Lecture 10
Chapter 10: Further abstraction techniques

Abstract classes and interfaces

Main concepts to be covered

- Abstract classes
  - Classes we can’t instantiate
  - Used only for inheritance
- Multiple inheritance
  - Inheriting from more than 1 place
- Interfaces
  - Like abstract classes, but without code
  - Allow multiple inheritance in Java

Simulations

- Programs regularly used to simulate real-world activities.
  - city traffic
  - the weather
  - nuclear processes
  - stock market fluctuations
  - environmental changes

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.

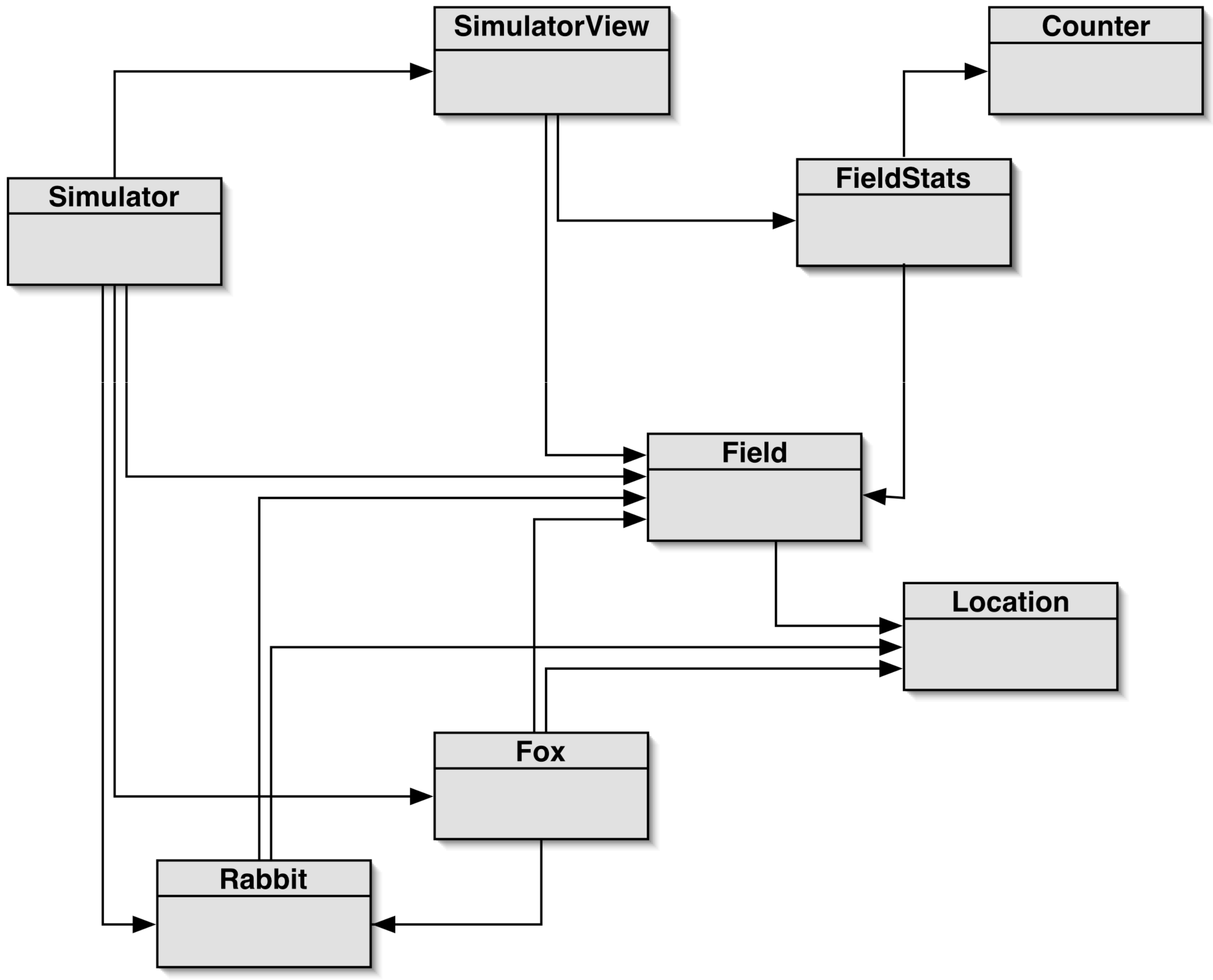
Benefits of simulations

- Support useful prediction.
  - The weather.
- Allow experimentation.
  - Safer, cheaper, quicker.
- 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



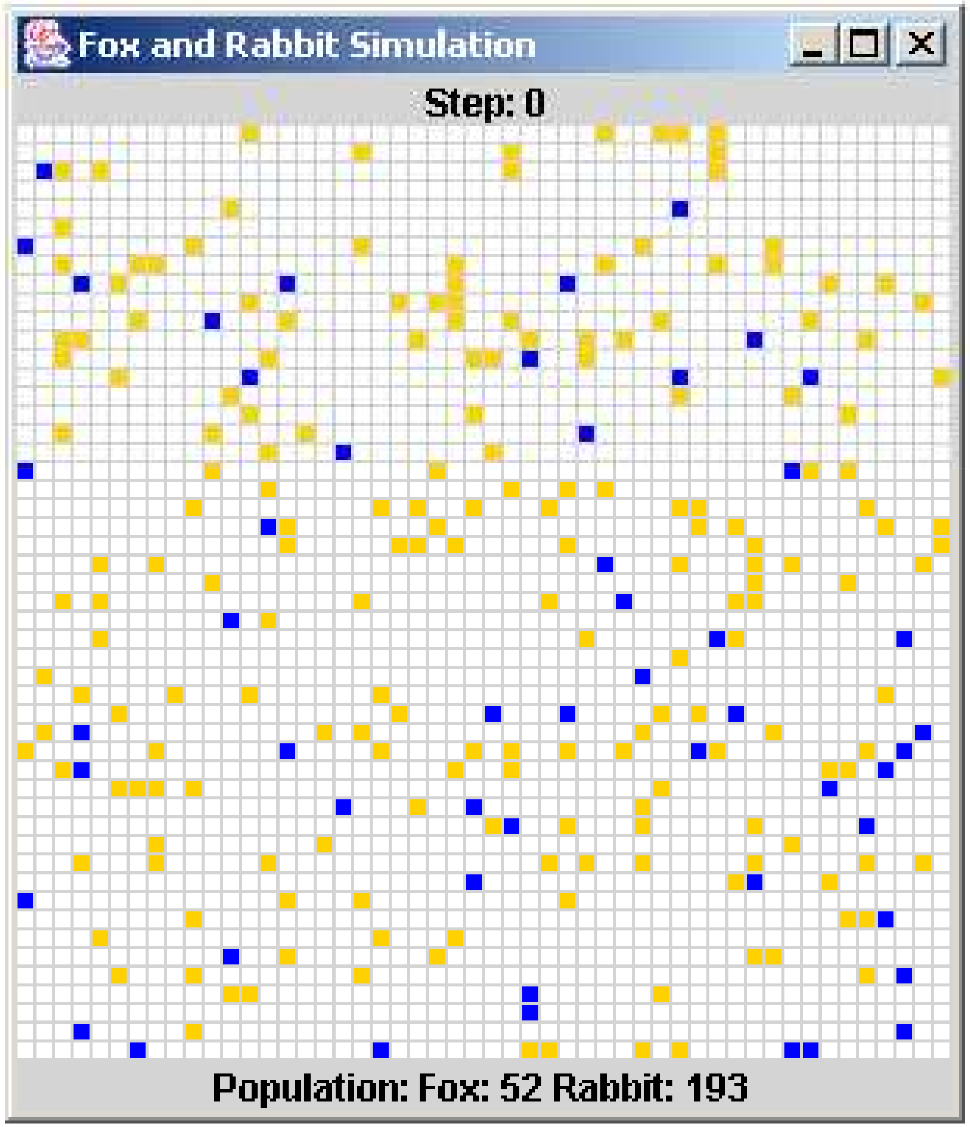
Main classes of interest

- `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.

The remaining classes

- `Field`
  - Represents a 2D field.
- `Location`
  - Represents a 2D position.
- `SimulatorView`, `FieldStats`, `Counter`
  - Maintain statistics and present a view of the field.

Example of the visualization



Step: 0

Population: Fox: 52 Rabbit: 193

A Rabbit’s state

```
public class Rabbit
{
    Static fields omitted.

    // Individual characteristics.

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

    Method omitted.
}
```

A Rabbit’s behavior

- Managed by `Rabbit.run()`
- On each time step:
  - Age incremented
  - Rabbits can die of old age
  - Rabbits that are old enough might breed
  - New rabbits can be born

Rabbit simplifications

- Rabbits do not have different genders.
- 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 fox's food level, which is increased
    // by eating rabbits.
    private int foodLevel;

    Methods omitted.
}
```

A Fox’s behavior

- Managed by `Fox.hunt()`
- Foxes also age and breed.
- They become hungry.
- They hunt for food in adjacent locations.

Configuration of foxes

- Similar simplifications to rabbits.
- Hunting and eating could be modeled in many different ways.
  - Should food level be additive?
  - Is a hungry fox more or less likely to hunt?
- Are simplifications ever acceptable?

The Simulator class

- Three key components:
  - Constructor creates field etc. objects
  - The `populate` method.
    - Each animal is given a random starting age.
  - The `simulateOneStep` method.
    - Initial version not very good
    - Iterates over the population.
    - Checks type of each object and handles them differently
    - Two `Field` objects are used: `field` and `updatedField`.

The update step

```
for(Iterator iter = animals.iterator(); iter.hasNext(); ) {
      Object animal = iter.next();

      if(animal instanceof Rabbit) {
            Rabbit rabbit = (Rabbit)animal;
            rabbit.run(updatedField, newAnimals);
      }
      else if(animal instanceof Fox) {
            Fox fox = (Fox)animal;
            fox.hunt(field, updatedField, newAnimals);
      }
}
```

- Foxes and rabbits are identified and handled differently in Simulator class

Room for improvement

- `Fox` and `Rabbit` have strong similarities but do not have a common superclass.
- The `Simulator` is tightly coupled to specific classes.
  - It ‘knows’ a lot about the behavior of foxes and rabbits.
- A better way is to make an Animal superclass

The Animal superclass

- Place common fields in `Animal`:
  - `age`, `alive`, `location`
- Method renaming to support information hiding:
  - `run` and `hunt` become `act`.
- `Simulator` can now be significantly decoupled.

Revised (decoupled) iteration

```
for(Iterator<Animal> iter = animals.iterator();
                            iter.hasNext(); ) {
    Animal animal = iter.next();
    animal.act(field, updatedField, newAnimals);
}
```

- Note use of Iterator object in for loop.
- While loop would be just as good.

The act method of Animal

- Static type checking requires an `act` method in `Animal`.
- There is no obvious shared implementation.
- We could implement an empty method:

```
public void act(Field currentField,
                         Field updatedField,
                         List newAnimals)
   {};
```

- But there’s no guarantee subclass will override it.

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.
- In order to instantiate, subclass must override with concrete (non-abstract) version.
- Abstract version of `act`:

```
abstract public void act(Field currentField,
                         Field updatedField,
                         List newAnimals);
```

The Animal class

```
public abstract class Animal {
  fields omitted

  /**
   * Make this animal act - make it do

   * whatever it wants/needs to do.*/

  abstract public void act(Field currentField,
                           Field updatedField,
                            List newAnimals);
    other methods omitted
}
```

Why use Abstract Classes?

Same reasons as other superclasses:

- Provide code and data to inherit (promote code reuse, avoid code duplication)
- Allow polymorphism (treat related types as same)

Extra reason:

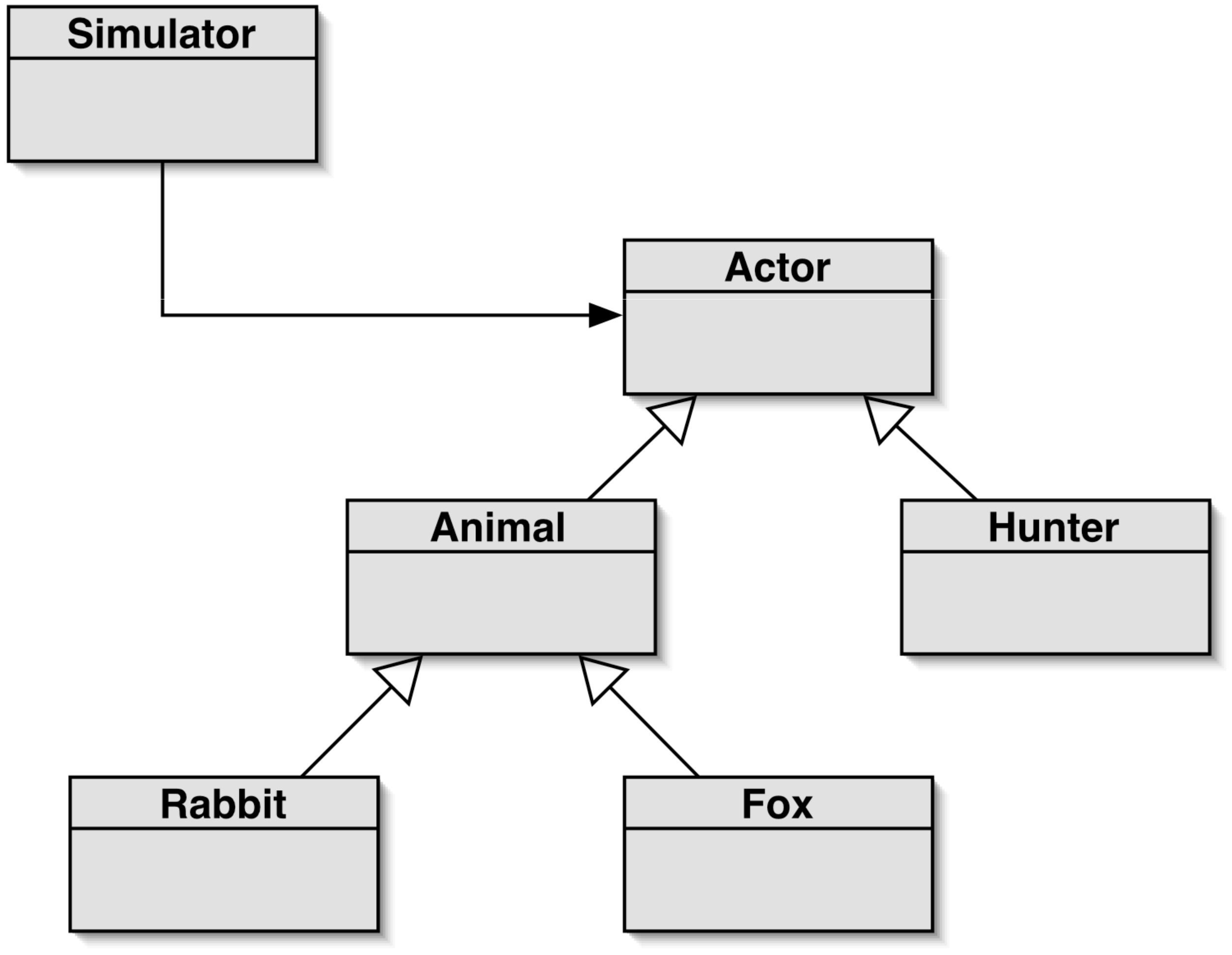
- Force/guarantee all subclasses to implement some functionality

Simulating overriding fields

- We want to move canBreed() into Animal
- We want each class to set its own static BREEDING_AGE field value
- Problem: you can’t override fields
- Solution:
  - use abstract getBreedingAge() in Animal
  - override in subclasses
  - add a BREEDING_AGE to each subclass

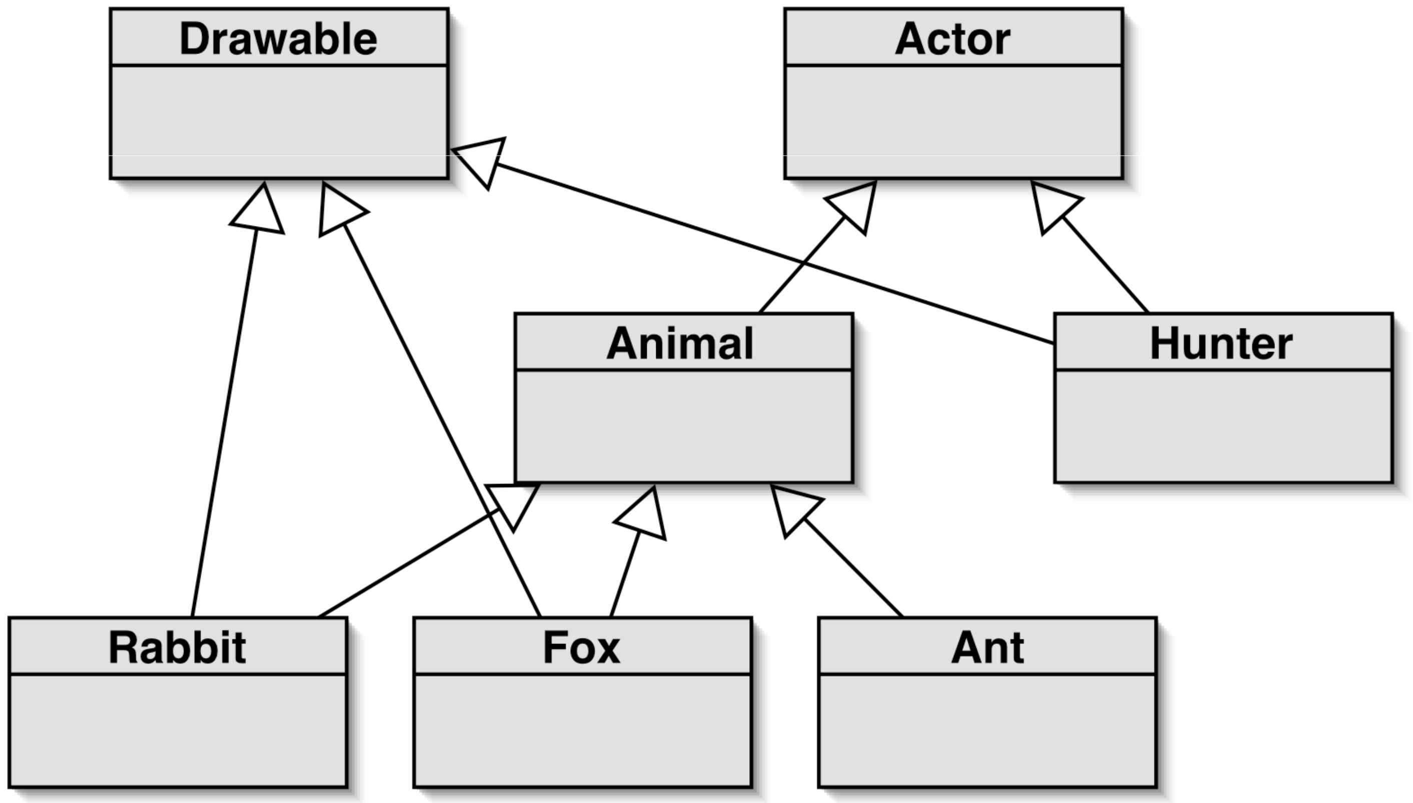
Further abstraction

- Let's add hunters and other actors
- Note abstract Animal class is part of hierarchy, like any other class



Selective drawing (multiple inheritance)

Suppose we only want to draw certain classes on screen



Multiple inheritance

- A class inheriting directly from multiple superclasses.
- Problem when 2 superclasses implement same method.
  - Which implementation to use?
  - Each language has its own rules.
- Java forbids multiple inheritance for classes.
- Java permits it for interfaces.
  - Have type, static fields, and method signatures only.
  - Hence no competing method implementations.

An Actor interface

```
public interface Actor
{
    /**
     * Perform the actor's daily behavior.
     * Transfer the actor to updatedField if it is
     * to participate in further steps of the simulation.
     * @param currentField The current state of the field.
     * @param location The actor's location in the field.
     * @param updatedField The updated state of the field.
     */
    void act(Field currentField, Location location,
             Field updatedField);
}
```

Classes implement an interface

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

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

Drawable and Actor are both made interfaces since they Implement no code

Interfaces

- Use keyword ‘interface’ instead of ‘class’
- All methods are abstract and public
- No constructors
- All fields are public, static and final
- Keywords abstract, public, static and final not needed for methods or fields
- Like abstract classes, they guarantee implementation

Interfaces as types

Implementing classes

- do not inherit code, but ...
- ... are subtypes of the interface type.

So, polymorphism is available with interfaces as well as classes.

Examples of Polymorphism

We can put any type of Drawable in a Drawable field:

```
Drawable d1;
d1 = new Rabbit();
d1 = new Fox();
```

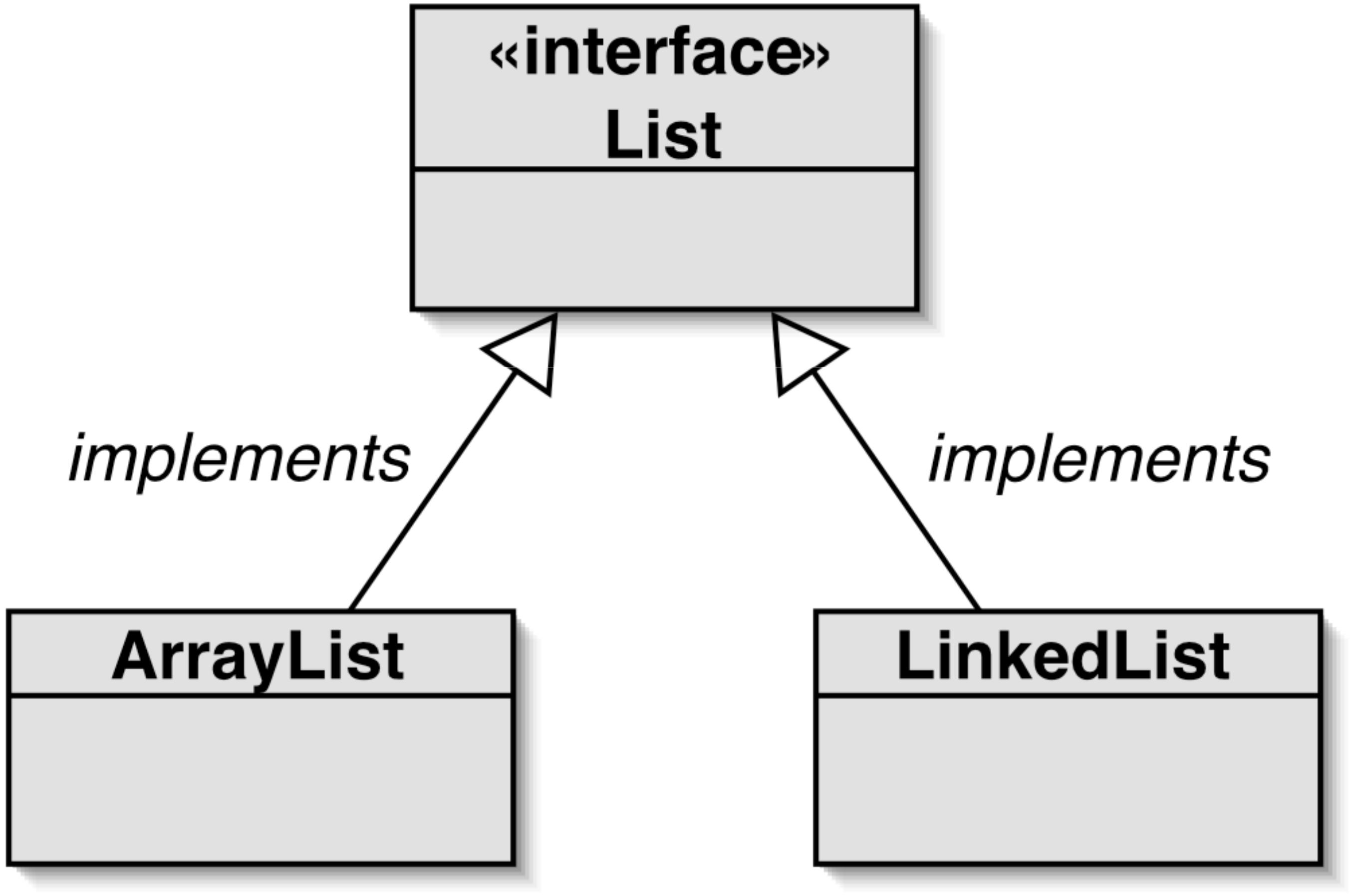
We can iterate over a collection of Drawables, and ignore their static type:

```
ArrayList<Drawable> drawables =
     new ArrayList<Drawable>();
Iterator<Drawable> iter = drawables.iterator();
while (iter.hasNext())
  {
     Drawable item = iter.next();
     item.draw();
  }
```

Interfaces as specifications

- Completely separate functionality from implementation.
  - Though parameter and return types are specified.
- Clients (classes that use interface) see only functionality.
  - But clients can choose from alternative implementations.
  - E.g. 2 types of List

Alternative implementations



Same Functionality, Different Implementations

- Random access to middle is much faster with ArrayList
- Insert/delete can be much faster with LinkedList
- Which should we use?
- Try both: only need to change 1 line:

```
private List myList = new xxx;
```

Review: inheritance

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

Review: Abstract Classes

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

Review: Interfaces

- Interfaces provide specification without implementation.
  - Interfaces are fully abstract.
- Interfaces support polymorphism.
- Java interfaces support limited multiple inheritance:
  - Type and constant fields only
  - No code

Should I use an Abstract Class or Interface?

Want to inherit code?

- Yes: use abstract class
- No: interface usually better, since they don’t prevent implementor from extending another class