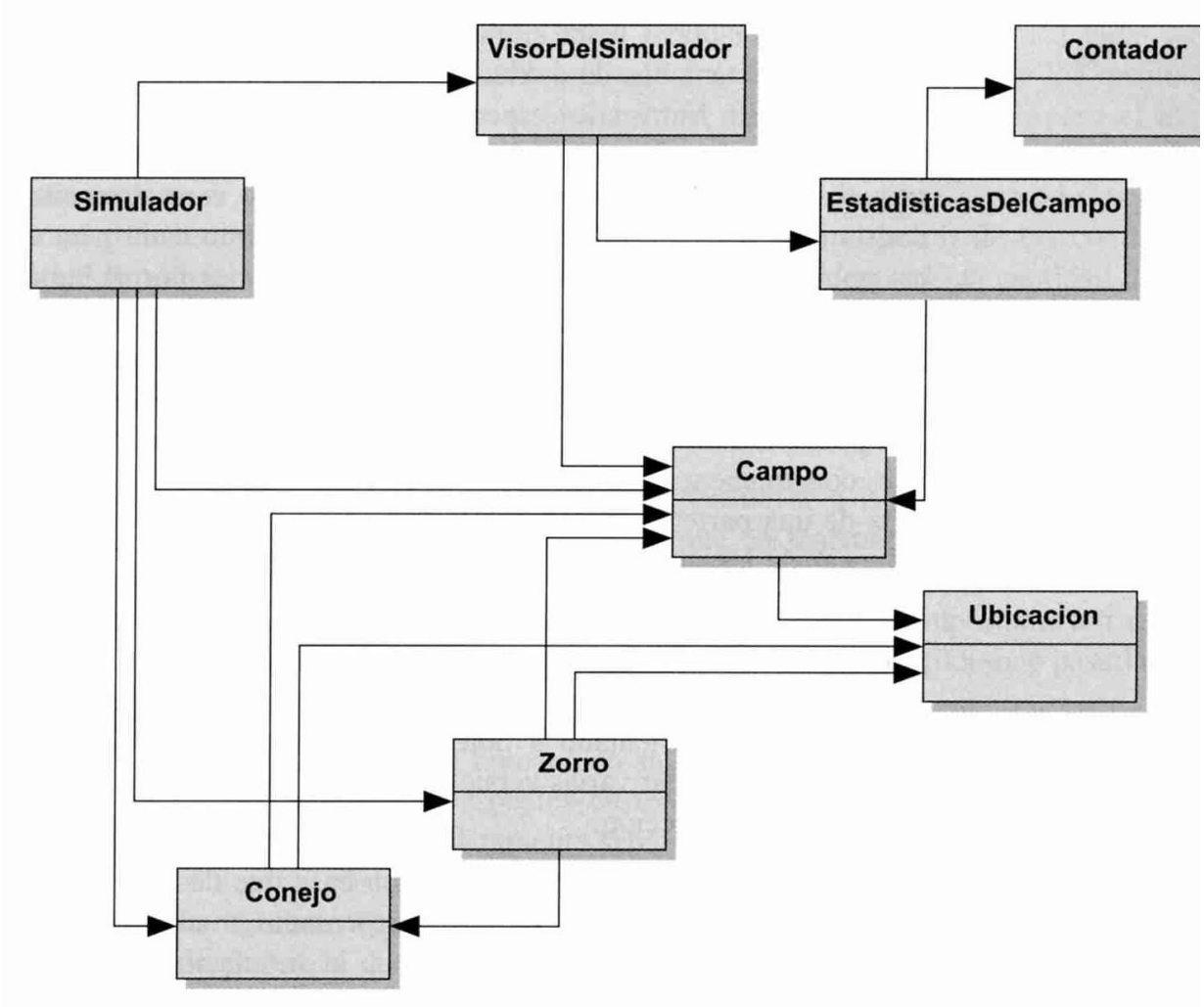
Programación orientada a objetos

Capítulo 10

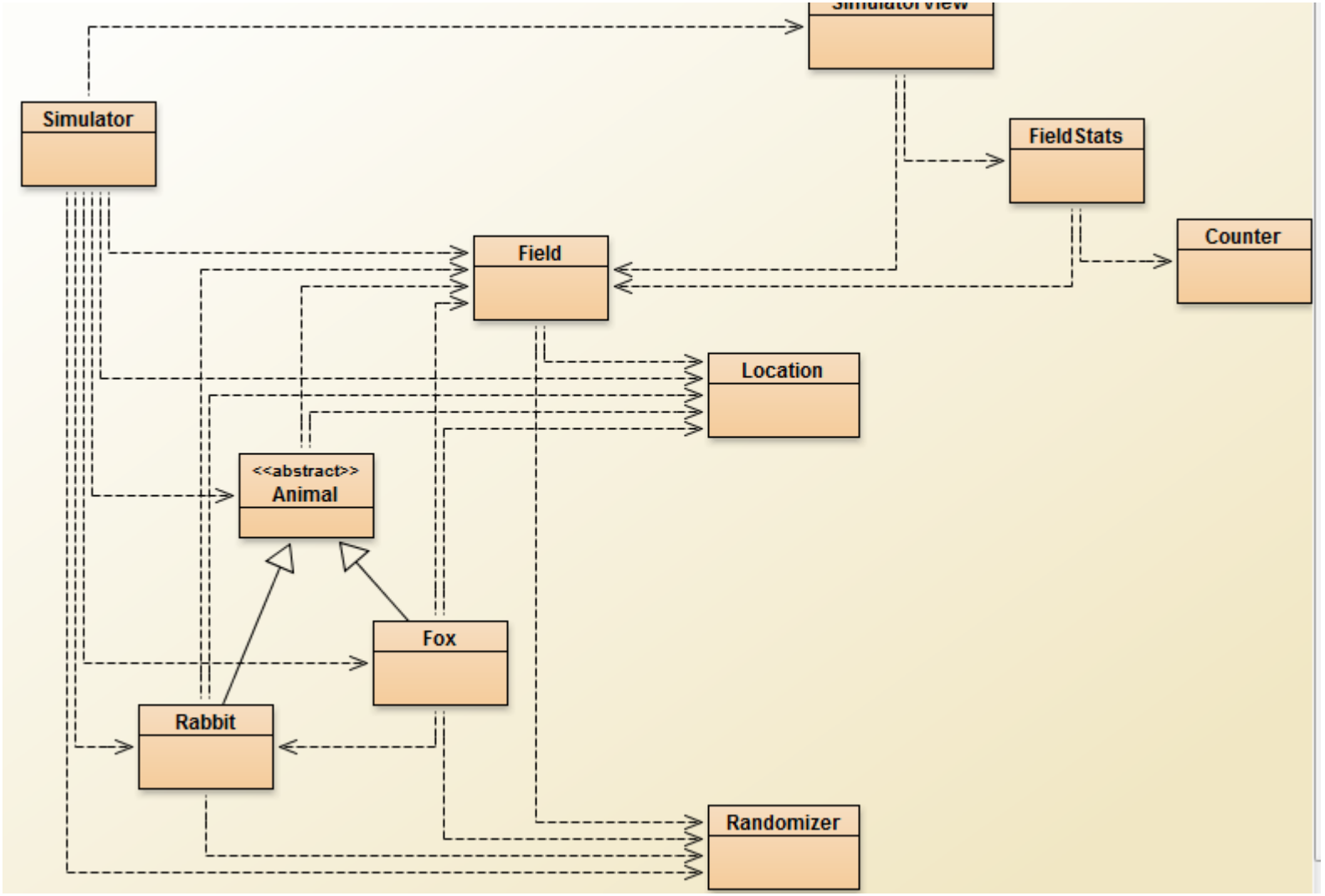
Más técnicas de abstracción

Tema 10: Más técnicas de abstracción. Semana 10	
1- Simulaciones. 2- El modelo de simulación predador-presa. 3- Clases abstractas. 4- Más métodos abstractos. 5- Herencia múltiple. 6- Interfaces. 7- Resumen de herencia.	1- Estudiar el capítulo 10 del libro base para la "Unidad Didáctica II". 2- Realizar los ejercicios correspondientes del libro base. 3- Realizar los ejercicios resueltos en examines de años anteriores en los que se utilice la herencia.

Implementación sin herencia



Con Herencia y clase abstracta



10.3.2 Solución con superclase Animal pero implementación insatisfactoria con "instanceof"

```
        for(Iterator<Animal> iter = animales.iterator();
iter.hasNext(); ) {
                    Animal animal = iter.next();
                    if(animal instanceof Conejo) {
                          Conejo conejo = (Conejo)animal;
                          conejo.correr(campoActualizado,
animalesNuevos);
                    }
                    else if(animal instanceof Zorro) {
                          Zorro zorro = (Zorro)animal;
                          zorro.cazar(campo, campoActualizado,
animalesNuevos);
                    }
                    else {
                          System.out.println("se encontró un
animal desconocido");
                    }
      // Elimina de la simulación los animales muertos.
      if(! animal.estaVivo()){
            iter.remove();
      }
   }
```

minamos el tipo mediante el uso del operador `instanceof`. El operador `instanceof` evalúa si un objeto determinado es una instancia de determinada clase, directa o indirectamente. La evaluación de la sentencia

```
obj instanceof MiClase
```

da resultado `true` si el tipo dinámico de `obj` es `MiClase` o cualquier subclase de `MiClase`.

Con métodos general “actuar” métodos abstractos

```
        // Permite que todos los animales actúen
for(Iterator<Animal> it = animales.iterator(); it.hasNext();
) {
                    Animal animal = it.next();
                    animal.actuar(campo, campoActualizado,
animalesNuevos);
    // Elimina de la simulación los animales muertos.
    if(! animal.estaVivo()){
         it.remove();
    }
         }
```

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

Método actuar

En este punto, son importantes varias observaciones:

- La variable que estamos usando para cada elemento de la colección (`animal`) es de tipo `Animal`. Esto es legal ya que todos los objetos de la colección son conejos o zorros, por lo que todos son subtipos de `Animal`.
- Asumimos que los métodos específicos de acción (`correr` para `Conejo`, `cazar` para `Zorro`) han sido renombrados como `actuar`. Esto es más adecuado: en lugar de decir exactamente lo que hace cada animal, estamos diciendo «actuar» y dejamos que el animal propiamente dicho decida exactamente lo que quiere hacer. Esto reduce el acoplamiento entre `Simulador` y las subclases individuales de los animales.
- El método `correr` de `Conejo` sólo tiene dos parámetros: `campoActualizado` y `animalesNuevos`. Hemos agregado un tercer parámetro, `campo`, para hacerlo consistente con el método `actuar` del zorro. Ahora, cada animal obtiene todos los parámetros que posiblemente necesite para implementar una acción flexible y cada clase puede elegir ignorar cualquiera de los parámetros.
- Dado que el tipo dinámico de la variable determina qué método es realmente ejecutado (como lo discutimos en el Capítulo 9), el método de acción del zorro se ejecutará para los zorros y el método de acción de los conejos, para los conejos.
- Dado que el control de tipos se realiza usando el tipo estático, este código compilará sólo si la clase `Animal` tiene un método `actuar` con la signatura correcta.

10.3.2 Método abstracto

La definición de un **método abstracto** consiste en la signatura de un método sin su correspondiente cuerpo. Se indica mediante la palabra clave `abstract`.

Un método abstracto se caracteriza por dos detalles:

- Está precedido por la palabra clave `abstract`.
- No tiene cuerpo y su encabezado termina con un punto y coma.

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

10.3.3 Clases abstractas

- No se creará ninguna instancia de clases abstractas
- Sólo las clases abstractas pueden tener métodos abstractos
- Las clases abstractas con métodos abstractos fuerzan a las subclases a sobreescribir una implementación de aquellos métodos declarados abstractos

Una **clase abstracta** es una clase de la que no se pretende crear instancias. Su propósito es servir como superclase a otras clases. Las clases abstractas pueden contener métodos abstractos.

Subclases abstractas. Para que una subclase de una clase abstracta se convierta en una subclase concreta, debe proveer las implementaciones de todos los métodos abstractos heredados. De lo contrario, será propiamente abstracta.

```
public abstract class Animal
{
   // Se omiten los campos
   /*
   * Hace que este animal actúe, es decir: hace que haga
lo que quiera
   * o necesita hacer.
   * @param campoActual El campo que ocupa actualmente
   * @param campoActualizado El campo al que se trasladará
   * @param animalesNuevos Una lista para agregar los
animales recién
          *                                    nacidos.
*/
abstract public void actuar (Campo campoActual,
                             Campo campoActualizado,
                             List<Animal> animalesNuevos);
// Se omiten los restantes métodos
}
```

- no se pueda crear directamente ninguna instancia de la clase `Animal` y
- todas las subclases concretas deban implementar el método `actuar`.

10.4 Más métodos abstractos

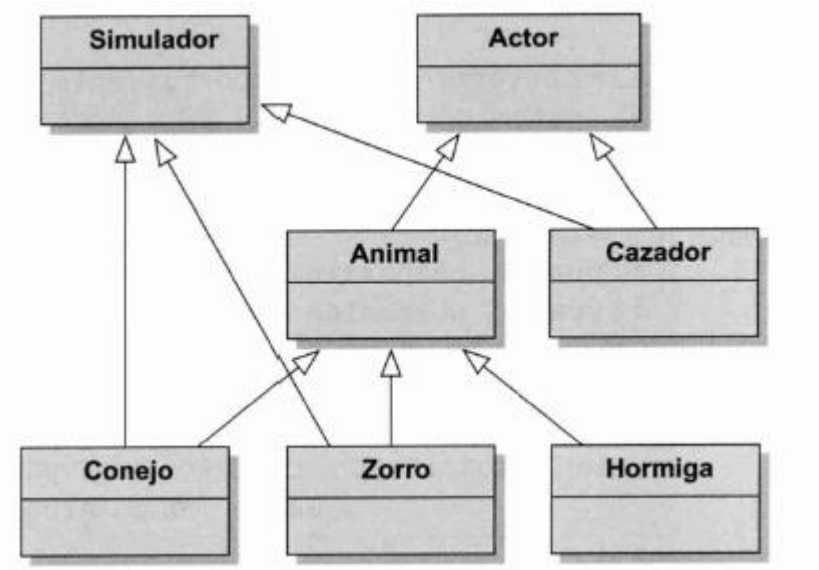
```
/**
* Un animal puede reproducirse si alcanzó la edad de
reproducción.
* @return true  si el animal puede reproducirse
*/
public boolean puedeReproducir()
{
  return edad >= getEdadDeReproduccion();
}
```

El método `puedeReproducir` ha sido sobrescrito para que use el valor que retorna una llamada a un método en lugar del valor de una variable de clase. Para que esto funcione, se debe definir el método `getEdadDeReproduccion` en la clase `Animal`. Dado que no podemos especificar una edad de reproducción para los animales en general, podemos usar nuevamente un método abstracto en la clase `Animal` y concretar las redefiniciones en las subclases. Tanto `Zorro` como `Conejo` definirán sus propias versiones del método `getEdadDeReproduccion` para devolver sus particulares valores de la variable `EDAD_DE_REPRODUCCION`:

```
/**
* @return La edad en que un conejo comienza a reproducirse
*/
public int getEdadDeReproduccion()
{
```

10.5 Herencia múltiple

Herencia múltiple. Una situación en la que una clase deriva de más de una superclase se denomina herencia múltiple.



El escenario aquí presentado usa una estructura que se conoce como *herencia múltiple*. La herencia múltiple existe en los casos en los que una clase deriva de más de una superclase. En consecuencia, la subclase tiene todas las características de ambas superclases y aquellas definidas en la subclase propiamente dicha.

Java no permite herencias múltiple pero proporciona otra construcción denominada “interfaces”

10.6 Interfaces

Una **interfaz** en Java es la especificación de un tipo (bajo la forma de un nombre de tipo y un conjunto de métodos) que no define ninguna implementación para los métodos.

- En el encabezado de la declaración se usa la palabra clave `interface` en lugar de `class`.
- Todos los métodos de una interfaz son abstractos; no se permiten métodos con cuerpo. No es necesaria la palabra clave `abstract`.
- Las interfaces no contienen ningún constructor.
- Todas las signaturas de los métodos de una interfaz tienen visibilidad pública. No es necesario declarar la visibilidad: por ejemplo, no es necesario que cada método contenga la palabra clave `public`.
- En una interfaz, sólo se permiten los campos constantes (campo público, estático y final). Pueden omitirse las palabras clave `public`, `static` y `final` pero todos los campos, igualmente, serán tratados como públicos, estáticos y finales.

Una clase puede derivar de una interfaz de la misma manera en que deriva de una clase. Sin embargo, Java utiliza una palabra clave diferente, `implements`, para la herencia a partir de interfaces.

Se dice que una clase *implementa* una interfaz si incluye una *cláusula implements* en su encabezado. Por ejemplo:

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

Como en este caso, si una clase extiende a una clase e implementa una interfaz, entonces la cláusula `extends` debe escribirse primero en el encabezado de la clase.

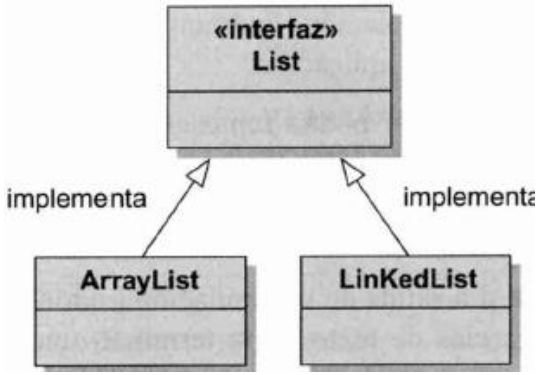
```
/**
* La interfaz que será implementada por cualquier clase
que
* que desee participar de la simulación.
*/
public interface Actor
{
   /**
   * Determina el comportamiento diario del actor. Traslada
al actor
* al campoActualizado si es que participa en otros pasos
de la
* simulación.
* @param campoActual El estado actual del campo
* @param ubicacion La ubicación del actor en el campo
actual
* @param campoActualizado El estado actualizado del campo
*/
void actuar(Campo campoActual, Ubicacion ubicacion,
                  Campo campoActualizado);
}
```

Herencia múltiple de interface

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

- Java permite que cada clase extienda como máximo a otra clase, pero permite que una clase implemente cualquier número de interface (además de la posibilidad de extender una clase)
- Interfaz como tipo
  - Una interfaz define un tipo exactamente igual que lo hace una clase
- Interfaz como especificación

La característica más importante de las interfaces es que separan completamente la definición de la funcionalidad (la clase «interfaz» en el sentido más amplio de la palabra) de su implementación. Un buen ejemplo de cómo pueden usarse las interfaces en la práctica se puede encontrar en la jerarquía de las colecciones de Java.



¿Clase abstracta o interfaz?

En algunas situaciones se tiene que elegir entre usar una clase abstracta o una interfaz. Algunas veces la elección es fácil: cuando se pretende que la clase contenga implementaciones para algunos métodos necesitamos usar una clase abstracta. En otros casos, tanto la clase abstracta como la interfaz pueden hacer el mismo trabajo.

Si tenemos que elegir, es preferible usar interfaces. Si proveemos un tipo mediante una clase abstracta, las subclases no pueden extender ninguna otra clase; dado que las interfaces permiten la herencia múltiple, el uso de una interfaz no crea tal restricción. Por lo tanto, el uso de interfaces da por resultado una estructura más flexible y más extensible.

Términos introducidos en este capítulo

método abstracto, clase abstracta, clase concreta, herencia múltiple, interfaz (construcción Java), implementa

Resumen de conceptos

- **método abstracto** Una definición de un método abstracto consiste en una signatura de método sin un cuerpo. Se marca con la palabra clave `abstract`.
- **clase abstracta** Una clase abstracta es una clase de la que no se tiene intención de crear instancias. Su propósito es servir como una superclase a otras clases. Las clases abstractas pueden contener métodos abstractos.
- **subclases abstractas** Para que una subclase de una clase abstracta se vuelva concreta, debe proveer implementaciones para todos los métodos abstractos heredados; de lo contrario, es propiamente abstracta.
- **herencia múltiple** Una situación en la que una clase deriva de más de una superclase se denomina herencia múltiple.
- **interfaz** Una interfaz en Java es la especificación de un tipo (bajo la forma de un nombre de tipo y un conjunto de métodos) que no define ninguna implementación para ningún método.