call it `BankAccount` - with constructor, getters, setters, `toStri`

But the two account types have diffe o need their own separate logic.

But we surely want both `CurrentAccount` and `DepositAccount` to have sensibly named methods for deposit and withdraw.

Bank Accounts

All bank accounts have a balance, account number, account holder.

So these all go in a base class - call it `BankAccount` - with constructor, getters, setters, `toStri`

But the two account types have diffe o need their own separate logic.

Notice also that in the hierarchy we are developing, there is actually no such thing as a `BankAccount`.

A Social Network

To illustrate this, imagine a simple "News Feed" on a social networking site.

Users add "Posts" to the feed.

Posts may be:

- Text
- Pictures
- Videos

A Social Network

To illustrate this, imagine a simple "News Feed" on a social networking site.

Users add "Posts" to the feed.

Posts may be:

- Text
- Pictures
- Videos

So we have a generic class "`Post`".
And there are three more specialised classes: `TextPost`, `PicturePost`, `VideoPost`.

A Social Network

To illustrate this, imagine a simple "News Feed" on a social networking site.

Users add "Posts" to the feed.

Posts may be:

- Text
- Pictures
- Videos

All Posts (of whatever type) have a user, a number of "likes" and might have comments added.

A Social Network

To illustrate this, imagine a simple "News Feed" on a social networking site.

Users add "Posts" to the feed.

Posts may be:

- Text
- Pictures
- Videos

Text posts contain a string of textual data, and nothing else.

A Social Network

To illustrate this, imagine a simple "News Feed" on a social networking site.

Users add "Posts" to the feed.

Posts may be:

- Text
- Pictures
- Videos

Picture posts don't have a message.
They have the name of an image file, and maybe a caption.

A Social Network

To illustrate this, imagine a simple "News Feed" on a social networking site.

Users add "Posts" to the feed.

Posts may be:

- Text
- Pictures
- Videos

Video posts are much the same as Picture posts, but may also have attributes such as duration.

Design

To keep things simple, we first ignore video posts.

As a first design, we could decide:

- A `NewsFeed` is made up of posts, stored in an `ArrayList`.
- Posts may be of one of two kinds:
  - `MessagePost`: a multiline text message.
  - `PhotoPost`: an image filename and a caption.

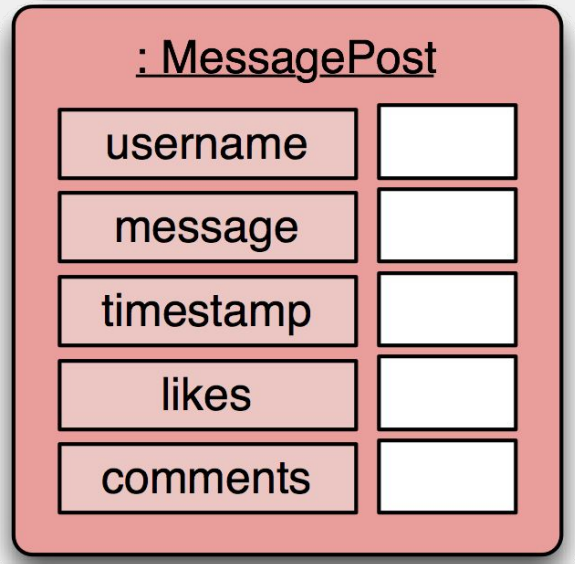
Design

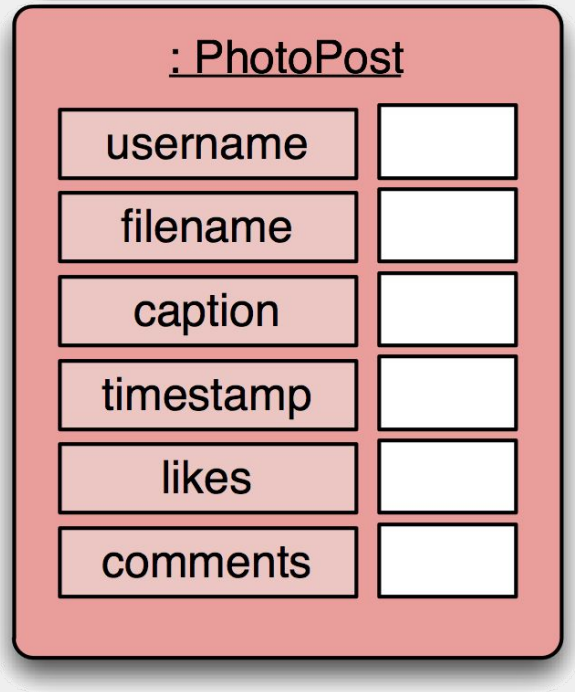
To keep things simple, we first ignore video posts.

As a first design, we could decide:

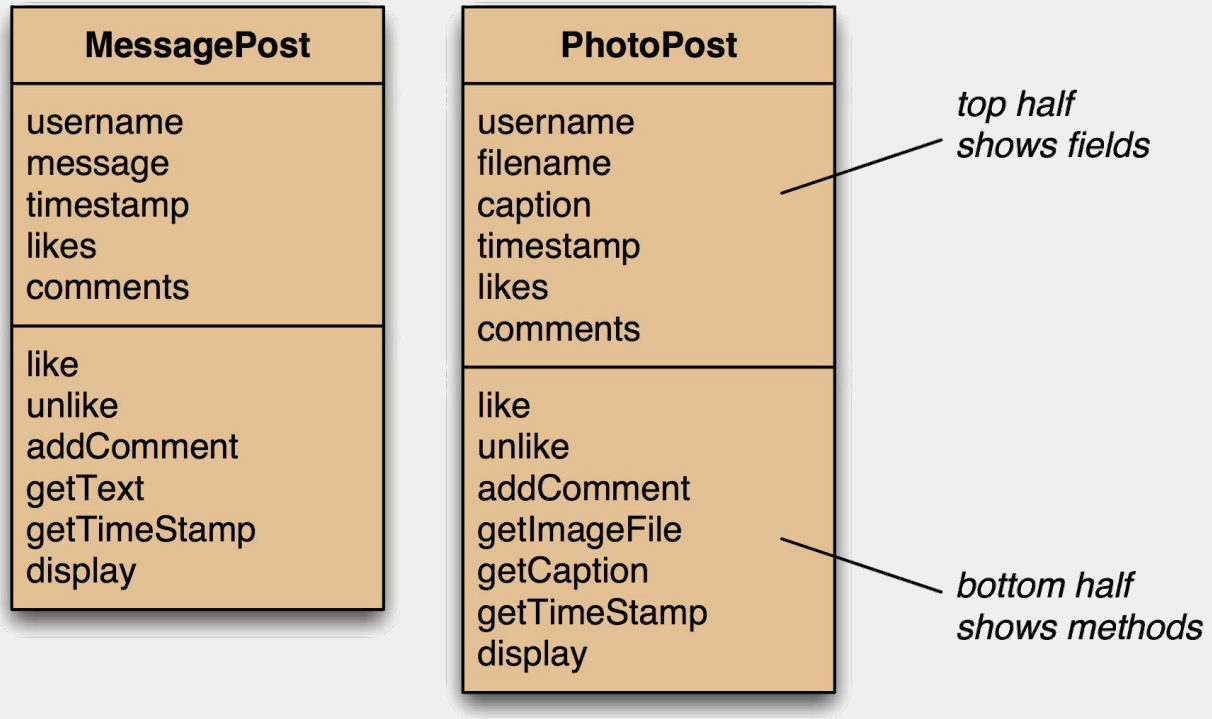
- A `NewsFeed` is made up of posts, stored in an `ArrayList`.
- Posts may be of one of two kinds:
  - `MessagePost`: a multiline text message.
  - `PhotoPost`: an image filename and a caption.
- Since the two kinds of post are different, two ArrayLists will be needed.

Design

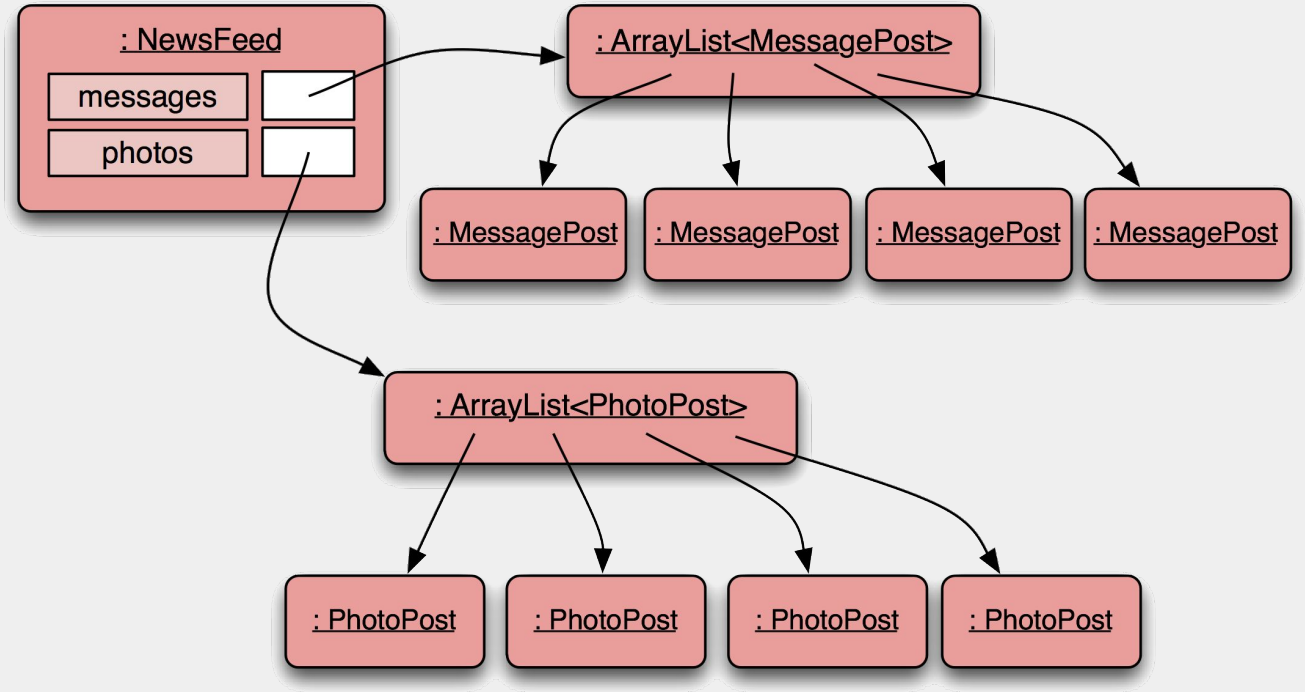

: MessagePost
username
message
timestamp
likes
comments


: PhotoPost
username
filename
caption
timestamp
likes
comments

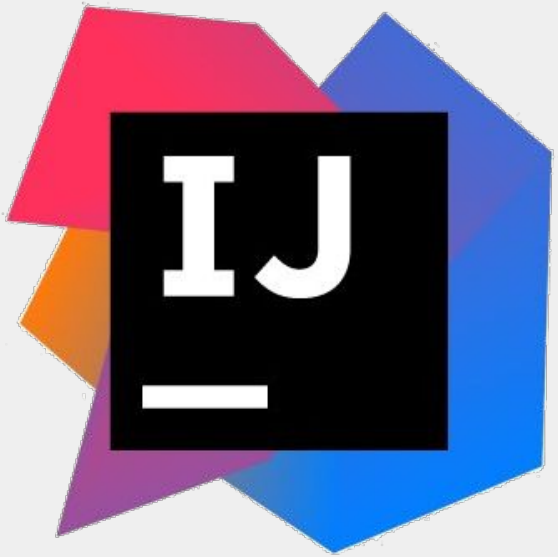
Design


MessagePost
username
message
timestamp
likes
comments
like
unlike
addComment
getText
getTimeStamp
display
PhotoPost
username
filename
caption
timestamp
likes
comments
like
unlike
addComment
getImageFile
getCaption
getTimeStamp
display
top half
shows fields
bottom half
shows methods

Design


: NewsFeed
messages
photos
: ArrayList<MessagePost>
: MessagePost
: MessagePost
: MessagePost
: MessagePost
: ArrayList<PhotoPost>
: PhotoPost
: PhotoPost
: PhotoPost
: PhotoPost

IntelliJ Demo Time



Assessment

We spot the following issues:

- Code duplication.
- The order in which posts are added is lost.
- Both `MessagePost` and `PhotoPost` contain `like` methods that do the same thing.
- *But* they both contain `display` methods that do very different things.

Assessment

We spot the following issues:

- Code duplication.
- The order in which posts are added is lost.
- Both `MessagePost` and Pho
  methods that do the same th
- *But* they both contain `displ`
  different things.

Moreover, this sample has implemented only two types of Post.
We know that there are in fact going to be several more.

NewsFeed

```
private ArrayList <MessagePost> messages;
private ArrayList <PhotoPost> photos;

public NewsFeed () {
    this.messages = new ArrayList <> ();
    this.photos = new ArrayList <> ();
}

public void addMessagePost (MessagePost m) {
    messages.add (m);
}

public void addPhotoPost (PhotoPost p) {
    photos.add (p);
}
```

NewsFeed

```
private ArrayList <MessagePost> messages;
private ArrayList <PhotoPost> photos;

public NewsFeed () {
    this.messages = new ArrayList <> ();
    this.photos = new ArrayList <> ();
}

public void addMessagePost (MessagePost
    messages.add (m);
}

public void addPhotoPost (PhotoPost p)
    photos.add (p);
}
```

Adding a New Post Type

1. Add a new ArrayList.
2. Extend the constructor.
3. Add a new "add" method.
4. Extend the "show" method.

NewsFeed

```
public void show () {

    for (MessagePost m : messages) {
        m.display ();
        System.out.println ();
    }

    for (PhotoPost p : photos) {
        p.display ();
        System.out.println ();
    }
}
```

Adding a New Post Type
And as we're limited to these loops, any new Post type would have to appear all together, losing the order of adding.

To Summarise

There is code duplication:

- `MessagePost` and `PhotoPost` are very similar (large parts are identical).
- `NewsFeed` must repeat code for each different Post type.

This makes maintenance harder, requires extra typing, induces RSI, and introduces a danger of bugs through sloppy maintenance.

To Summarise

There is code duplication:

- `MessagePost` and `PhotoPost` are very similar (large parts are identical).
- `NewsFeed` must repeat code for each different Post type.

This makes maintenance harder, rec
RSI, and introduces a danger of bugs
maintenance.

Anyone remember the idea of a "Code Smell"?
We have a big one here.

A Better Design

We reason like this:

- `NewsFeed` is a collection of posts.
- Some posts are messages, some are photos.
- `NewsFeed` will be easier if it handles just Posts.
- `Post` will be a class containing all the common parts.
- `MessagePost` and `PhotoPost` will contain the specialised parts.

A Better Design

We reason like this:

- `NewsFeed` is a collection of posts.
- Some posts are messages, some are photos.
- `NewsFeed` will be easier if it han
- `Post` will be a class containing a
- `MessagePost` and `PhotoPos`
  parts.

This is, of course, inheritance.
In this scheme adding a new type of post will be easy.

A Better Design

We reason like this:

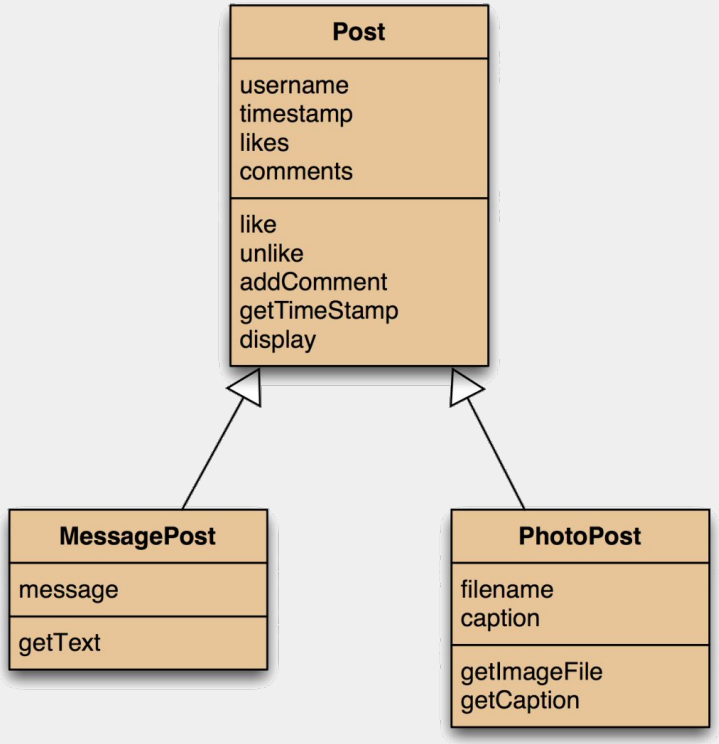
- `NewsFeed` is a collection of posts.
- Some posts are messages, some are photos.
- `NewsFeed` will be easier if it han
- `Post` will be a class containing a
- `MessagePost` and `PhotoPos` parts.

A Subtle Point

There is actually no such thing as a `Post`. Every post in the news feed is going to be one of the specialised types.

We say that `Post` is *abstract*.

Using Inheritance



Implementation

1. Define the *superclass*: `Post`.
2. Define *subclasses* for `MessagePost` and `PhotoPost`.

The superclass defines common attributes and methods.

The subclasses *inherit* the superclass attributes and methods.

The subclasses add their own, specialised, attributes and methods.

Implementation

First, define your superclass:

```
public class Post
{
    // Common attributes and methods here.
}
```

Nothing new here, so move along.

Implementation

First, define your superclass:

```
abstract class Post
{
    // Common attributes and methods
}
```

Nothing new here, so move along.

Actually there is something new.
Spot it?

Implementation

Now, add a subclass:

```
public class MessagePost extends Post
{
    // Specific attributes and methods here.
}
```

Simples.

Implementation

And, add a second subclass:

```
public class PhotoPost extends Post
{
    // Specific attributes and methods here.
}
```

Still simples.

Implementation

The two subclasses have all the attributes of the superclass, along with their specific ones.

They also have their own methods, plus the methods from the superclass.

They *cannot* access the attributes of the superclass directly - they use the public interface.

Implementation

The two subclasses have all the attributes of the superclass, along with their specific ones.

They also have their own methods, plus the methods from the superclass.

They *cannot* access the attributes of [illegible] use the public interface.

The bottom statement on this slide is not strictly true.
But this version of the truth will do until next week.

Implementation: Constructors

For `Post`, this is just the same as before:

```
public Post (String author)
{
    username = author;
    likes = 0;
    comments = new ArrayList <String> ();
}
```

Implementation: Constructors

In the subclasses, there must be a call to the superclass constructor:

```
public MessagePost (String author, String text)
{
    super (author);
    message = text;
}
```

Implementation: Constructors

In the subclasses, there must be a call to the superclass constructor:

```
public MessagePost (String author, String text)
{
    super (author);
    message = text;
}
```

Fiddly Detail
The `super` call must be the first statement in the subclass constructor.

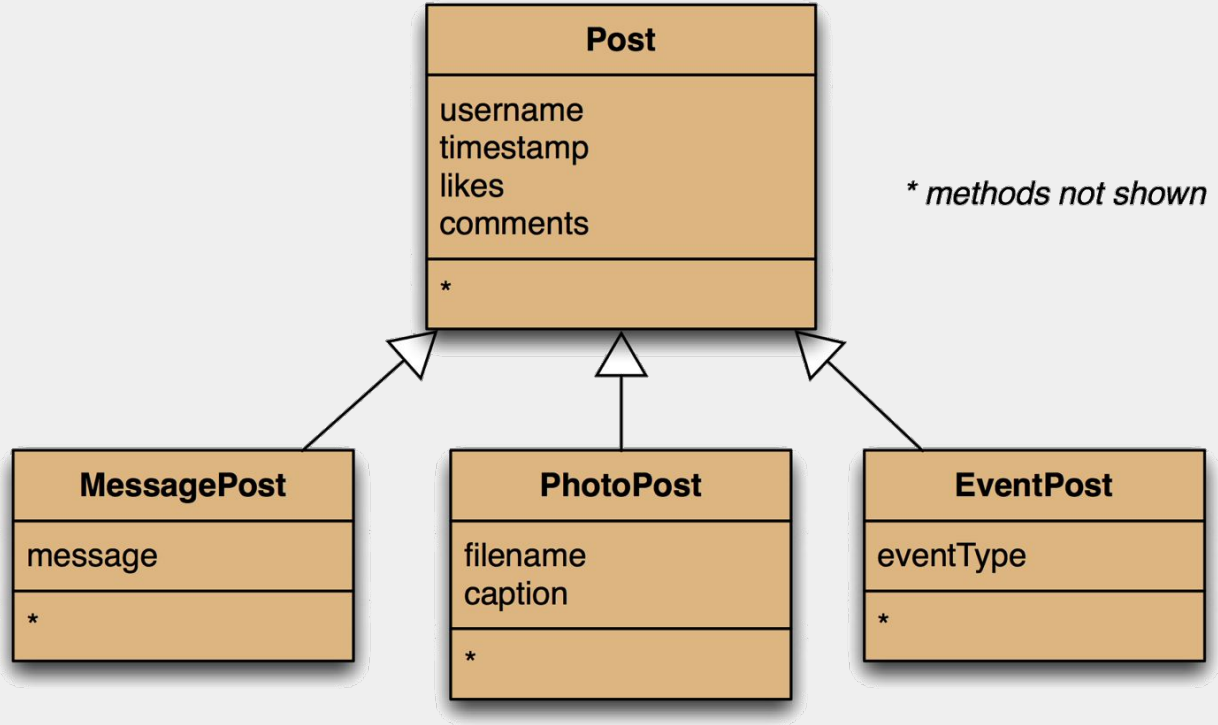
Implementation: Constructors

In the subclasses, there must be a call to the superclass constructor:

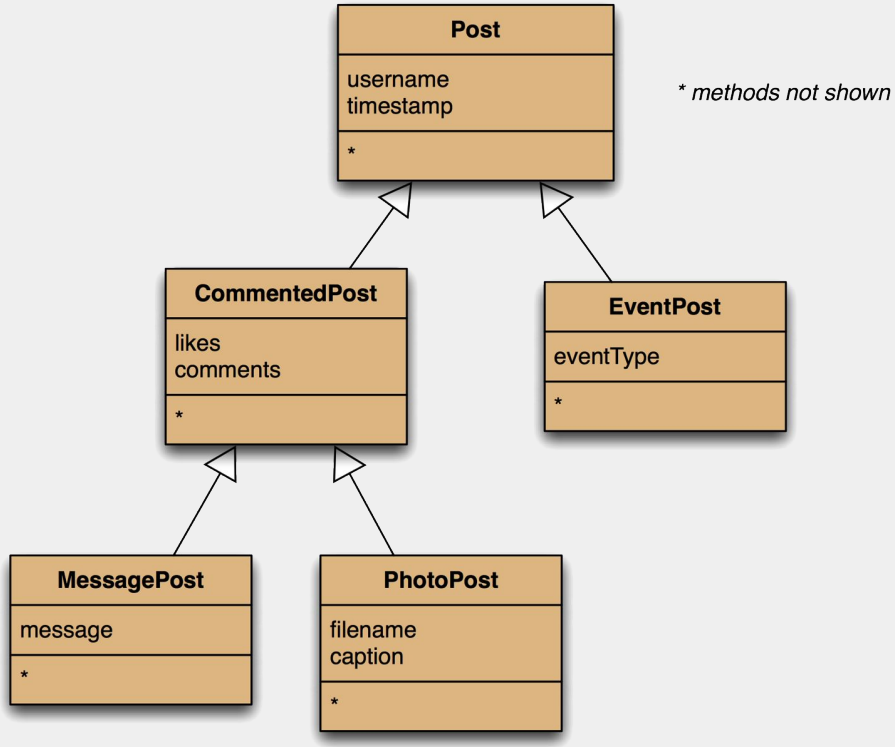
```
public MessagePost (String author, String text)
{
    super (author);
    message = text;
}
```

Fiddly Detail
If it is missing, Java will insert one implicitly (with no parameters), which is rarely what is wanted.

Adding More Post Types

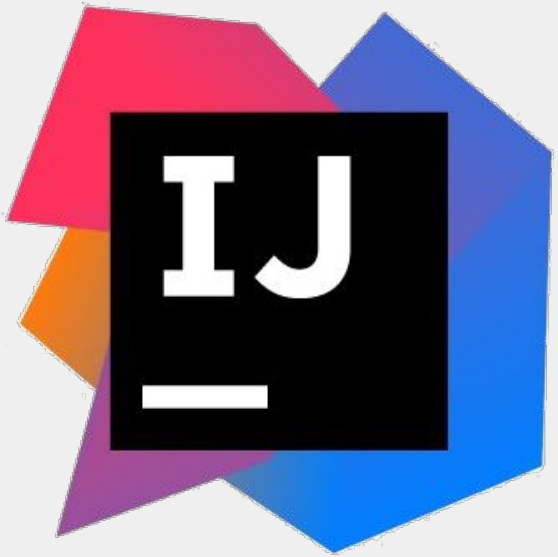


Deeper Hierarchies



** methods not shown*

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 sort next week.

IntelliJ Demo Time