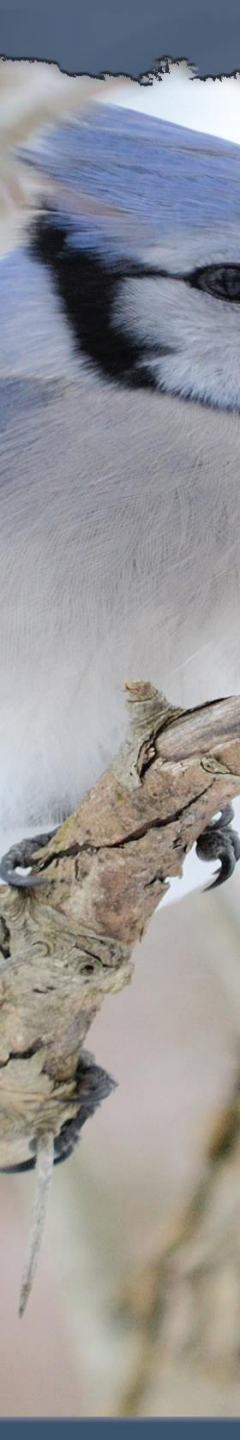
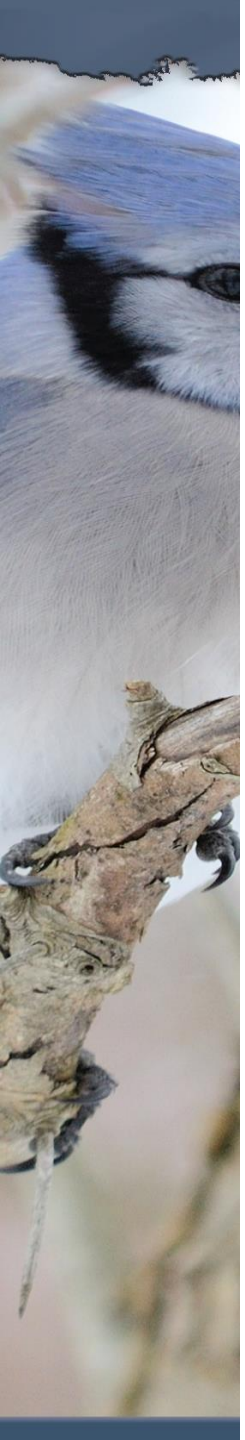


PGR101 – Objektorientert programmering 2

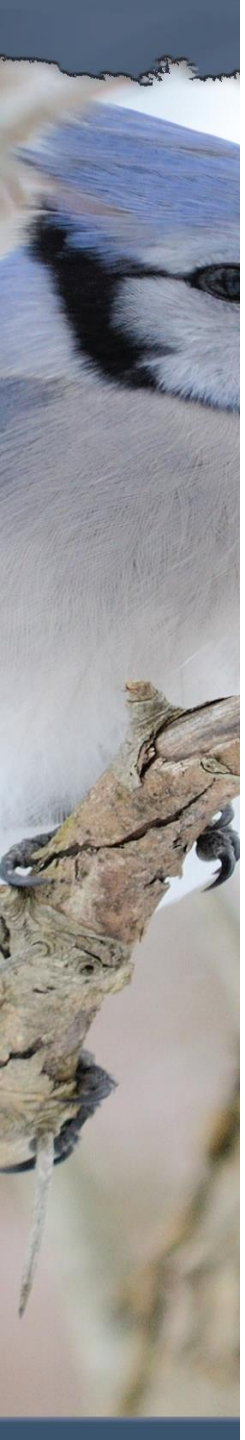
Repetisjonsforelesning



Arbeidskrav 1

a)

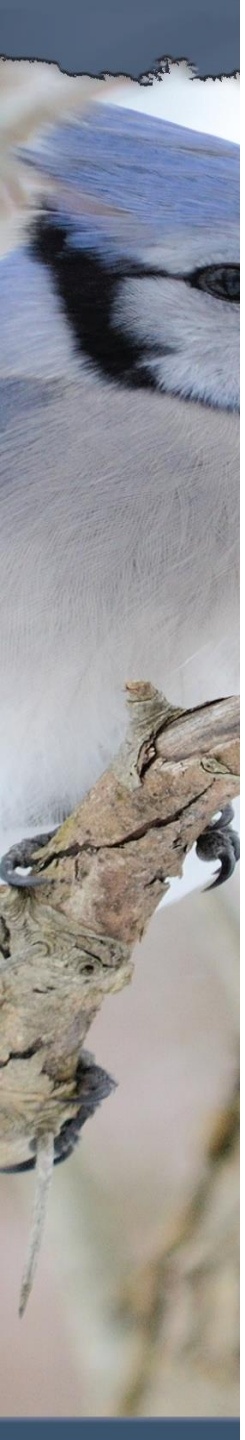
- Hvordan skal klassene `Meter`, `Weight`, `Clock` og `Thermometer` se ut?
- Hvorfor er det hensiktsmessig å arve fra `Meter`?
- Hva betyr det at “Klassene har standard parametrisk og ikke-parametrisk konstruktør i tillegg til standard tilgangsmetoder, `toString`-metode og passende `equals`-metode.”



Arbeidskrav 1

b)

- Hvordan skal klassen `MeterArchive` se ut? Hvilke metoder inneholder den (ihvertfall)?
- Hvilke typer forholder den seg til?
- Er det bare jeg som stusser over de to endringsmetodene?

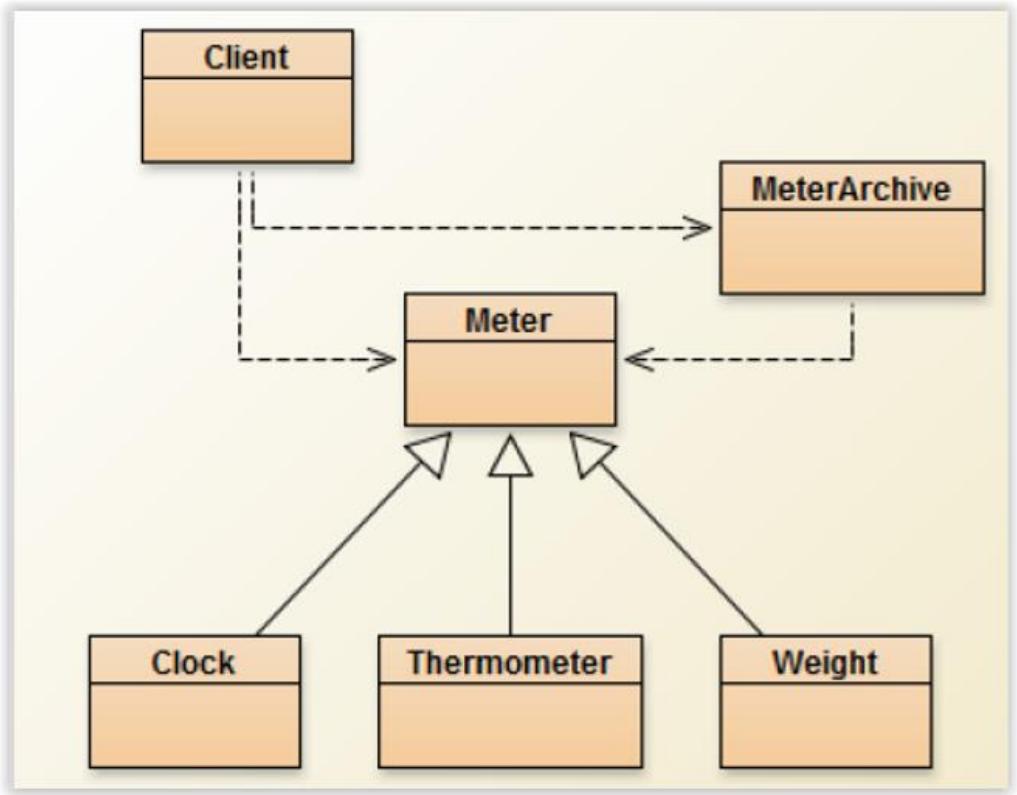


Arbeidskrav 1

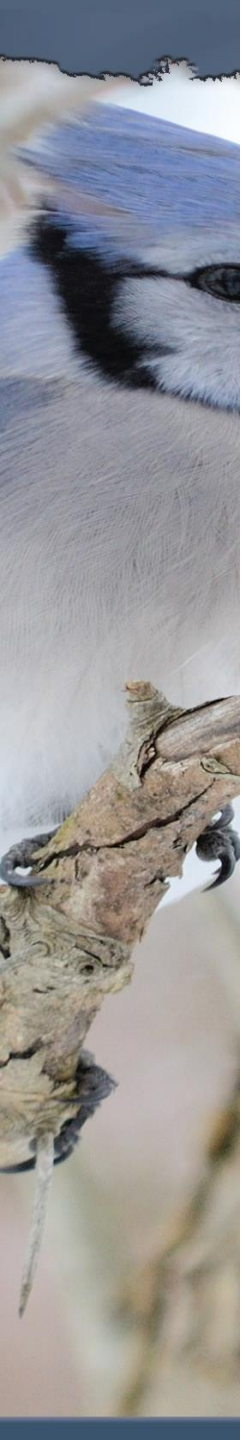
c)

- Hvordan skal klassen `Client` se ut? Hvilke metoder inneholder den (ihvertfall)?
- Hvordan kan vi registrere måleinstrumenter?
- Hvordan kan vi vise at all funksjonalitet i `MeterArchive` fungerer som tiltenkt?

Arbeidskrav 1



- Hvordan kan vi skrive ut innholdet i hele arkivet?

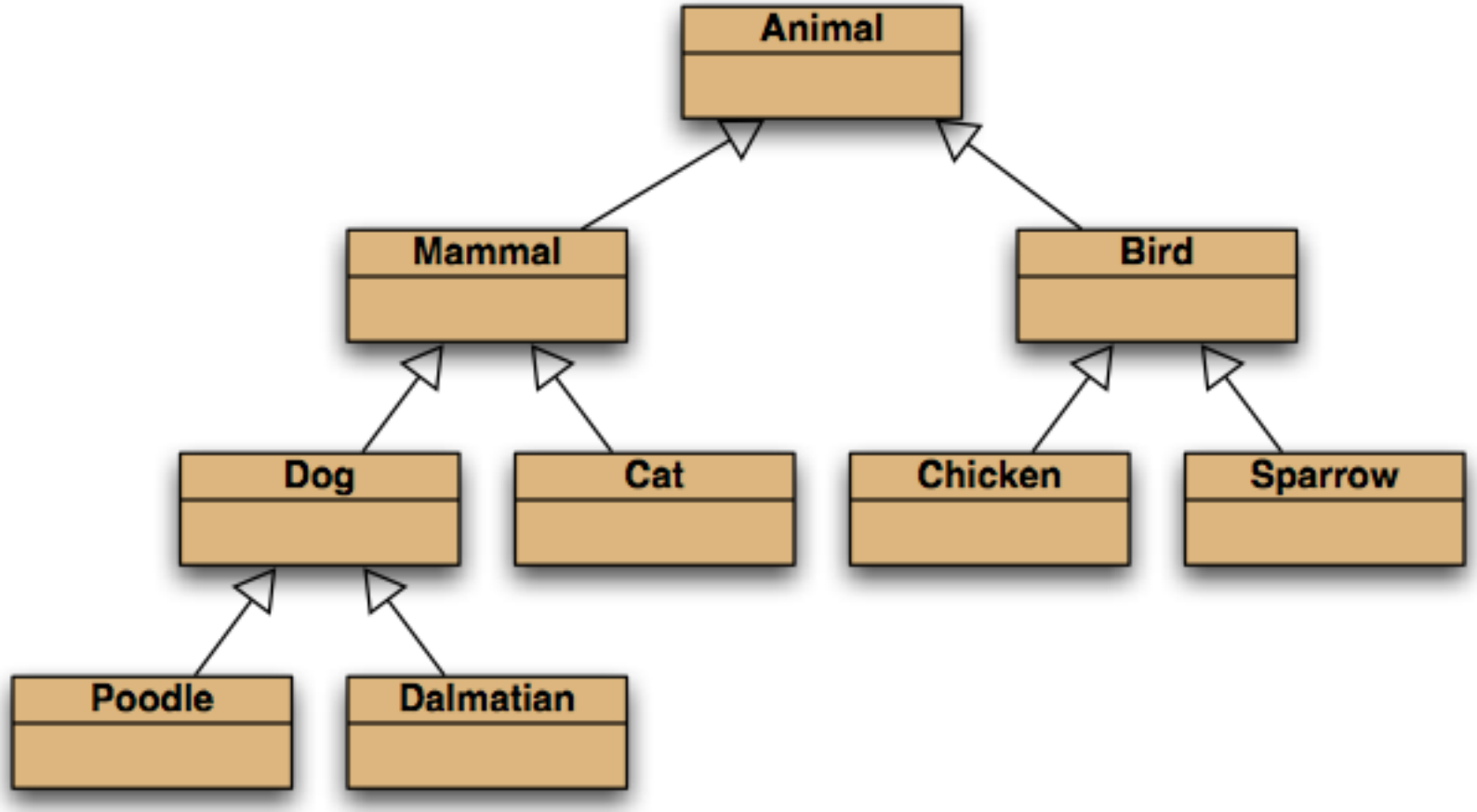


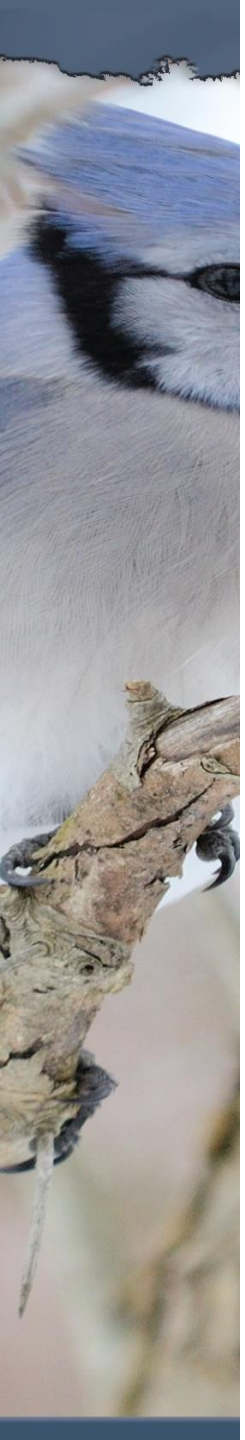
Arbeidskrav 1

Utvidelser:

- Lese fra fil.
- Automatiserte tester.

Inheritance hierarchies



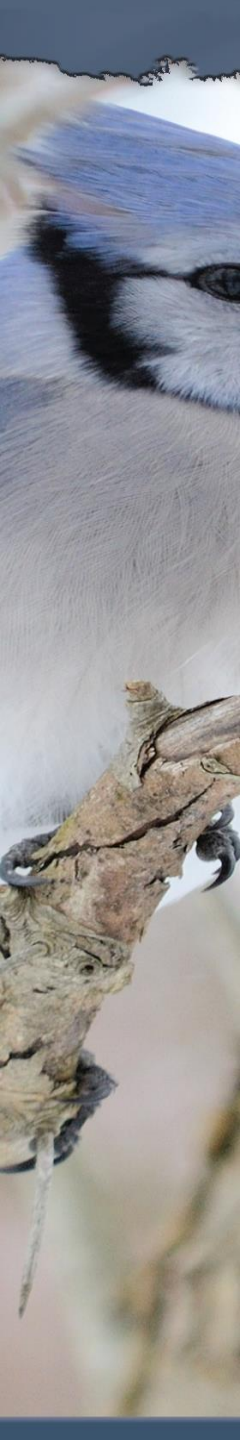


is-a? has-a? relationships

- Is-a: Arv. Subklassen **er** en spesialisering av superklassen. Eks: En hund ***er*** et pattedyr.
- Has-a: Komposisjon. En klasse kan ***ha*** tilgang til en annen klasse. En hund kan ***ha*** en eier.

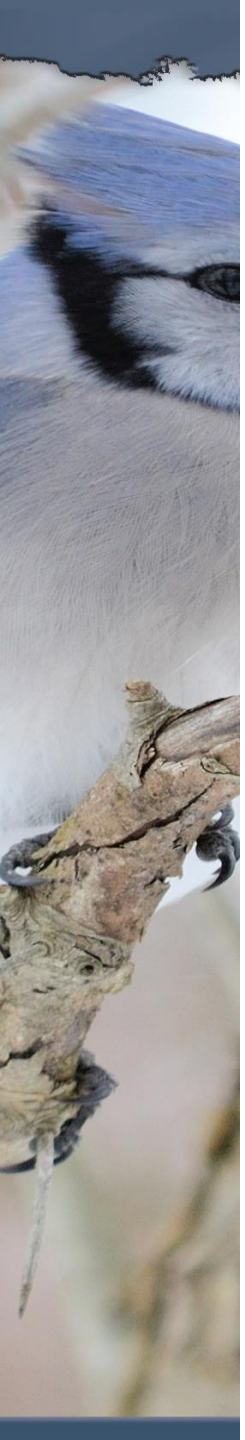
Inheritance and constructors

```
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<>();
    }

    // methods omitted
}
```



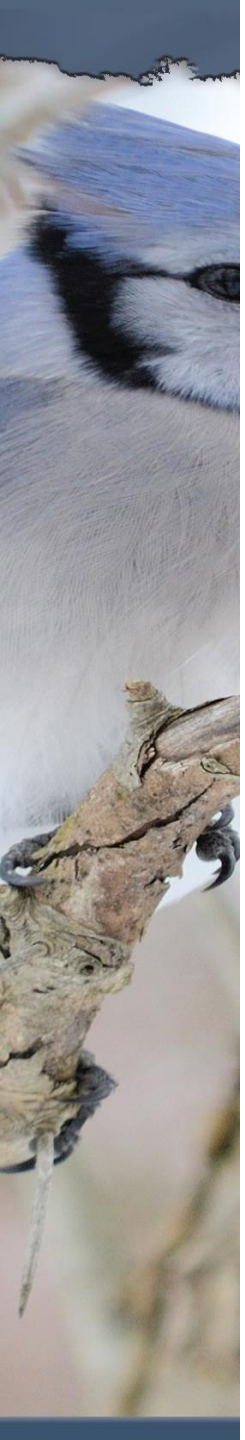
Inheritance and constructors

```
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
}
```



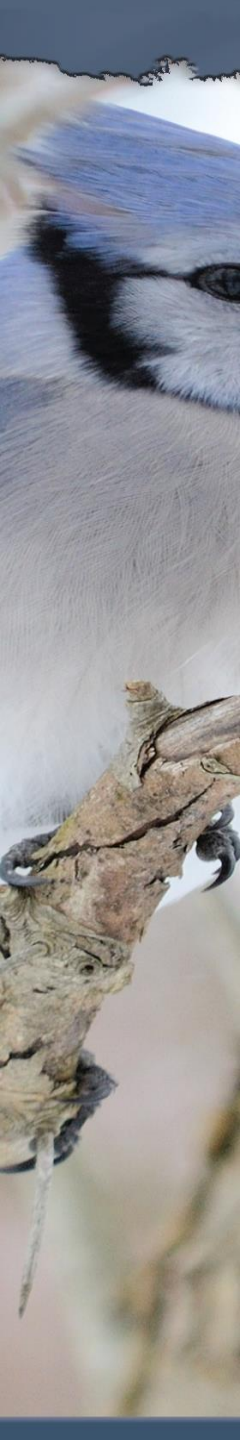
Superclass constructor call

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



Private access

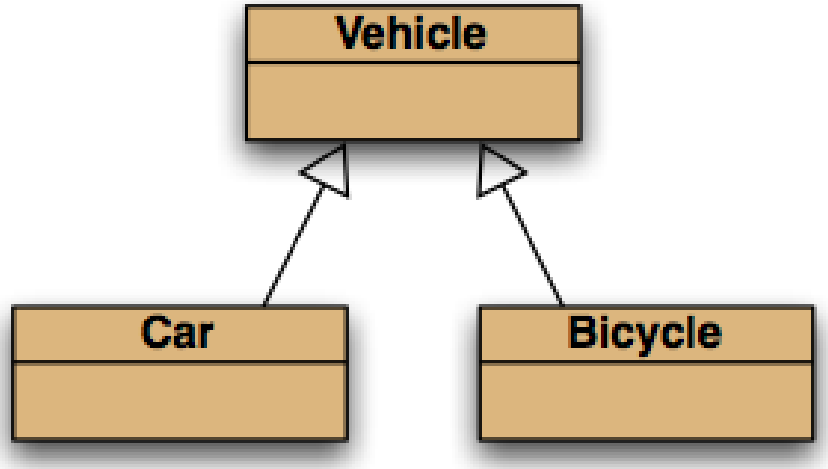
- Subclasses cannot access private fields in a superclass.
- Use getters in superclass if you want access…



Subclasses and subtyping

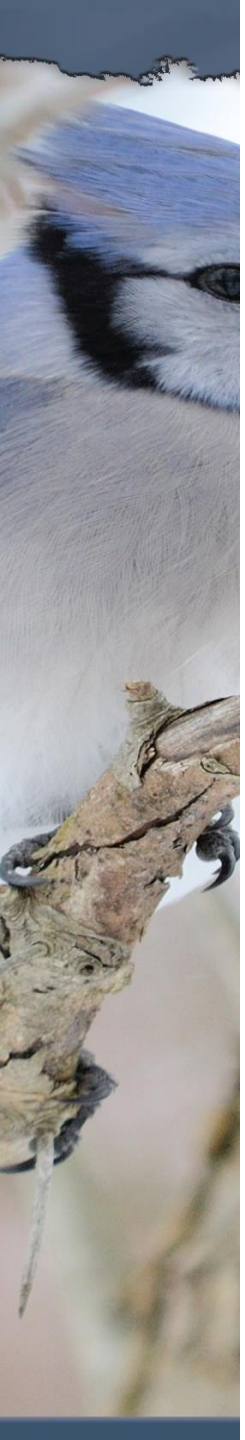
- Classes define types.
- Subclasses define *subtypes*.
- Objects of subclasses can be used where objects of supertypes are required.
  (This is called **substitution** .)

Subtyping and assignment



subclass objects may be assigned to superclass variables

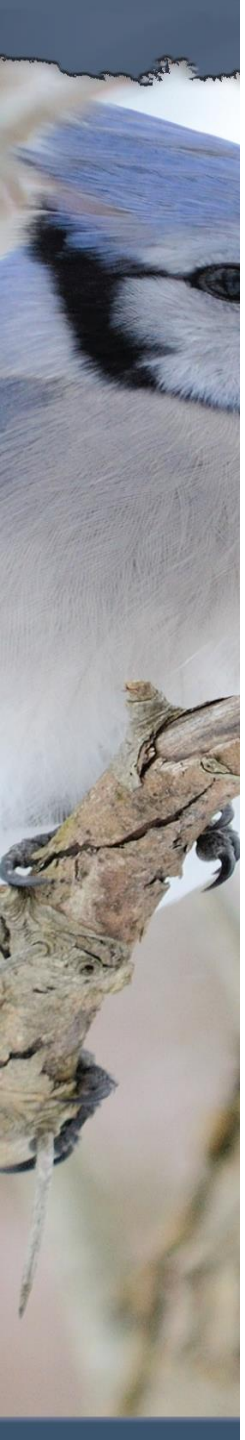
```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```



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!

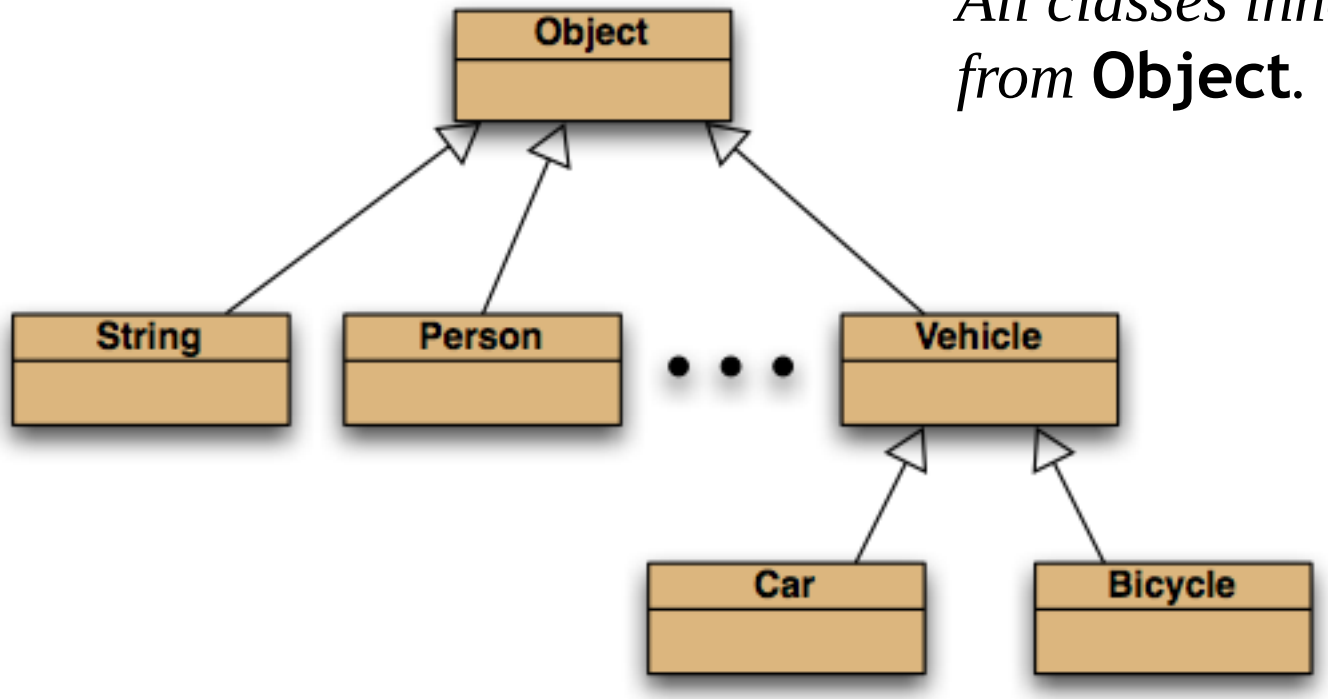
```
Vehicle v;
Car c = new Car();
v = c;  // correct
c = v;  // compile-time error!
```

- Casting fixes this:

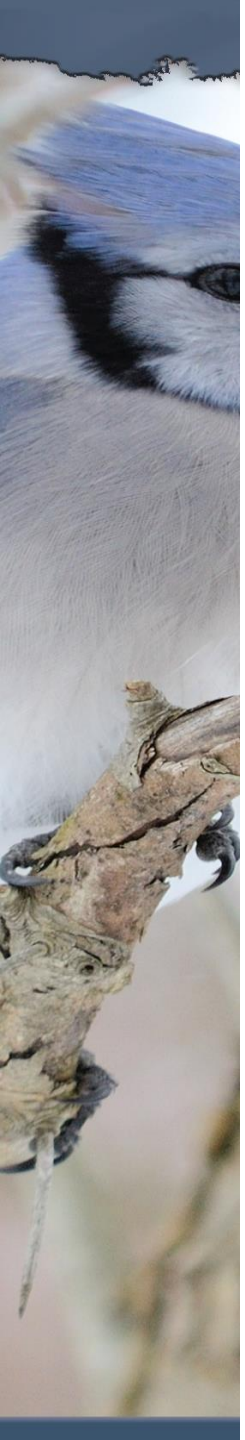
```
c = (Car) v;
```

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

The Object class



All classes inherit from **Object**.

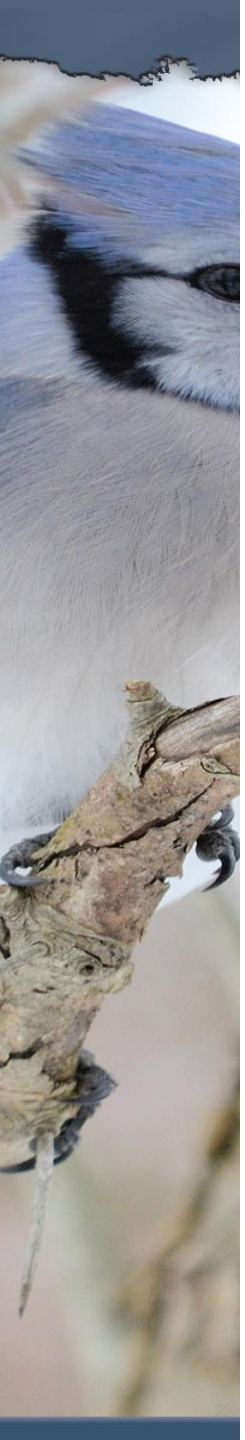


Polymorphic collections

- 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.

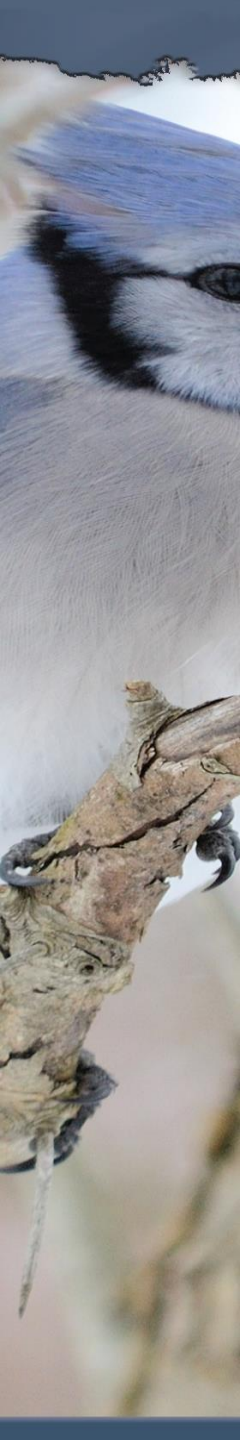
Static and dynamic type

What is the type of c1?

```
Car c1 = new Car();
```

What is the type of v1?

```
Vehicle v1 = new Car();
```

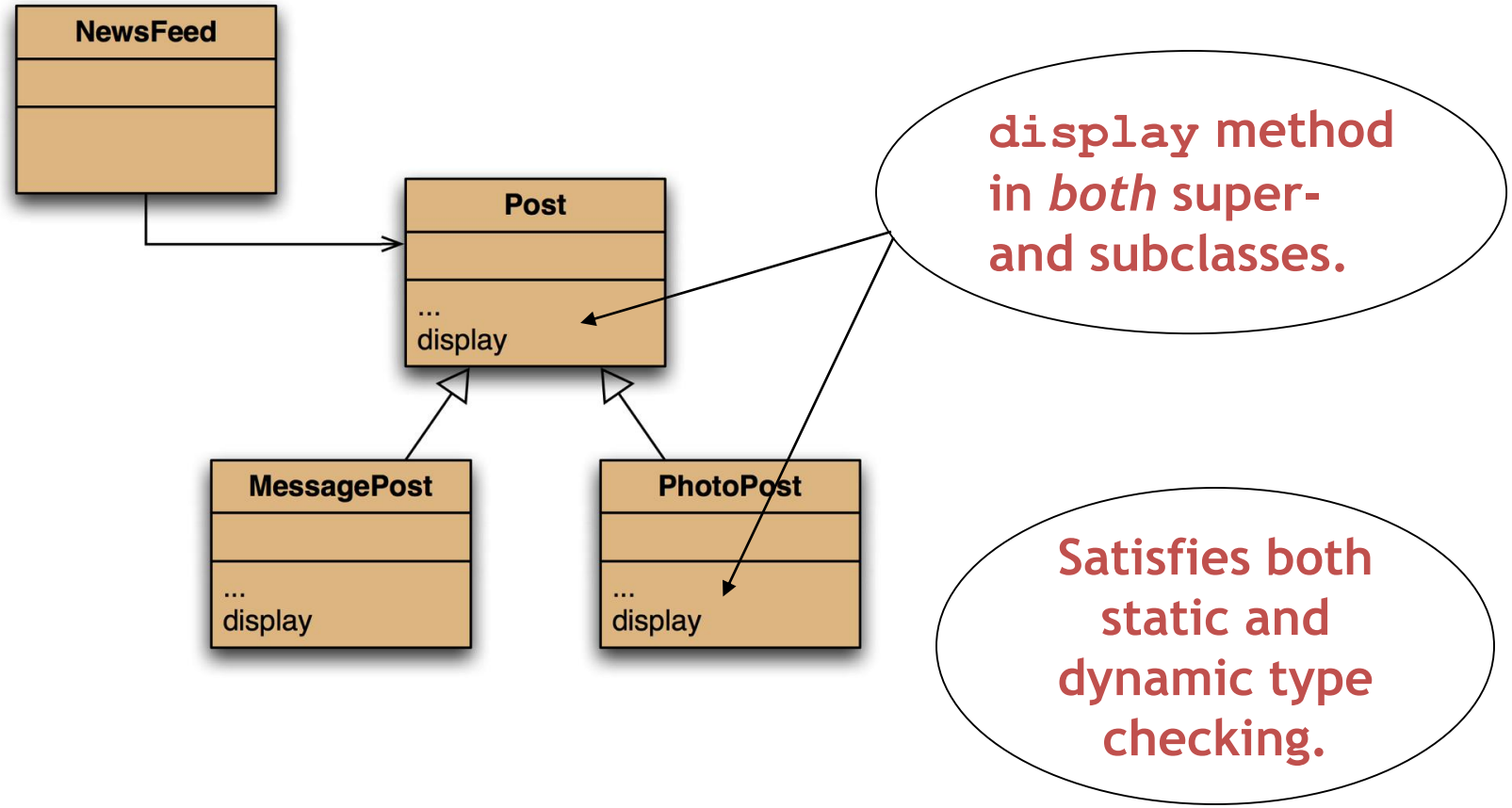


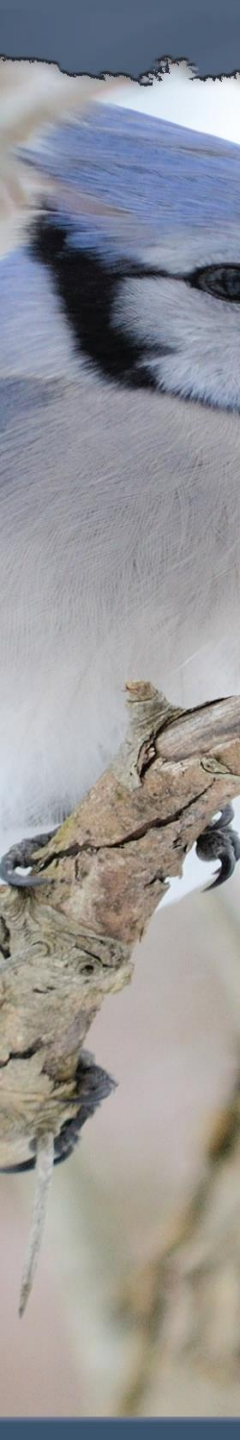
Static and dynamic type

- The declared type of a variable is its *static type*.
- The type of the object a variable refers to is its *dynamic type*.
- The compiler's job is to check for static-type violations.

```
for(Post post : posts) {
    post.display();   // Compile-time error.
}
```

Overriding: the solution

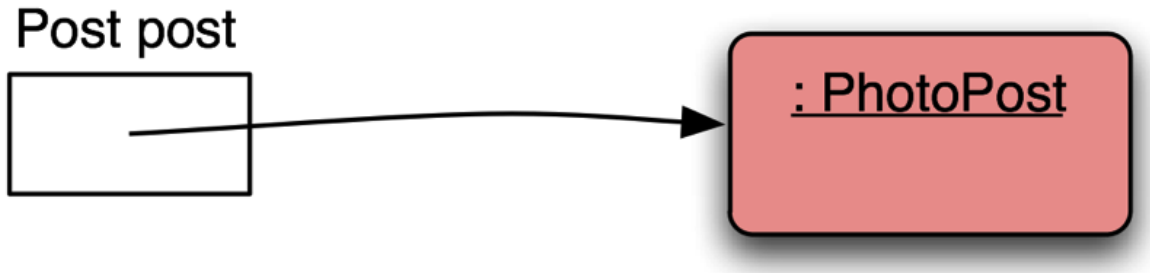




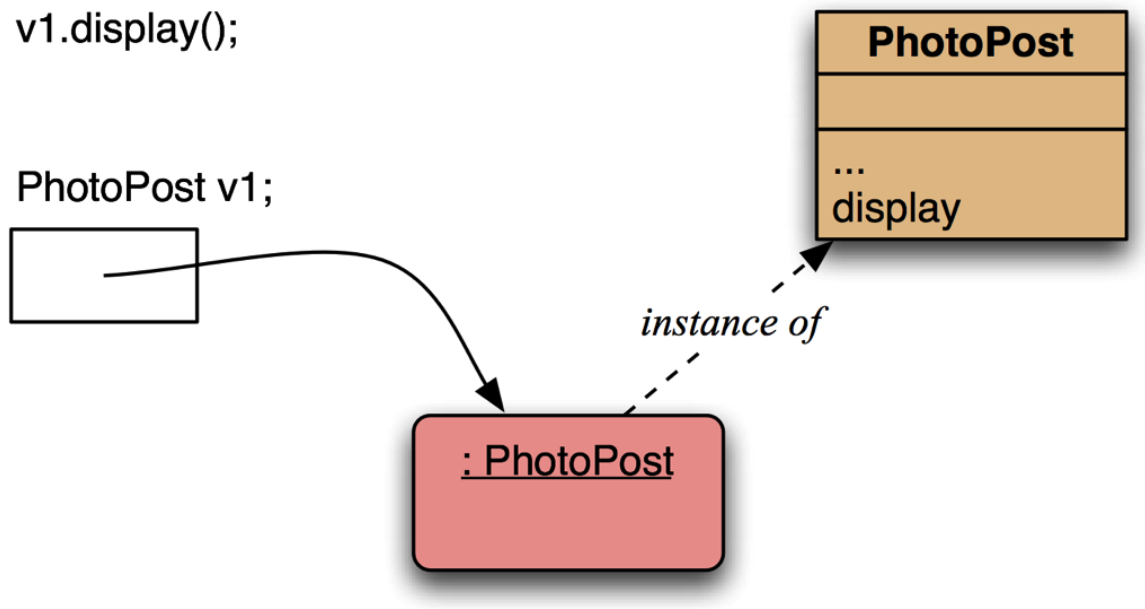
Overriding

- Superclass and subclass define methods with the same signature.
- Each has access to the fields of its class.
- Superclass satisfies static type check.
- Subclass method is called at runtime – it *overrides* the superclass version.
- What becomes of the superclass version?

Distinct static and dynamic types



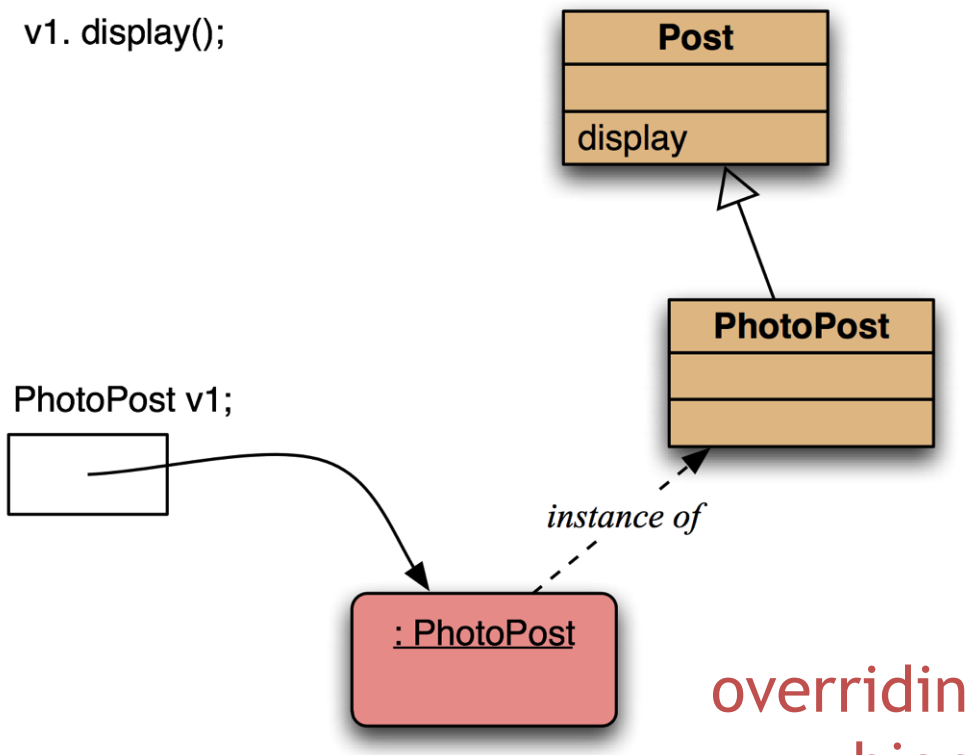
Method lookup



No inheritance or polymorphism.

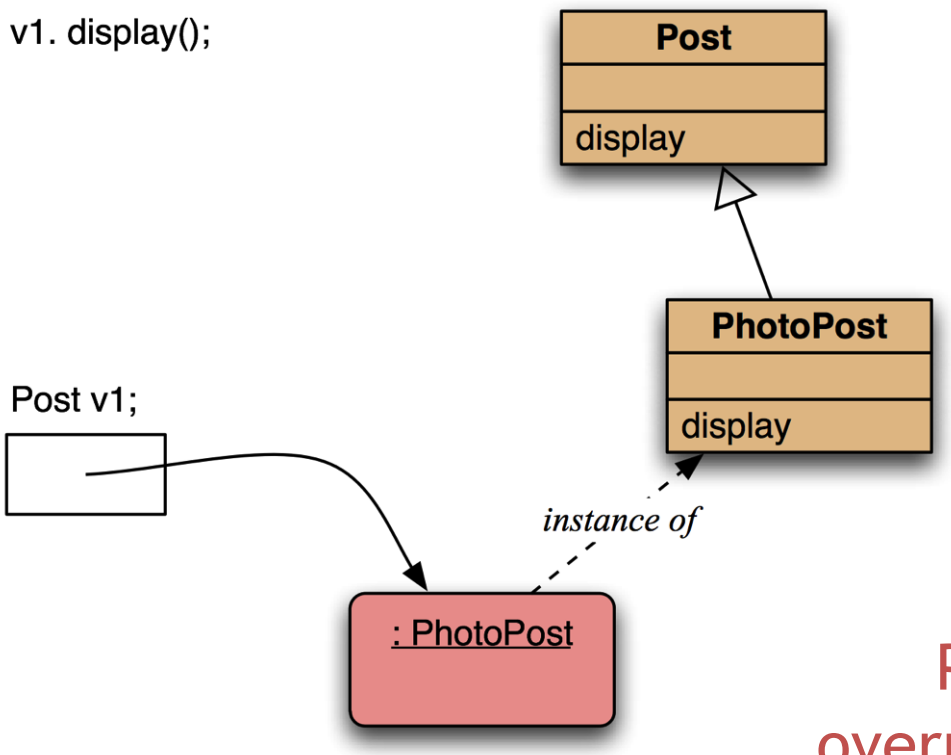
The obvious method is selected.

Method lookup

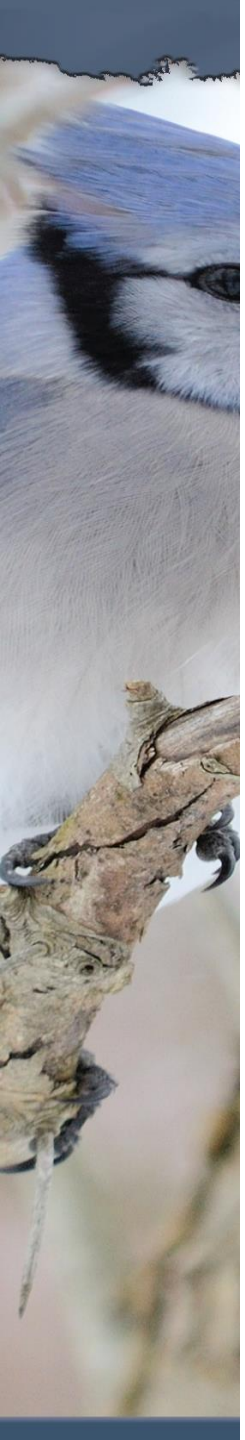


Inheritance but no overriding. The inheritance hierarchy is ascended, searching for a match.

Method lookup

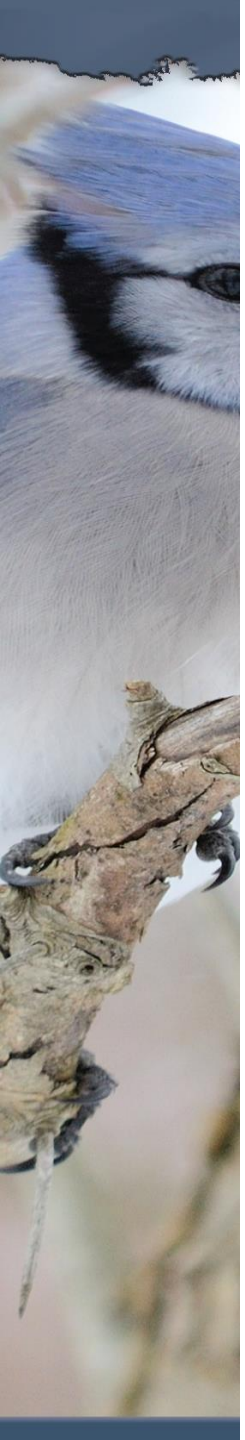


Polymorphism and overriding. The 'first' version found is used.



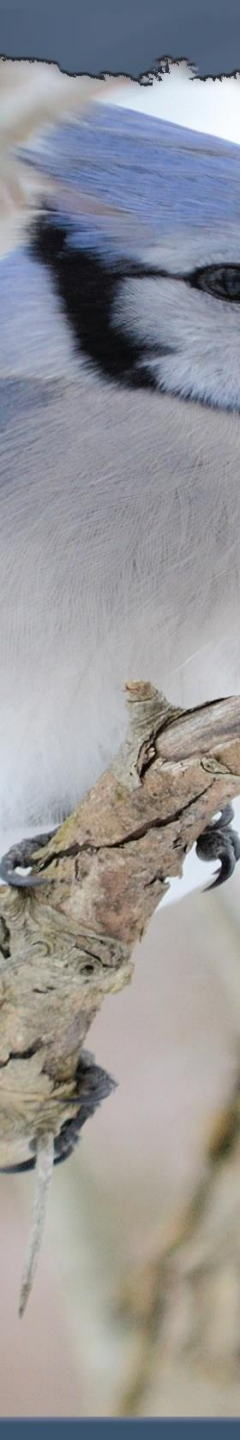
Super call in methods

- Overridden methods are hidden ...
- ... but we often still want to be able to call them.
- An overridden method *can* be called from the method that overrides it.
  - **`super.method(...)`**
  - Compare with the use of **`super`** in constructors.



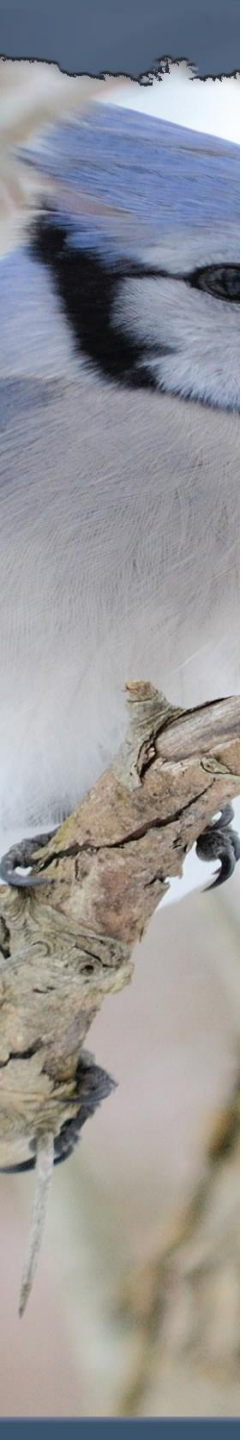
Calling an overridden method

```
public void display()
{
    super.display();
    System.out.println(" [" +
                       filename +
                       "]");
    System.out.println(" " + caption);
}
```



Method polymorphism

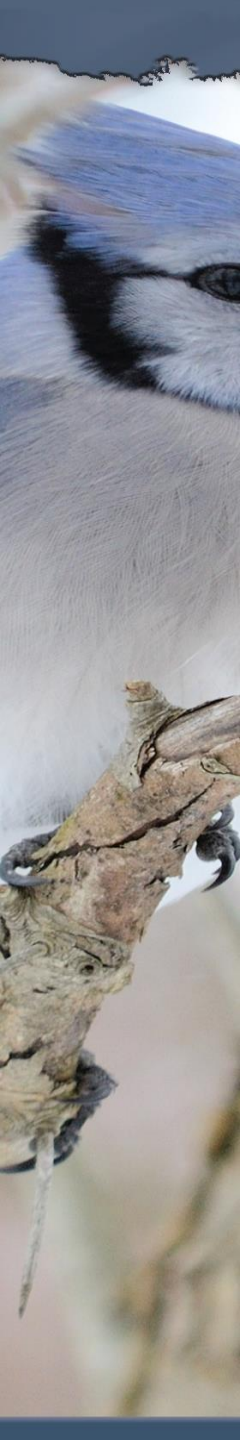
- We have been discussing *polymorphic method dispatch*.
- A polymorphic variable can store objects of varying types.
- Method calls are polymorphic.
  - The actual method called depends on the dynamic object type.



The `instanceof` operator

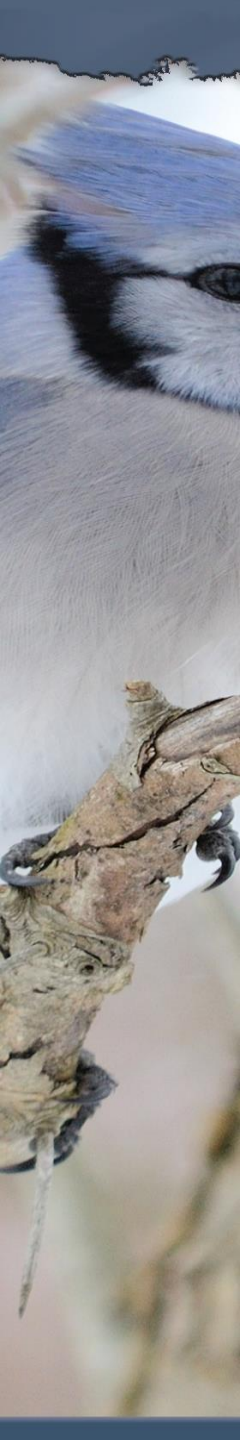
- Used to determine the dynamic type.
- Identifies ‘lost’ type information.
- Usually precedes assignment with a cast to the dynamic type:

```
if(post instanceof MessagePost) {
    MessagePost msg =
        (MessagePost) post;
    … access MessagePost methods via msg …
}
```



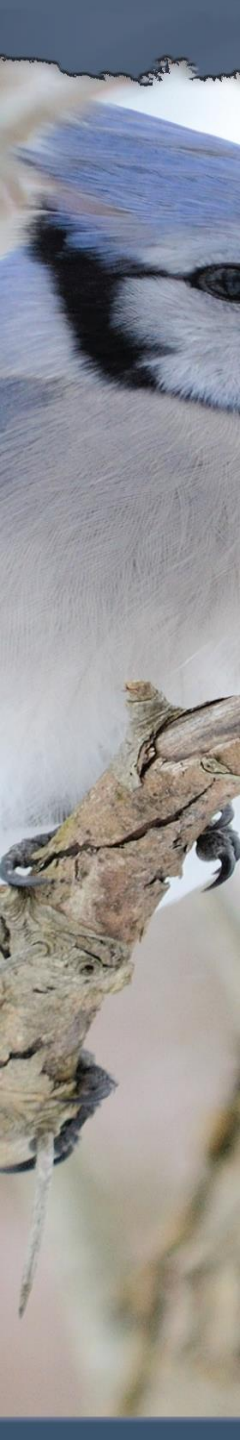
The `Object` class's methods

- Methods in **`Object`** are inherited by all classes.
- Any of these may be overridden.
- The **`toString`** method is commonly overridden:
  - **`public String toString()`**
  - Returns a string representation of the object.



Overriding `toString` in `Post`

```
public String toString()
{
    String text = username + "\n" +
                  timeString(timestamp);
    if(likes > 0) {
        text += " - " + likes + " people like this.\n";
    }
    else {
        text += "\n";
    }
    if(comments.isEmpty()) {
        return text + " No comments.\n";
    }
    else {
        return text + " " + comments.size() +
               " comment(s). Click here to view.\n";
    }
}
```



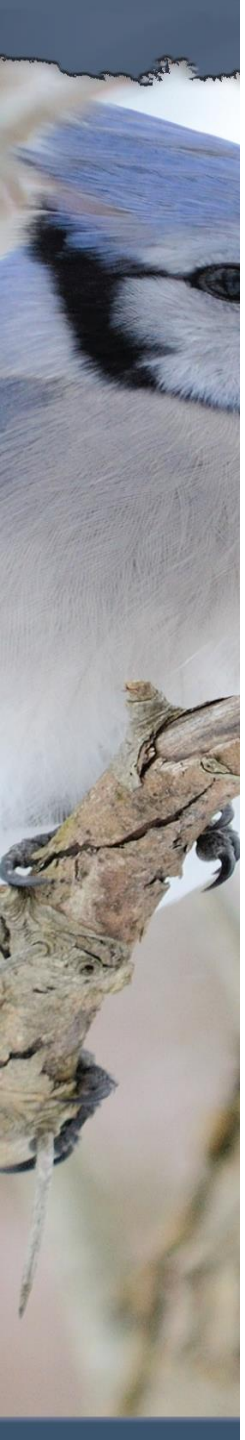
Overriding `toString`

- Explicit print methods can often be omitted from a class:

```
System.out.println(post.toString());
```

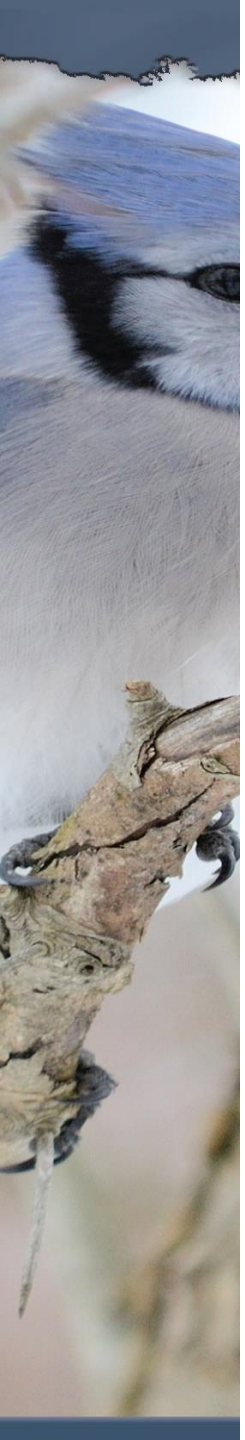
- Calls to **`println`** with just an object automatically result in **`toString`** being called:

```
System.out.println(post);
```



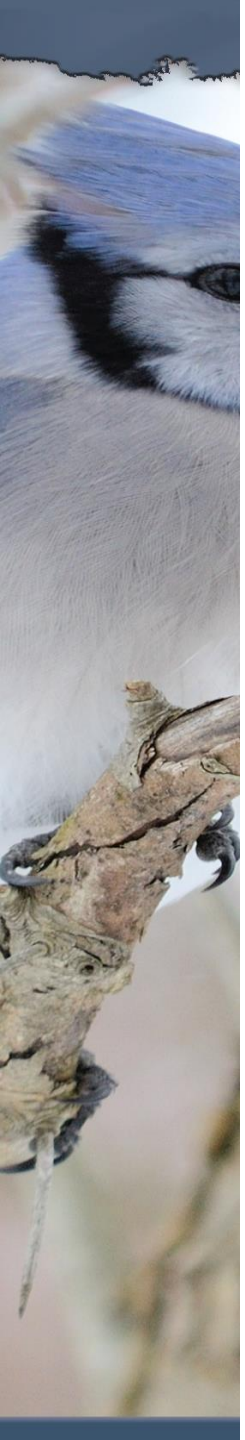
Object equality

- What does it mean for two objects to be ‘the same’?
  - Reference equality.
  - Content equality.
- Compare the use of `==` with `equals()` between strings.



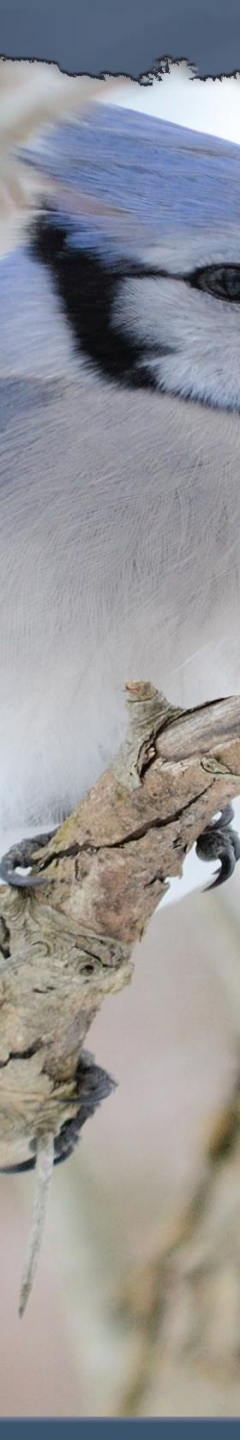
Overriding `equals`

```
public boolean equals(Object obj)
{
    if(this == obj) {
        return true;
    }
    if(!(obj instanceof ThisType)) {
        return false;
    }
    ThisType other = (ThisType) obj;
    … compare fields of this and other
}
```



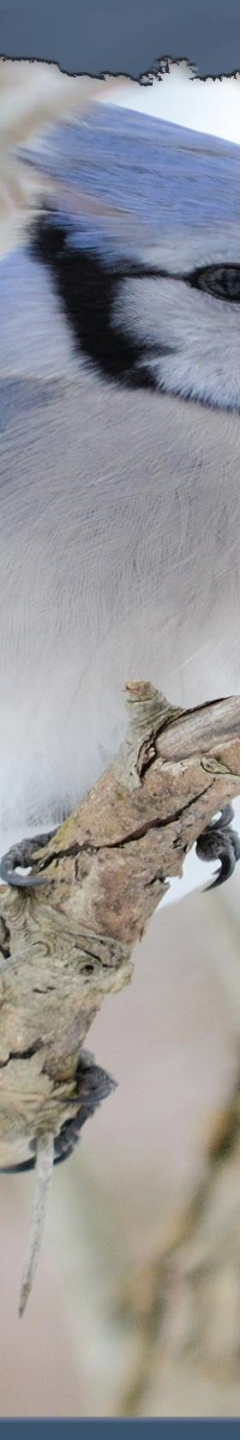
Overriding `equals` in `Student`

```
public boolean equals(Object obj)
{
    if(this == obj) {
        return true;
    }
    if(!(obj instanceof Student)) {
        return false;
    }
    Student other = (Student) obj;
    return name.equals(other.name) &&
           id.equals(other.id) &&
           credits == other.credits;
}
```

Overriding `hashCode` in `Student`

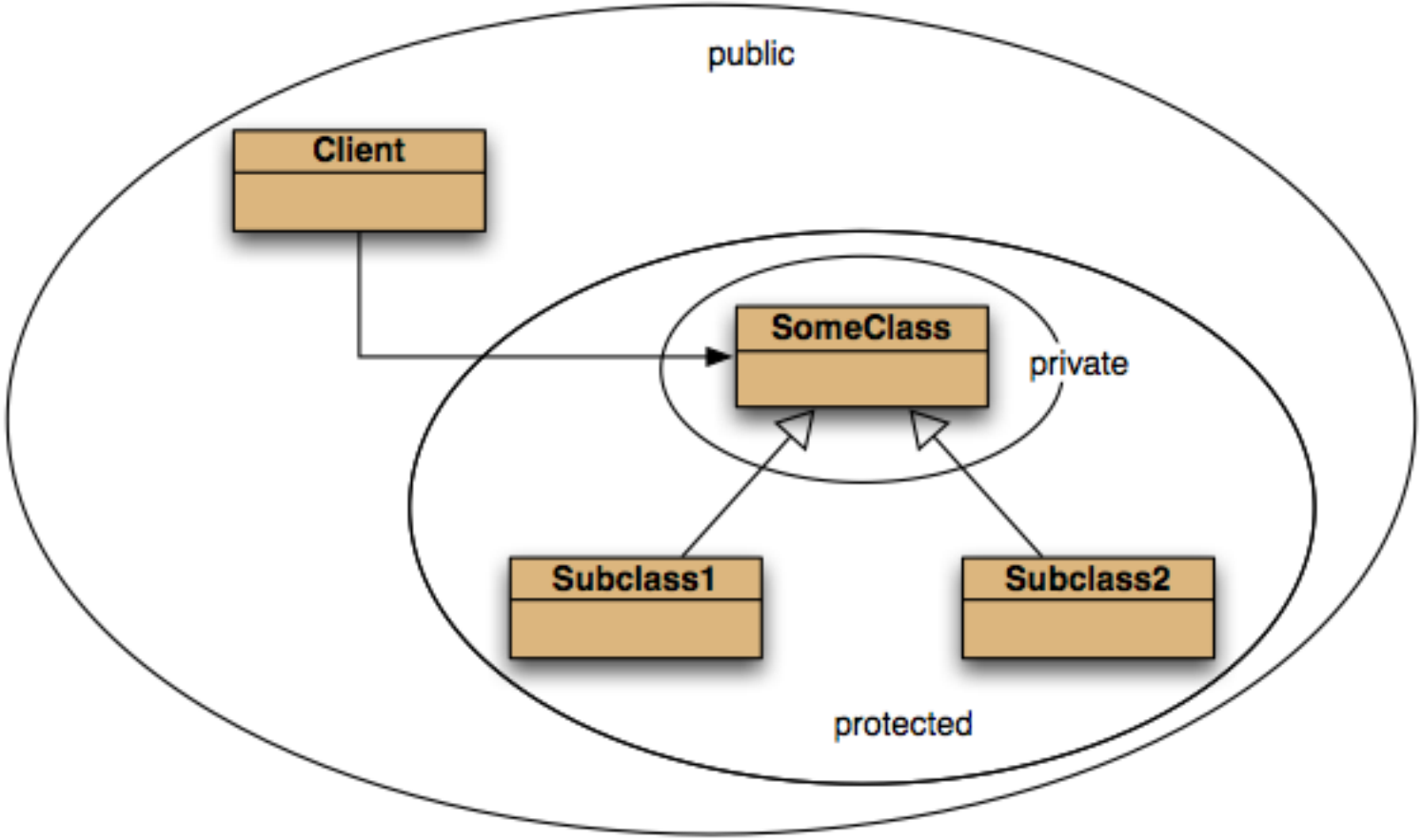
```
/**
 * Hashcode technique taken from
 * Effective Java by Joshua Bloch.
 */
public int hashCode()
{
    int result = 17;
    result = 37 * result + name.hashCode();
    result = 37 * result + id.hashCode();
    result = 37 * result + credits;
    return result;
}
```

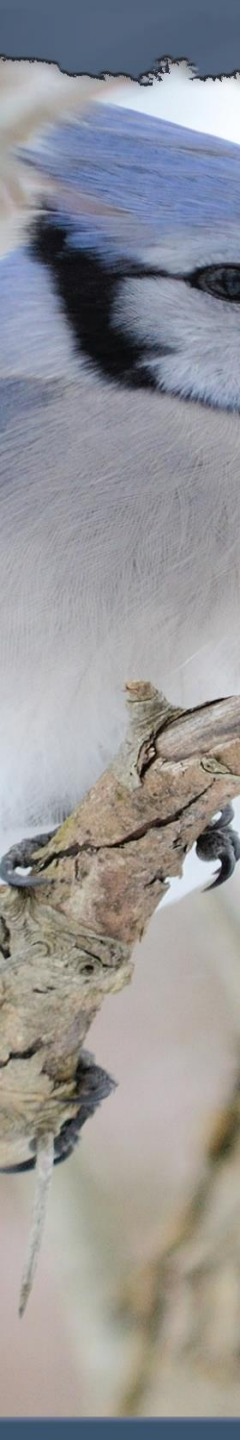


Protected access

- Private access in the superclass may be too restrictive for a subclass.
- The closer inheritance relationship is supported by *protected access*.
- Protected access is more restricted than public access.
- We still recommend keeping fields private.
  - Define protected accessors and mutators.

Access levels

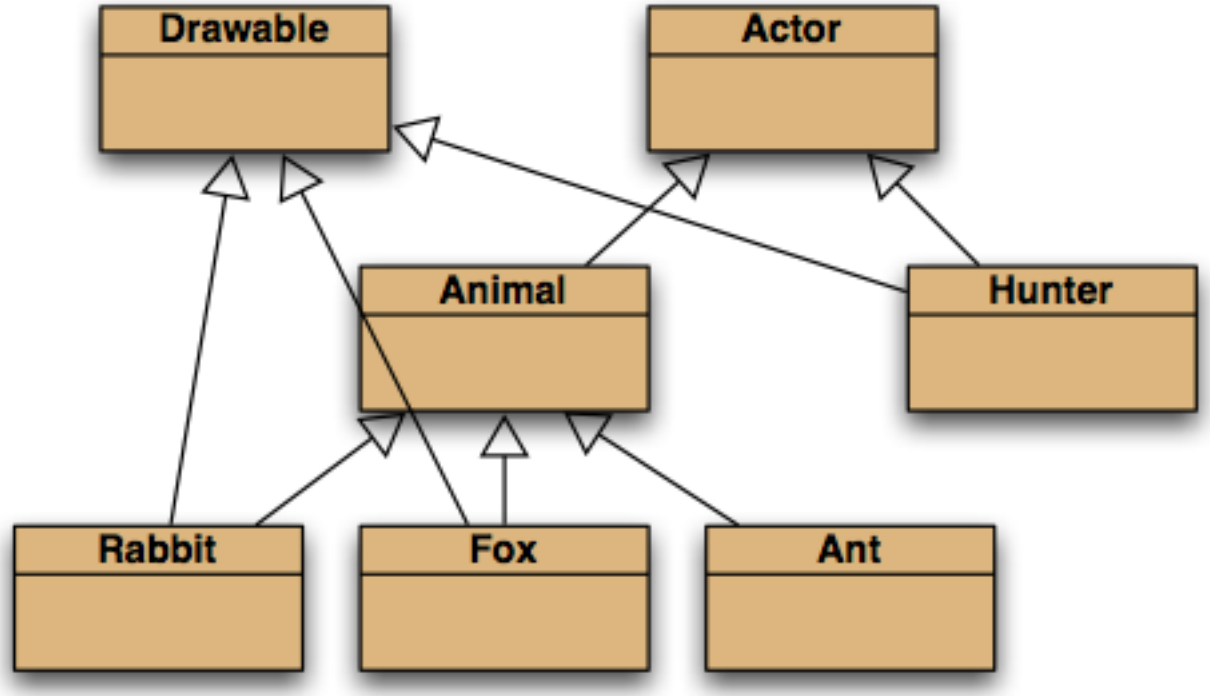


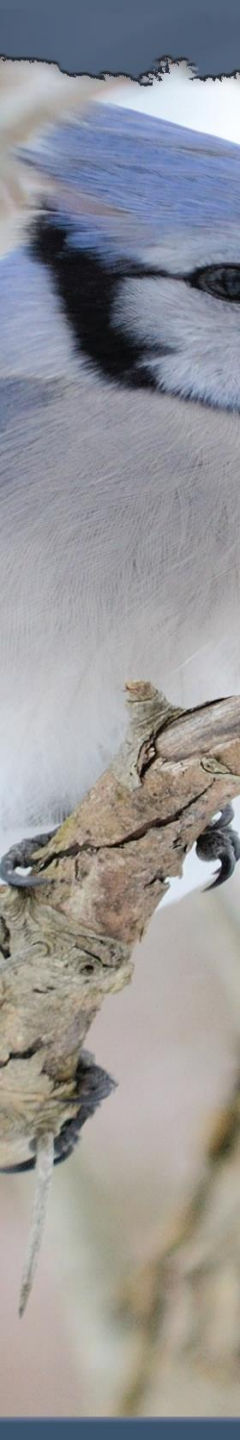


Abstract classes and methods

- Abstract methods have **`abstract`** in the signature.
- Abstract methods have no body.
- Abstract methods make the class abstract.
- *Abstract classes cannot be instantiated.*
- Concrete subclasses complete the implementation.

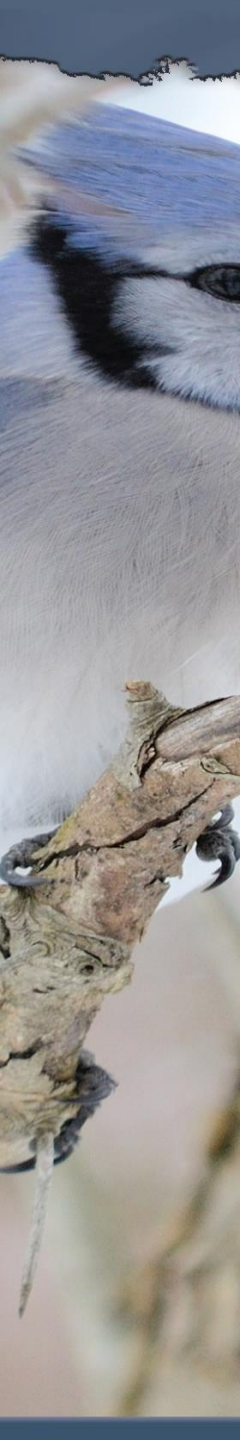
Selective drawing (multiple inheritance)





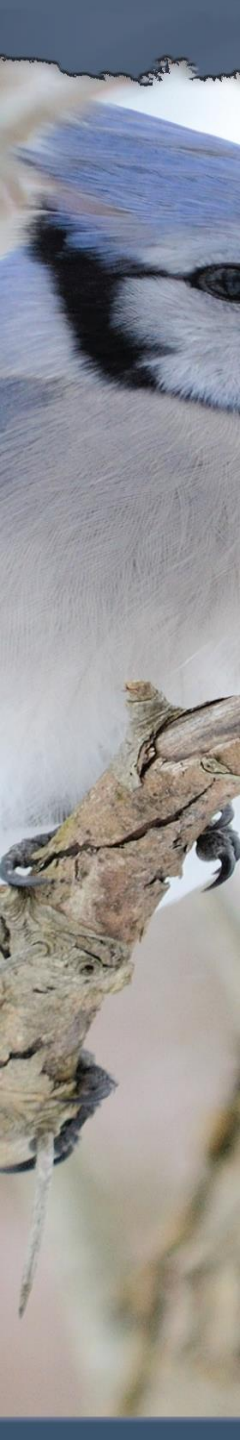
Multiple inheritance

- Having a class inherit directly from multiple ancestors.
- Each language has its own rules.
  - How to resolve competing definitions?
- Java forbids it for classes.
- Java permits it for interfaces.



Interfaces as types

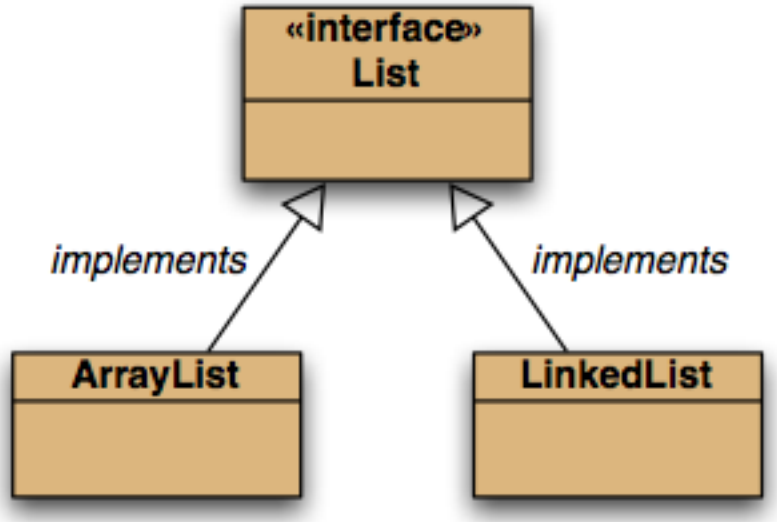
- Implementing classes are subtypes of the interface type.
- So, polymorphism is available with interfaces as well as classes.

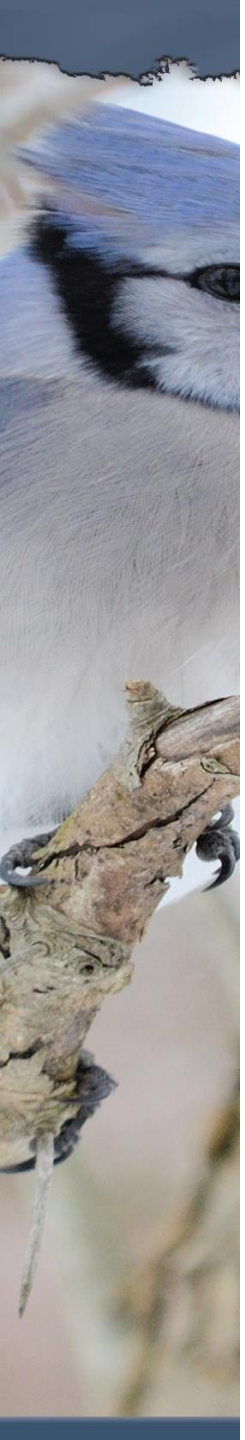


Features of interfaces

- Use **`interface`** rather than **`class`** in their declaration.
- They do not define constructors.
- All methods are **`public`**.
- All fields are **`public`**, **`static`** and **`final`**. (Those keywords may be omitted.)
- Abstract methods may omit **`abstract`**.

Alternative implementations





Nå

- Kahoot☺
- Deretter øving her på Fjerdingen (sjekk TimEdit for rom)
  - Ingen spesifikke øvingsoppgaver denne uken. Jobb med arbeidskravet (eller andre oppgaver av eget ønske).

- Neste uke: JavaFX