Más técnicas de Abstracción

Clases abstractas e interfaces

Principales conceptos a ser cubiertos

- Clases abstractas
- Interfaces
- Herencia múltiple

Simulaciones

- Frecuentemente, se desarrollan y utilizan programas para simular actividades de la vida real:
  - tráfico
  - clima
  - procesos nucleares
  - análisis de stock
  - cambios ambientales

Simulaciones

- En general, son sólo simulaciones parciales
- Las simulaciones involucran simplificaciones.
  - Mayor detalle en la simulación ofrece mayor precisión en los resultados.
  - Pero también consume más recursos
    - Capacidad de procesamiento
    - Tiempo de simulación.

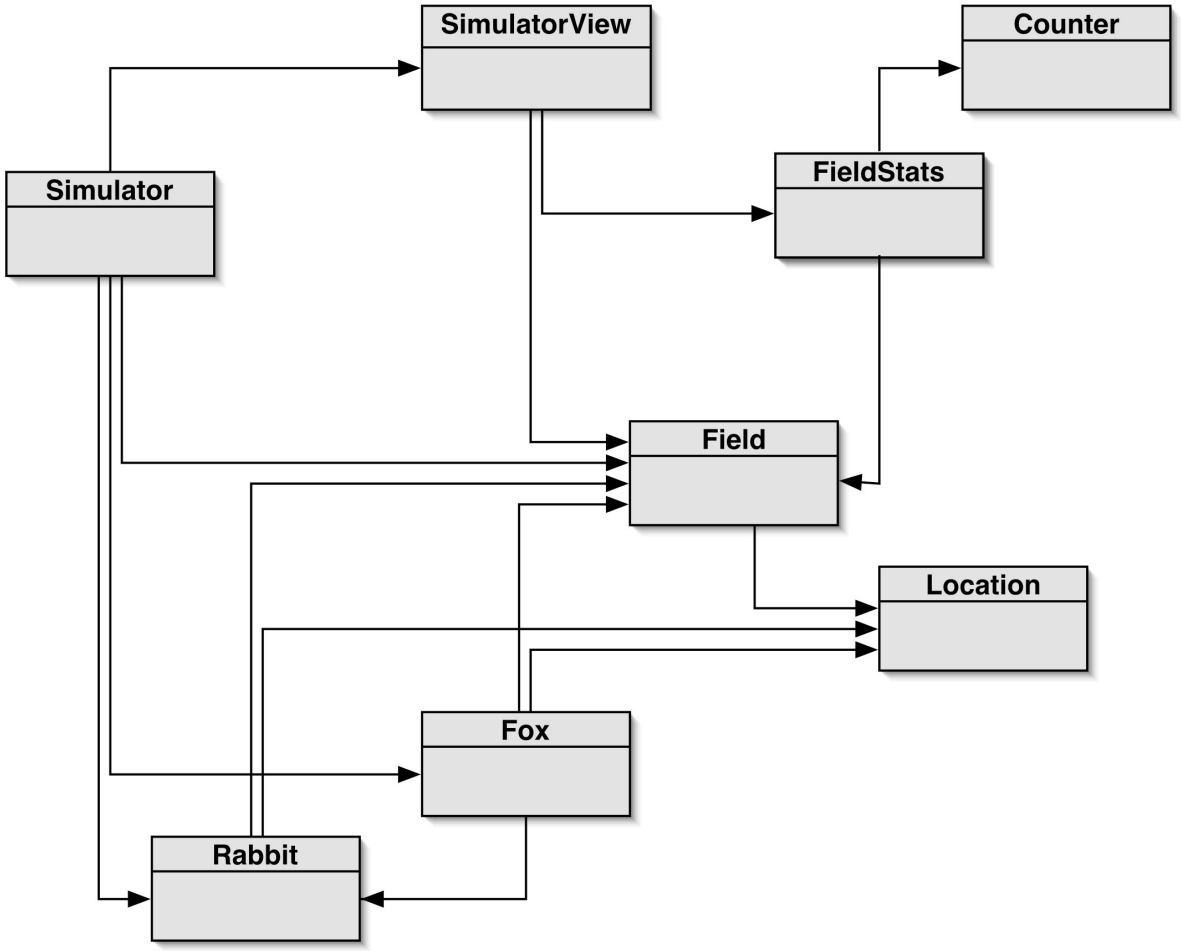
Beneficios de las simulaciones

- Permiten realizar predicciones.
  - El pronóstico meterológico.
- Permiten la experimentación
  - Seguro, rápido, bajo costo.
- Ejemplo:
  - ¿Cómo afectará a la vida salvaje si atravesamos un parque nacional con una autopista?

Simulaciones *predador-presa*

- Existe a menudo un delicado balance entre especies:
  - Muchas presas implican mucha comida.
  - Mucha comida favorece la existencia de muchos predadores.
  - Muchos predadores comen muchas presas.
  - Menos presas significa menos comida
  - Menos comida significa ...

El proyecto zorros y conejos



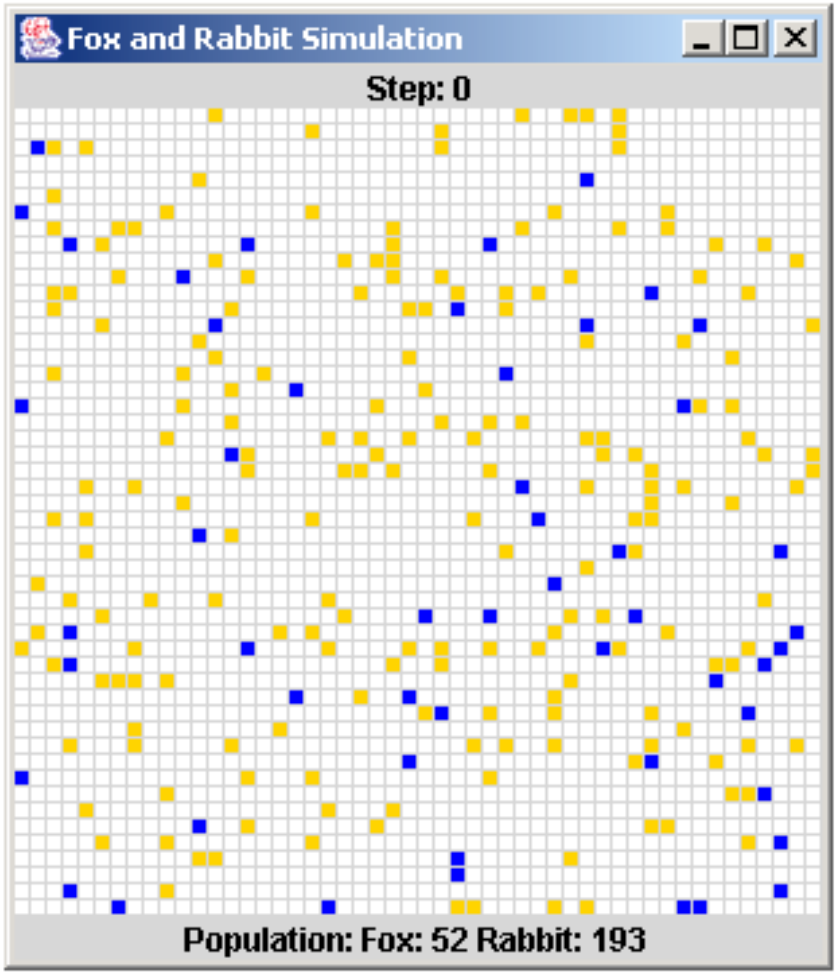
Clases principales

- **`Zorro`**
  - Modelo sencillo de un tipo de predador.
- **`Conejo`**
  - Modelo sencillo de un tipo de presa.
- **`Simulador`**
  - Lleva a cabo la tarea de simulación.
  - Contiene una colección de conejos y zorros.

Las clases restantes

- **`Campo`**
  - Representa un terreno 2D
- **`Ubicacion`**
  - Representa una posición en 2D.
- **`VisorDeSimulador`, `EstadisticasdelCampo`, `Contador`**
  - Llevan estadísticas y presentan una vista del Campo

Ejemplo de visualización



Estado del Conejo

```
public class Conejo
{
    Se omitieron los campos estáticos.

    // Caracteristicas individuales
    // (campos de instancia).

    // Edad del conejo.
    private int edad;
    // Si el conejo está vivo o no
    private boolean vive;
    // La posición del conejo
    private Ubicacion ubicacion;

    Se omiten los métodos.
}
```

Comportamiento del Conejo

- Definido en el método **`correr`**.
- La edad se incrementa en cada "paso" de la simulación
  - Un conejo puedo morir en este punto.
- Los conejos de edad suficiente pueden reproducirse en cada paso
  - Pueden nacer nuevos conejos en este punto.

Simplificaciones de Conejo

- Los conejos no tienen diferenciación por sexo.
  - Son todos hembras.
- El mismo conejo puede reproducirse en cada paso.
- Todos los conejos mueren a la misma edad (excepto que se los coman)
- ¿Otros?

Estado del zorro

```
public class Zorro
{
    Se omitieron los campos estáticos

    // La edad del zorro.
    private int edad;
    // Si el zorro está vivo o no
    private boolean vive;
    // La ubicación del zorro
    private Ubicacion ubicacion;
    // El nivel de comida del zorro, que
    // se incrementa comiendo conejos.
    private int nivelDeComida;

    Se omiten los métodos.
}
```

Comportamiento del Zorro

- Definido en el método **`cazar`**.
- Los zorros también pueden reproducirse.
- Pueden tener hambre.
- Cazan en ubicaciones adyacentes.

Configuración de los zorros

- Presentan simplificaciones similares a las de los conejos
- Cazar y comer pueden modelarse de diferentes formas
  - ¿El nivel de comida debe ser incremental?
  - ¿Un zorro hambriento es más probable que cace?
- ¿Cuándo son aceptables las simplificaciones?

La clase simulador

- Tres componentes claves:
  - El constructor.
  - El método **`poblar`**.
    - Cada animal recibe una edad inicial.
  - El método **`simularUnPaso`**.
    - Itera sobre poblaciones de zorros y conejos
    - Se usan dos objetos **`Campo`**: **`campo`** y **`campoActualizado`**.

Paso de Simulador

```
for(Iterator<Conejo> it = conejos.iterator();
        it.hasNext(); ) {
    Conejo conejo = it.next();
    conejo.correr(campoActualizado, nuevosConejos);
    if(! conejo.vive()) {
        it.remove();
    }
}
…
for(Iterator<Zorro> it = zorros.iterator();
        it.hasNext(); ) {
    Zorro zorro = it.next();
    zorro.cazar(campo, campoActualizado, nuevosZorros);
    if(! zorro.vive()) {
        it.remove();
    }
}
```

Oportunidades de mejora

- **`Zorro`** y **`Conejo`** poseen grandes similitudes pero no tienen una superclase común.
- El **`pasoDeSimulador`** incluye código similar.
- La clase **`Simulador`** está muy acoplada a clases específicas:
  - Sabe demasiado del comportamiento de los zorros y los conejos.

La superclase *Animal*

- Ubicar atributos comunes en **`Animal`**:
  - **`edad`**, **`vive`**, **`ubicacion`**
- Renombrar métodos para lograr ocultamiento de informacion
  - **`correr`** y **`cazar`** se convierten en **`actuar`**.
- **`Simulador`** se desacopla.

Iteración revisada (desacoplada)

```
for(Iterator<Animal> it = animales.iterator();
        it.hasNext(); ) {
    Animal animal = iter.next();
    animal.actuar(campo,
                  campoActualizado,
                  nuevosAnimales);
    if(! animal.vive()) {
        it.remove();
    }
}
```

El método actuar de *Animal*

- El chequeo estático requiere de un método **`actuar`** en **`Animal`**.
- No existe una implementación común para este método.
- Definimos **`actuar`** como abstracta:

```
abstract public void actuar(Campo campoActual,
                            Campo campoActualizado,
                            List<Animal> nuevosAnimales);
```

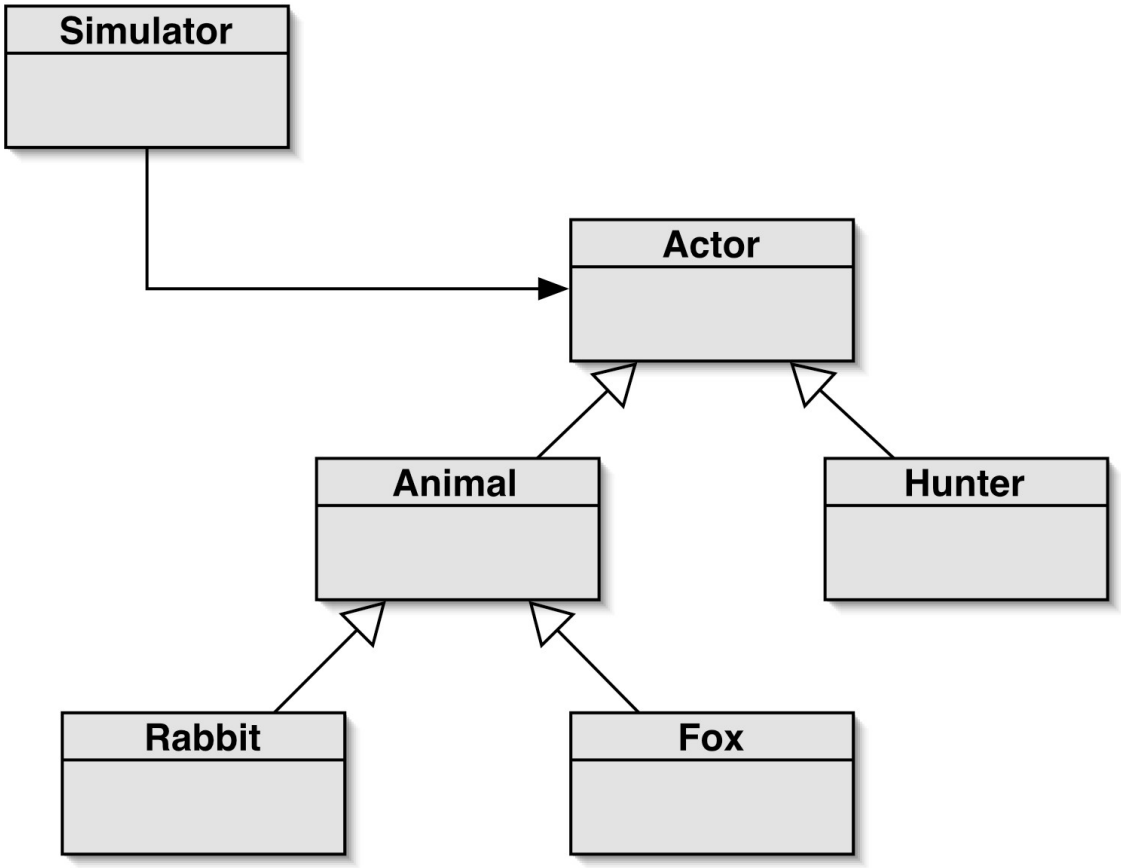
Clases y métodos abstractos

- Los métodos abstractos llevan la palabra clave **`abstract`** en su signatura
- Los métodos abstractos no tienen cuerpo
- Los métodos abstractos hacen que la clase sea abstracta
- Las clases abstractas no pueden instanciarse
- Las subclases concretas completan la implementación faltante

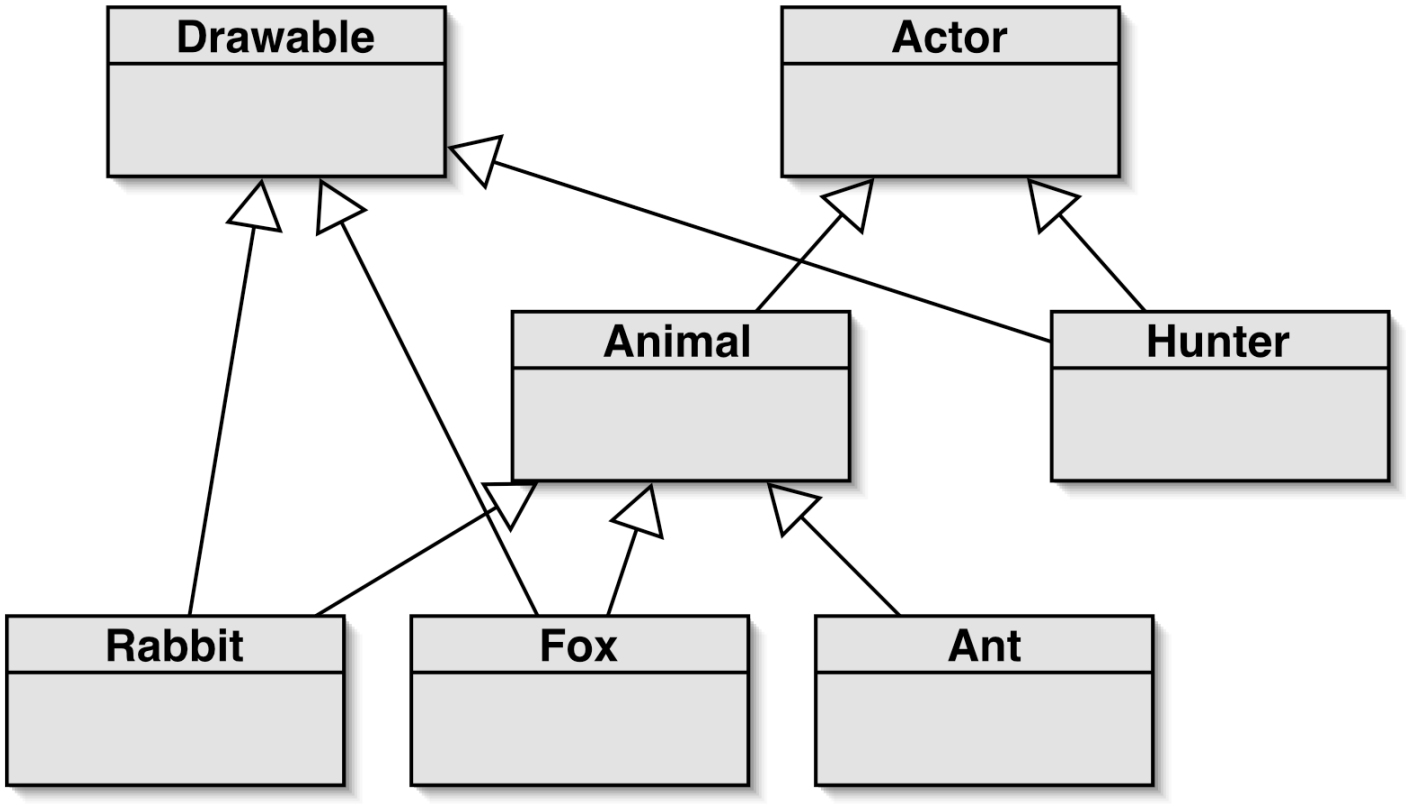
La clase Animal

```
public abstract class Animal
{
    atributos omitidos

    /**
     * Hace que este animal actúe, es decir: hace
     * que haga lo que quiera o necesita hacer
     */
    abstract public void actuar(Campo campoActual,
                                Campo campoActualizado,
                                List<Animal> nuevosAnimales);

    se omiten los métodos restantes
}
```

Más abstracciones



Dibujo selectivo (herencia múltiple)



Herencia múltiple

- Una clase deriva de más de una superclase.
- Cada lenguaje tiene sus propias reglas
- Java prohibe herencia múltiple de clases.
- Java permite herencia múltiple de interfaces

La interfaz Actor

```
public interface Actor
{
    /**
     * Determina el comportamiento diario del actor.
     * Traslada al actor al campoActualizado si es que
     * participa en otros pasos de la simulación.
     * @param campoActual El estado actual del campo
     * @param campoActualizado El estado actualizado
     *                del campo
     * @param nuevosActores Nuevos actores creados como
     *                resultado de las acciones del Actor
     */
    void actuar(Campo campoActual,
                Campo campoActualizado,
                List<Actor> nuevosActores);
}
```

Las clases implementan interfaces

```
public class Zorro extends Animal implements Dibujable
{
    ...
}

 public class Cazador implements Actor, Dibujable
 {
     ...
 }
```

Interfaces como tipos

- Las clases que implementan interfaces no heredan código, pero...
- ... las clases que implementan interfaces son subtipos de la interfaz.
- El polimorfismo se puede obtener tanto de las clases como de las interfaces.

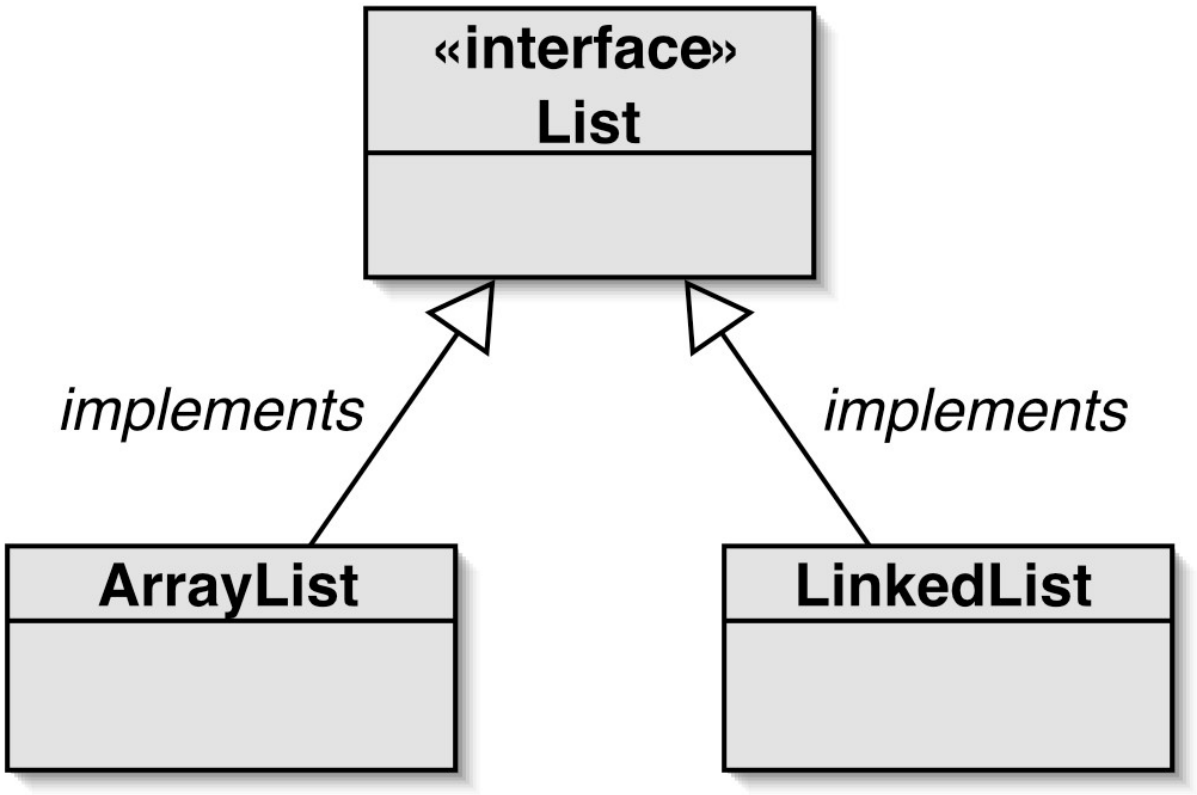
Características de las Interfaces

- Todos sus métodos son abstractos.
- No tienen constructores.
- Todos sus métodos son públicos.
- Todos sus atributos deben ser **`public`**, **`static`** y **`final`**.

Interfaces como especificaciones

- Separan fuertemente la funcionalidad de la implementación.
  - Sin embargo, se respetan los parámetros y tipos retornados.
- Los clientes de las interfaces interactúan independientemente de la implementación.
  - Pero pueden elegir entre distintas implementaciones.

Implementaciones alternativas



Resumen

- La herencia puede usarse para heredar código.
  - Clases abstractas y concretas.
- La herencia puede usarse para heredar tipo.
  - Clases e interfaces.

Resumen

- Los métodos abstractos permiten chequeo estático de tipos sin que exista una implementación.
- Las clases abstractas funcionan como superclases incompletas
  - No pueden instanciarse.
- Las clases abstractas permiten el polimorfismo.

Interfaces

- Las Interfaces permiten especificación sin implementación.
  - Las Interfaces son completamente abstractas.
- Las Interfaces permiten el polimorfismo.
- Java permite herencia múltiple a través de las interfaces.