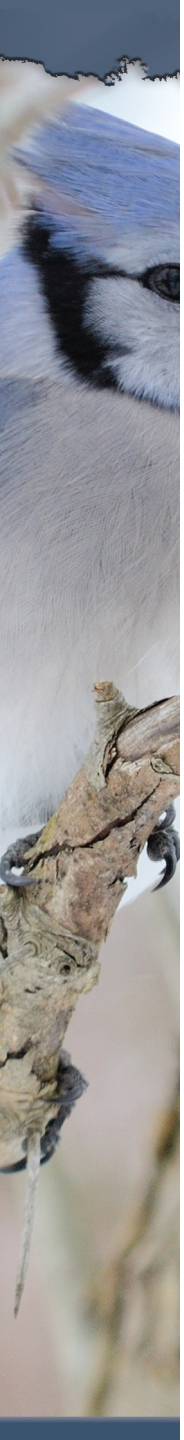
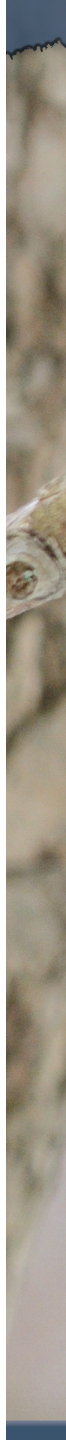
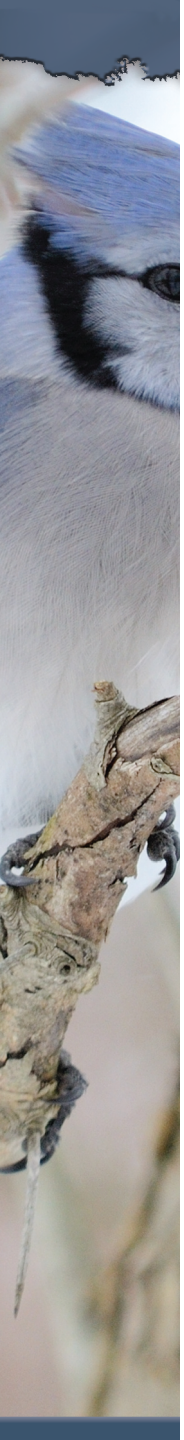


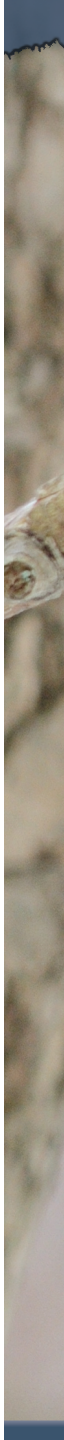
Improving structure with inheritance

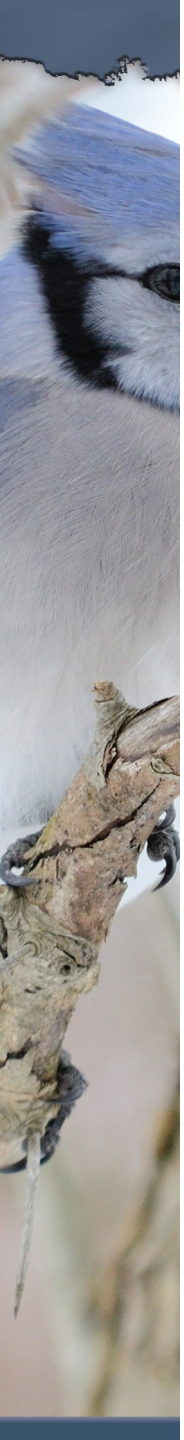




Main concepts to be covered

- Inheritance
- Subtyping
- Substitution
- Polymorphic variables
- Code recycling

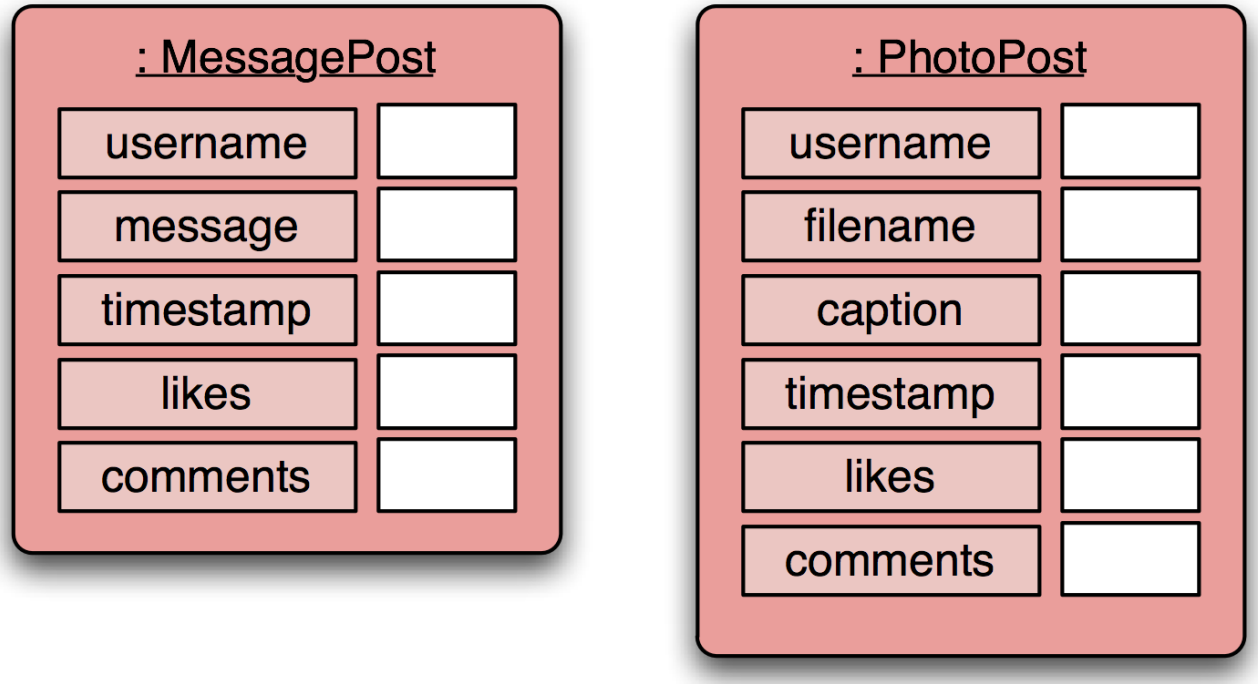




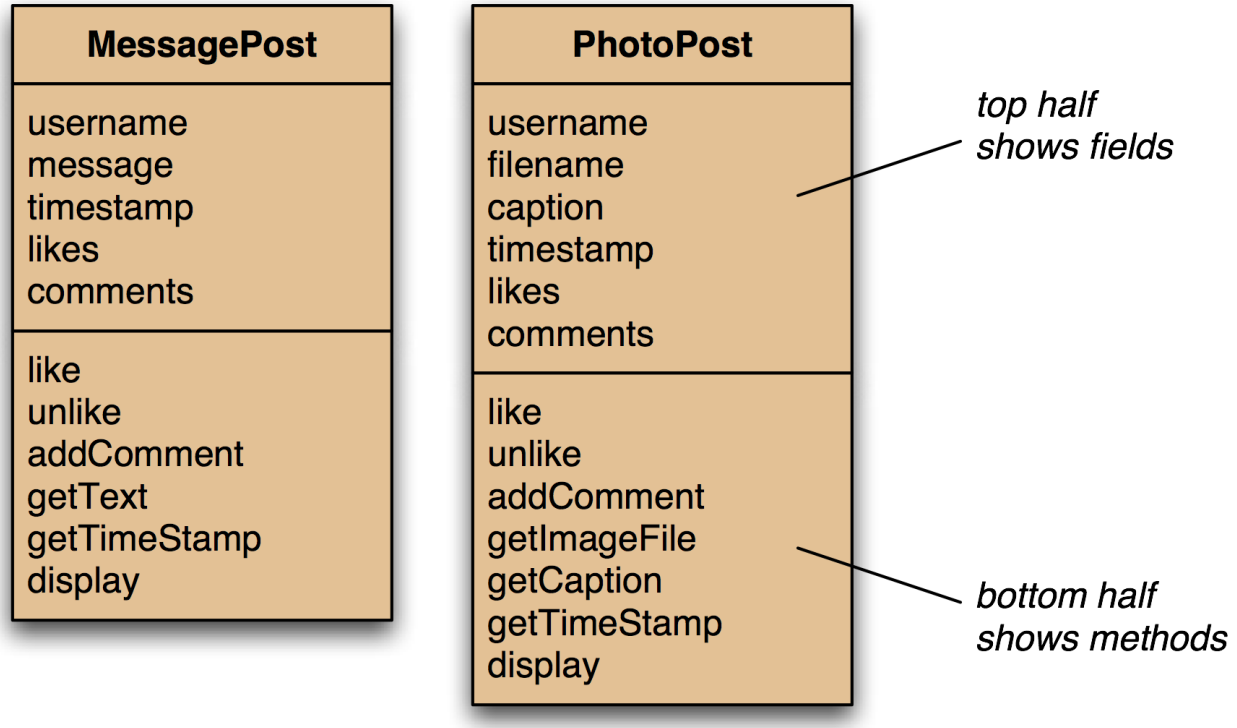
The Network example

- 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.

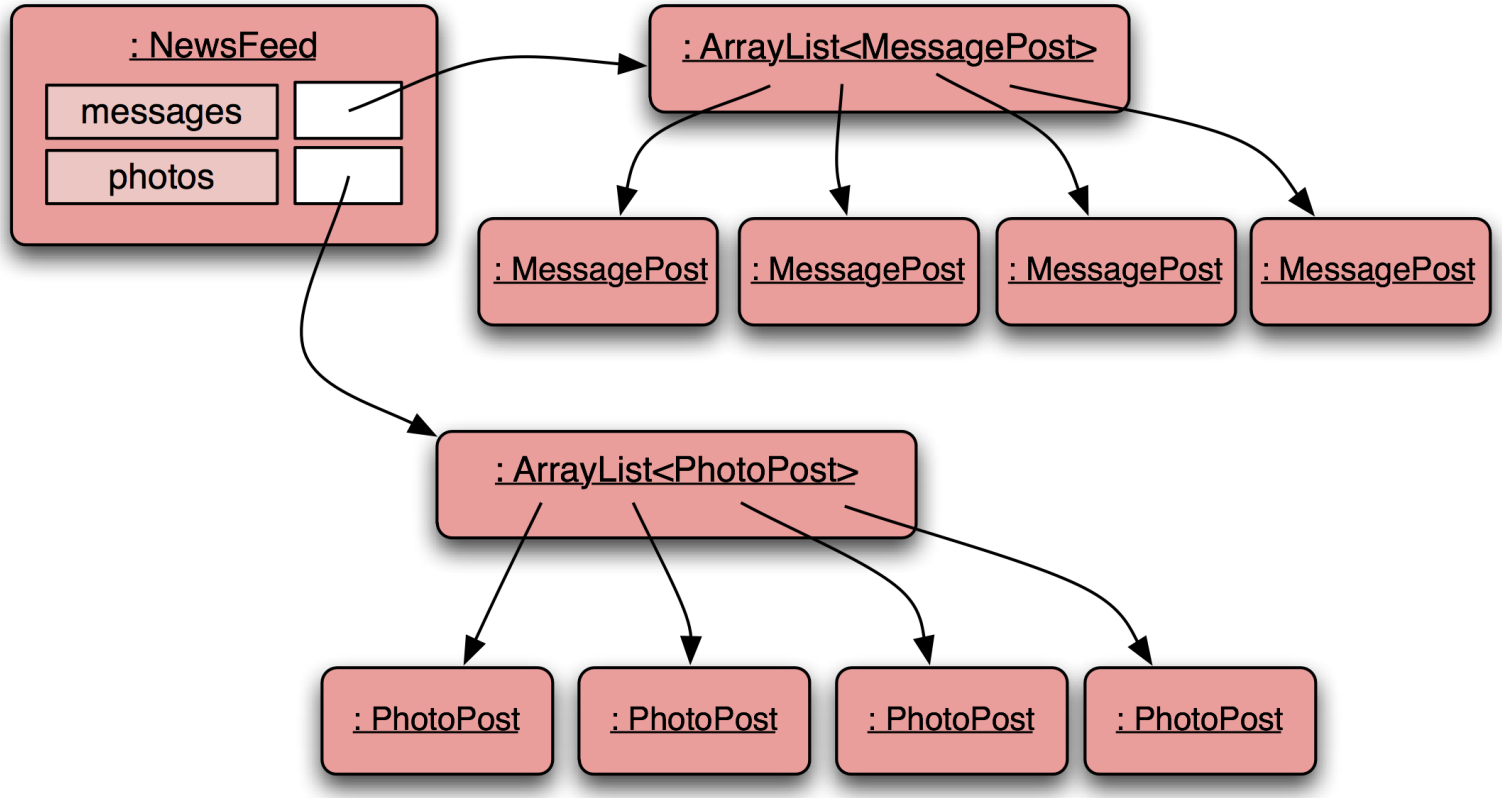
Network objects



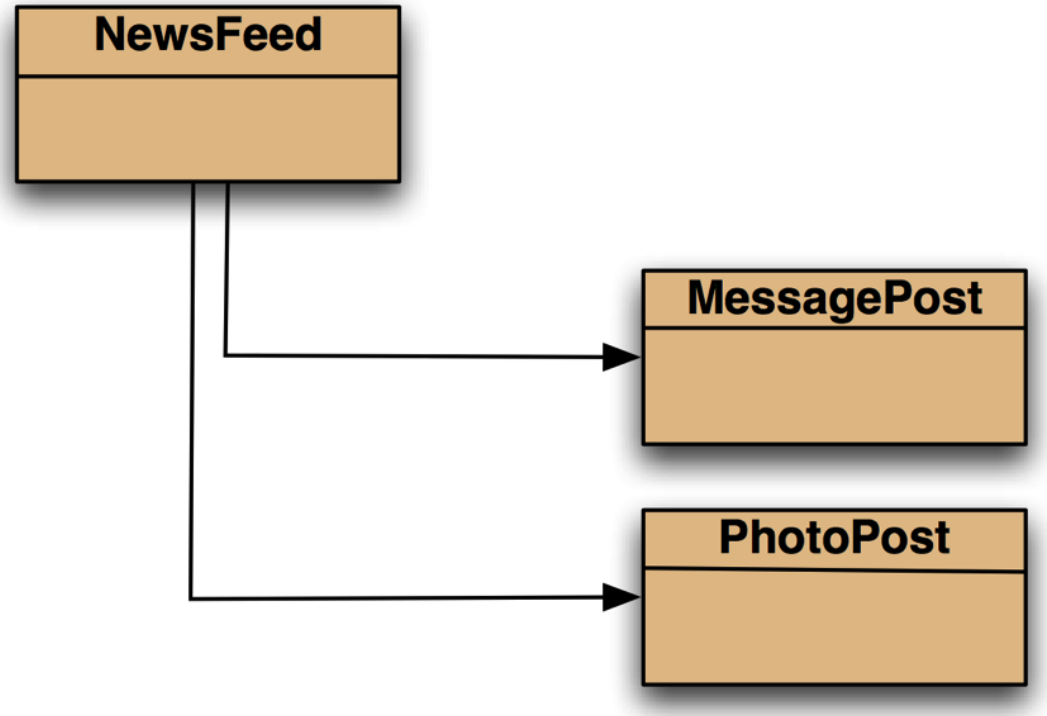
Network classes

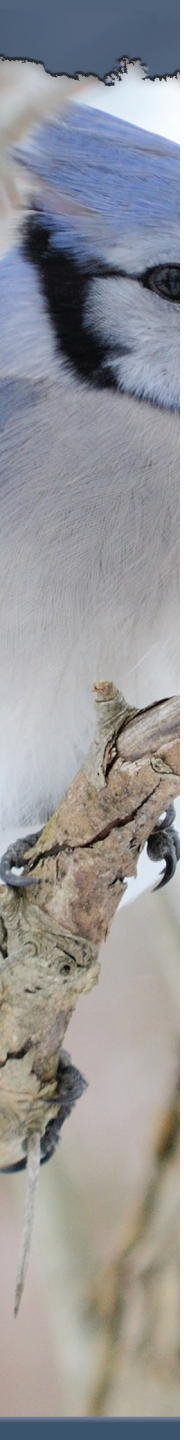


Network object model



Class diagram





Message-Post source code

Just an outline

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

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

   void addComment(String text) ...

   void like() ...

   void display() ...

   ...
}
```

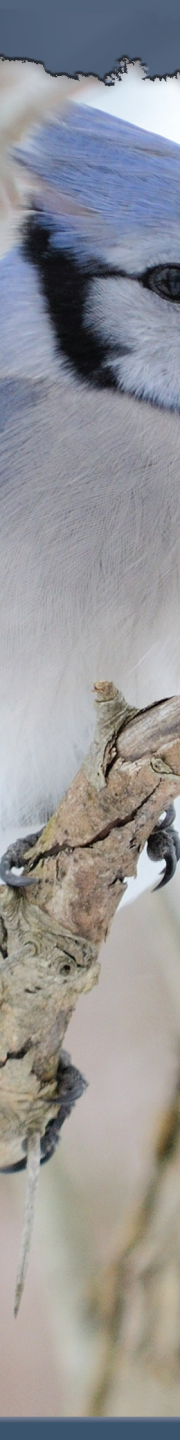
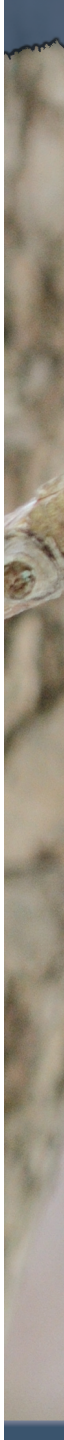


Photo-Post source code

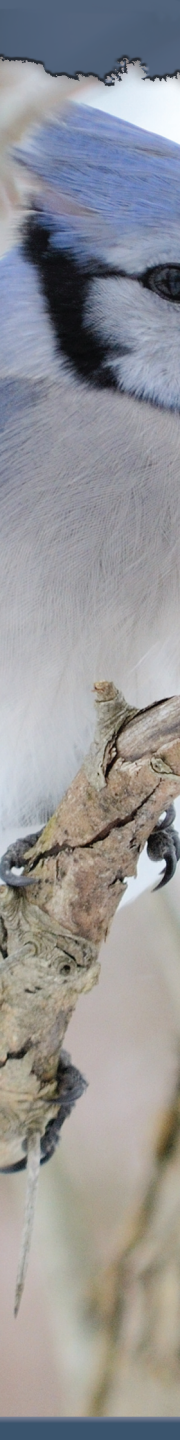
Just an outline

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

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

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



NewsFeed

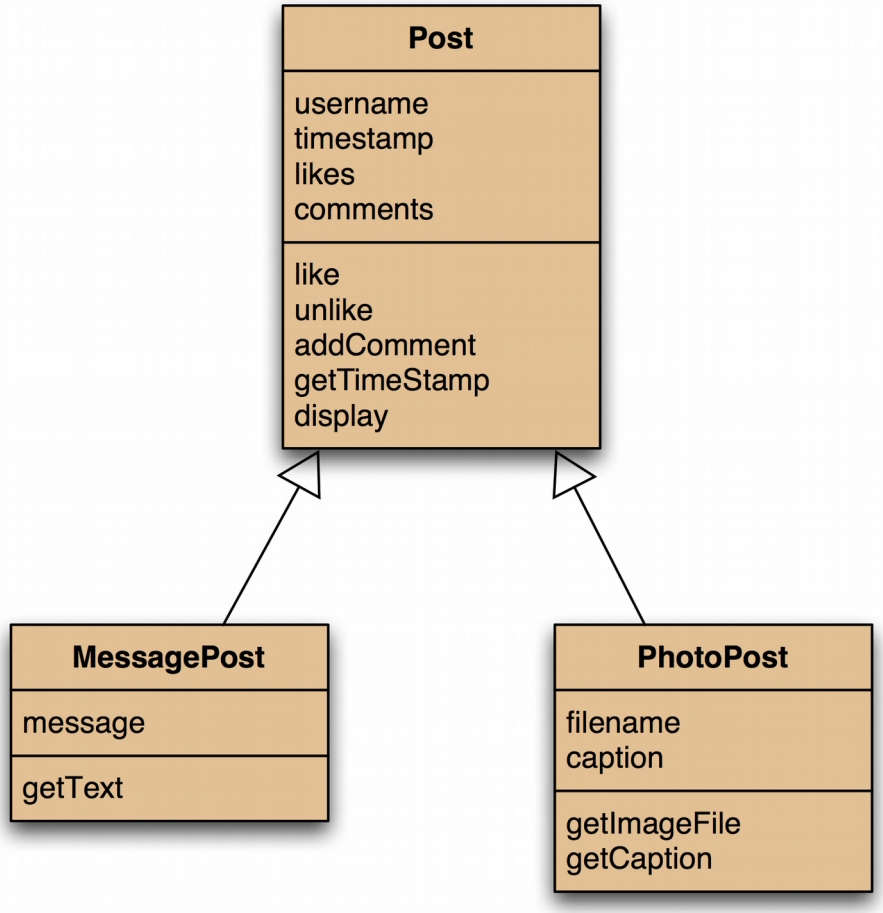
```
class NewsFeed {
   private final ArrayList<MessagePost> messages;
   private final ArrayList<PhotoPost> photos;
   ...
   void show() {
       for(MessagePost message : messages) {
          message.display();
          System.out.println();  // empty line between posts
       }

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

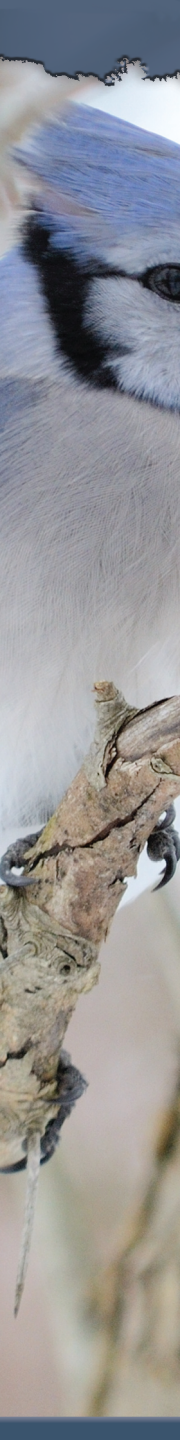


Critique of Network

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

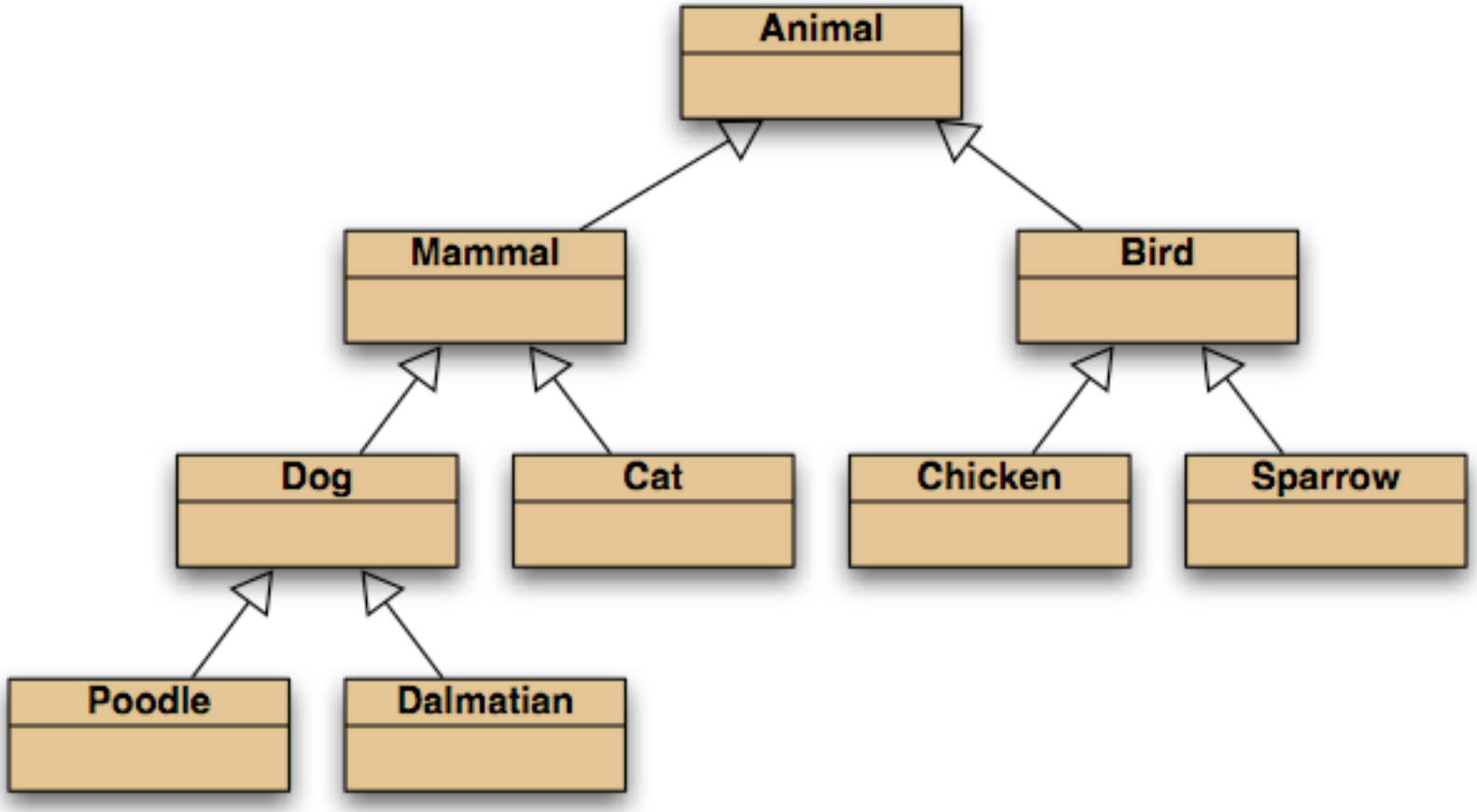




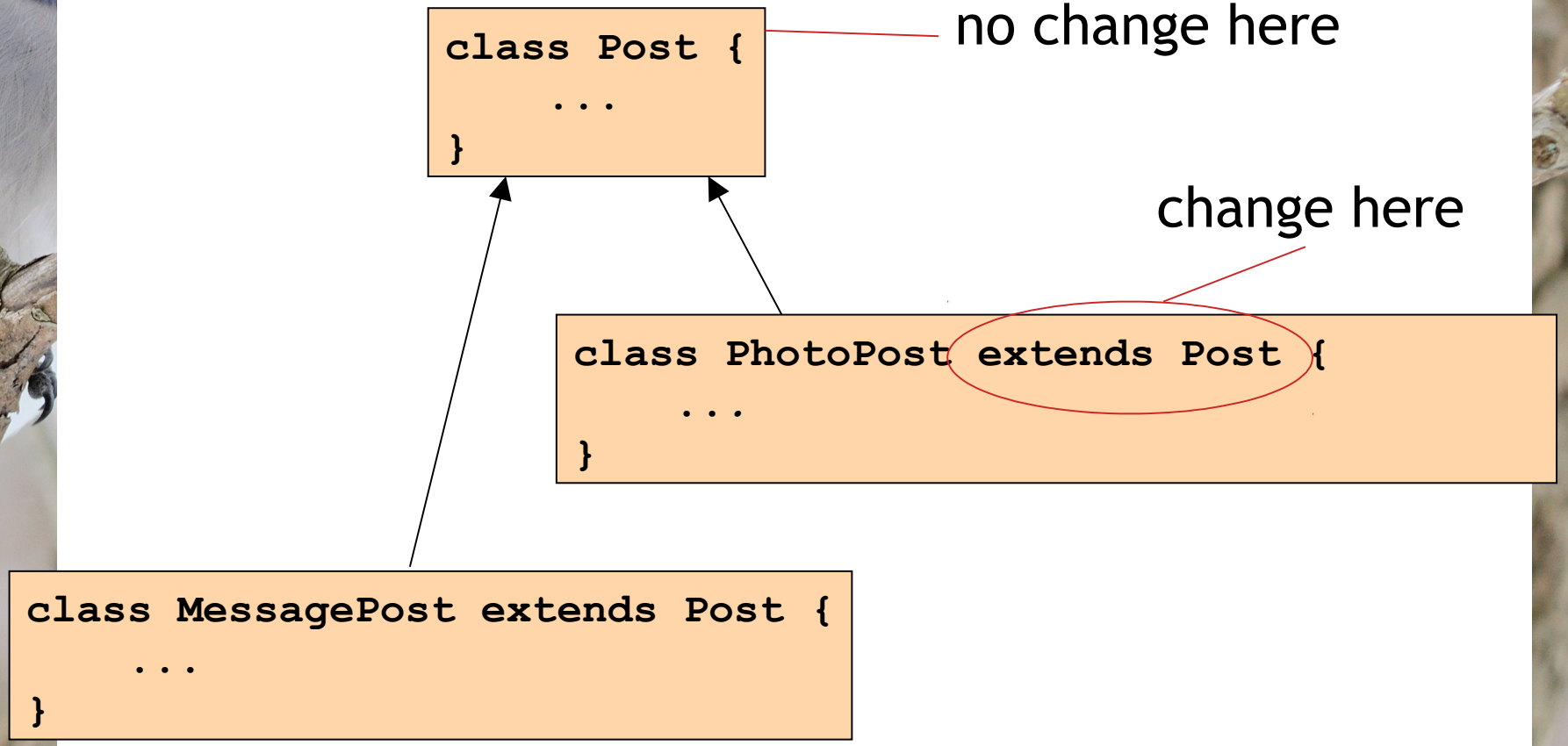
Using inheritance

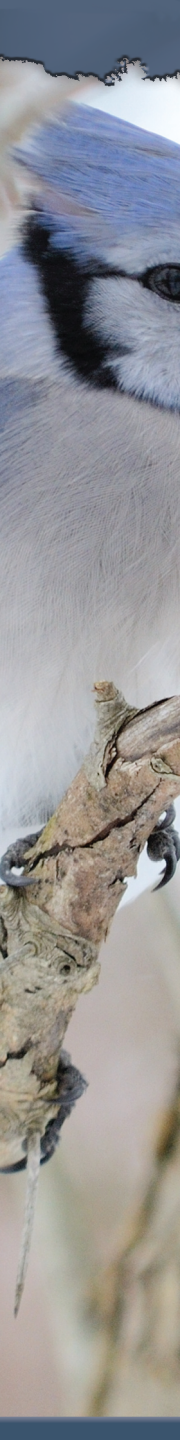
- define one **superclass** : `Post`
- define **subclasses** for `MessagePost` and `PhotoPost`
- the superclass defines common attributes (via fields)
- the subclasses **inherit** the superclass characteristics
- the subclasses add other characteristics

Inheritance hierarchies



Inheritance in Java





Superclass

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

    // constructor and methods omitted.
}
```

Subclasses

```
class MessagePost extends Post {
    private final String message;

    // constructor and methods omitted.
}
```

```
class PhotoPost extends Post {
    private final String filename;
    private final String caption;

    // constructor and methods omitted.
}
```

Inheritance and constructors

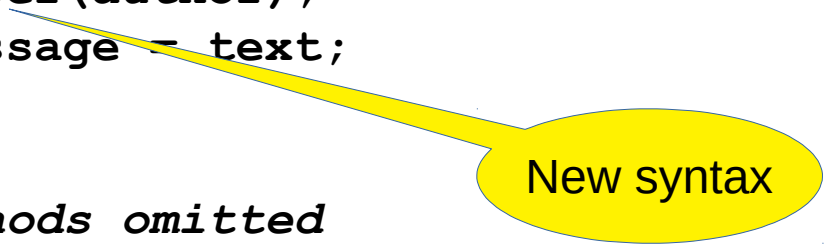
```
class Post {
    private final String username;
    private final long timestamp;
    private int likes;
    private final ArrayList<String> comments;

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

    // methods omitted
}
```

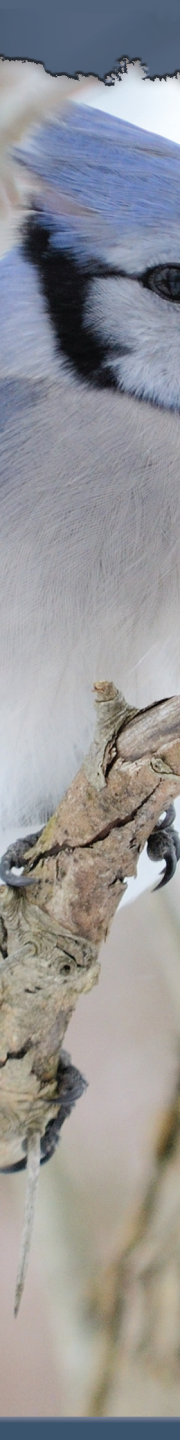
Inheritance and constructors

```
class MessagePost extends Post {
    private final String message;

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

    // methods omitted
}
```

Inheritance and constructors

```
class MessagePost extends Post {
    private final String message;

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

    // methods omitted
}
```

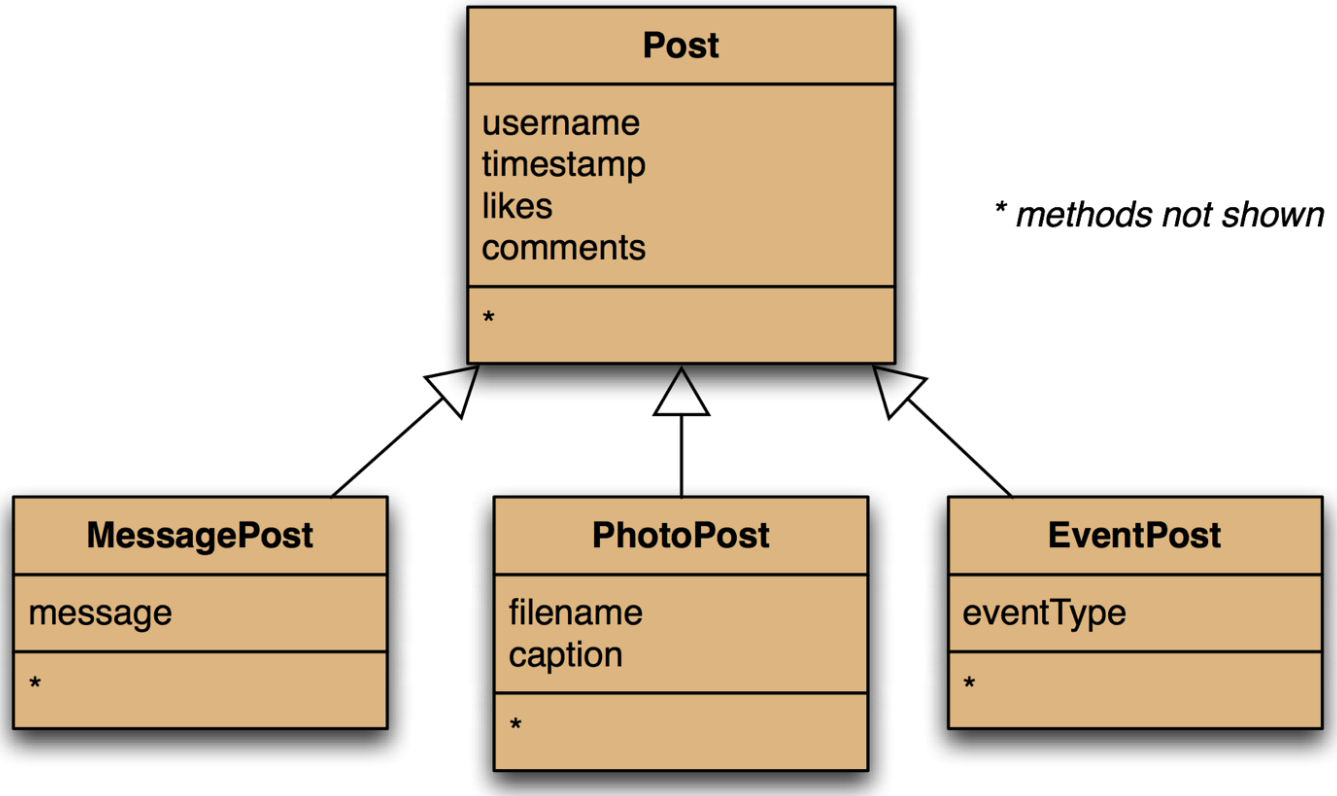




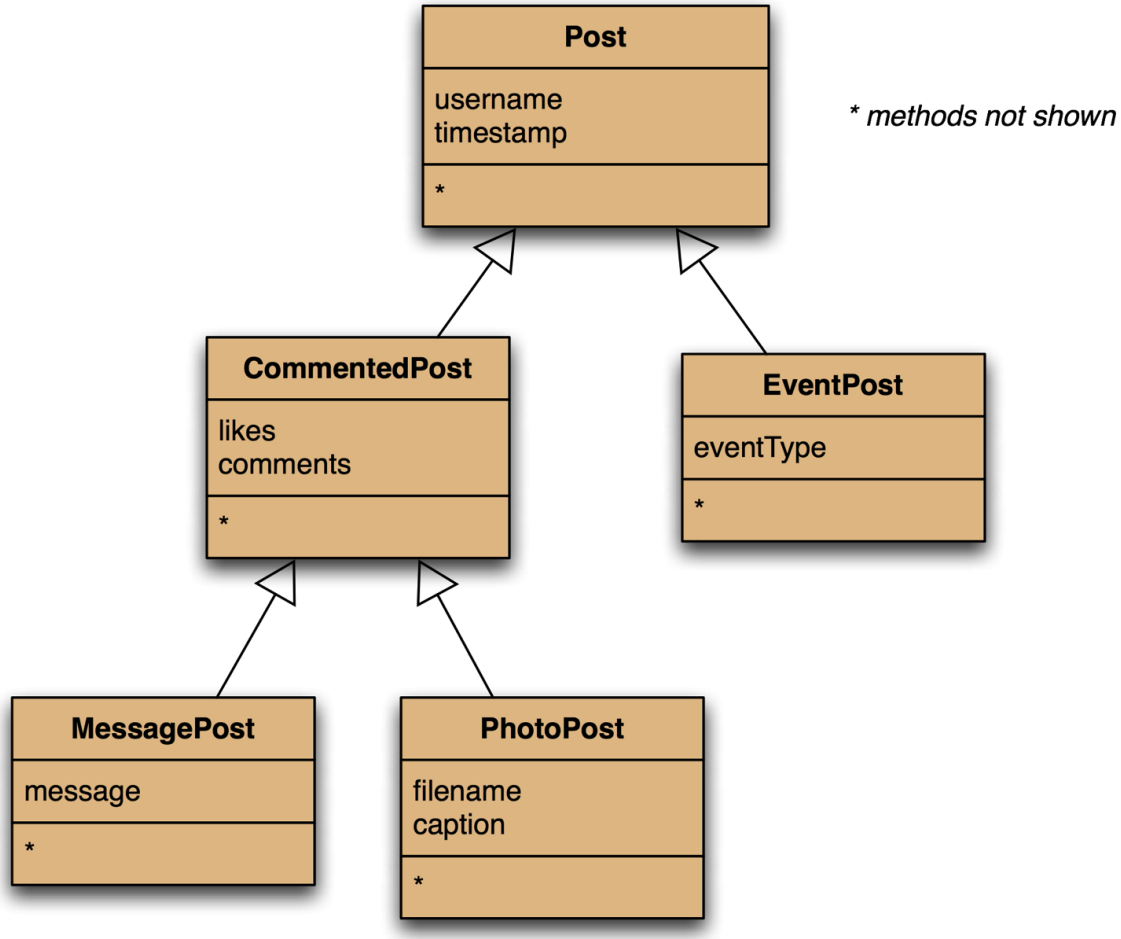
Superclass constructor call

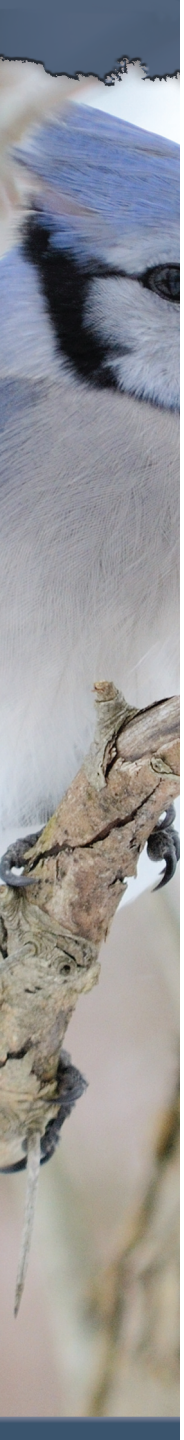
- Subclass constructors must always contain a 'super' call.
- If none is written, the compiler inserts one (without parameters)
  - only compiles if the superclass has a constructor without parameters
- Must be the first statement in the subclass constructor.

Adding more item types



Deeper hierarchies

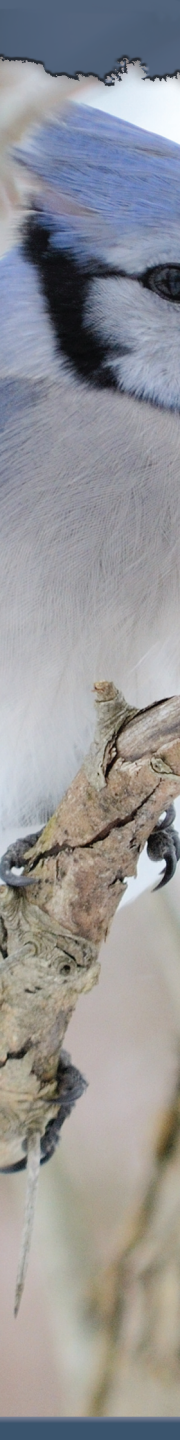




Review (so far)

Inheritance (so far) helps with:

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



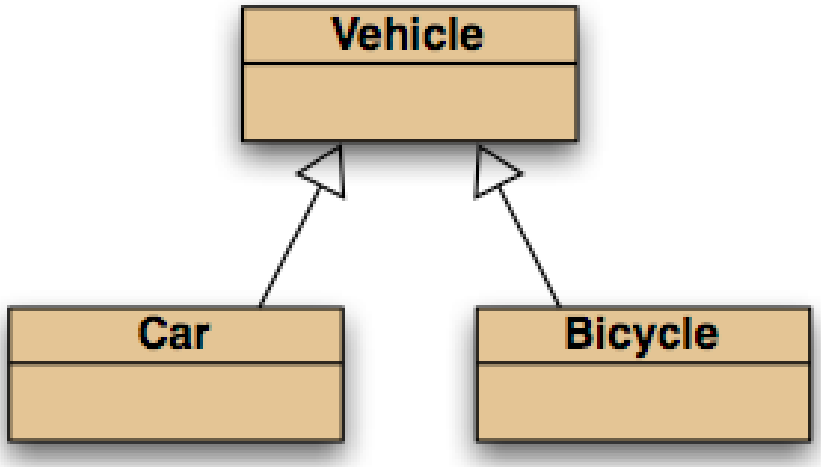
Subclasses and subtyping

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

Subclasses and subtyping

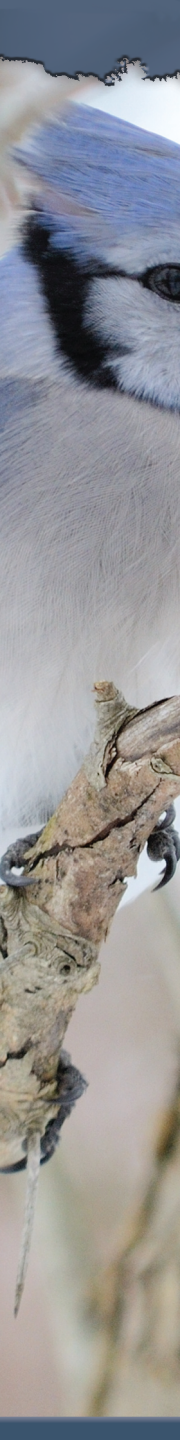
- Classes define types.
- Subclasses define *subtypes*.
- Objects of subclasses can be used where objects of supertypes are required.
  (This is called **substitution** .)

Subtyping and assignment



subclass objects may be assigned to superclass variables

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



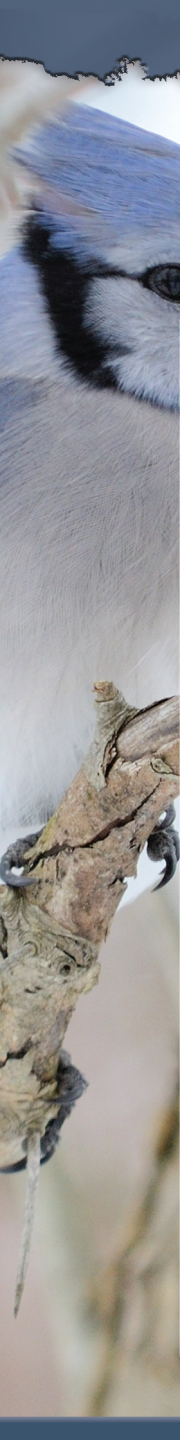
```
class NewsFeed {
    private final ArrayList<Post> posts;

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

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

Revised **`NewsFeed`** source code

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.)
 */
void show() {
    for (Post post : posts) {
        post.display();
        System.out.println(); // Empty line ...
    }
}
```



Subtyping

First, we had:

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

Now, we have:

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

We call this method with:

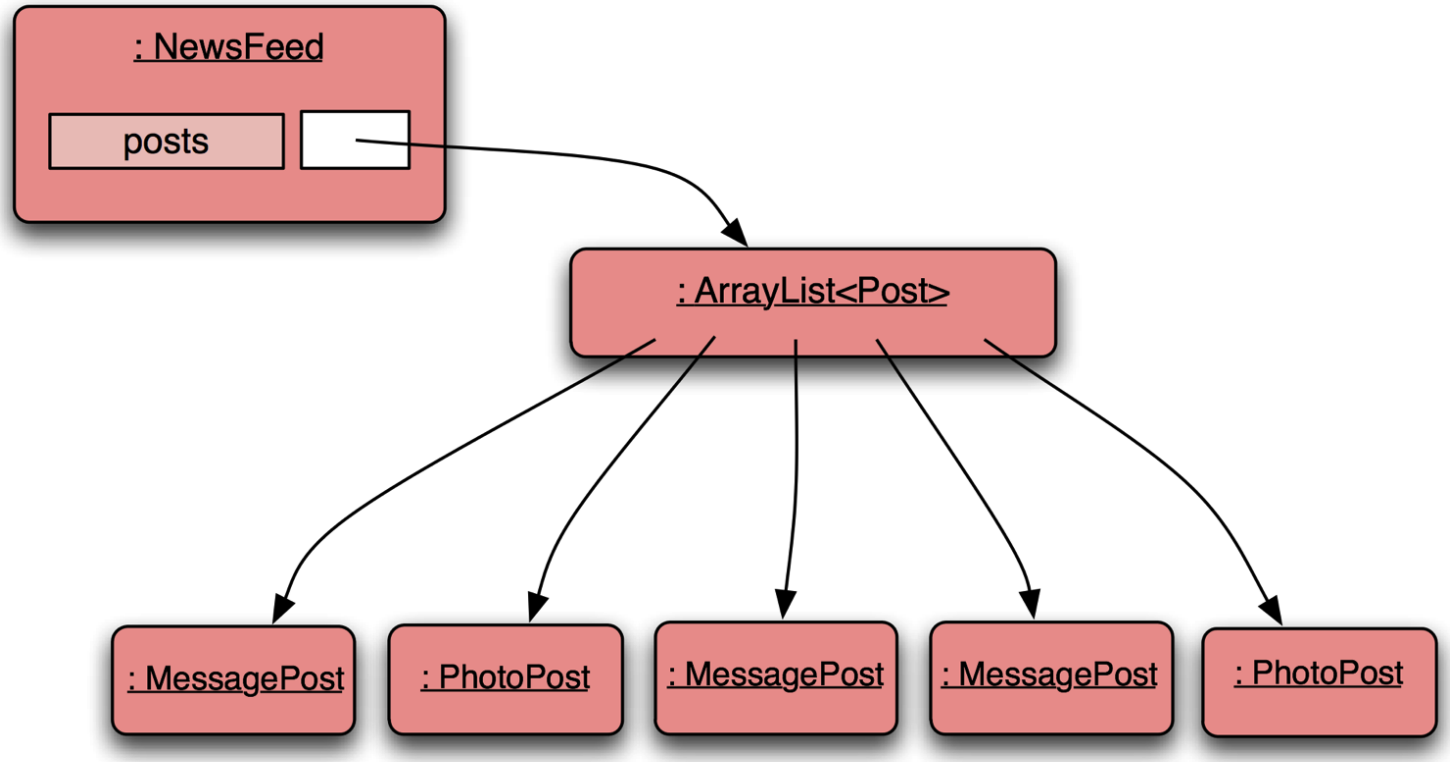
```
PhotoPost myPhoto = new PhotoPost(...);
feed.addPost(myPhoto);
```

Subtyping and parameter passing

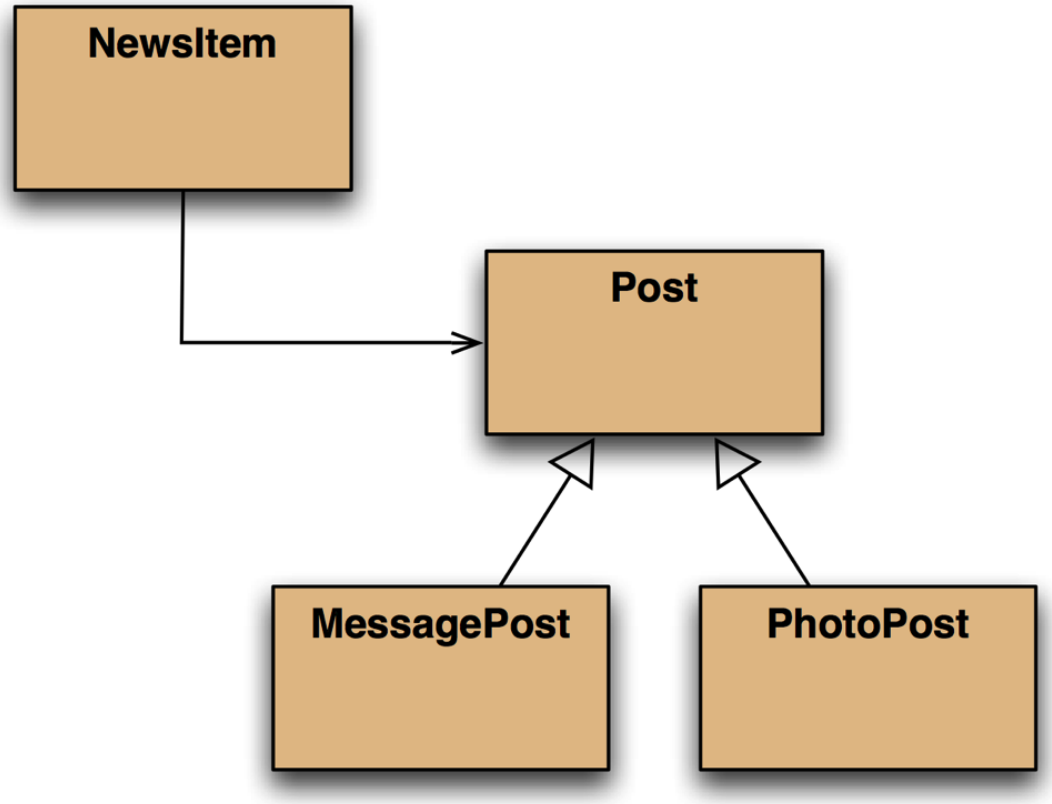
```
class NewsFeed {

    void addPost(Post post) {
        ...
    }
}

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

feed.addPost(photo);
feed.addPost(message);
```

subclass objects may be used as actual parameters for the superclass

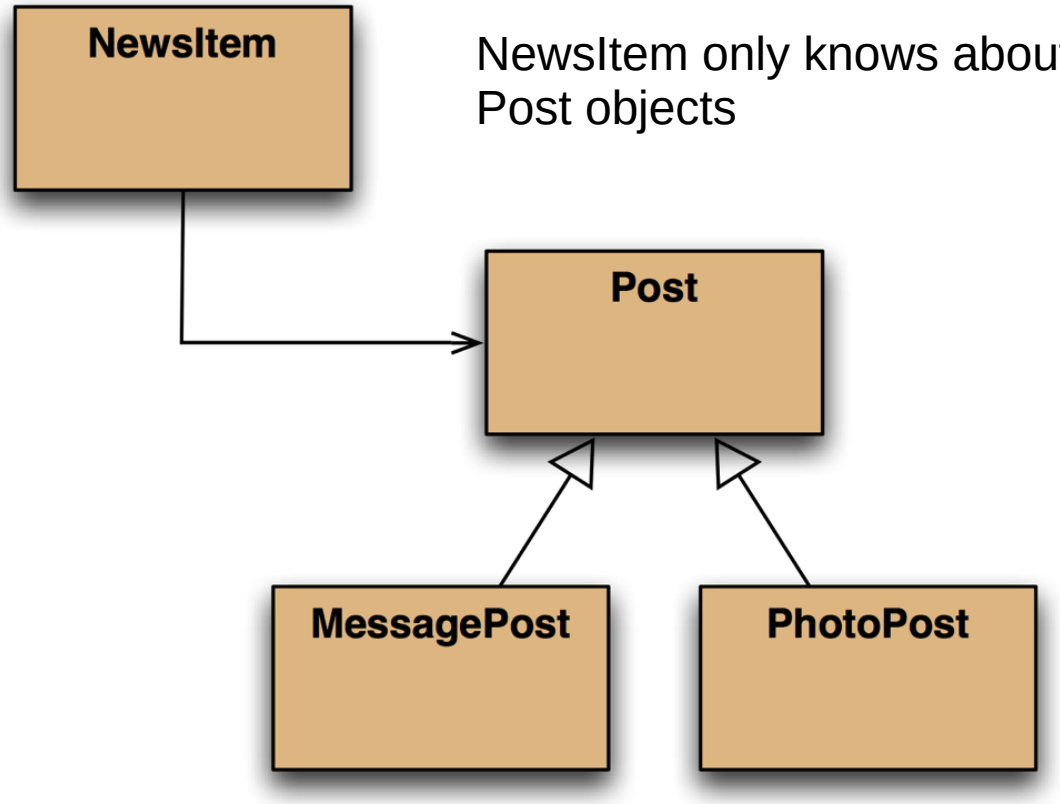
Object diagram



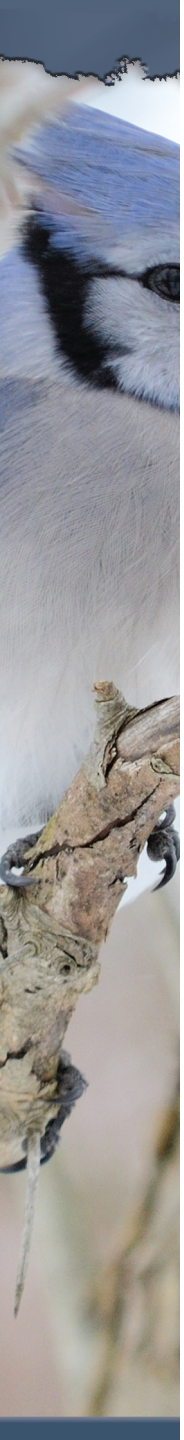
Class diagram



Class diagram



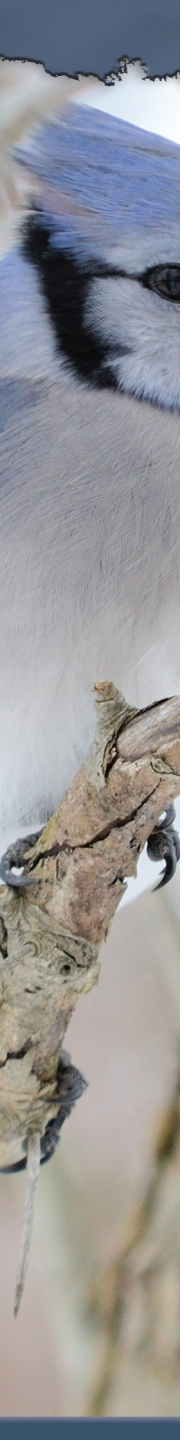
NewsItem only knows about Post objects



Polymorphic variables

- Object variables in Java are **polymorphic**.

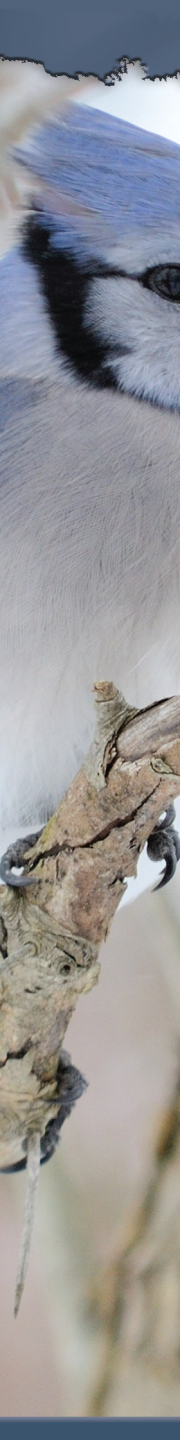
  (They can hold objects of more than one type.)
- They can hold objects of the declared type, or of subtypes of the declared type.



Casting

- We can assign 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

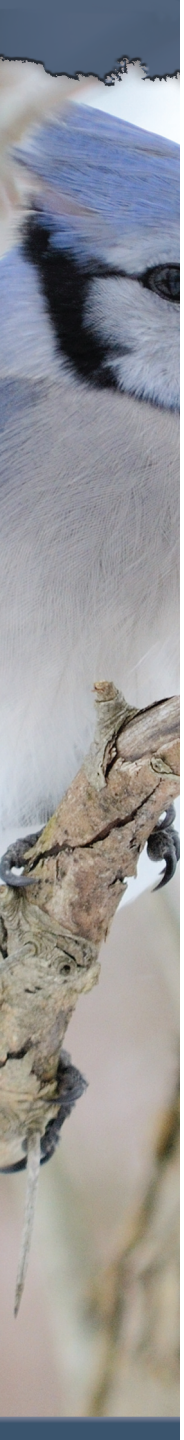
- We can assign subtype to supertype …
- … but we cannot assign supertype to subtype!

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

- Casting fixes this:

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

(but only ok if the vehicle really is a **`Car`**!)

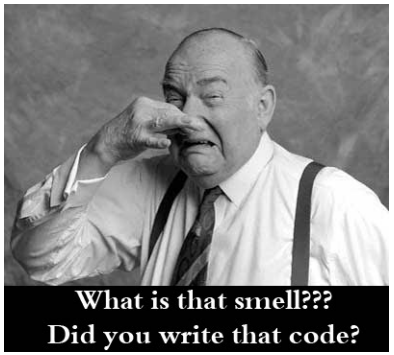


Casting

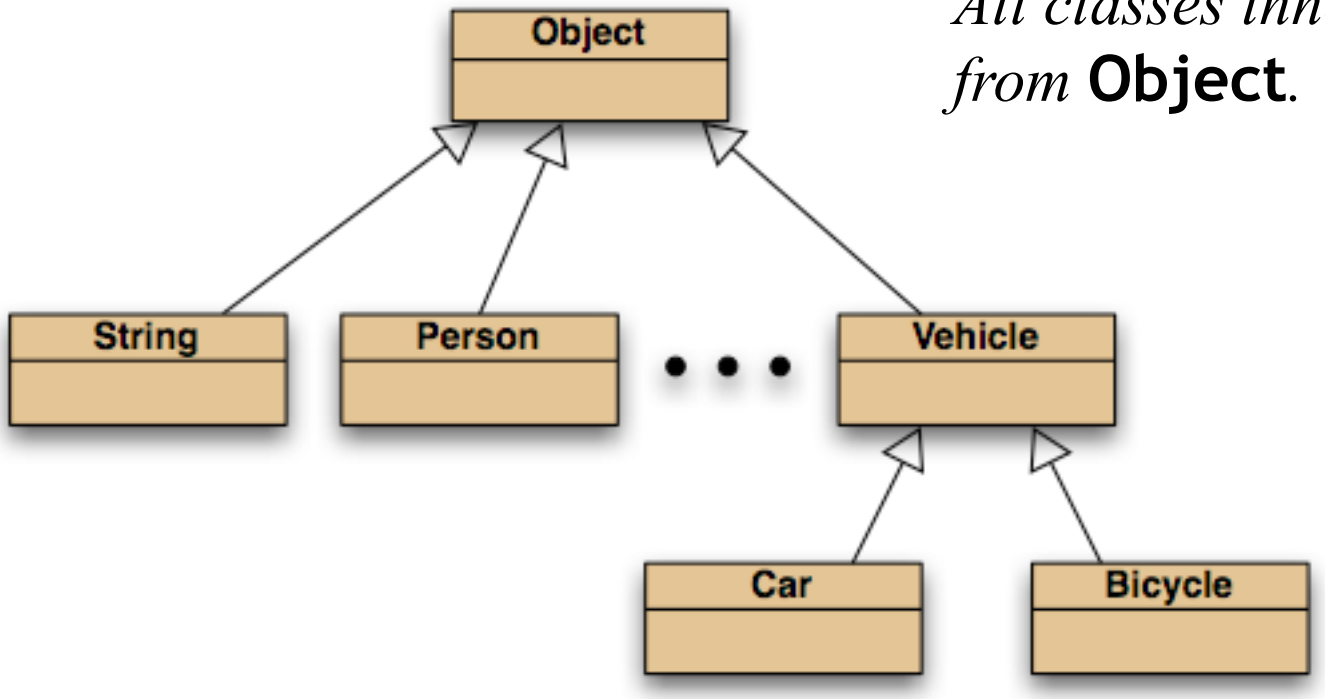
- An object type in parentheses.
- Used to overcome 'type loss'.
- The object is not changed in any way.
- A runtime check is made to ensure the object really is of that type:
  - `ClassCastException` if it isn't!
- Use it sparingly.
- Often a sign of code smell.

Casting

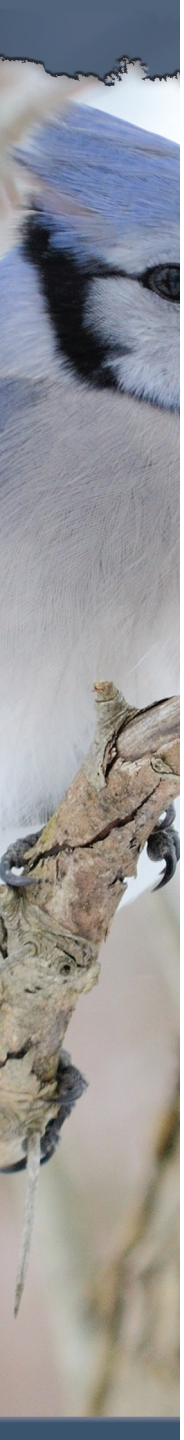
- An object type in parentheses.
- Used to overcome 'type loss'.
- The object is not changed in any way.
- A runtime check is made to ensure the object really is of that type:
  - `ClassCastException` if it isn't!
- Use it sparingly.
- Often a sign of code smell.



The Object class



All classes inherit from **Object**.

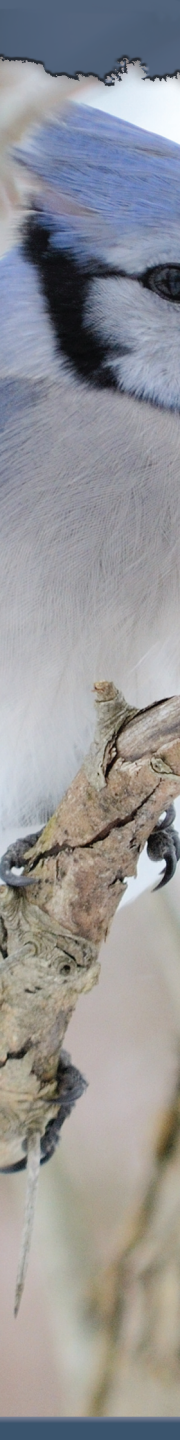


Polymorphic collections

- All collections are polymorphic.
- The elements could simply be of type **`Object`**.

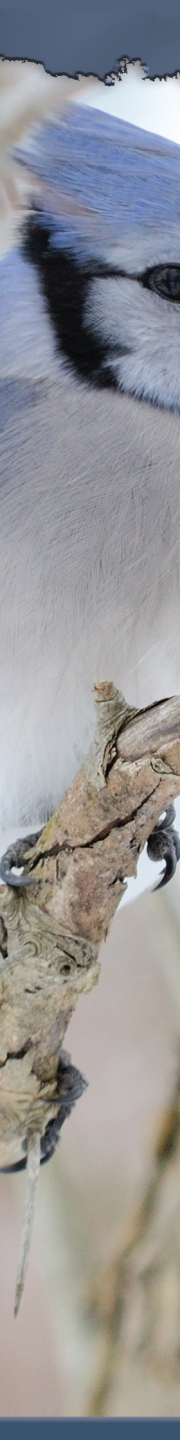
```
public void add(Object element)

public Object get(int index)
```

- Usually avoided by using a type parameter with the collection.



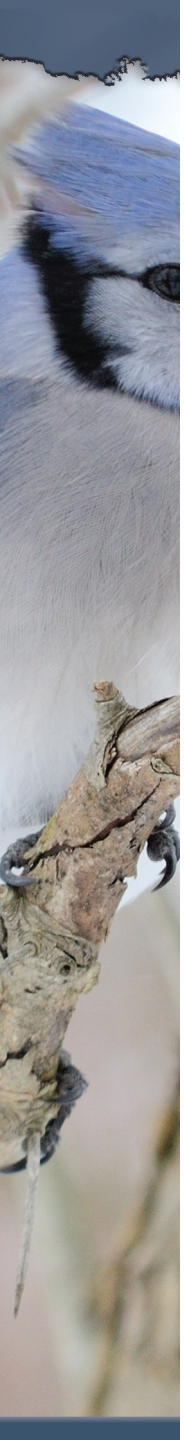
Polymorphic collections

- 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.



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).

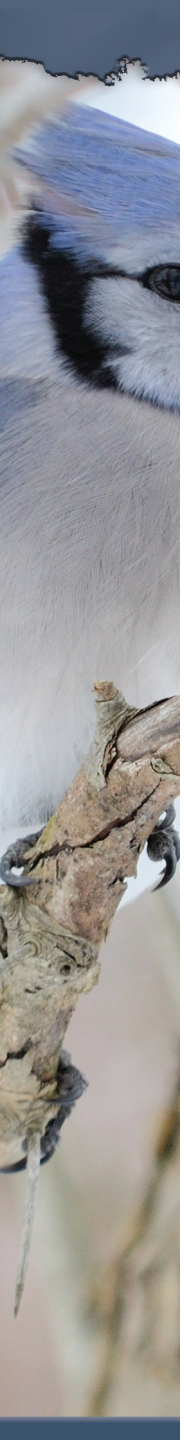


Caveats – isa

- An object of a subclass **isa**n object of its superclass
- ⇒ every method of the superclass must apply to the subclass.
- Eg, all methods of `Object` inherited by all classes.

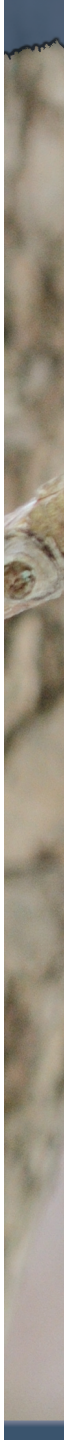
Caveats – isa counter-example

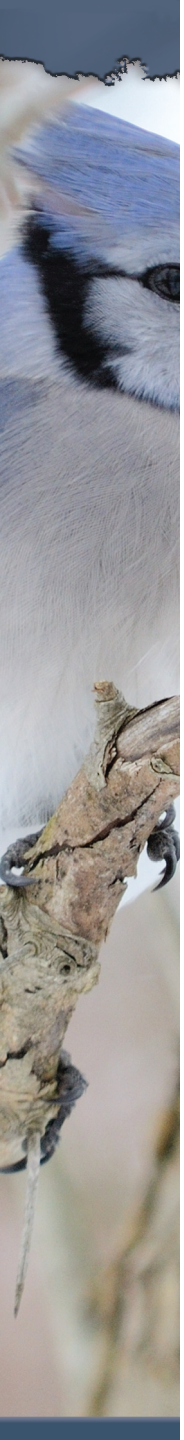
- Have **`Rectangle`**, want **`Square`**.
- But...**`Rectangle`** with **`setWidth`**, **`setHeight`** should not be a superclass of **`Square`**.
- **`Rectangle`** has one method too many.
- Would need to prevent setters acting independently – can get ugly.



Caveats – isa example

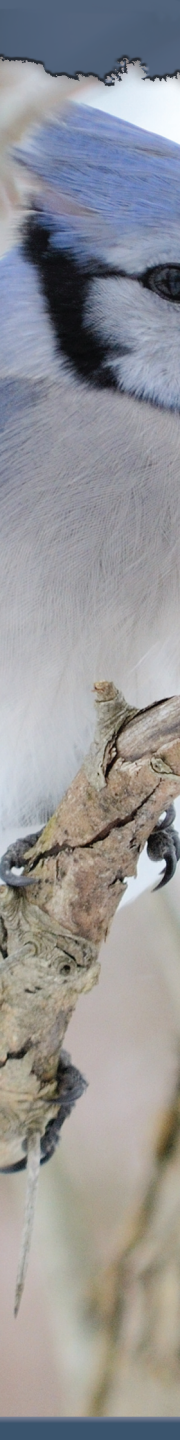
- However, **`Rectangle`** can be subclass of **`Square`**.
- Just needs to add code to manage second dimension independently.





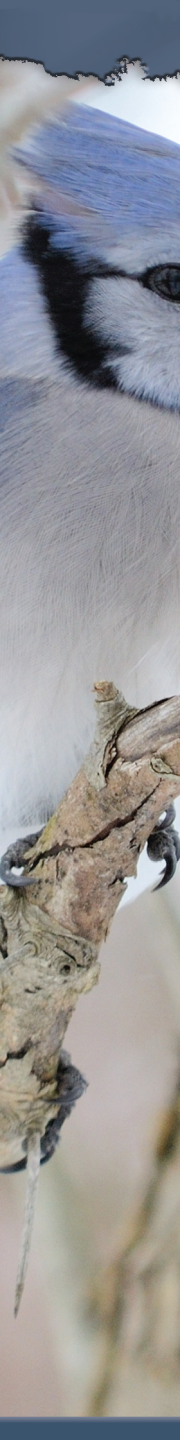
Caveats – inheritance issues

- Subclass can only inherit from one superclass.
  - `extends` _____ only has one available slot.
- Inheritance breaks encapsulation – subclass can depend on superclass implementation.
  - *Fragile superclass* problem.
- And more...



Ways to recycle code

- Inheritance – recycle superclass code.
- But also without inheritance:
  - composition,
  - aggregation,
  - association,
  - code forwarding.



Composition

- An object is made up of other objects, eg,

```
class Car {
    private final Engine e = new Engine();
    void accelerate() {e.revFaster();}
}
```

- Car "owns" **`Engine e`**.
- Lifetime of **`Engine e`** is same as **`Car`**.

Aggregation

- An object is made up of other objects, eg,

```
class Car {
    private final Engine e;
    Car(Engine e) {this.e = e;}
}
```

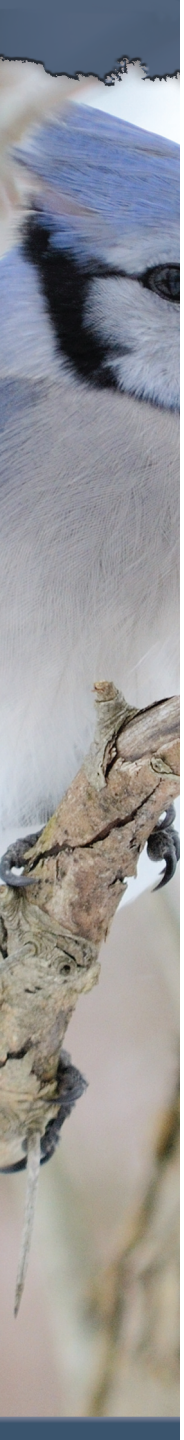
- Car owns **`Engine e`**.
- Lifetime of **`Engine e`** is independent of **`Car`** – may continue to exist after **`Car`** is finished.

Association

- An object uses other objects, eg,

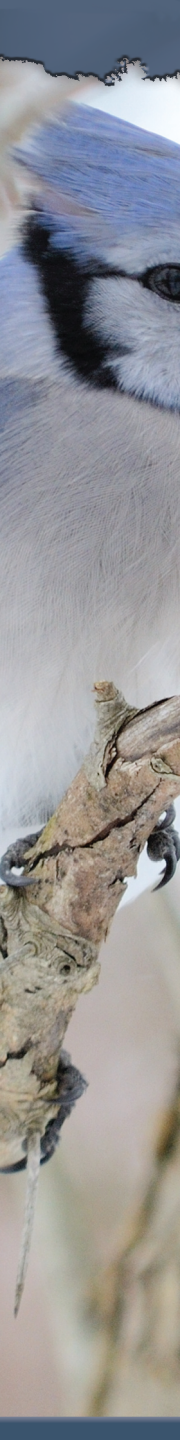
```
class Car {
    void goForDrive(Driver d) {
        d.drive(this);
    }
}
```

- Car doesn't own **`Driver d`**.
- Lifetime of **`Driver d`** is independent of **`Car`** – may continue to exist after **`Car`** is finished.



Pro tip - prefer composition to inheritance

- Subclass can only inherit from one superclass.
  - `extends` ____ only has one available slot.
- Inheritance breaks encapsulation – subclass can depend on superclass implementation.
  - *Fragile superclass* problem.
- And more...



Method forwarding

- Recycle **`Rectangle`** to create a **`Square`**.

```
class Square {
    private final Rectangle;

    Square(int side) {
        rectangle = new Rectangle(side, side);
    }

    int getSide() {return rectangle.getWidth();}

    void setSide(int side) {
        rectangle.setWidth(side);
        rectangle.setHeight(side);
    }

    int area() {return rectangle.area();}
}
```