9 Further abstraction techniques

Abstract classes and interfaces

Main concepts to be covered

- Abstract classes
- Interfaces
- Multiple inheritance

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.

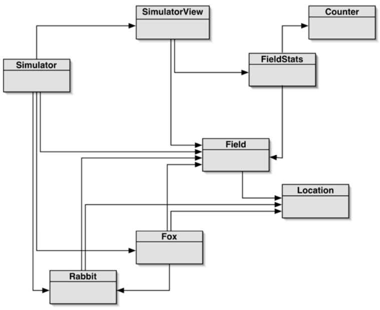
Benefits of simulations

- 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



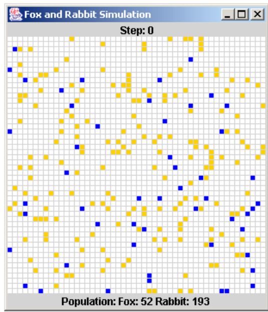
Main classes of interest

- `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



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;

    Method 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 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.

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:
  - 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

```
Iterator<Rabbit> it = rabbits.iterator();
while ( it.hasNext() ) {
    Rabbit rabbit = it.next();
    rabbit.run(updatedField, newRabbits);
    if(! rabbit.isAlive()) {
        it.remove();
    }
}
...
Iterator<Fox> it = foxes.iterator();
while ( it.hasNext() ) {
    Fox fox = it.next();
    fox.hunt(field, updatedField, 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.
- *Refactor!*

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

```
Iterator<Animal> it = animals.iterator();
while ( it.hasNext() ) {
    Animal animal = it.next();

    animal.act(field, updatedField, newAnimals);

    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(Field currentField,
                         Field updatedField,
                         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.

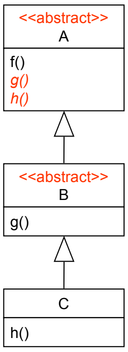
The Animal class

```
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(Field currentField,
                             Field updatedField,
                             List<Animal> newAnimals);

    other methods omitted
}
```

Ex. Abstract and Concrete classes

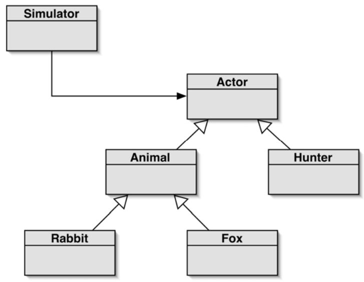



```
public abstract class A {
    public void f() { ... }
    public abstract int g();
    public abstract boolean h(float x);
}
public abstract class B extends A {
    public int g() { ... }
}
public class C extends B {
    public boolean h(float x) { ... }
}
```

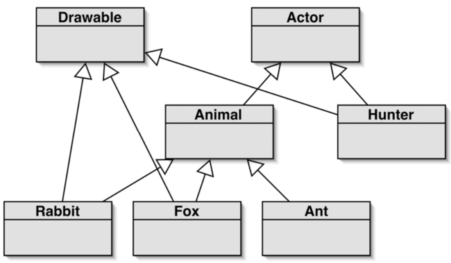
```
A a = new A();
B b = new B();     NOT ALLOWED! A and B are abstract

C c = new C();
c.f();                        // from A
int x = c.g();                // from B
boolean flag = c.h(3.14);     // from C
```


Further abstraction



Multiple inheritance



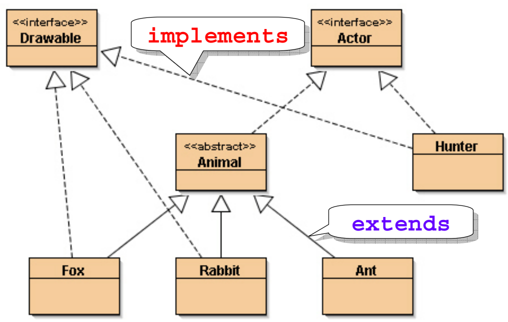
Multiple inheritance

- 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.
  - No competing implementation.

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 updatedField The updated state of the field.
     * @param newActors New actors created as a result
     *                  of this actor's actions.
     */
    void act(Field currentField, Field updatedField,
             List<Actor> newActors);
}
```


Classes implement an interface



Classes implement an interface (2)

```
public interface Actor { ... }
public interface Drawable { ... }

public abstract class Animal implements Actor
{ ... }

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

public class Rabbit extends Animal implements Drawable
{ ... }

public class Ant extends Animal
{ ... }

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


Interfaces as types

- Implementing classes do not inherit code, but ...
- ... implementing classes are subtypes of the interface type.
- So, polymorphism is available with interfaces as well as classes.

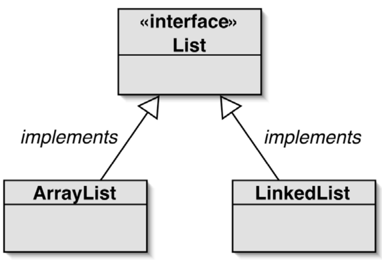
Features of interfaces

- All methods are abstract.
- There are no constructors.
- All methods are public.
- All fields are public, static and final.
  - So they are public constants.

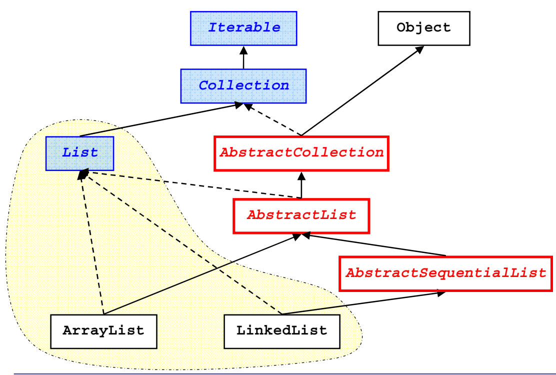
Interfaces as specifications

- 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.

Alternative implementations



Lists – the (nearly) whole truth



Ex. Some legal combinations

```
Map<String,List<Integer>> m;
m = new HashMap<String,List<Integer>>();
```

OK!
`HashMap<String,List<Integer>>` **is** a subtype of `Map<String,List<Integer>>`

```
m.put("first",new ArrayList<Integer>());
m.put("second",new LinkedList<Integer>());
```

OK!
`ArrayList` and `LinkedList` **are** subtypes of `List`

Ex. Some ILLEGAL combinations

```
Map<String,List<Integer>> m;
m = new HashMap<String,ArrayList<Integer>>();
```

Wrong!
`HashMap<String,ArrayList<Integer>>`
is not a subtype of
`Map<String,List<Integer>>`
or of
`HashMap<String,List<Integer>>`
(the same applies to `LinkedList`)

Review (1)

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

Review (2)

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

Review (3)

- Interfaces provide specification without implementation.
  - Interfaces are fully abstract.
- Interfaces support polymorphism.
- Java interfaces support multiple inheritance.