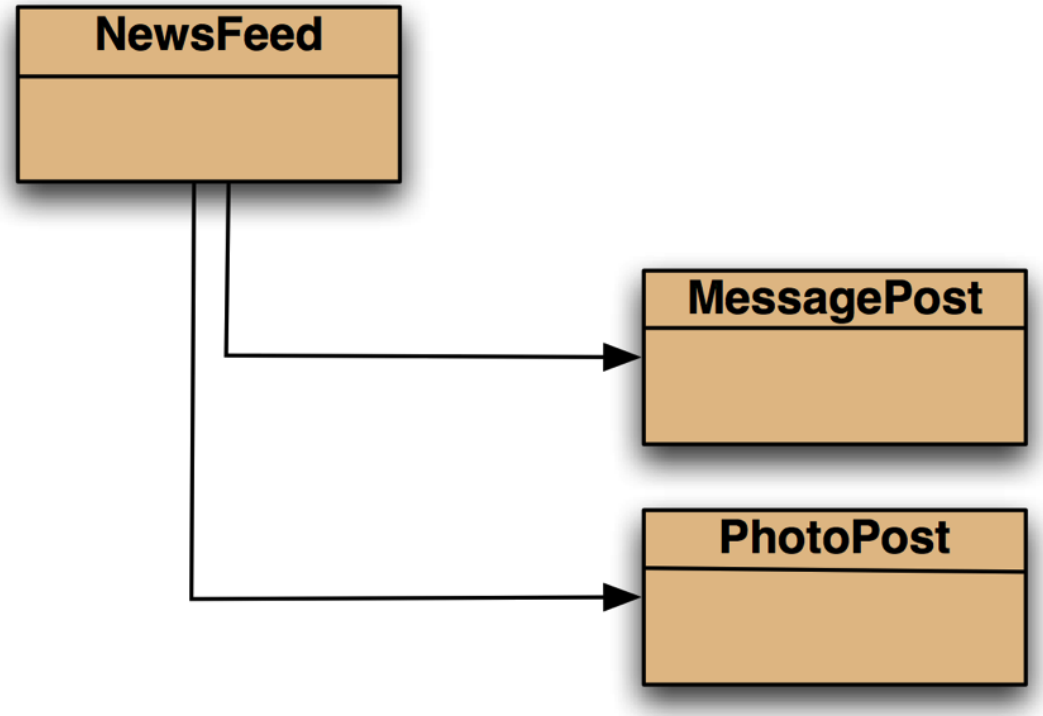
Improving Structure with Inheritance

Chapter-8;
Objects First with Java using BlueJ

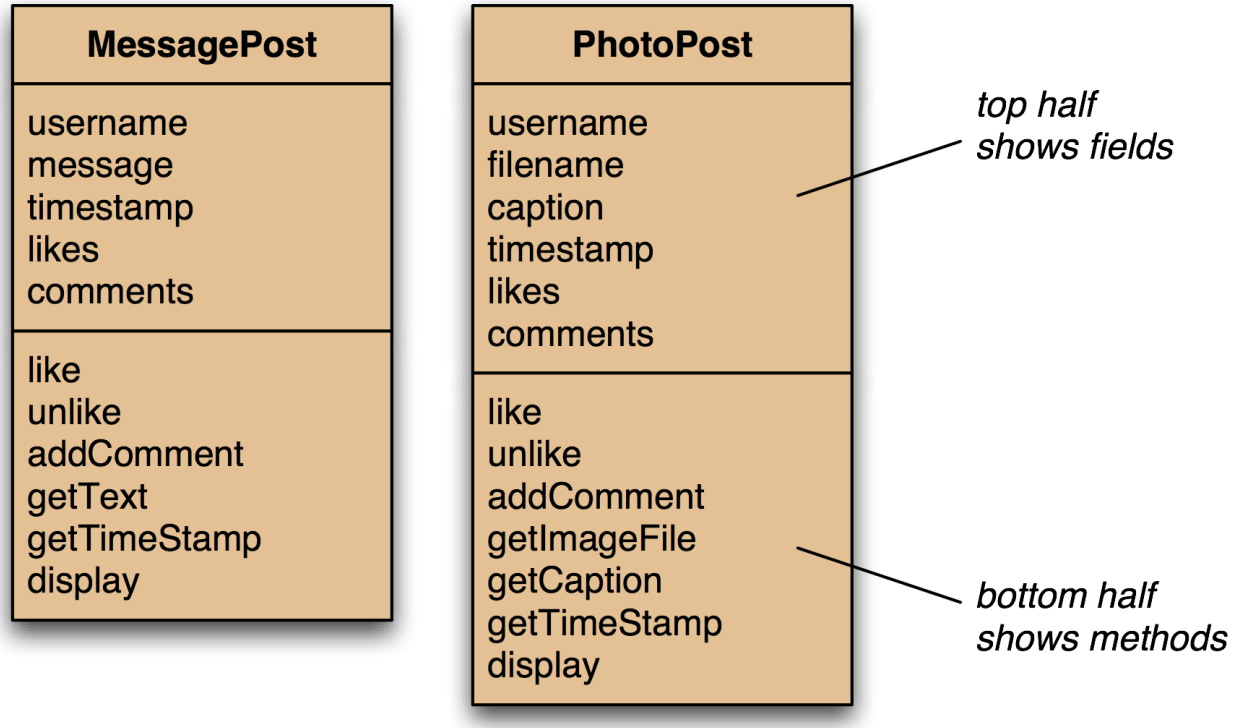
The Network example

- 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`.

Class diagram



Network classes



Message-Post source code

```
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<String>();
   }

   public void addComment(String text) { ... }

   public void like() { ... }

   public void display() { ... }

   ...
}
```

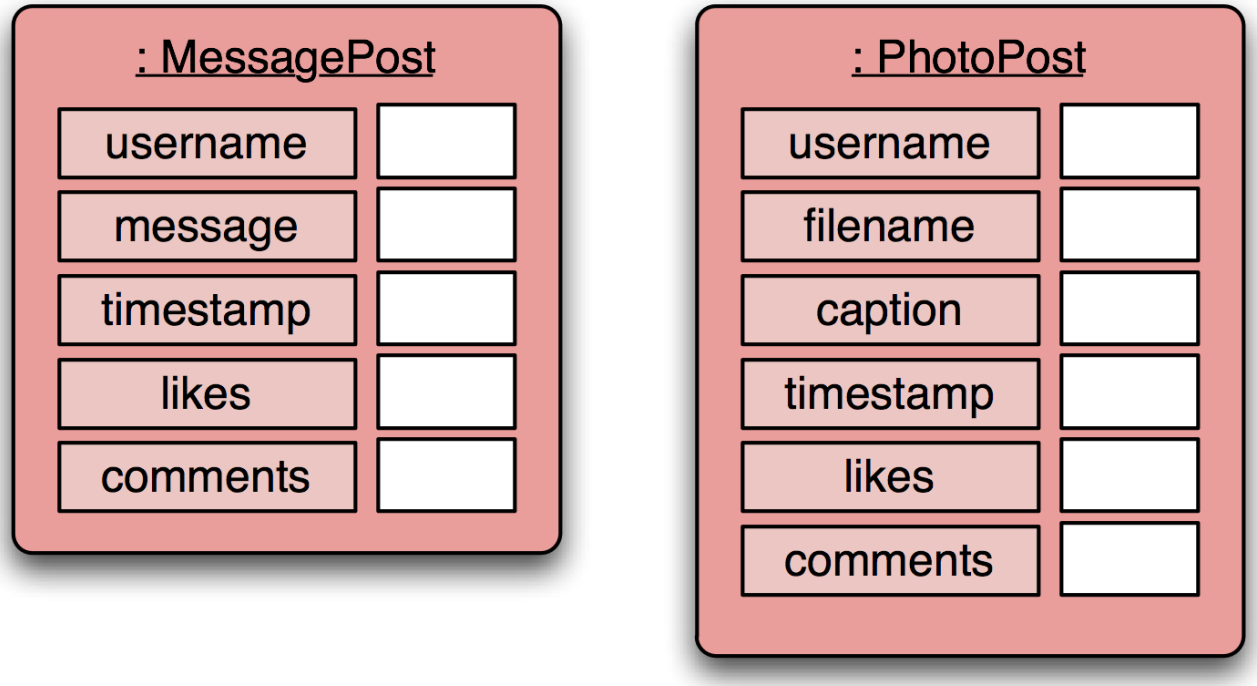
Photo-Post source code

```
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<String>();
   }

   public void addComment(String text) { ... }

   public void like() { ... }

   public void display() { ... }

   ...
}
```

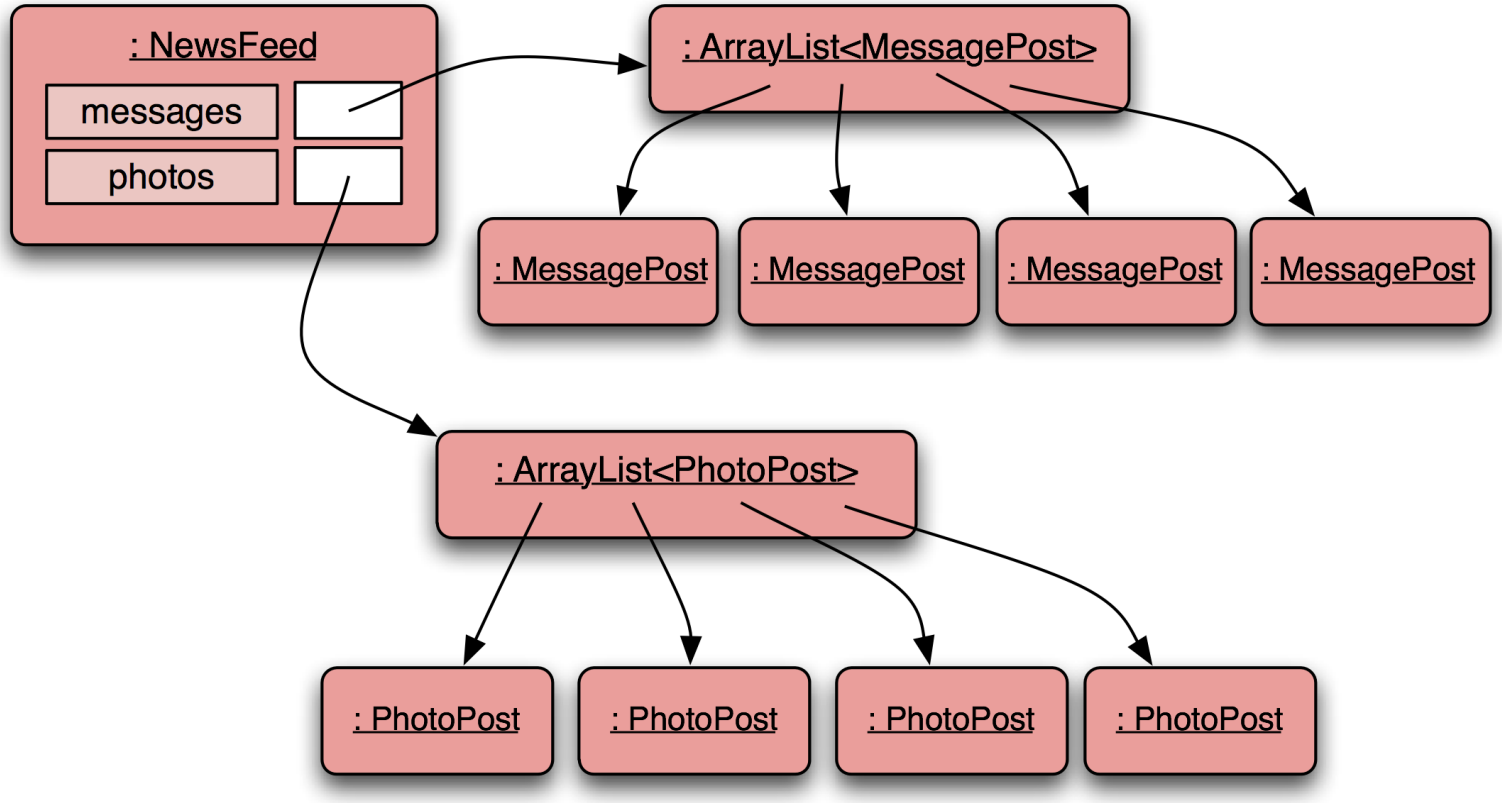
NewsFeed

```
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
       }
   }
}
```

Network objects



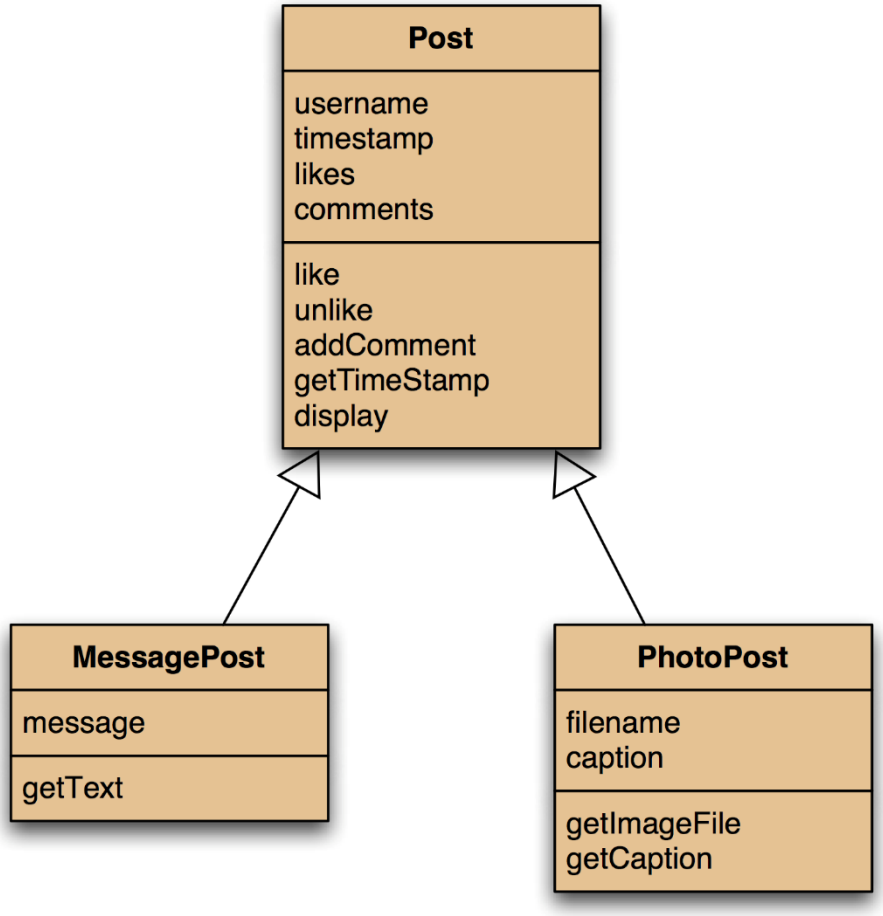
Network object model



Critique of Network

- Code duplication:
  - **`MessagePost`** and **`PhotoPost`** classes 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.

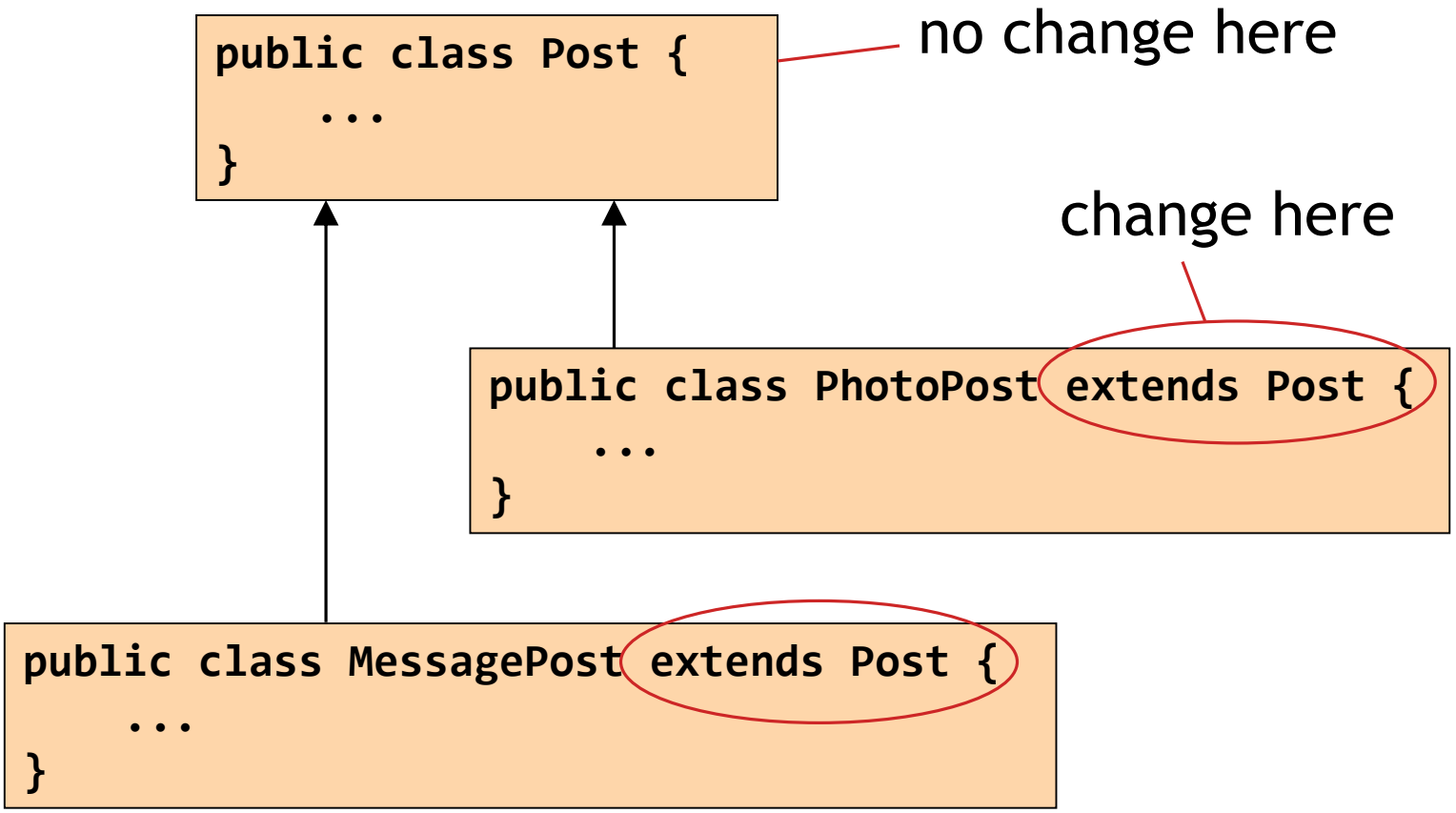
Using inheritance



Using inheritance

- define one **superclass** : **`Post`**
- define **subclasses** for **`MessagePost`** and **`PhotoPost`**
- the superclass defines common attributes (via fields)
- the subclasses **inherit** the superclass attributes
- the subclasses add other attributes

Inheritance in Java



Superclass

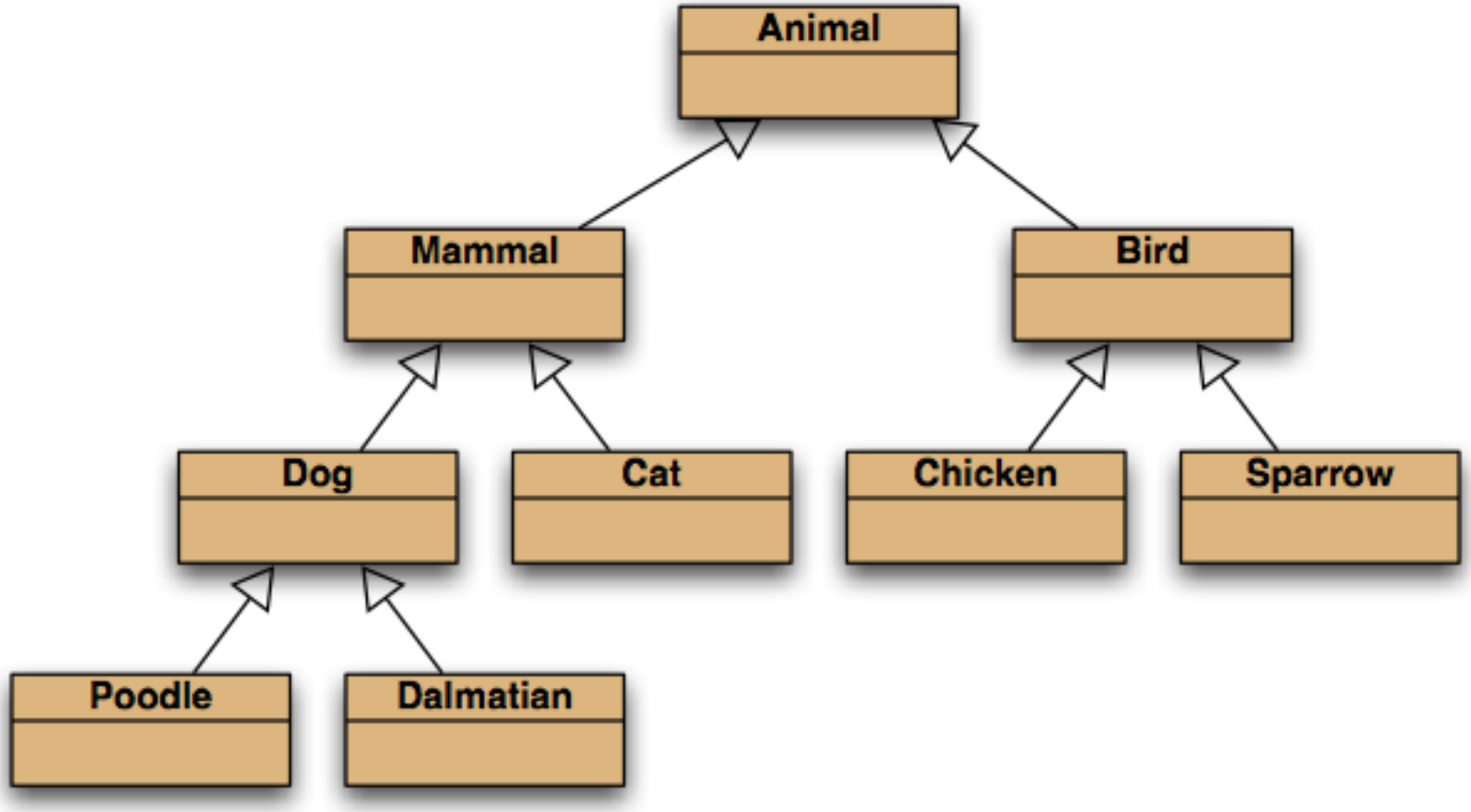
```
public class Post {

    private String username;
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    // constructor and methods omitted.
}
```

Subclasses

```
public class MessagePost extends Post {

    private String message;

    // constructor and methods omitted.
}
```

```
public class PhotoPost extends Post {

    private String filename;
    private String caption;

    // constructor and methods omitted.
}
```

Inheritance Hierarchies



Inheritance & constructors

```
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
}
```

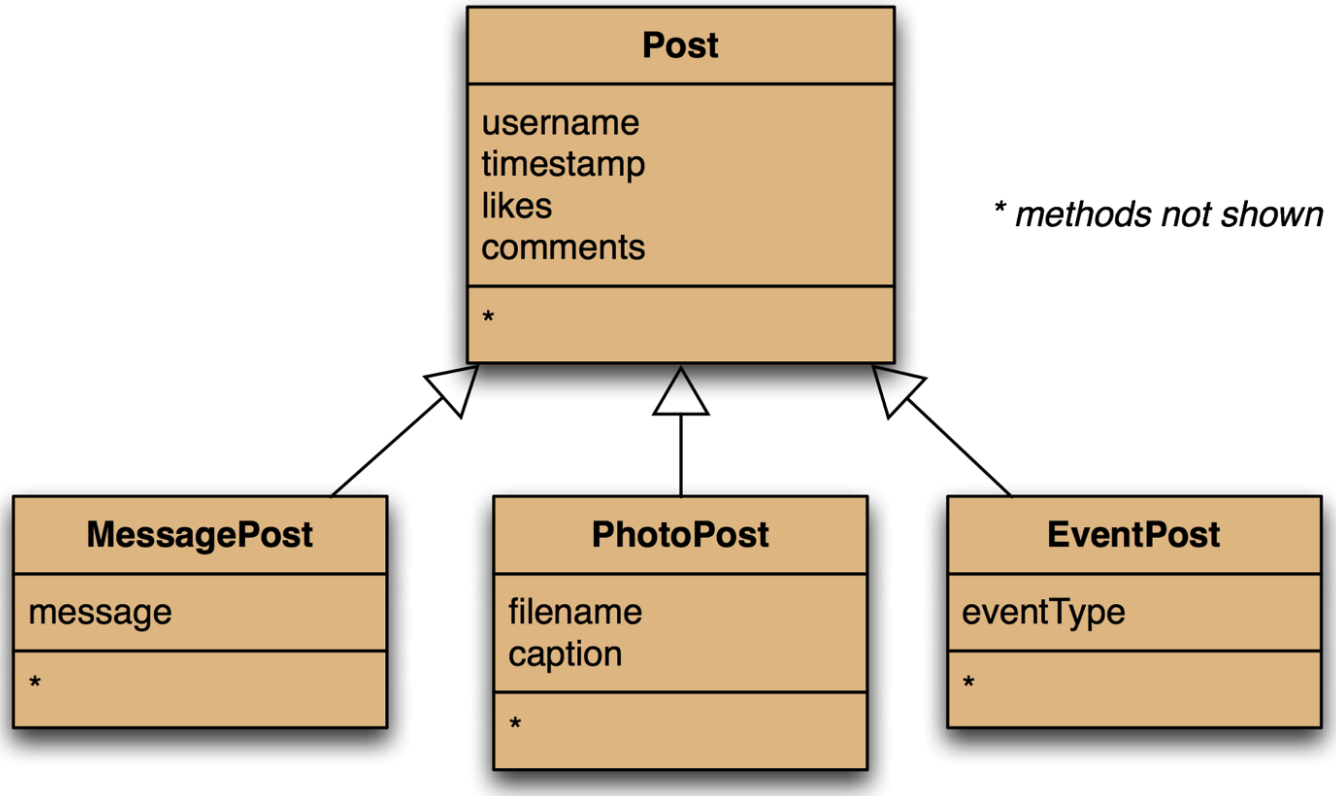
Inheritance & constructors

```
public class MessagePost extends Post {

    private String message;

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

    // methods omitted
}
```

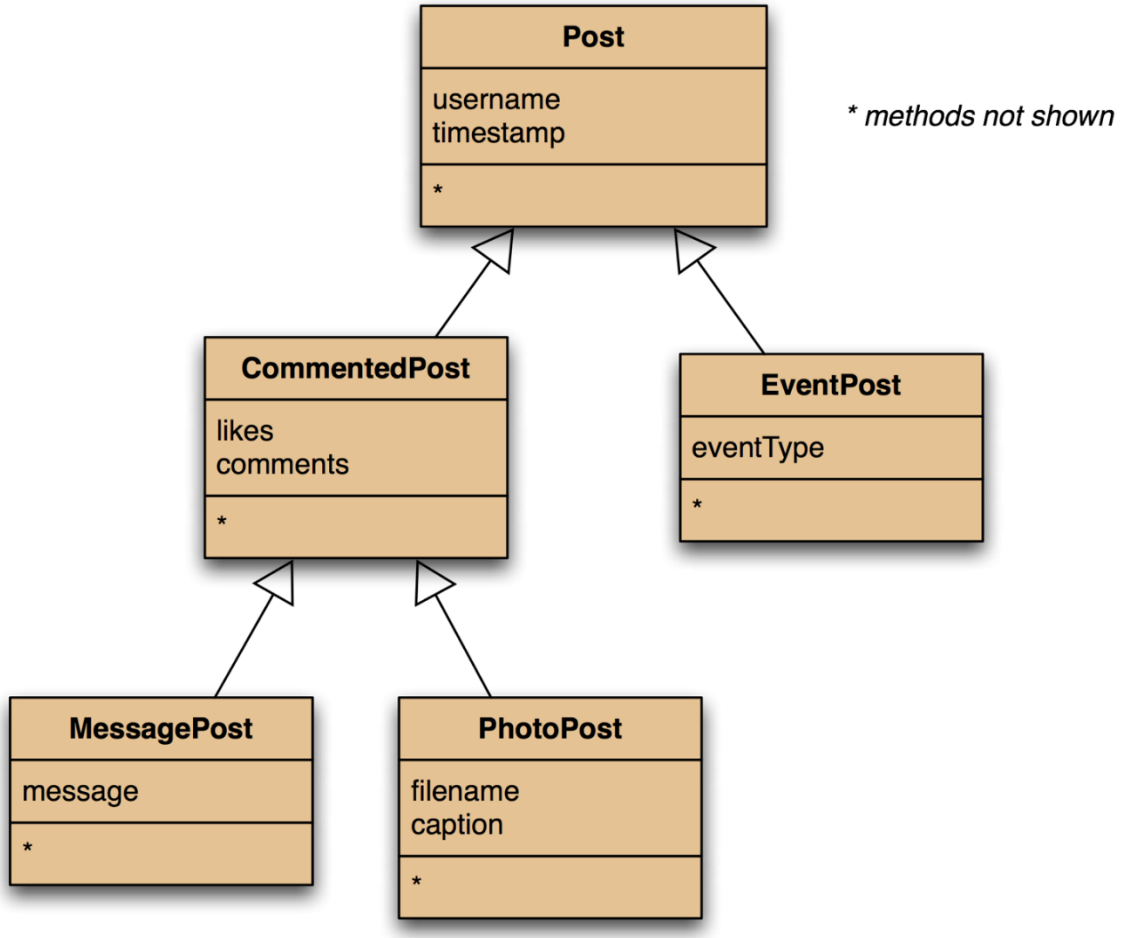
Superclass constructor call

- Subclass constructors must always contain a '**`super`**' call.
- If none is written, the compiler inserts one (without parameters)
  - works only, if the superclass has a constructor without parameters
- Must be the first statement in the subclass constructor.

Adding more item types



Deeper hierarchies



Review (so far)

Inheritance (so far) helps with:

- Avoiding code duplication
- Code reuse
- Easier maintenance
- Extendibility

Revised NewsFeed source code

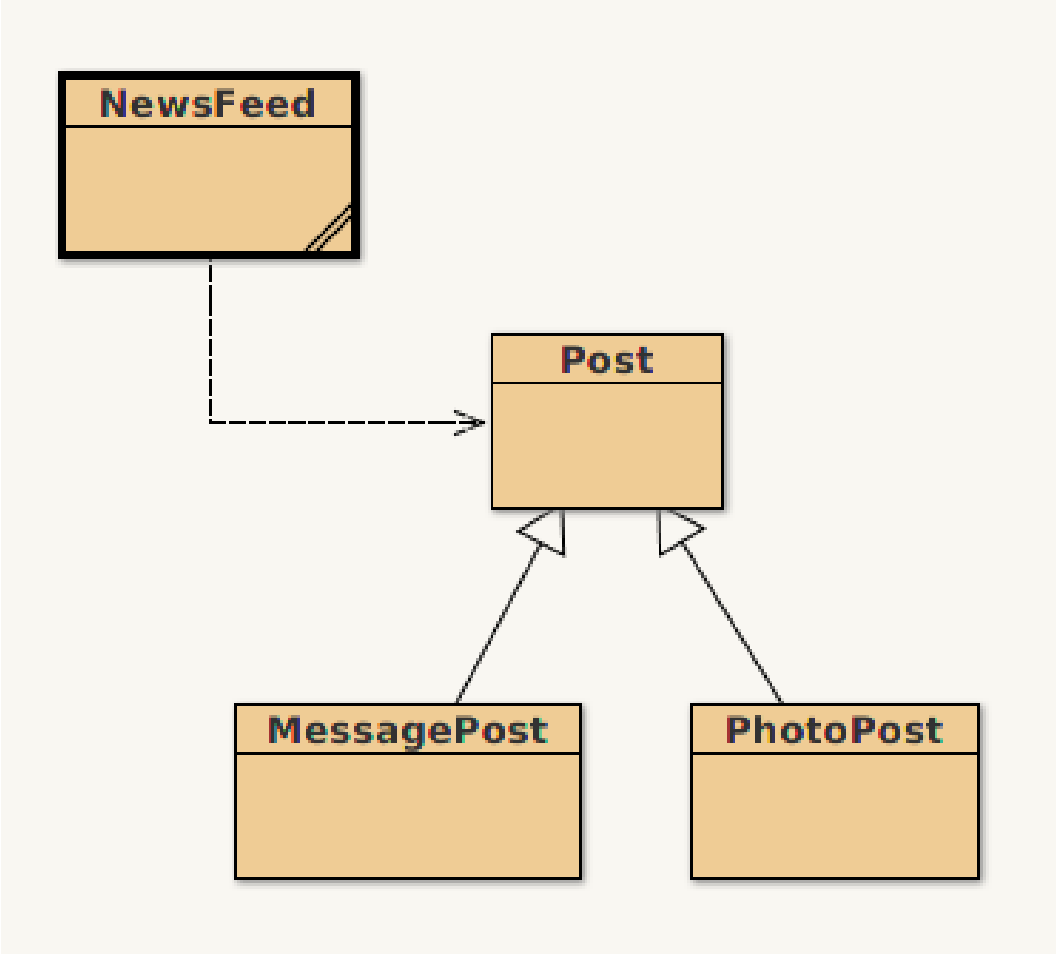
```
public class NewsFeed {

    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!

New NewsFeed source code

```
/**
 * 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 ...
    }
}
```

Improved Class diagram



Subtyping

First, we had:

```
public void addMessagePost(MessagePost message)
public void addPhotoPost(PhotoPost photo)
```

Now, we have:

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

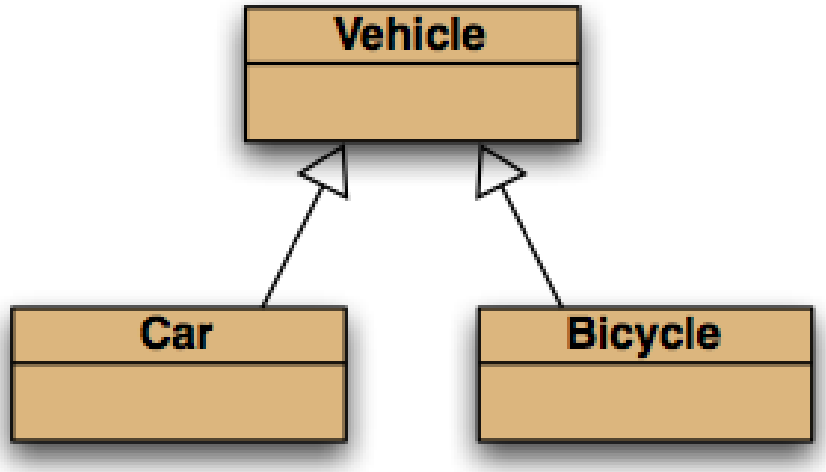
We call this method with:

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

Subclasses and subtyping

- Classes define types.
- Subclasses define *subtypes*.
- Objects of subclasses can be used where objects of super-types are required.
  (This is called **substitution**.)

Subtyping and assignment



subclass objects may be assigned to superclass variables

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

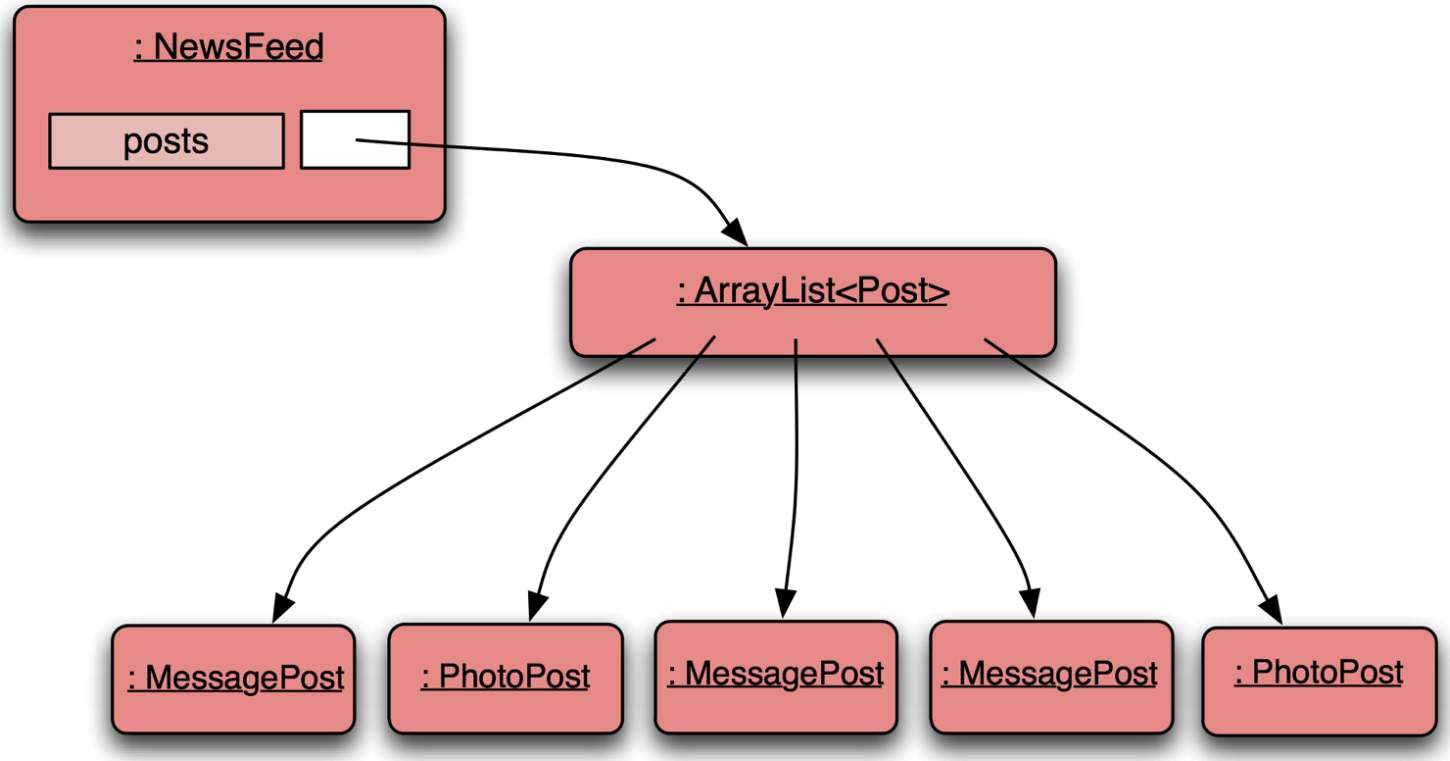
Subtyping & parameter passing

```
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(...);

NewsFeed feed = new NewsFeed();
feed.addPost(photo);
feed.addPost(message);
```

Object diagram



Polymorphic variables

- Variables in Java are **polymorphic.**

  They can hold objects of more than one type.

- They can hold objects of the declared type, or of subtypes of the declared type.

Casting

- We can assign sub-type to super-type;
- But we cannot assign super-type to sub-type!

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

- Casting fixes this:

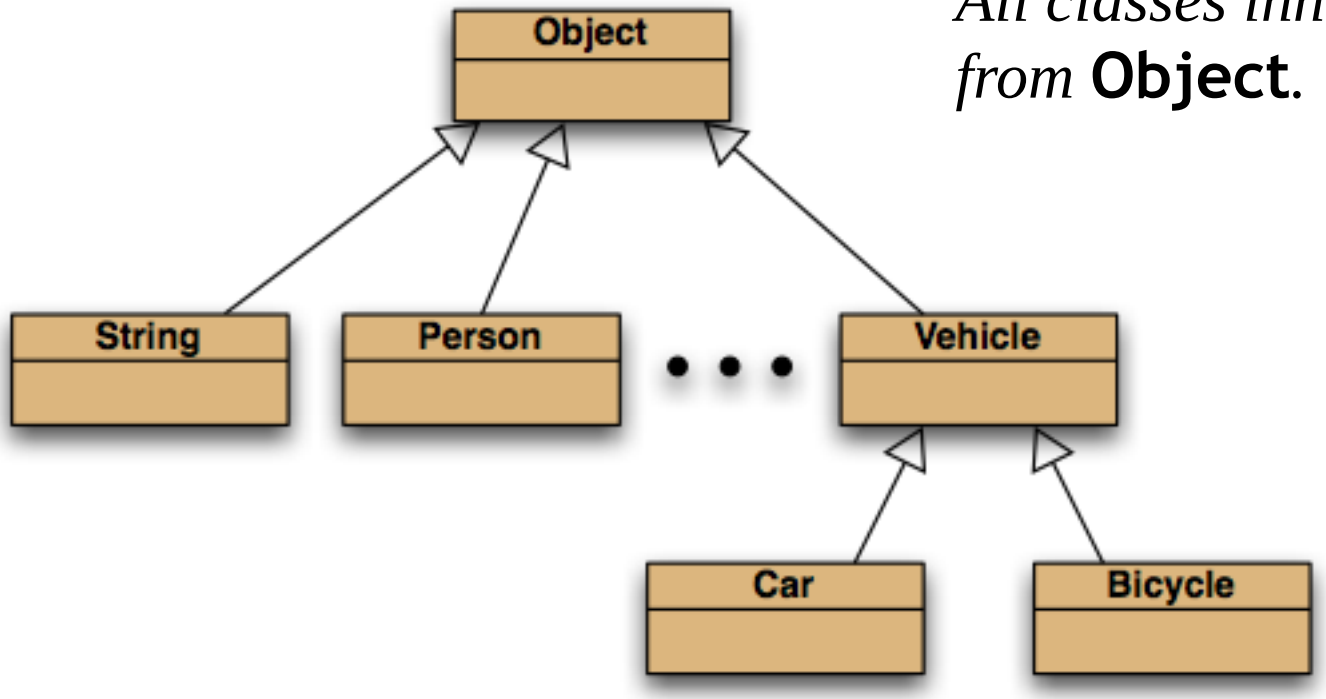
```
c = (Car) v;
```

(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**.

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.

Collections and primitive types

- Potentially, all objects can be entered into collections ... because collections can accept elements of type **`Object`** ... and all classes are subtypes of **`Object`**.
- But what about *the primitive types*: **`int`**, **`boolean`**, etc ?

Wrapper classes

- Primitive types are not objects types. Primitive-type values must be wrapped in objects to be stored in a collection!
- Wrapper classes exist for all primitive types:

simple type	*wrapper class*
`int`	`Integer`
`float`	`Float`
`char`	`Character`
`...`	`...`

Wrapper classes

wrap the value

```
int i = 18;
Integer iwrap = new Integer(i);
…
int value = iwrap.intValue();
```

unwrap it

In practice, *autoboxing* and *unboxing* mean we don't often have to do this explicitly

Autoboxing and unboxing

```
private ArrayList<Integer> markList;

…

public void storeMark(int mark) {
    markList.add(mark);
}
```

autoboxing

```
int firstMark = markList.remove(0);
```

unboxing

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).