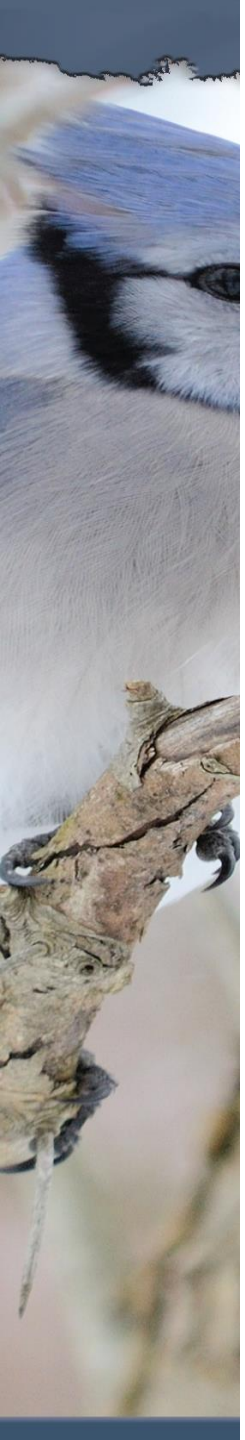
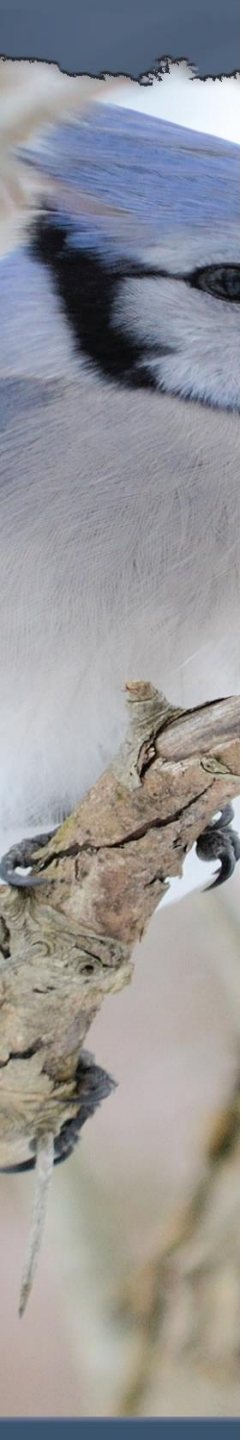


Further abstraction techniques

Abstract classes and interfaces



Forelesningsplan PGR101 V2018

Det kommer helt sikkert noen endringer underveis. Husk at for tid og sted er det TimeEdit som gjelder.

Lærebok: Samme som i forrige semester: BlueJ-boka.

Økt1: Arv i objektorientert programmering (1 av 4). Kapittel 10.

Økt2: Arv i objektorientert programmering (2 av 4). Kapittel 10.

Økt3: Arv i objektorientert programmering (3 av 4). Kapittel 11.

Økt4: Arv i objektorientert programmering (4 av 4). Kapittel 11.

Økt5: Abstrakte klasser og grensesnitt (1 av 2). Kapittel 12.

Økt6: Abstrakte klasser og grensesnitt (2 av 2). Kapittel 12.

Økt7: Brukergrensesnitt - GUI (Java FX) (1 av 2). Her må vi nok benytte nettressurser.

Økt8: Brukergrensesnitt - GUI (Java FX) (2 av 2). Her må vi nok benytte nettressurser.

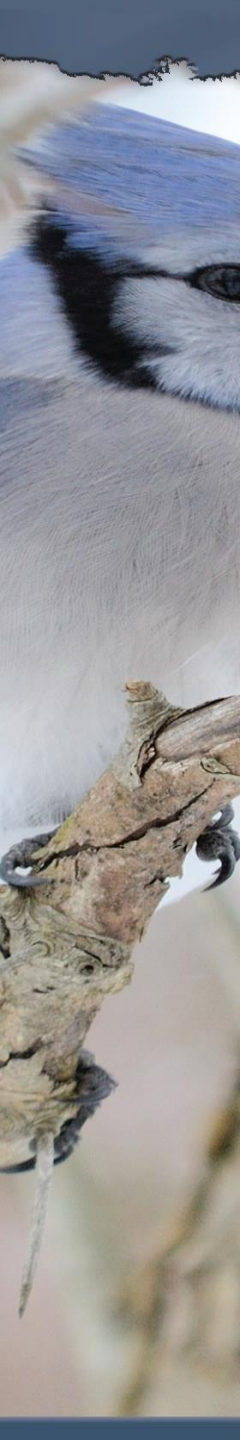
Økt9: Unntakshåndtering (1 av 2). Kapittel 14.

Økt10: Unntakshåndtering (2 av 2). Kapittel 14.

Økt11: Å designe kode. Kapittel 15.

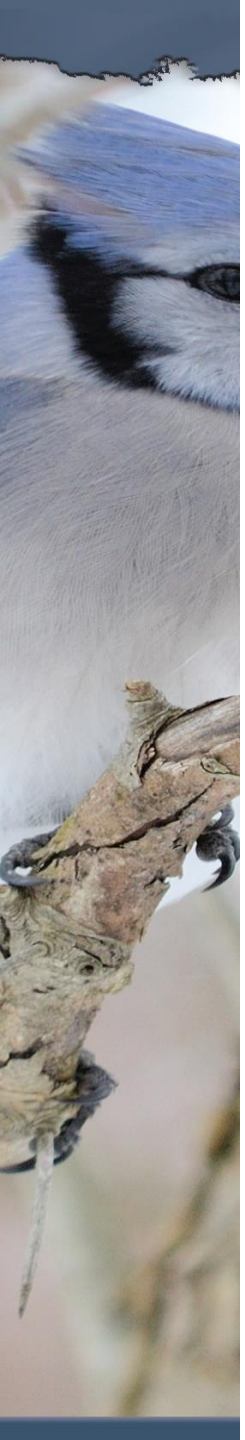
Økt12: Repetisjon. Kapittel 16.

Det er to arbeidskrav i emnet. De benyttes som i forrige semester. Det blir trolig ett arbeidskrav som omhandler arv, og ett som omhandler GUI.



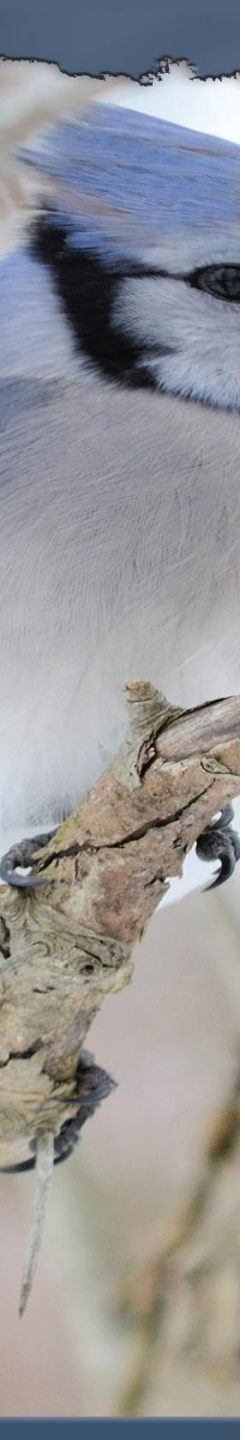
Main concepts to be covered

- Abstract classes
- Interfaces
- Multiple inheritance



Abstract classes and methods

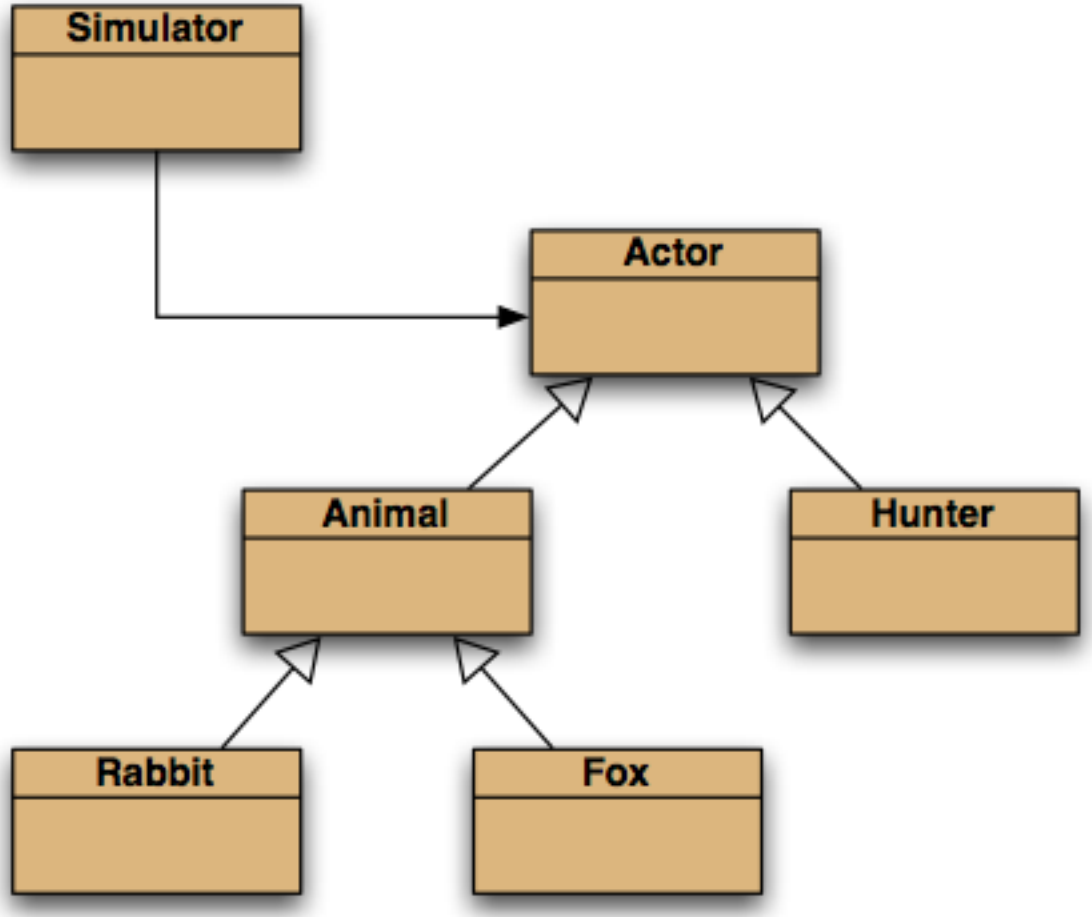
- Abstract methods have **`abstract`** in the signature.
- Abstract methods have no body.
- Abstract methods make the class abstract.
- *Abstract classes cannot be instantiated.*
- Concrete subclasses complete the implementation.

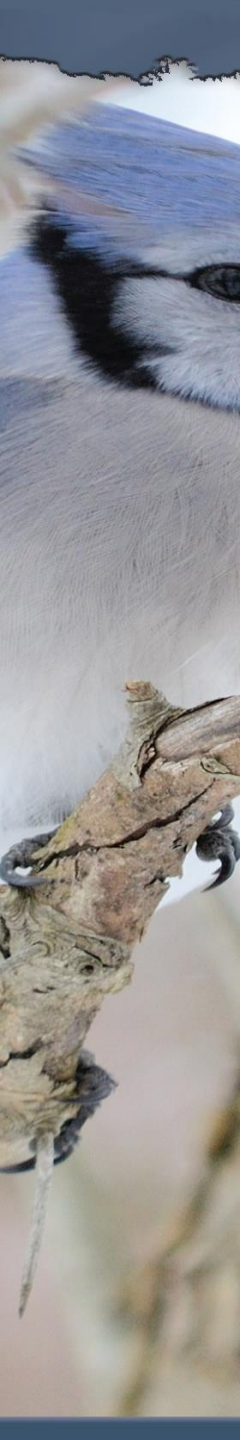


The Animal class

```
public abstract class Animal
{
    fields omitted

    /**
     * Make this animal act - that is: make it do
     * whatever it wants/needs to do.
     */
    abstract public void act(List<Animal> newAnimals);

    other methods omitted
}
```

Further abstraction





An Actor class

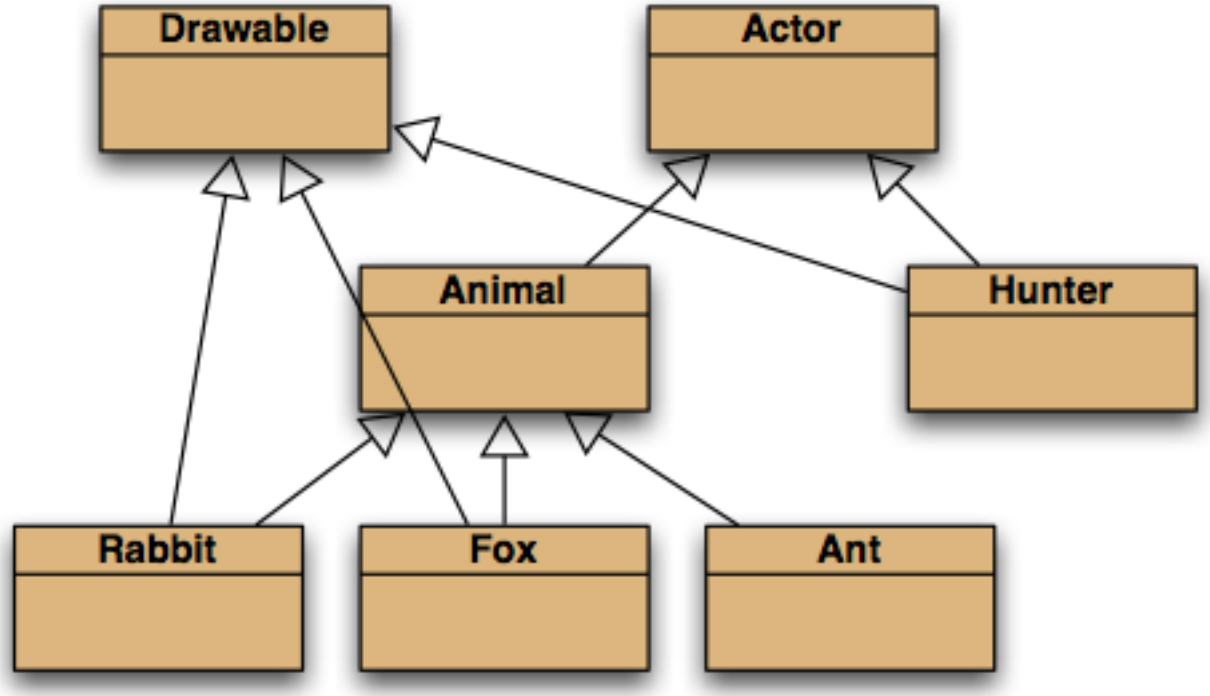
```
public abstract class Actor
{
        abstract public void act(List<Actor> newActors);

        abstract public boolean isActive();
}
```

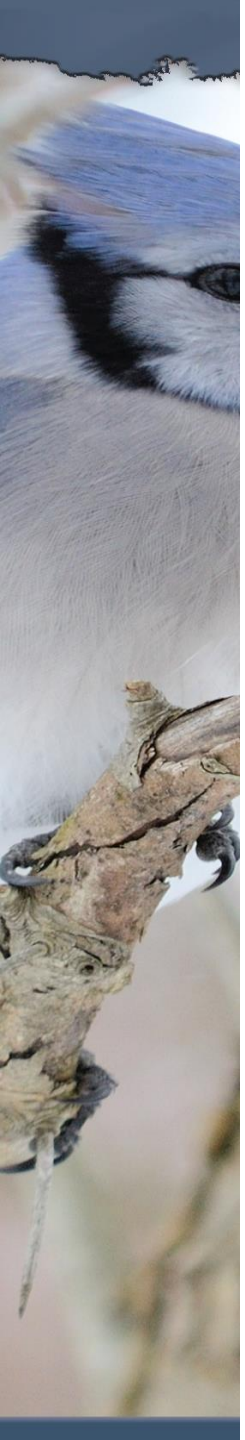
Oppgave 1 (12.53 i boka):

- Lag klassen `Actor` som `Animal` arver fra.
- Endre `simulateOneStep` i `Simulator` slik at `Actor` benyttes.

(Ta utgangspunkt i `foxes-and-rabbits-v2`)

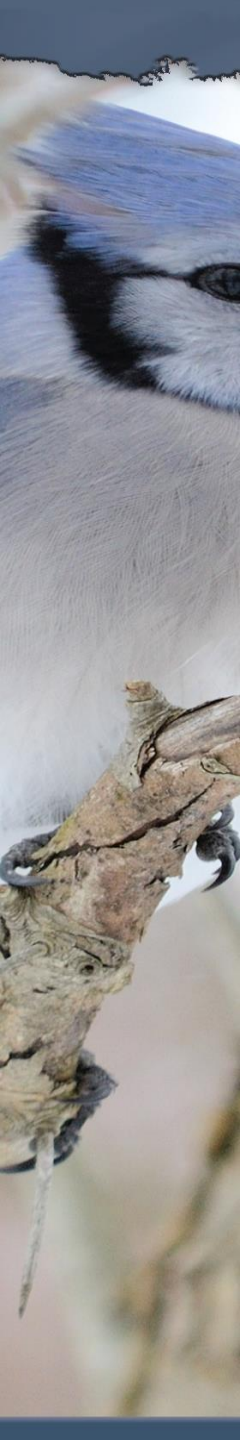
Selective drawing (multiple inheritance)





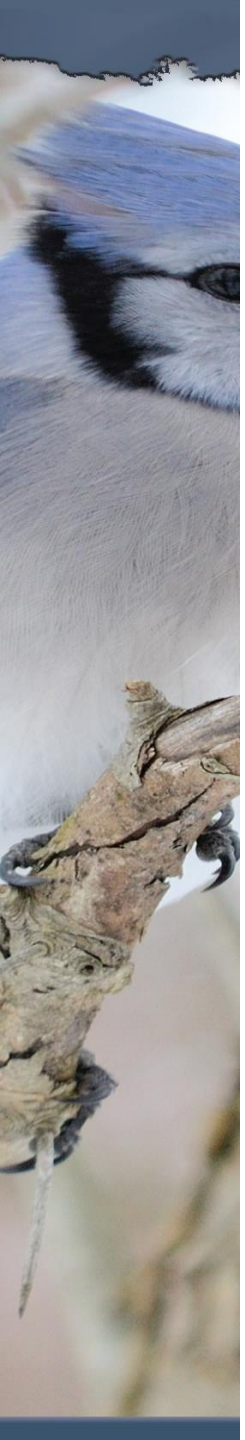
Multiple inheritance

- 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.



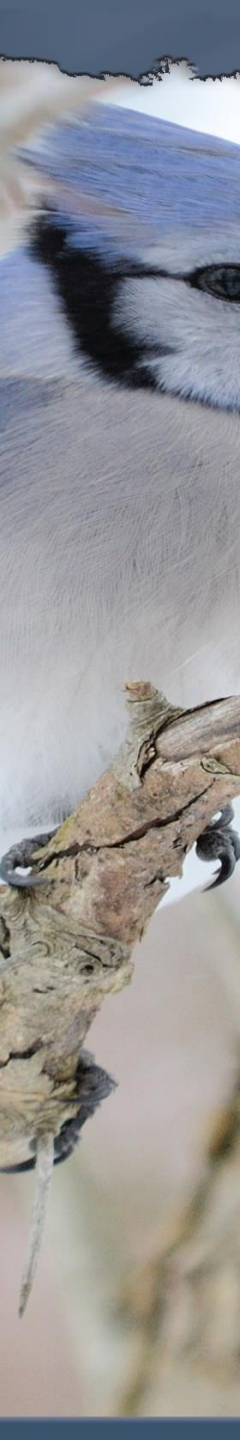
An Actor interface

```
public interface Actor
{
    /**
     * Perform the actor's regular behavior.
     * @param newActors A list for storing newly created
     *                  actors.
     */
    void act(List<Actor> newActors);

    /**
     * Is the actor still active?
     * @return true if still active, false if not.
     */
    boolean isActive();
}
```



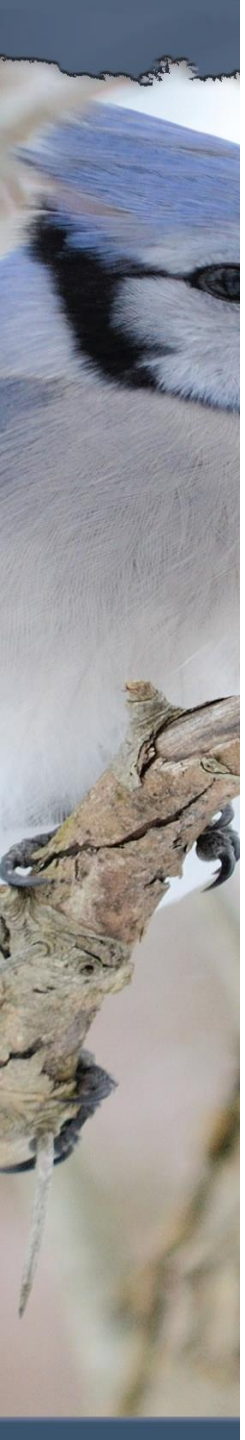
Classes *implement* an interface

```
public class Fox extends Animal
    implements Drawable
{
   ...
}

public class Hunter
    implements Actor, Drawable
{
   ...
}
```



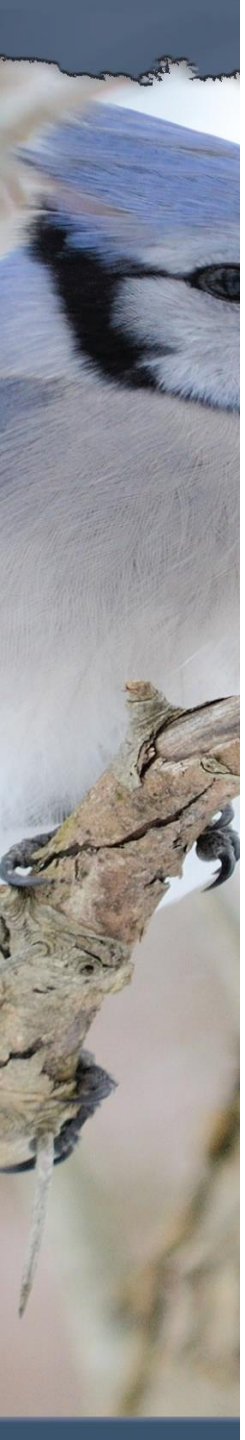
Interfaces as types

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



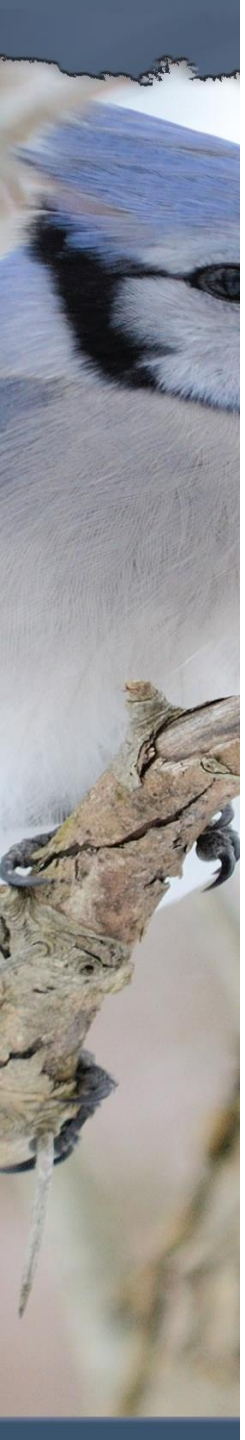
Features of interfaces

- 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`**.



Features of interfaces

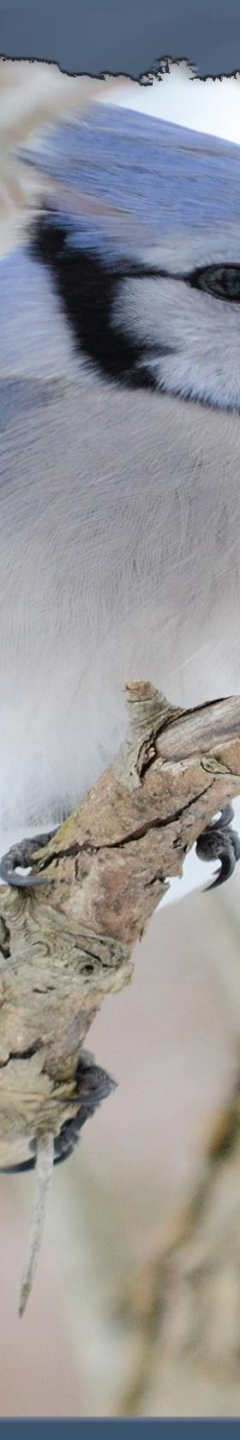
- Methods marked as **`default`** have a method body – they are not abstract.
- Methods marked as **`static`** have a method body.
- Default and static methods could complicate multiple inheritance of interfaces.



Default methods

- Introduced in Java 8 to adapt legacy interfaces; e.g., **`java.util.List`**.
- Classes inheriting two with the same signature must override the method.
- Syntax for accessing the overriden version:

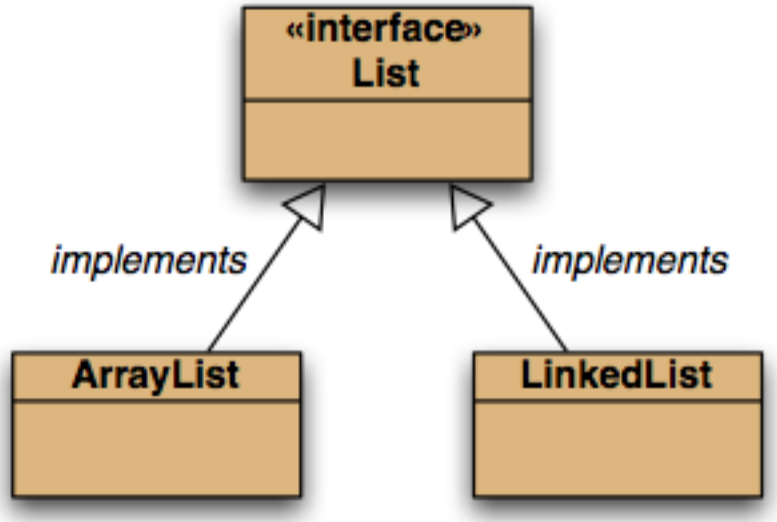
***`InterfaceType`*`.super.`*`method`*`(…)`**

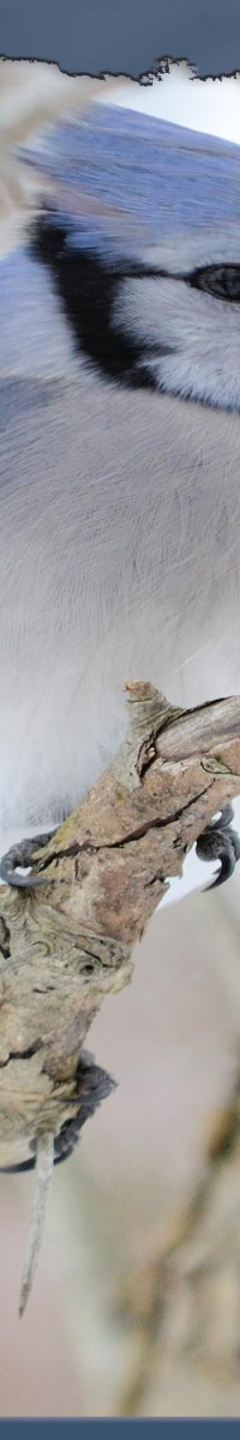


Interfaces as specifications

- Strong separation of functionality from implementation.
  - Though parameter and return types are mandated.
- Clients interact independently of the implementation.
  - But clients can choose from alternative implementations.
- **`List`**, **`Map`** and **`Set`** are examples.

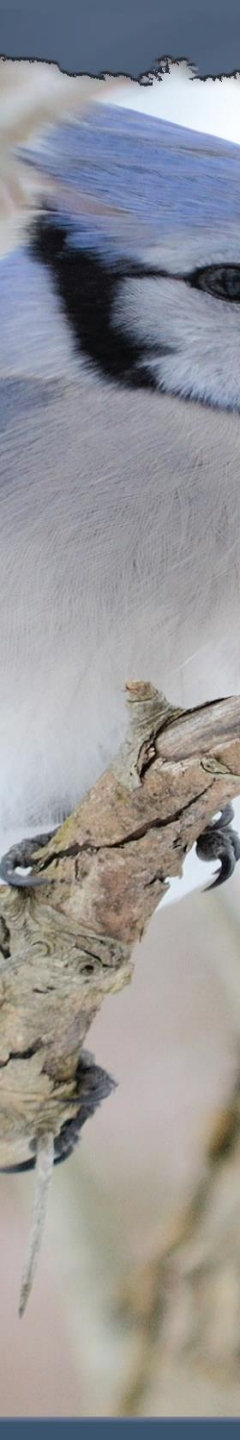
Alternative implementations





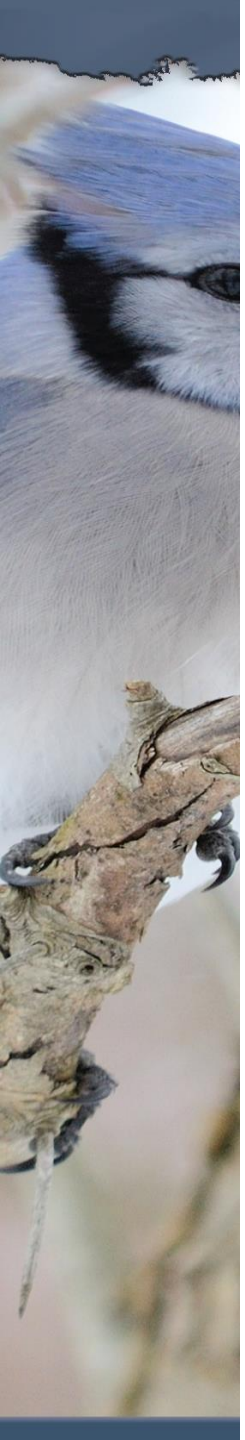
Oppgave 2

- Lag et lite program med én klasse.
- Lag en metode (A) som oppretter en variable av type List og instansierer den med en implementasjon du velger selv.
- Send listen til en metode (B) som fyller den med noe innhold.
- Skriv ut innholdet av listen i A.



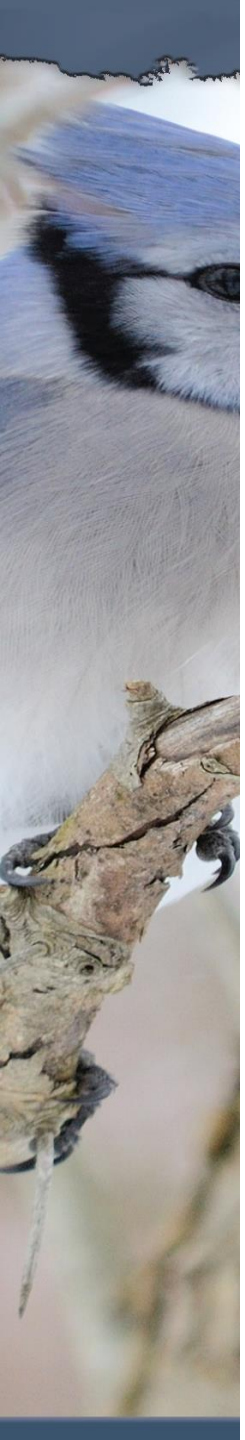
Oppgave 3

- Endre slik at du instansierer listen med en annen implementasjon.
- Ble det noe forandring andre steder i programmet ditt?



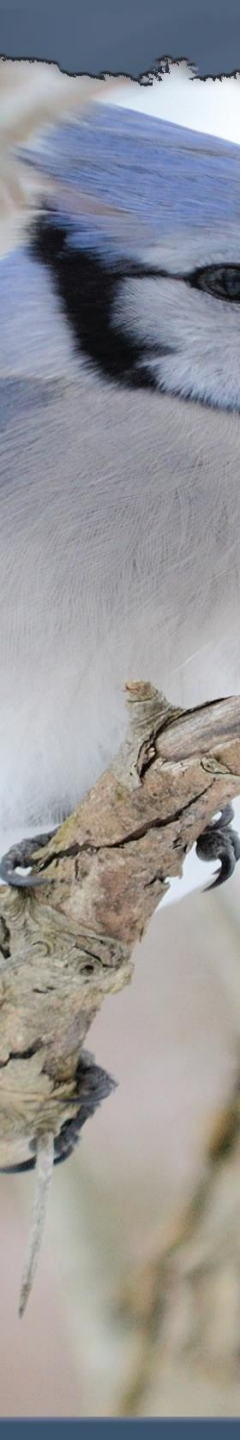
Functional interfaces and lambdas (advanced)

- Interfaces with a single abstract method are *functional interfaces*.
- **`@FunctionalInterface`** is the associated annotation.
- A lambda may be used where a functional interface is required.
- **`java.util.function`** defines some functional interfaces.



Common functional interfaces (advanced)

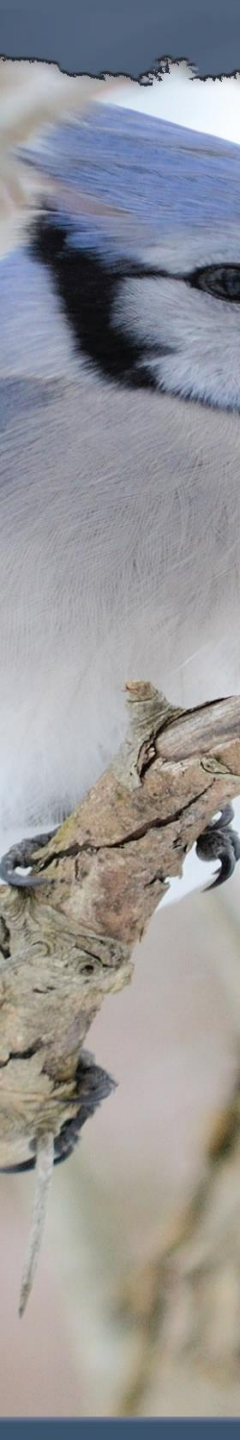
- **`Consumer`**: for lambdas with a **`void`** return type.
- **`BinaryOperator`**: for lambdas with two parameters and a matching result type.
- **`Supplier`**: for lambdas returning a result.
- **`Predicate`**: returns a **`boolean`**.



Functional interfaces (advanced)

- Having functional types for lambdas means we can assign them to variables, or pass them as parameters; e.g.:

```
BinaryOperator<String> aka =
    (name, alias) -> {
        return name +  " (AKA " +
               alias + ")"; };
```

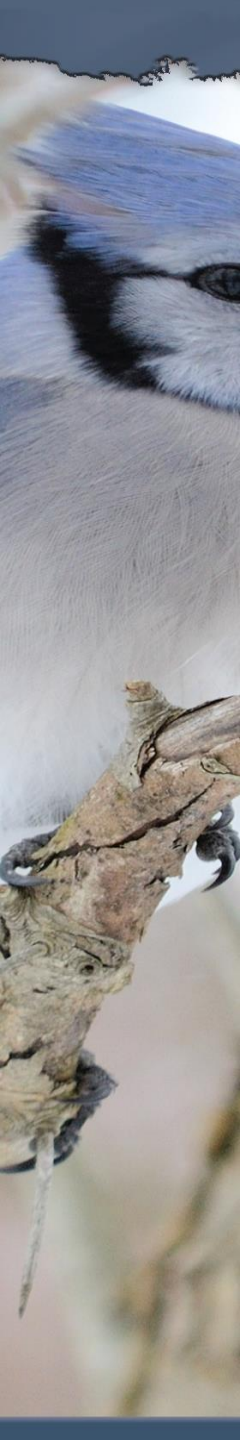
Oppgave (vanskelig)

- Vi skal forsøke å bare skrive ut tekster med "b" I seg. Vi skal bruke Streams og lambda☺
- Tips:
  - Lage Stream fra en collection: <collection>.stream()
  - filter
  - Predicate
  - forEach

```
filter(Predicate<? super T> predicate)
Returns a stream consisting of the elements of this stream that match the given predicate.
```

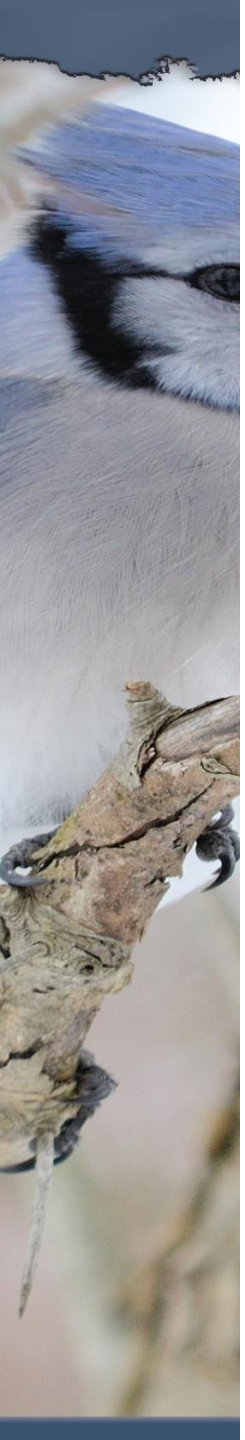
```
boolean test(T t)

Evaluates this predicate on the given argument.
```

```
void forEach(Consumer<? super T> action)

Performs an action for each element of this stream.
```



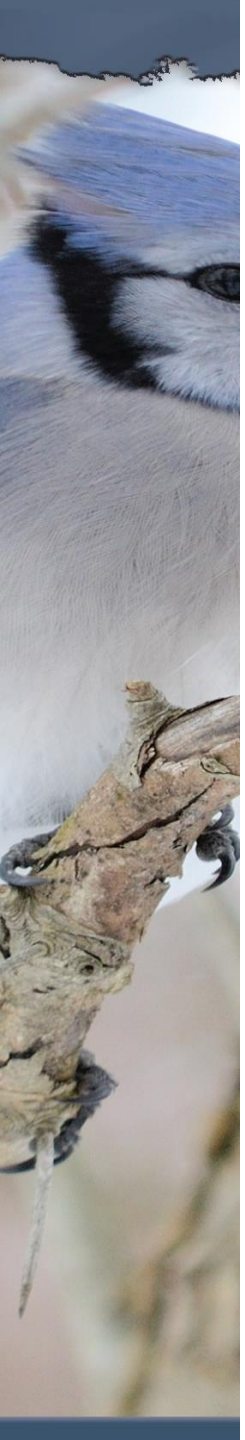
The `Class` class

- A **`Class`** object is returned by **`getClass()`** in **`Object`**.
- The **`.class`** suffix provides a **`Class`** object:
  `Fox.class`
- Used in **`SimulatorView`**:
  `Map<Class, Color> colors;`
- **`String getName()`** for the class name.



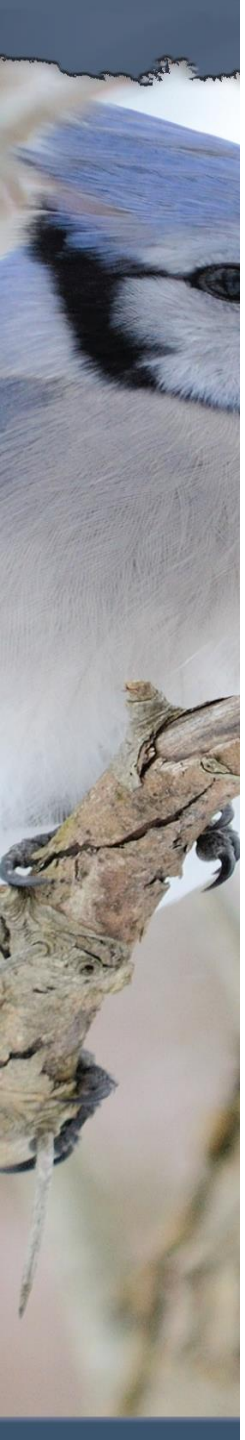
Review

- Inheritance can provide shared implementation.
  - Concrete and abstract classes.
- Inheritance provides shared type information.
  - Classes and interfaces.



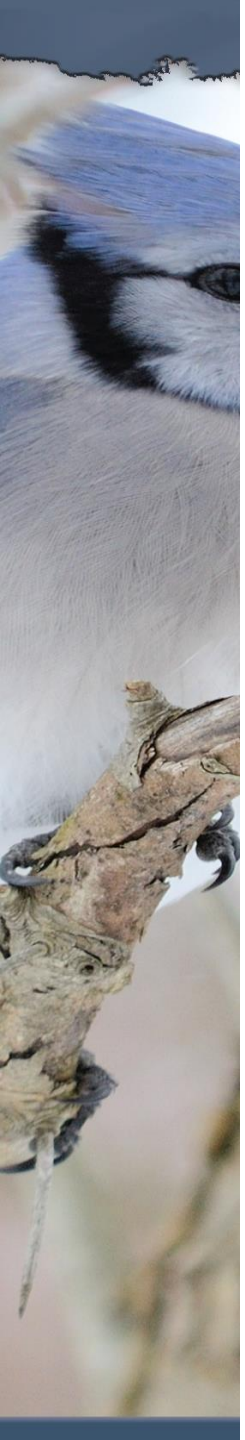
Review

- Abstract methods allow static type checking without requiring implementation.
- Abstract classes function as incomplete superclasses.
  - No instances.
- Abstract classes support polymorphism.



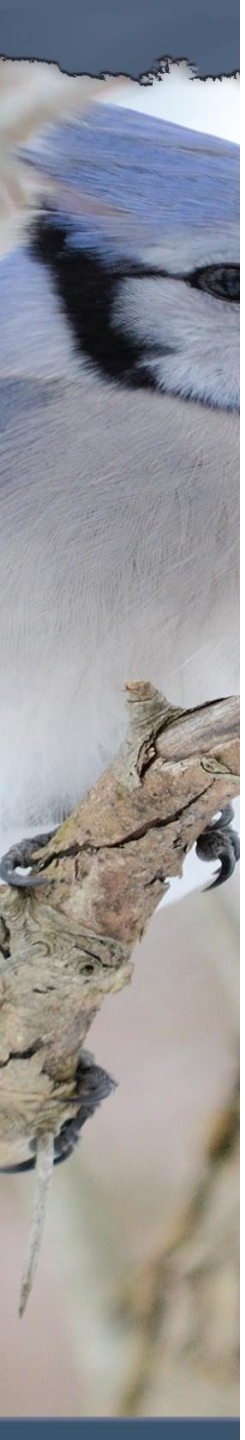
Review

- Interfaces provide specification – usually without implementation.
  - Interfaces are abstract apart from their default methods.
- Interfaces support polymorphism.
- Java interfaces support multiple inheritance.



Arbeidskrav 1

- Publiseres i morgen.
- Frist 21. mars.
- Jeg anbefaler at dere først gjør denne ukes oppgaver før dere starter på arbeidskravet.
- Neste uke blir det ingen øvingsoppgaver.
- Neste forelesning blir repetisjon/arbeidskrav.



Nå

- Kahoot☺
- Deretter øving her på Vulkan (sjekk TimEdit for rom)