Improving structure with inheritance

The Network example

- A small, prototype social network

- Supports a news feed with posts

- Stores *text posts* and *photo posts*
  - `MessagePost`: multi-line text message
  - `PhotoPost`: photo and caption

- Allows operations on the posts:
  - e.g. search, display and remove

2

Network objects

: MessagePost

username	
message	
timestamp	
likes	
comments	

: PhotoPost

username	
filename	
caption	
timestamp	
likes	
comments	

3

Network classes

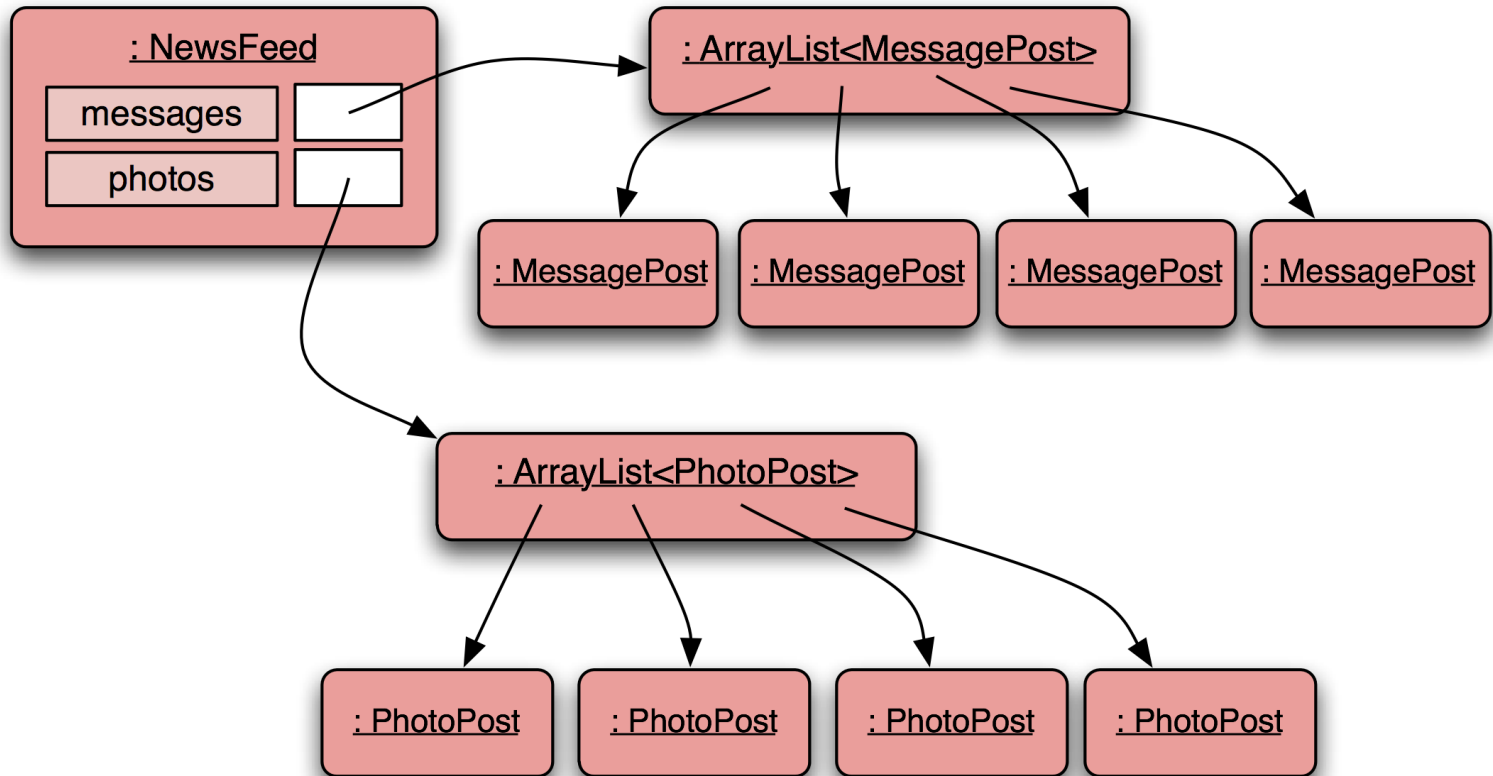| MessagePost |
|---|
| username
message
timestamp
likes
comments |
| like
unlike
addComment
getText
getTimeStamp
display |

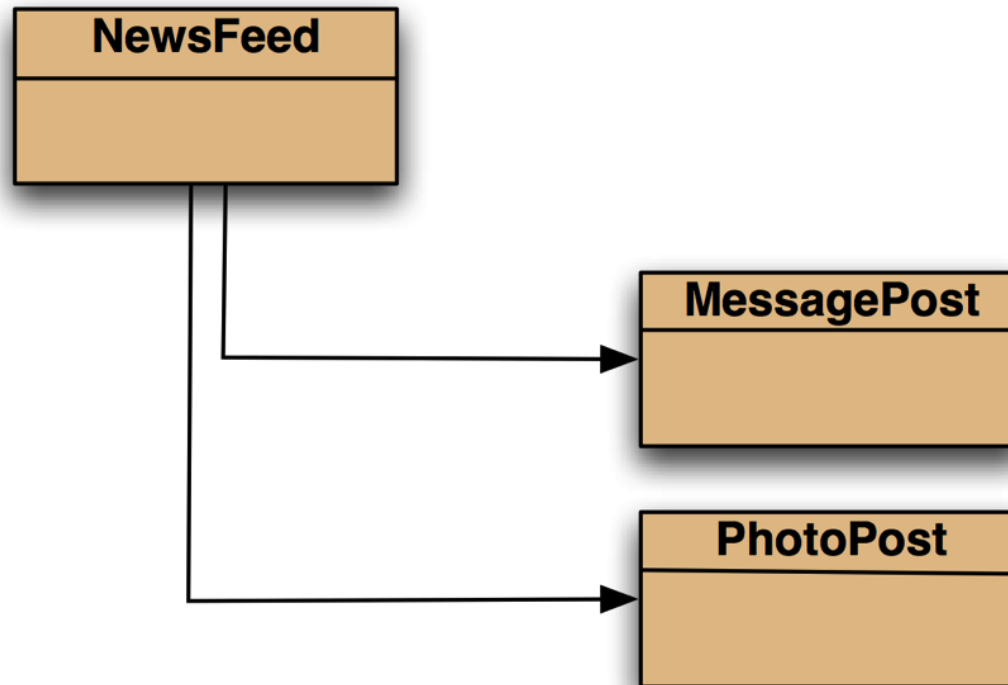| PhotoPost |
|---|
| username
filename
caption
timestamp
likes
comments |
| like
unlike
addComment
getImageFile
getCaption
getTimeStamp
display |

*top half
shows fields*

*bottom half
shows methods*

Network object model

5

Class diagram

6

Message-Post source code

Just an outline

```
public class MessagePost
{
    private String username;
    private String message;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    public MessagePost(String author, String text)
    {
        username = author;
        message = text;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }

    public void addComment(String text) ...

    public void like() ...

    public void display() ...

    ...
}
```

Photo-Post source code

Just an outline

```java
public class PhotoPost
{
    private String username;
    private String filename;
    private String caption;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    public PhotoPost(String author, String filename,
                     String caption)
    {
        username = author;
        this.filename = filename;
        this.caption = caption;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }

    public void addComment(String text) ...
    public void like() …
    public void display() …
    ...
}
```

NewsFeed

```java
public class NewsFeed
{
    private ArrayList<MessagePost> messages;
    private ArrayList<PhotoPost> photos;
    ...
    public void show()
    {
        for(MessagePost message : messages) {
            message.display();
            System.out.println();  // empty line between posts
        }

        for(PhotoPost photo : photos) {
            photo.display();
            System.out.println();  // empty line between posts
        }
    }
}
```
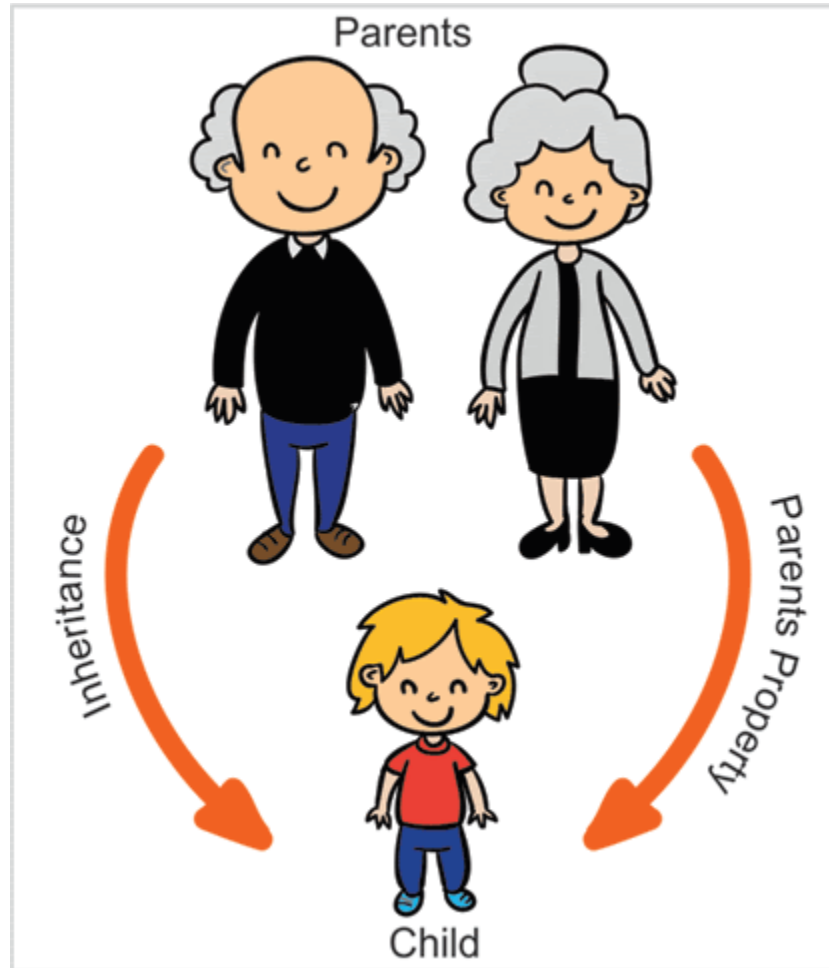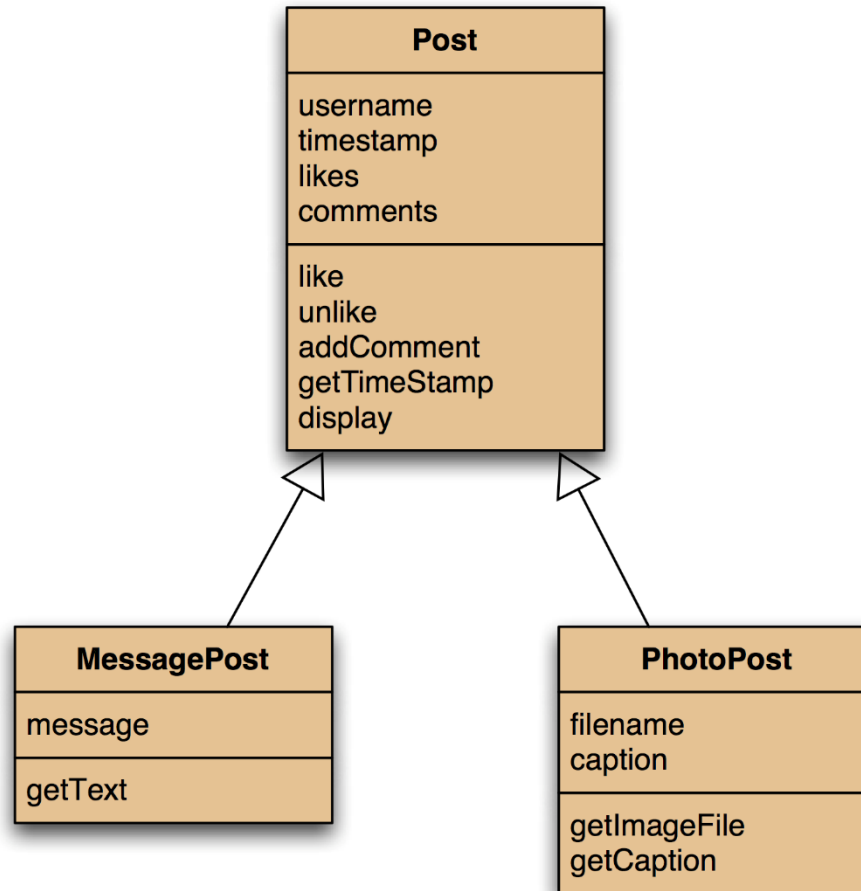
Critique of Network

- Code duplication:
  - **MessagePost** and **PhotoPost** classes are very similar (large parts are identical)
  - makes maintenance difficult/more work
  - introduces danger of bugs through incorrect maintenance

- Code duplication in **NewsFeed** class as well

Using inheritance

Using inheritance

Using inheritance

- define one **superclass** : `Post`

- define **subclasses** for `MessagePost` and `PhotoPost`

- the superclass defines common attributes (via fields)

- the subclasses **inherit** the superclass characteristics

- the subclasses add other characteristics

13

Inheritance hierarchies
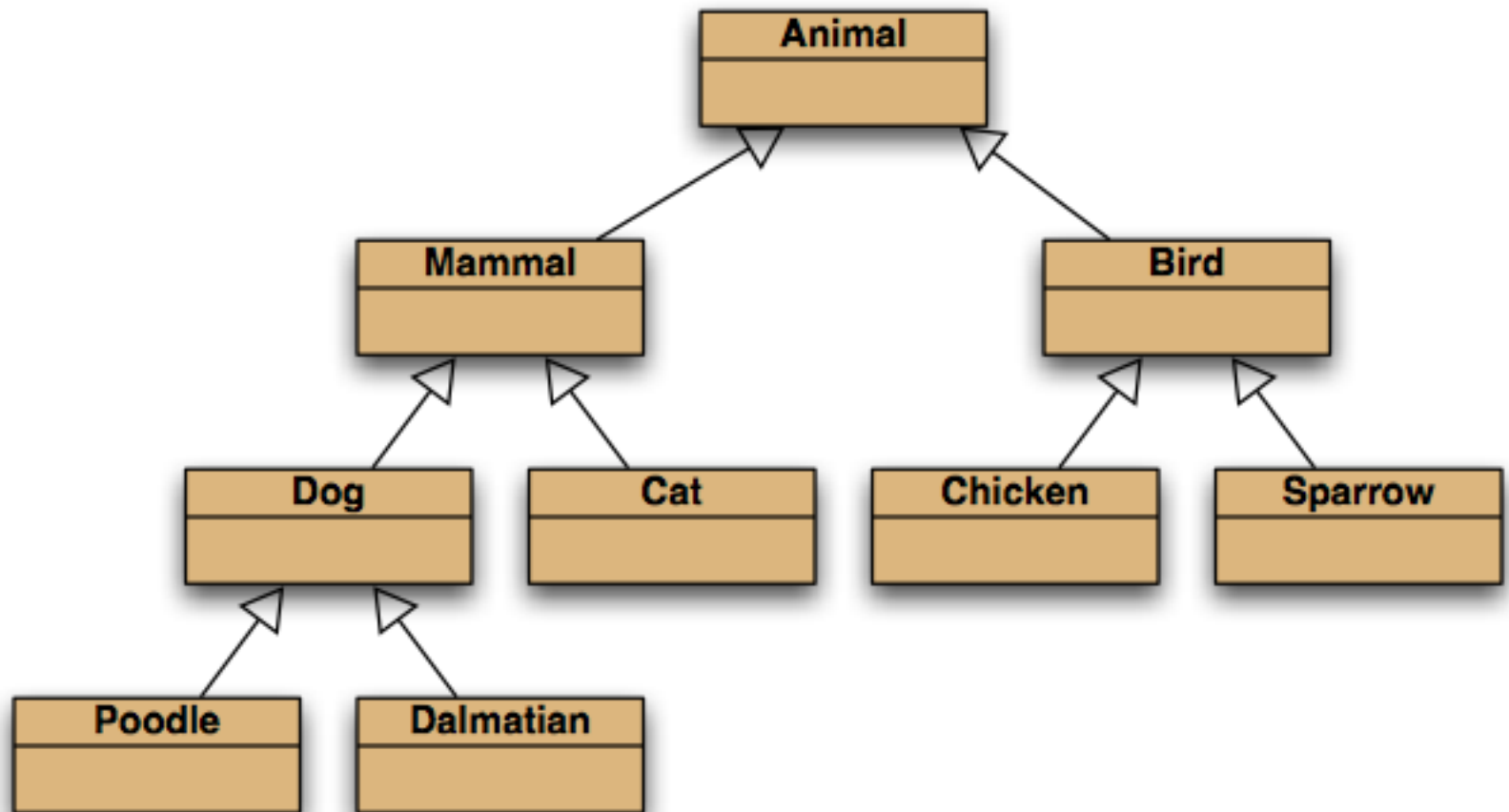
Inheritance in Java

```
public class Post
{
    ...
}
```
no change here

```
public class PhotoPost extends Post
{
    ...
}
```

```
public class MessagePost extends Post
{
    ...
}
```
change here

15

Superclass

```java
public class Post
{
    private String username;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    // constructor and methods omitted.
}
```

16

Subclasses

```java
public class MessagePost extends Post
{
    private String message;

    // constructor and methods omitted.
}
```

```java
public class PhotoPost extends Post
{
    private String filename;
    private String caption;

    // constructor and methods omitted.
}
```

17

```
public class Post
{
    private String username;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Initialise the fields of the post.
     */
    public Post(String author)
    {
        username = author;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<String>();
    }

    // methods omitted
}
```

18

Inheritance and constructors

```java
public class MessagePost extends Post
{
    private String message;

    /**
     * Constructor for objects of class MessagePost
     */
    public MessagePost(String author, String text)
    {
        super(author);      // MUST be first statement
        message = text;
    }

    // methods omitted
}
```

- Subclass must call superclass constructor!
- Must take values for all fields that we want to initialize!

Superclass constructor call

- Subclass constructors must always contain a *super* call

- If none is written, the compiler inserts one (without parameters)
  - only compiles if the superclass has a constructor without parameters

- Must be the first statement in the subclass constructor

Adding more item types

© 2017 Pearson Education, Inc. Hoboken, NJ. All rights reserved.

21
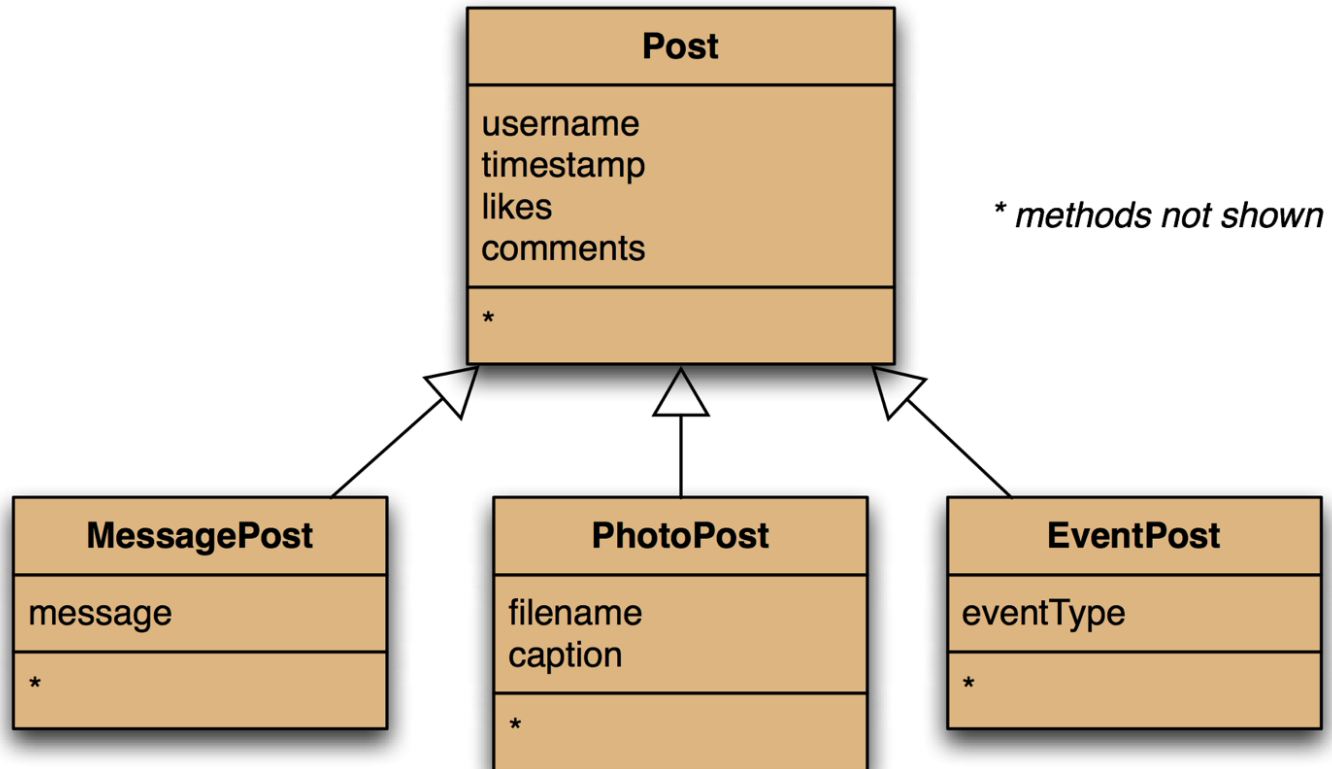
Deeper hierarchies
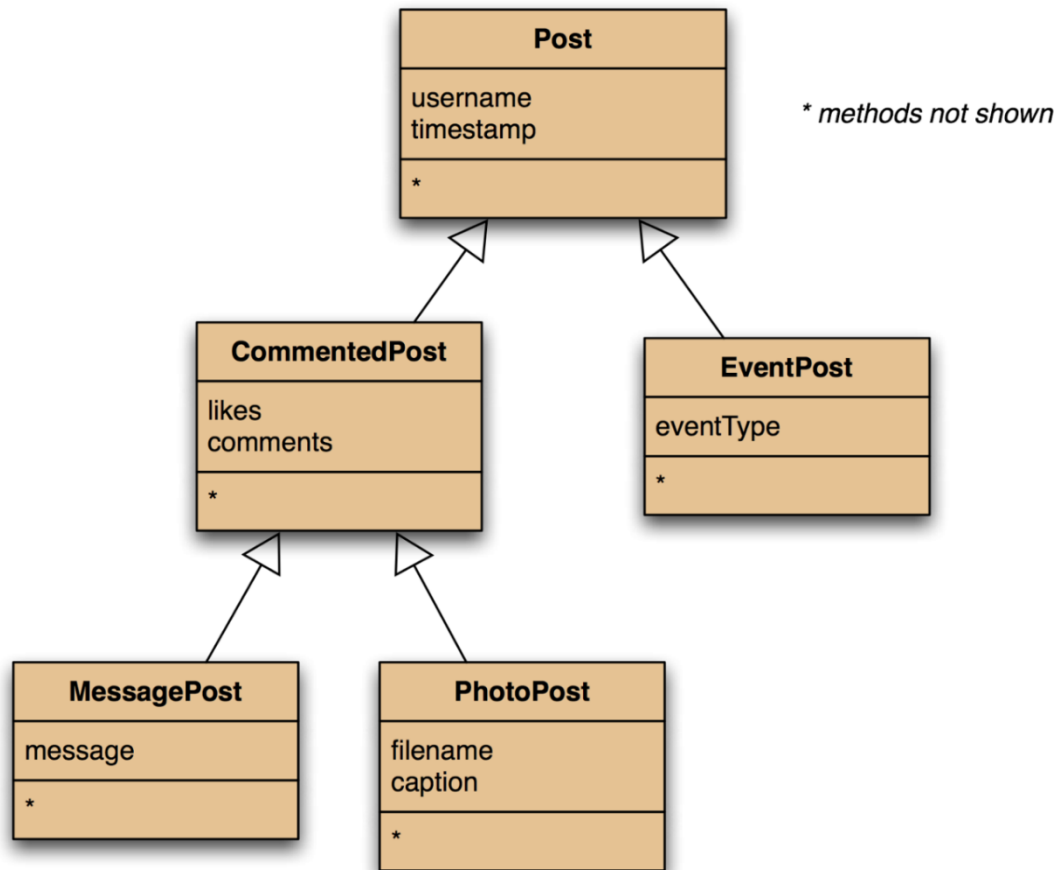
Abstract classes serve solely as a superclass and are not intended to be used to create instances.

Review (so far)

Inheritance (so far) helps with:

- Avoiding code duplication

- Code reuse

- Easier maintenance

- Extendibility

Revised NewsFeed source code

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

    /**
     * Construct an empty news feed.
     */
    public NewsFeed()
    {
        posts = new ArrayList<Post>();
    }

    /**
     * Add a post to the news feed.
     */
    public void addPost(Post post)
    {
        posts.add(post);
    }
    ...
}
```

avoids code duplication in the client class!

No longer a *messages* AND *photos* ArrayLists!!

New NewsFeed source code

```java
/**
 * Show the news feed. Currently: print the
 * news feed details to the terminal.
 * (Later: display in a web browser.)
 */
public void show()
{
    for(Post post : posts) {
        post.display();
        System.out.println(); // Empty line ...
    }
}
```

Now only 1 loop in the show method!!

Subtyping

First, we had:

```
public void addMessagePost(
                MessagePost message)
public void addPhotoPost(
                PhotoPost photo)
```
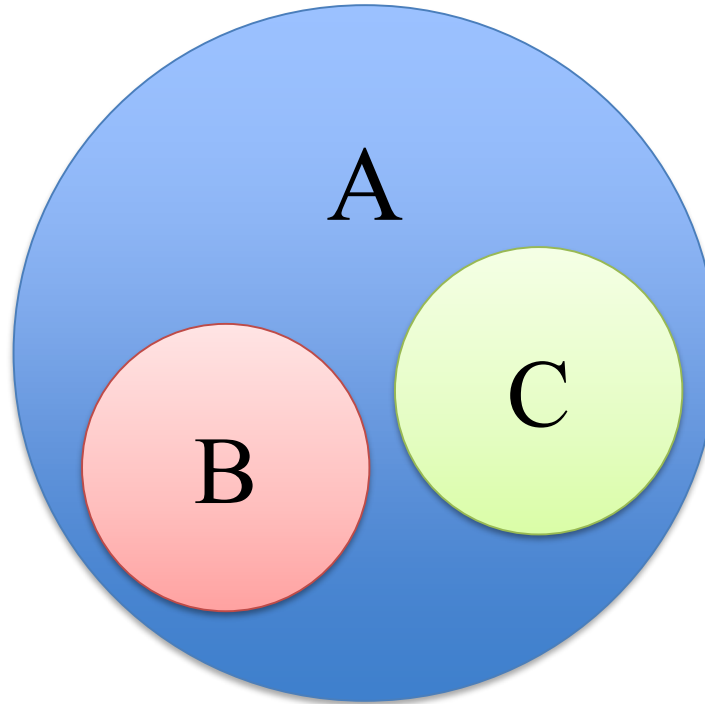
Now, we have:

```
public void addPost(Post post)
```

We call this method with:

```
PhotoPost myPhoto = new PhotoPost(...);
feed.addPost(myPhoto);
```

PhotoPost is a subtype of *Post*
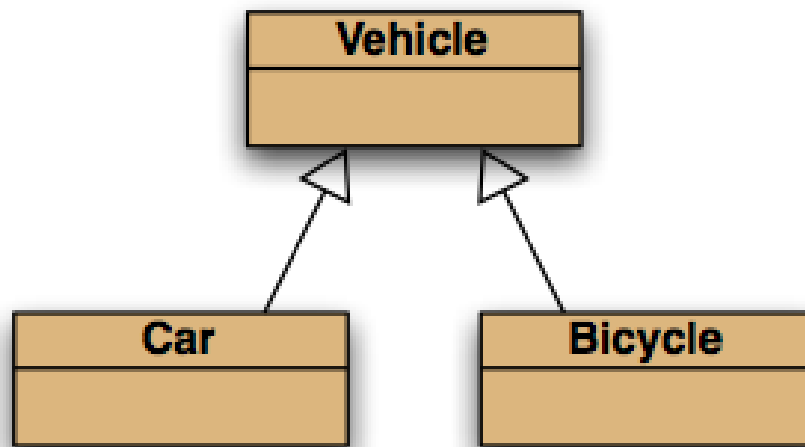
Subclasses and subtyping

Subclasses and subtyping

- Classes define types

- Subclasses define *subtypes*

- Objects of subclasses can be used where objects of supertypes are required ... called **substitution**

- But supertypes may NOT be used in place of a subtype

Subtyping and assignment

subclass objects may be assigned to superclass variables

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

Subtyping and parameter passing

```
public class NewsFeed
{

    public void addPost(Post post)
    {
        ...
    }
}
```
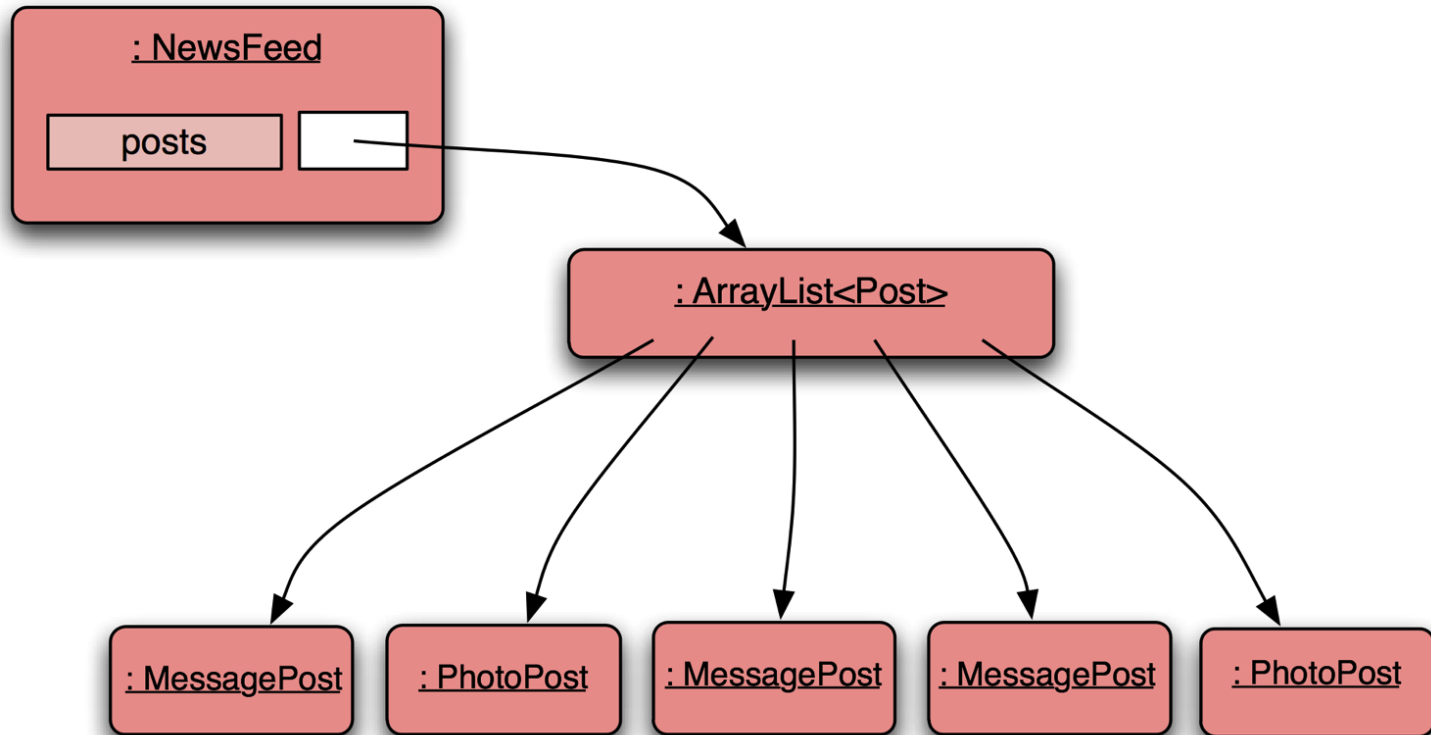
subclass objects may be used as actual parameters for the superclass

```
PhotoPost photo = new PhotoPost(...);
MessagePost message = new MessagePost(...);

feed.addPost(photo);
feed.addPost(message);
```
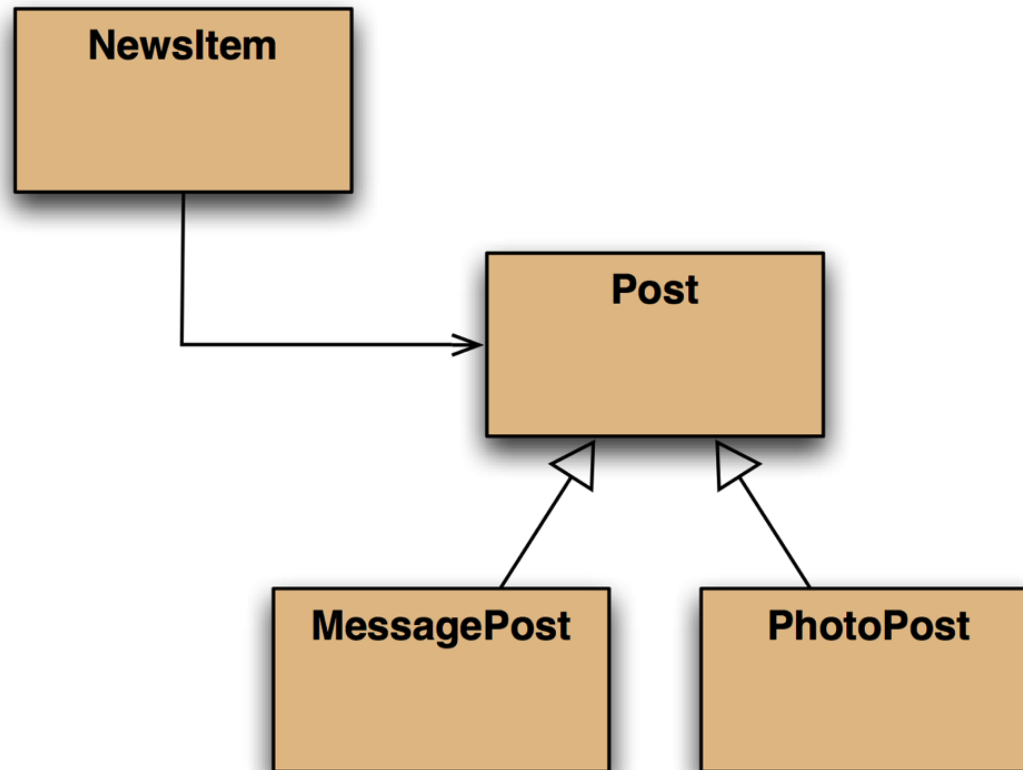
PhotoPost & *MessagePost* are both subtypes of *Post*

Object diagram

NewsFeed object can hold a single or mixed collection of supertype *Post* and subtypes PhotoPost/MessagePost

Class diagram

NewsItem **now only knows about** ***Post***
rather than the subclasses

Polymorphic variables

- Object variables in Java are **polymorphic** (many shapes)
  – Can hold objects of more than one type

- Can hold objects of the declared type, or of subtypes of the declared type

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

Variables of supertype *Post* may hold objects of subtypes PhotoPost/MessagePost

33

Casting

- We can assign subtype to supertype …

- … but we cannot assign supertype to subtype!

```
Vehicle v;
Car c = new Car();
v = c;              // correct
c = v;              // compile-time error!
```

- Casting fixes this:
  ```
  c = (Car) v;
  ```
  (only ok if the vehicle really is a Car!)

Casting

- An object type in parentheses

- Used to overcome 'type loss'

- The object is not changed in any way

- A runtime check is made to ensure the object really is of that type:

  - `ClassCastException` if it isn't!

- Use it sparingly

The Object class

All classes inherit from **Object**
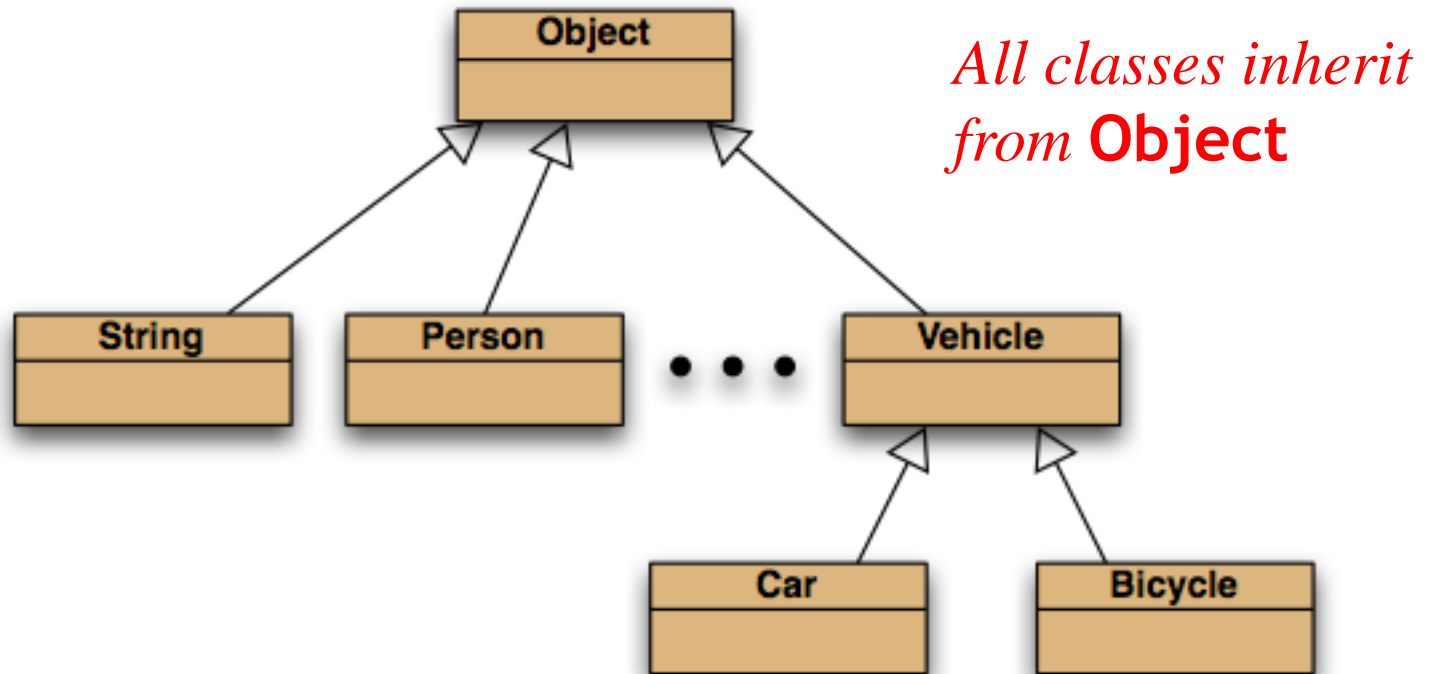
Object class from Java standard library

Polymorphic collections

- All collections are polymorphic

- The elements could simply be of type **Object**

```
public void add(Object element)

public Object get(int index)
```

- Usually avoided by using a type parameter with the collection

Polymorphic collections

- A type parameter limits the degree of polymorphism:
    ArrayList<Post>

- Collection methods are then typed

- Without a type parameter, **ArrayList<Object>** is implied

- Likely to get an *"unchecked or unsafe operations"* warning

- More likely to have to use casts

Review

- Inheritance allows the definition of classes as extensions of other classes

- Inheritance
  - avoids code duplication
  - allows code reuse
  - simplifies the code
  - simplifies maintenance and extending

- Variables can hold subtype objects

- Subtypes can be used wherever supertype objects are expected (substitution)