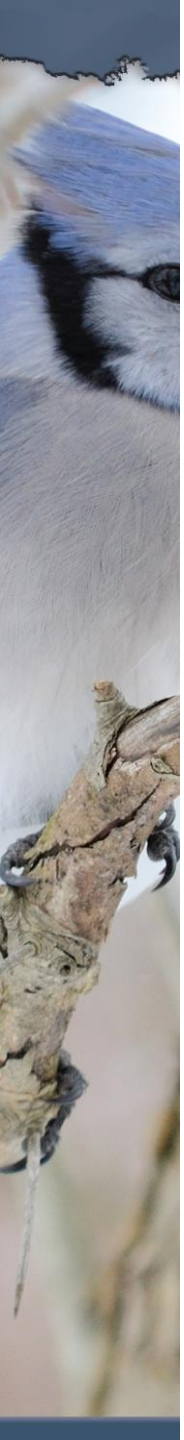
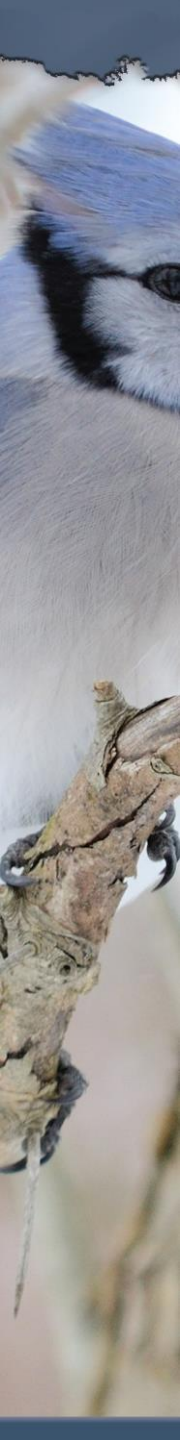


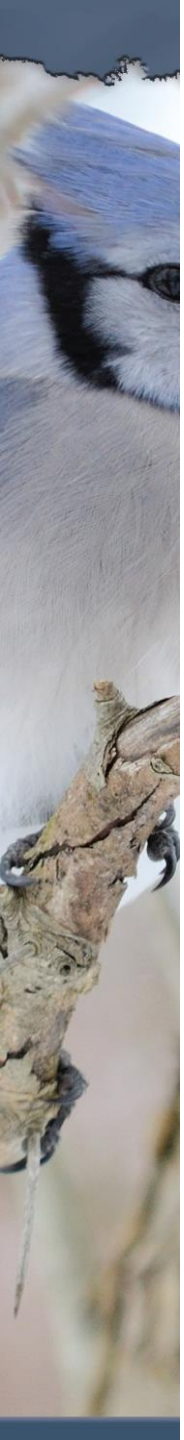
Further abstraction techniques

Abstract classes and interfaces



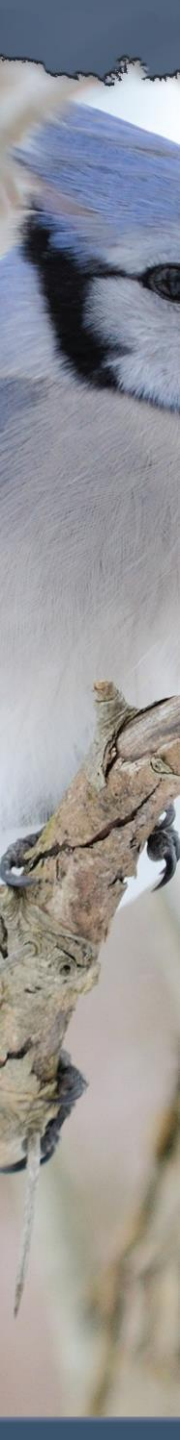
Simulations

- Programs regularly used to simulate real-world activities:
  - city traffic;
  - the weather;
  - nuclear processes;
  - stock market fluctuations;
  - environmental impacts;
  - space flight.



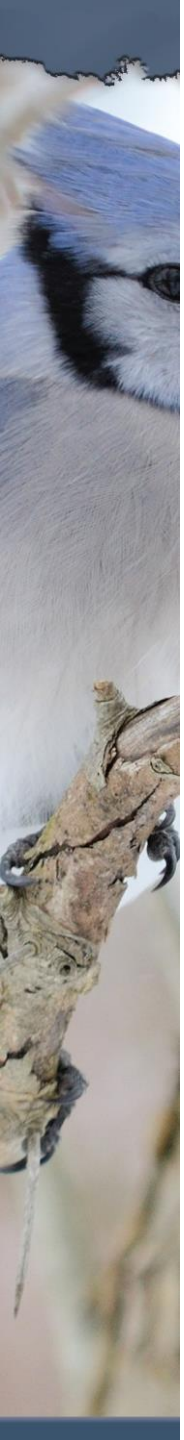
Simulations

- They are often only partial simulations.
- They often involve simplifications.
  - Greater detail has the potential to provide greater accuracy.
  - Greater detail typically requires more resource:
    - Processing power;
    - Simulation time.



Benefits of simulations

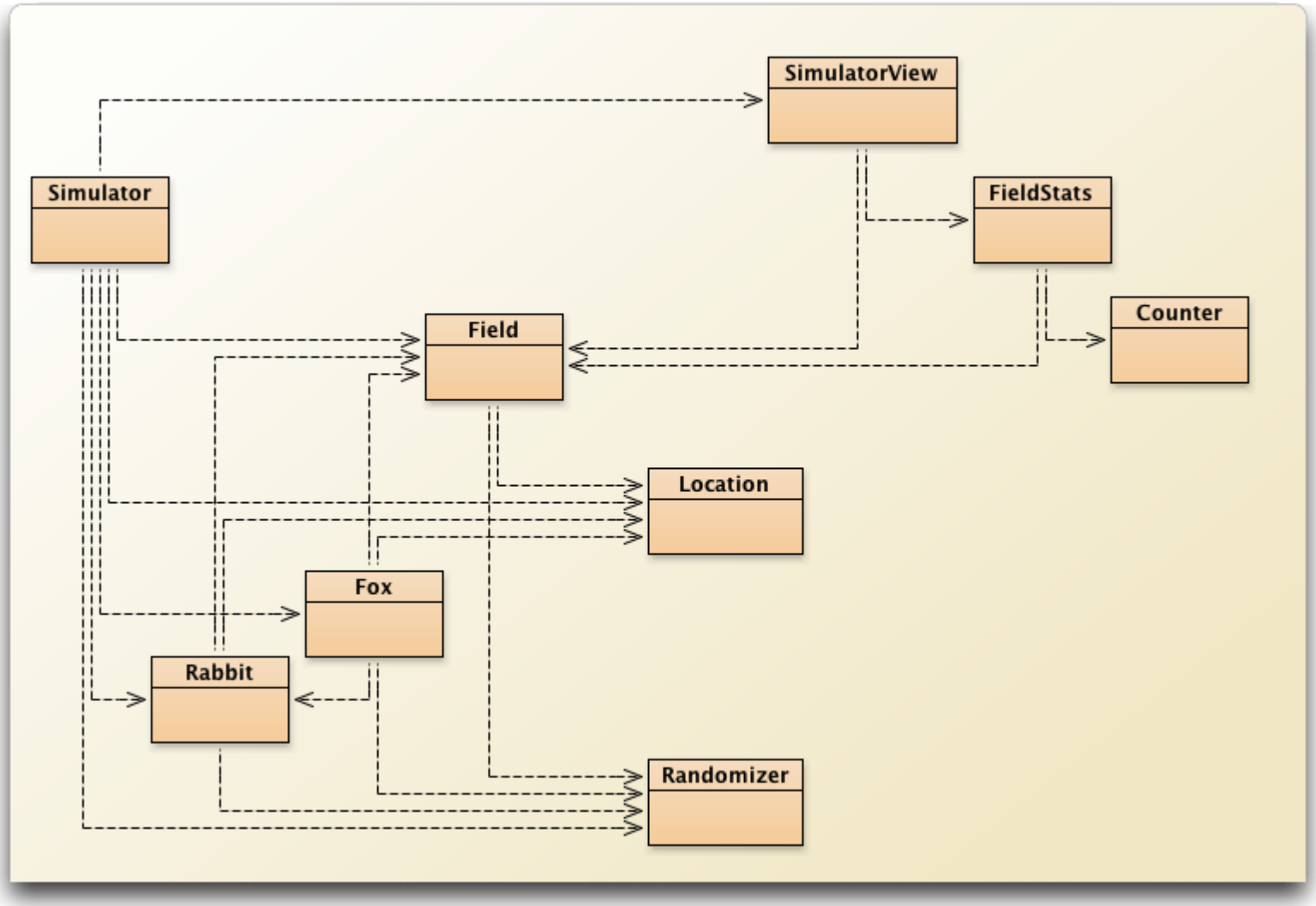
- Support useful prediction.
  - E.g., the weather.
- Allow experimentation.
  - Safer, cheaper, quicker.
- Our example:
  - ‘How will the wildlife be affected if we cut a highway through the middle of this national park?’

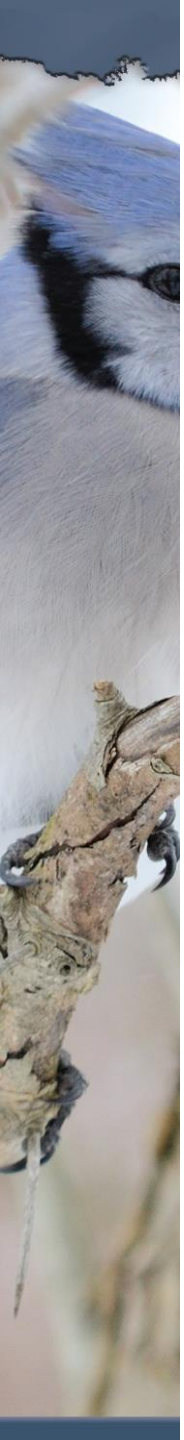


Predator-prey simulations

- There is often a delicate balance between species.
  - A lot of prey means a lot of food.
  - A lot of food encourages higher predator numbers.
  - More predators eat more prey.
  - Less prey means less food.
  - Less food means ...

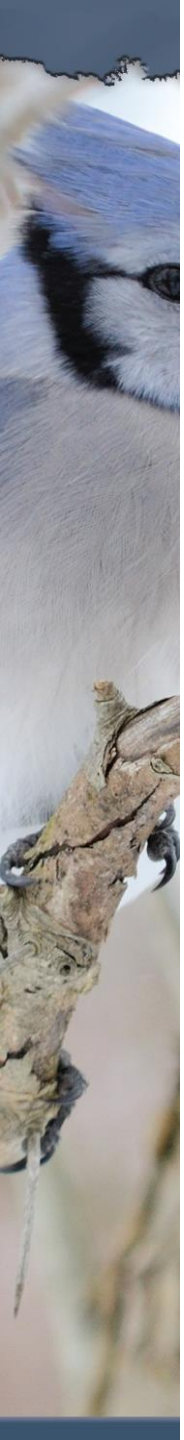
The foxes-and-rabbits project





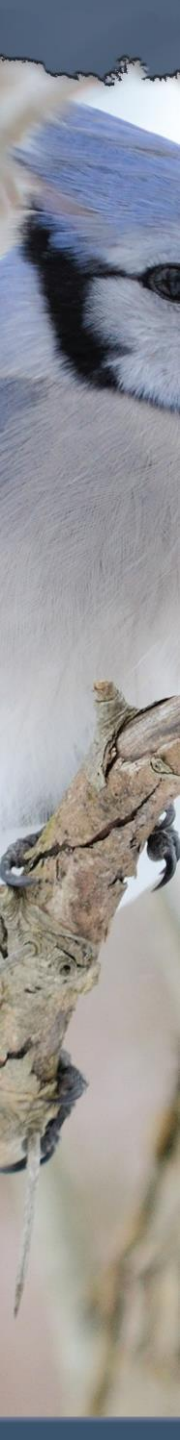
Main classes of interest

- **`Fox`**
  - Simple model of a type of predator.
- **`Rabbit`**
  - Simple model of a type of prey.
- **`Simulator`**
  - Manages the overall simulation task.
  - Holds a collection of foxes and rabbits.



Modeling the environment

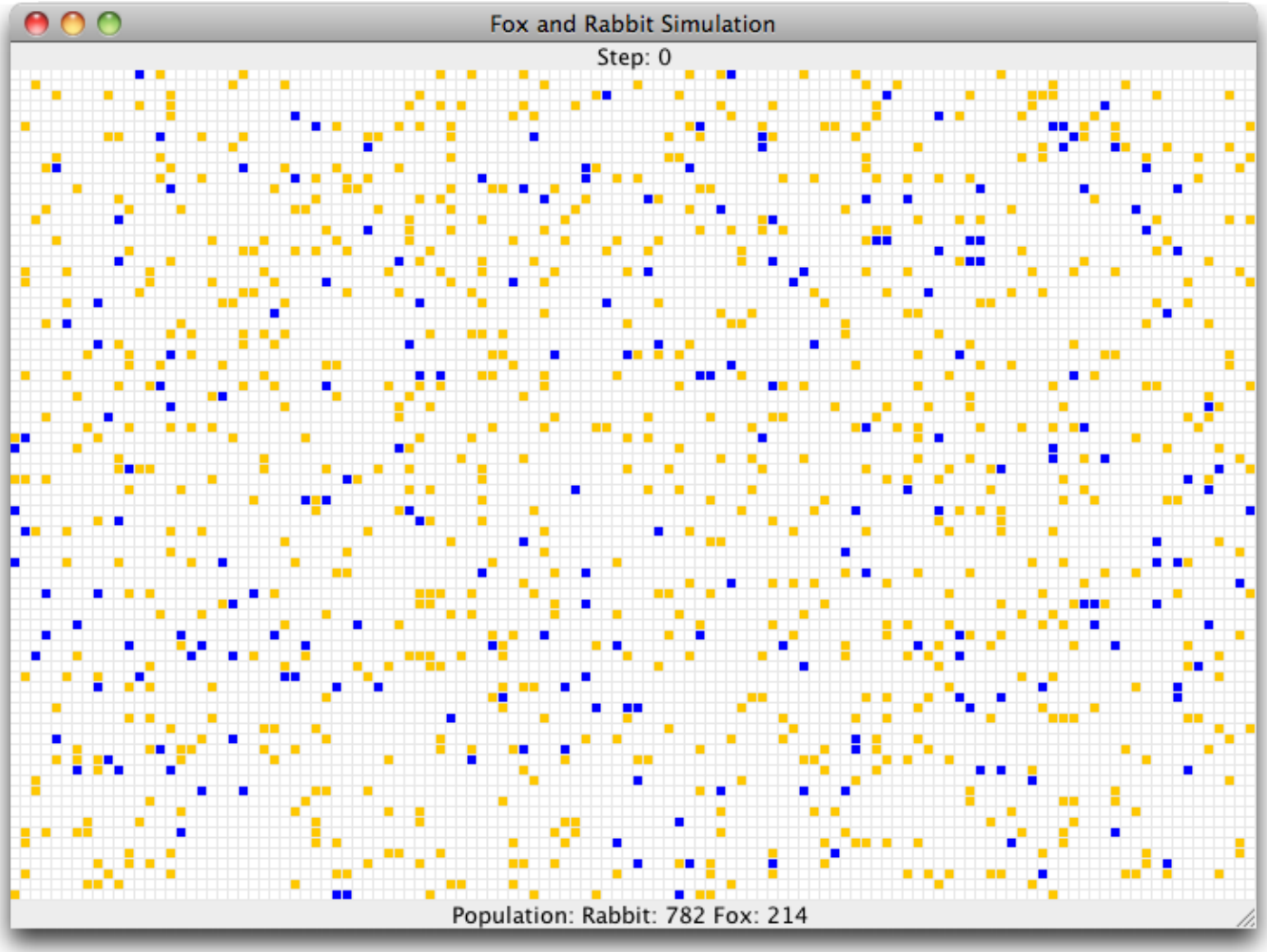
- **`Field`**
  - Represents a 2D field.
- **`Location`**
  - Represents a 2D position in the environment.



Monitoring the simulation

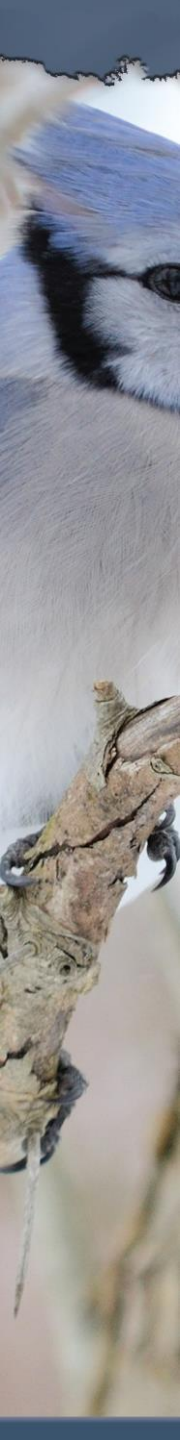
- **`SimulatorView`**
  - Presents a view of the environment.
- **`FieldStats`**, **`Counter`**
  - Maintain statistics.
- **`Randomizer`**
  - Supports reproducibility.

Example of the visualization



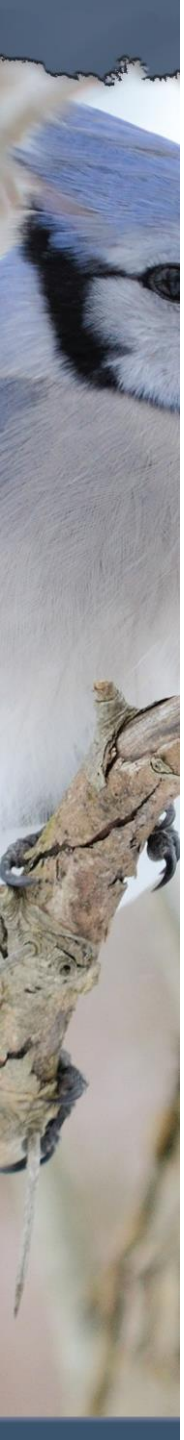
A Rabbit's state

```
public class Rabbit
{
    Static fields omitted.

    // Individual characteristics (instance fields).

    // The rabbit's age.
    private int age;
    // Whether the rabbit is alive or not.
    private boolean alive;
    // The rabbit's position
    private Location location;
    // The field occupied
    private Field field;

    Methods omitted.
}
```



A Rabbit's behavior

- Managed from the **`run`** method.
- Age incremented at each simulation 'step'.
  - A rabbit could die at this point.
- Rabbits that are old enough might breed at each step.
  - New rabbits could be born at this point.

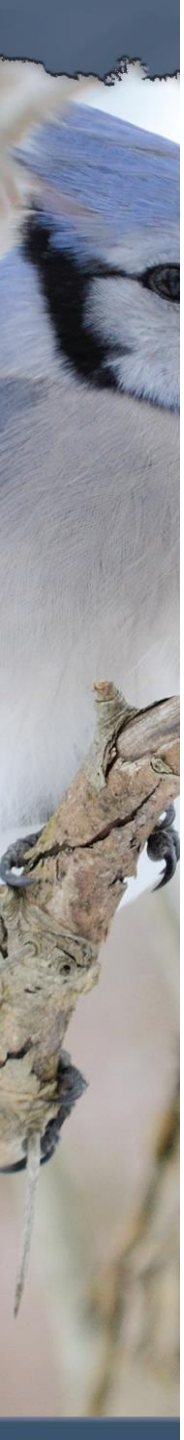


Rabbit simplifications

- Rabbits do not have different genders.
  - In effect, all are female.
- The same rabbit could breed at every step.
- All rabbits die at the same age.
- Others?

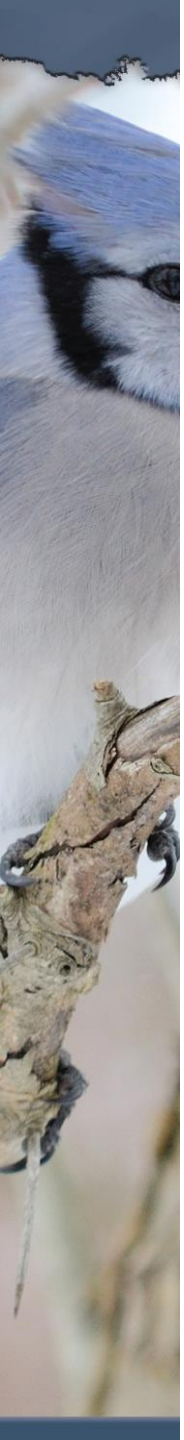
A Fox's state

```
public class Fox
{
    Static fields omitted

    // The fox's age.
    private int age;
    // Whether the fox is alive or not.
    private boolean alive;
    // The fox's position
    private Location location;
    // The field occupied
    private Field field;
    // The fox's food level, which is increased
    // by eating rabbits.
    private int foodLevel;

    Methods omitted.
}
```



A Fox's behavior

- Managed from the **`hunt`** method.
- Foxes also age and breed.
- They become hungry.
- They hunt for food in adjacent locations.

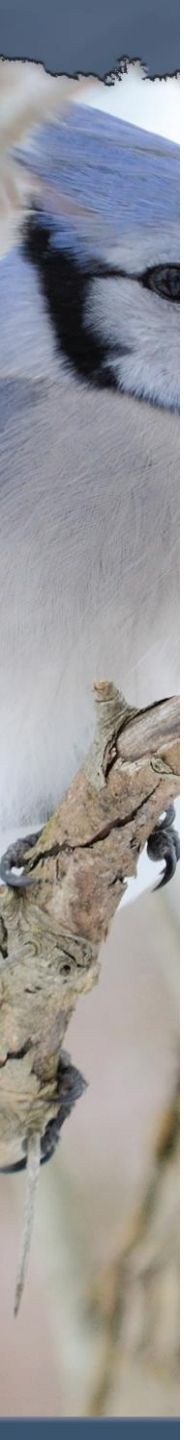


The Simulator class

- Three key components:
  - Setup in the constructor.
  - The **`populate`** method.
    - Each animal is given a random starting age.
  - The **`simulateOneStep`** method.
    - Iterates over separate populations of foxes and rabbits.
    - Two **`Field`** objects are used: **`field`** and **`updatedField`**.

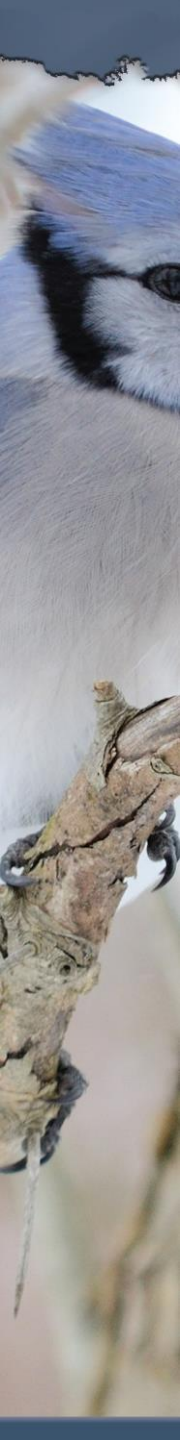
The update step

```
for(Iterator<Rabbit> it = rabbits.iterator();
        it.hasNext(); ) {
    Rabbit rabbit = it.next();
    rabbit.run(newRabbits);
    if(! rabbit.isAlive()) {
        it.remove();
    }
}
…
for(Iterator<Fox> it = foxes.iterator();
        it.hasNext(); ) {
    Fox fox = it.next();
    fox.hunt(newFoxes);
    if(! fox.isAlive()) {
        it.remove();
    }
}
```



Room for improvement

- **`Fox`** and **`Rabbit`** have strong similarities but do not have a common superclass.
- The update step involves similar-looking code.
- The **`Simulator`** is tightly coupled to specific classes.
  - It 'knows' a lot about the behavior of foxes and rabbits.

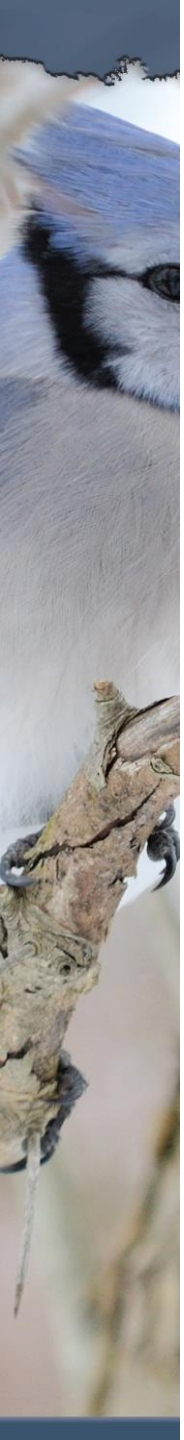


The Animal superclass

- Place common attributes in **`Animal`**:
  - **`age, alive, location`**
- Keep the remaining in subclasses:
  - **`run`** and **`hunt`** stay in **`Fox`** and **`Rabbit`**.

Revised iteration

```
for (Iterator<Animal> it = animals.iterator();
it.hasNext(); ) {
    Animal animal = it.next();
    if (animal instanceof Rabbit) {
        Rabbit rabbit = (Rabbit) animal;
        rabbit.run(newAnimals);
    }
    else if (animal instanceof Fox) {
        Fox fox = (Fox) animal;
        fox.hunt(newAnimals);
    }
    // Remove dead animals from the simulation.
    if (! animal.isAlive())
        it.remove();
}
```

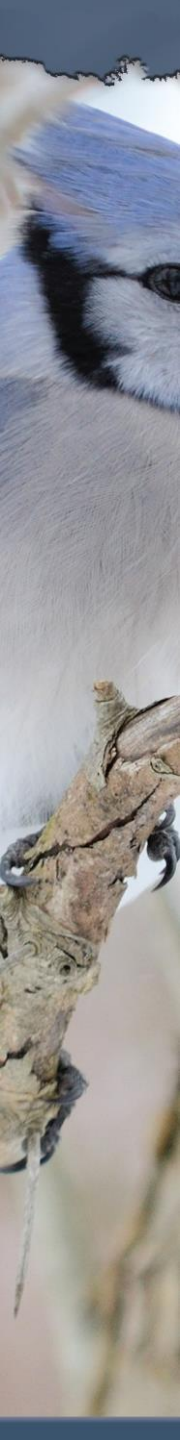


The better Animal superclass

- Method renaming to support information hiding:
  - **`run`** and **`hunt`** become **`act`**.
- **`Simulator`** can now be significantly decoupled.

Revised (decoupled) iteration

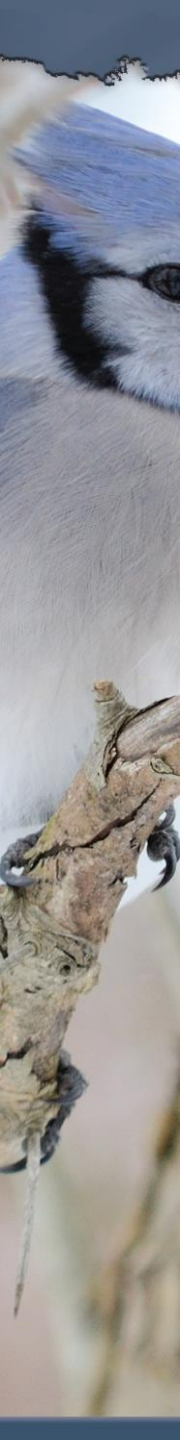
```
for(Iterator<Animal> it = animals.iterator();
        it.hasNext(); ) {
    Animal animal = iter.next();
    animal.act(newAnimals);
    // Remove dead animals from simulation
    if(! animal.isAlive()) {
        it.remove();
    }
}
```



The `act` method of Animal

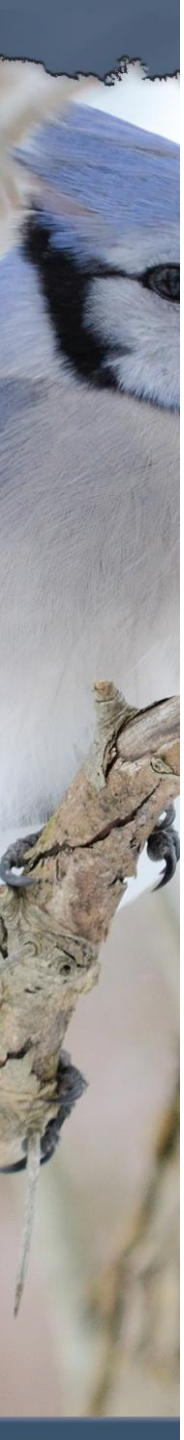
- Static type checking requires an **`act`** method in **`Animal`**.
- There is no obvious shared implementation.
- Define **`act`** as abstract:

```
abstract public void act(List<Animal> newAnimals);
```



Abstract classes and methods

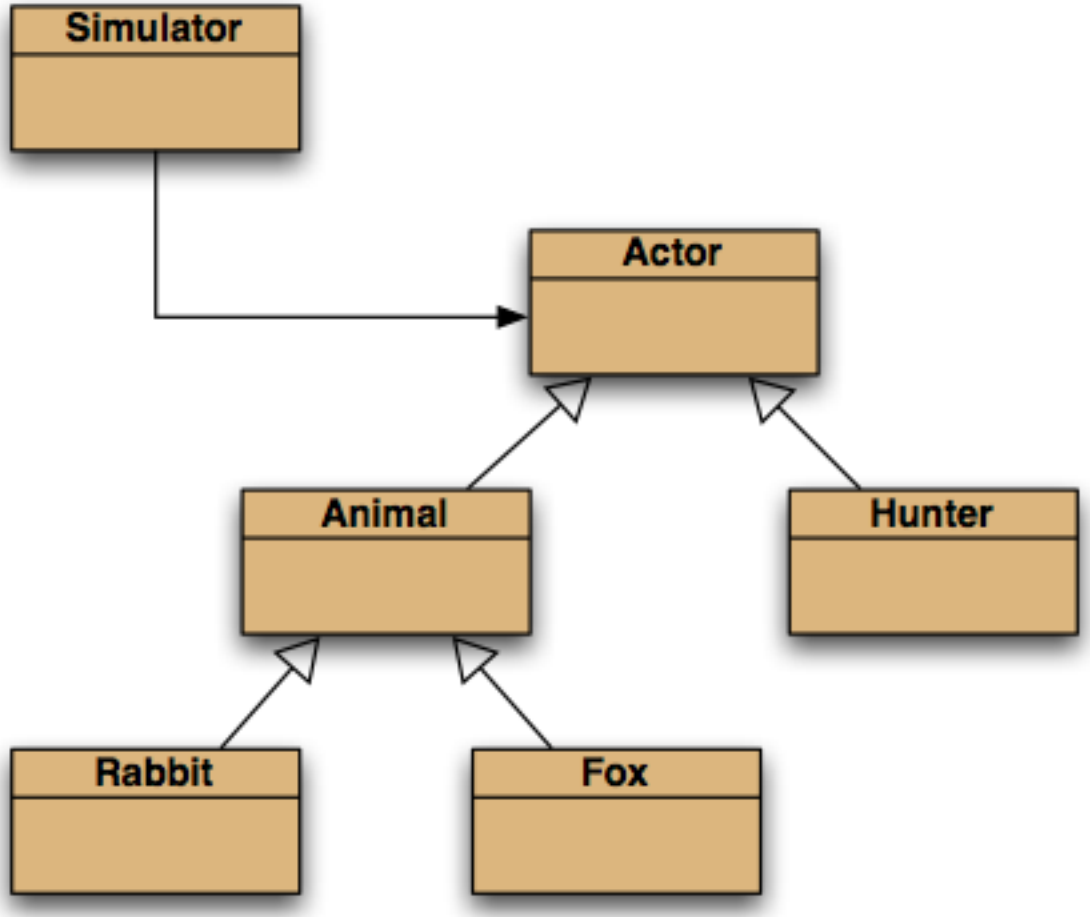
- Abstract methods have **`abstract`** in the signature.
- Abstract methods have no body.
- Abstract methods make the class abstract.
- *Abstract classes cannot be instantiated.*
- Concrete subclasses complete the implementation.



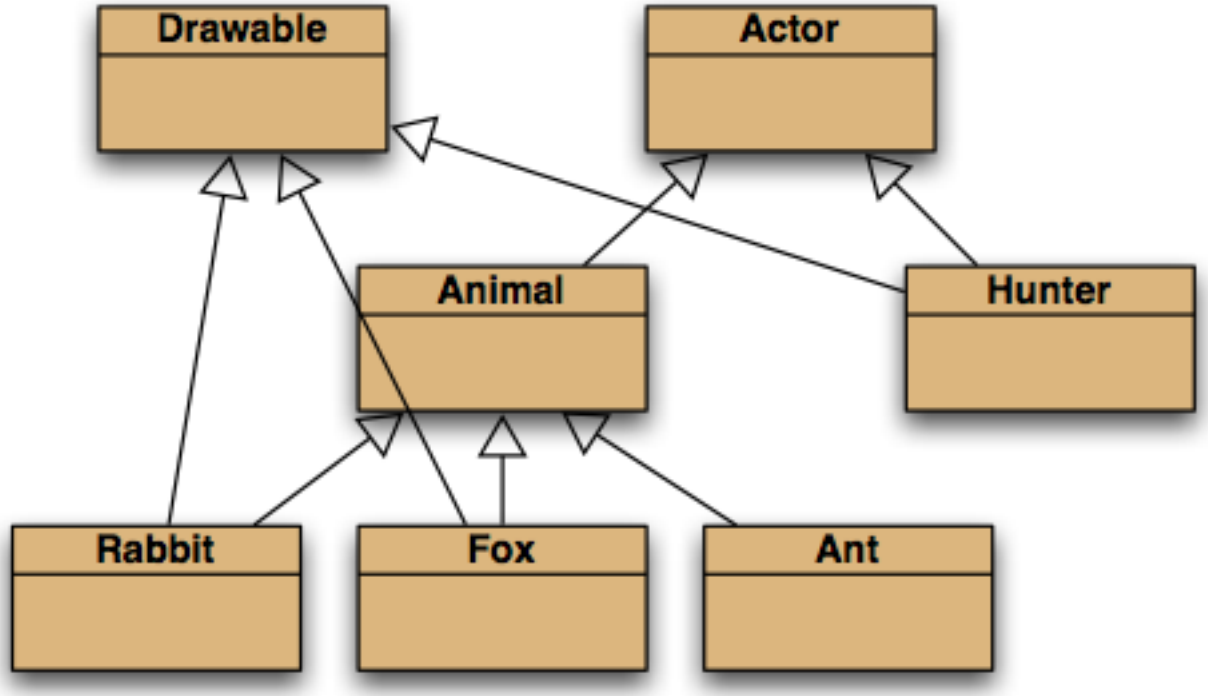
The Animal class

```
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

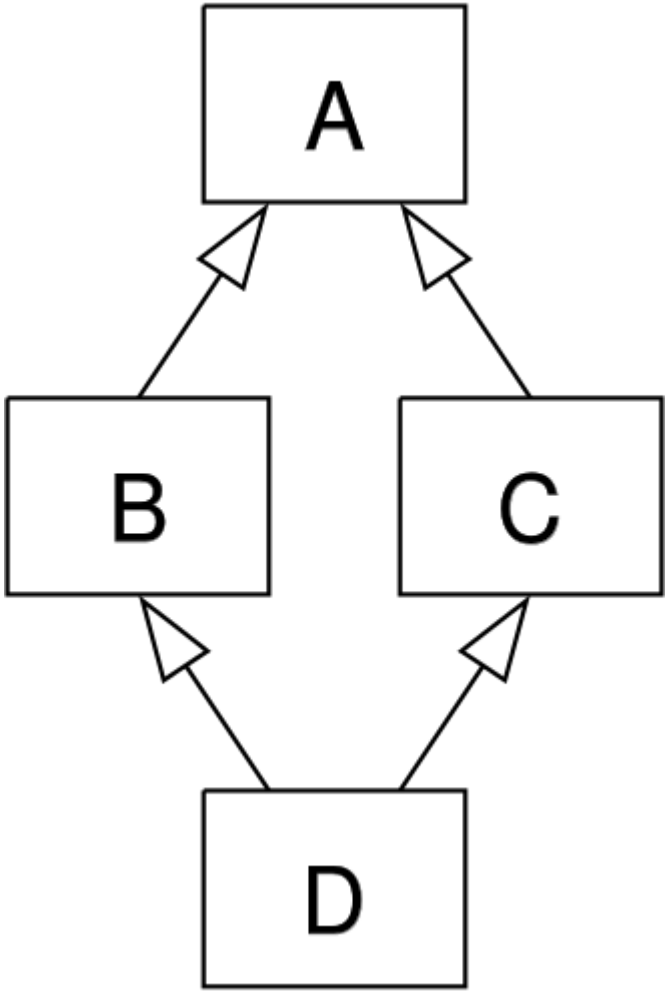


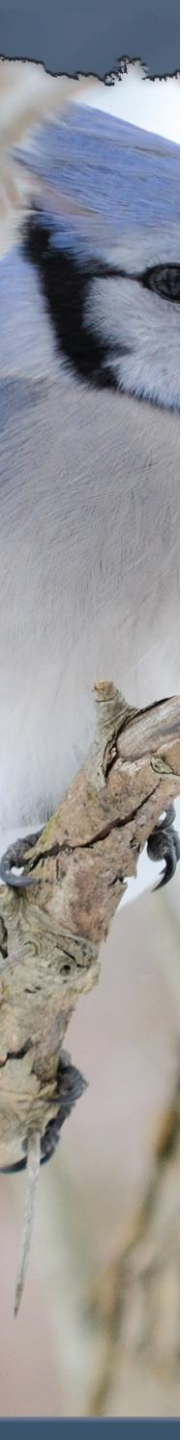
Selective drawing (multiple inheritance)



Multiple Inheritance (diamond problem)

- If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?



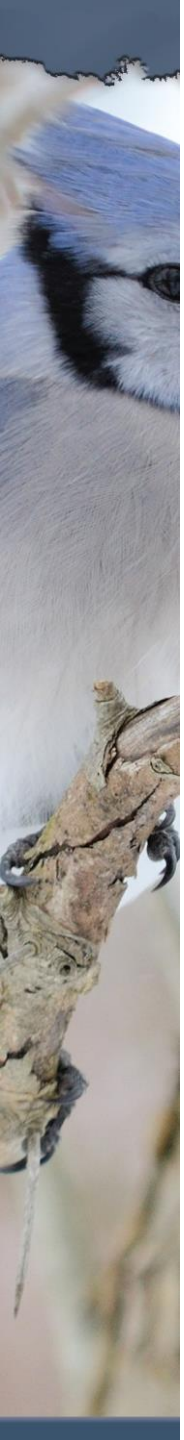


Multiple inheritance

- 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.

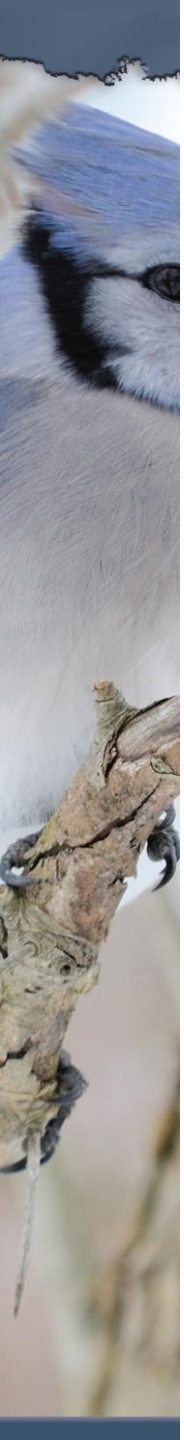
An Actor interface

```
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();
}
```



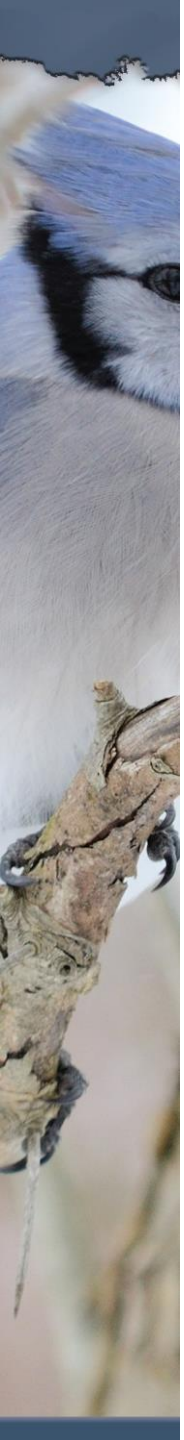
Classes *implement* an interface

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

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



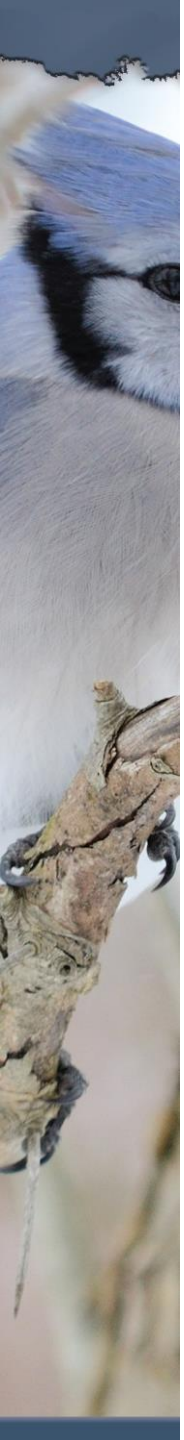
Interfaces as types

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



Features of interfaces

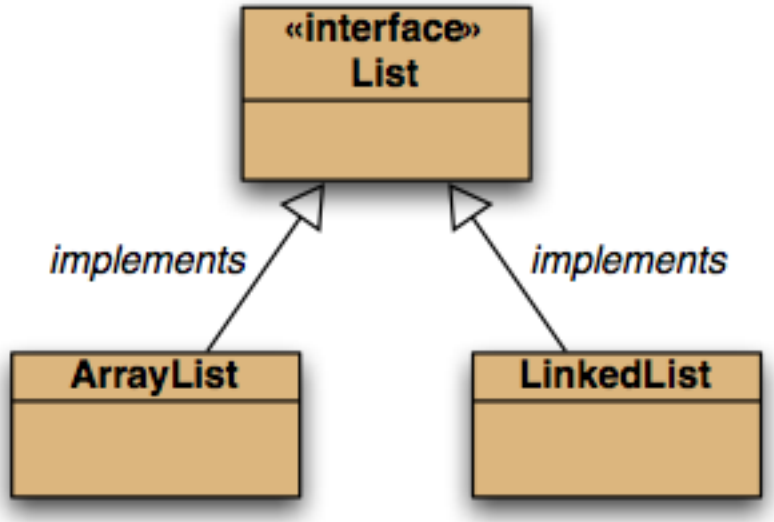
- Use **`interface`** rather than **`class`** in their declaration.
- They do not define constructors.
- All methods are **`public`**.
- All fields are **`public`**, **`static`** and **`final`**. (Those keywords may be omitted.)
- Abstract methods may omit **`abstract`**.

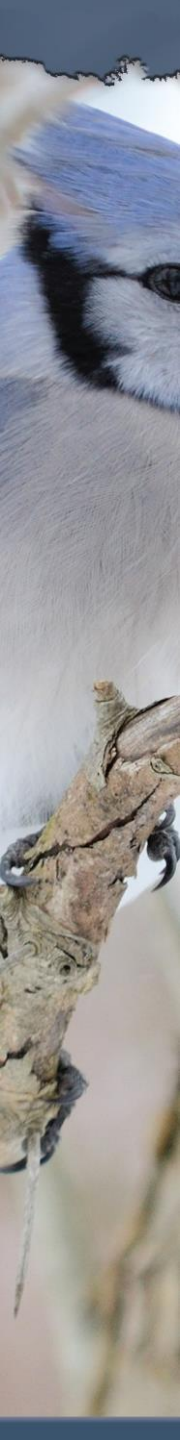


Interfaces as specifications

- 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.

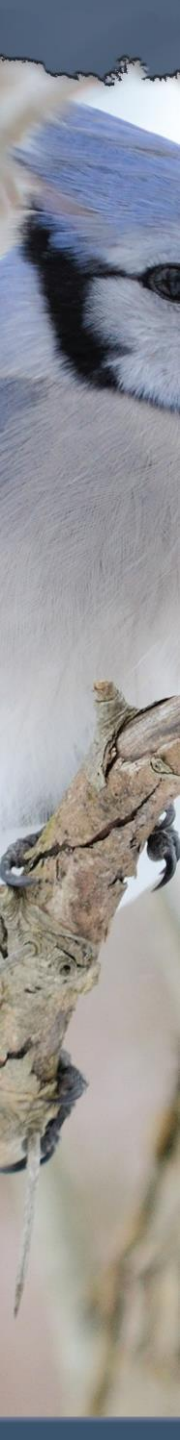
Alternative implementations





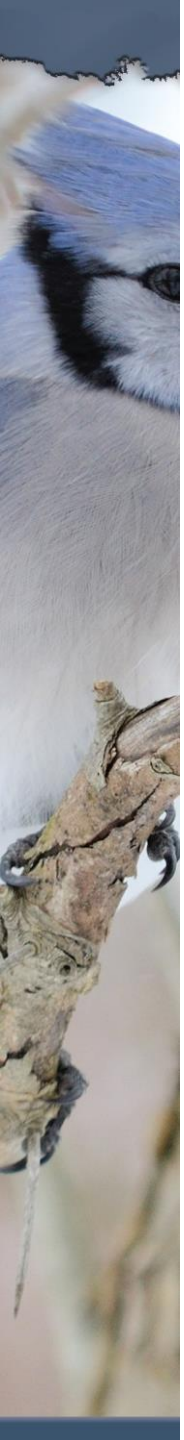
Review

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



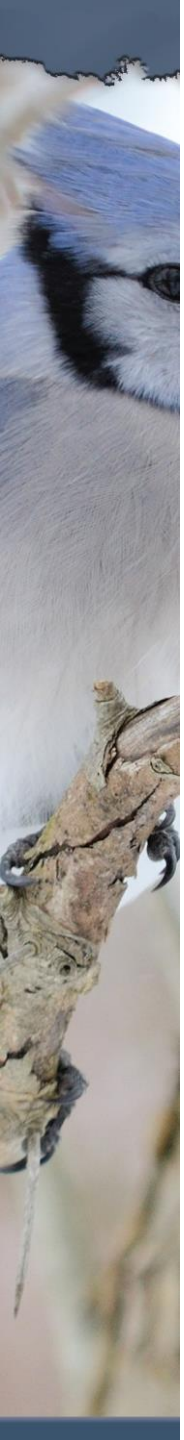
Review

- 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.



شعر امروز

چشم در راه کسی هستم
کوله بارش بر دوش
آفتابش در دست
خنده بر لب ، گل به دامن ، پیروز
کوله بارش سرشار از عشق ، امید
آفتابش نوروز
با سلامش ، شادی
در کلامش ، لبخند
از نفس‌هایش گُل می‌بارد
با قدم‌هایش گُل می‌کارد
مهربان ، زیبا ، دوست
روح هستی با اوست
قصه ساده ست ، معما مشمار
چشم در راه بهارم آری
چشم در راهِ بهار ...!

فریدون مشیری