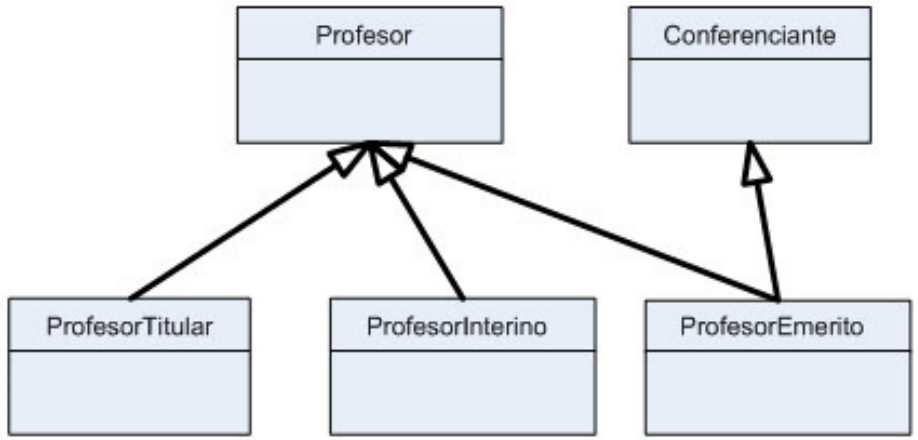
Programación Orientada a Objetos

Capítulo 12: Técnicas de abstracción adicionales

Abstracción

- **Clase abstracta:** clase de la que no se tiene intención de crear instancias. Su propósito es ser una superclase para otras clases. Las clases abstractas pueden contener métodos abstractos.
- **Método abstracto:** método sin implementación (sólo signatura). Se indica con la palabra clave "*abstract*".
- **Subclases abstractas:** Para que una subclase de una clase abstracta se vuelva concreta, debe implementar todos los métodos abstractos heredados; de lo contrario, es abstracta.
- **Herencia múltiple:** la posibilidad de que una clase derive de más de una superclase.
- Java no permite la herencia múltiple, pero a cambio dispone de la construcción denominada "*interfaz*" que permite una forma de simulación o implementación limitada de la herencia múltiple.



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.

Una clase sólo puede derivar de otra clase, pero puede implementar varias interfaces.

- Características:
  - En el encabezado se usa la palabra clave ***interfaz*** en lugar de ***class***.
  - Las interfaces **no tienen constructor.**
  - Todos los **métodos** son **abstractos y públicos** sin necesidad de especificarlo así. Ninguno de sus métodos tiene implementación. Por lo tanto, cuando una clase implementa una interfaz no hereda ninguna implementación.
  - Todos los **campos** son **públicos y constantes** (public static final) sin necesidad de especificarlo.
  - Para declarar la herencia de una interfaz se usa la palabra clave **implements** en lugar de **extends**.
  - La característica más importante de las interfaces es que separan completamente la definición de la funcionalidad de su implementación.
- ¿Clase abstracta o interfaz?
  - Clase abstracta si se pretende que la clase contenga la implementación de algún método
  - En otros casos es preferible usar interfaces:
    - 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.

Otros conceptos importantes

- Usos de la herencia:
  - heredar el código (a partir de clases y parcialmente de clases abstractas)
  - heredar el tipo (a partir de clases, clases abstractas e interfaces)

- Los campos de una superclase no pueden ser sobrescritos por las versiones de las subclases, pero sí se pueden utilizar métodos en la superclase que se rescriban en las subclases y modifiquen los campos.

- El operador instanceof evalúa si un objeto determinado es una instancia de determinada clase, directa o indirectamente:

  obj instanceof MiClase

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

- Una clase, método o campo declarado como **estático** puede ser accedido o invocado sin la necesidad de tener que instanciar un objeto de la clase.
  - Se identifican con la palabra “static”.
  - Las variables estáticas son compartidas entre todos los objetos de una clase.
  - Los métodos estáticos no se pueden sobrescribir ni ser abstractos.

Fox

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
public class Fox
{
    // Characteristics shared by all foxes (class variables).

    // The age at which a fox can start to breed.
    private static final int BREEDING_AGE = 15;
    // The age to which a fox can live.
    private static final int MAX_AGE = 150;
    // The likelihood of a fox breeding.
    private static final double BREEDING_PROBABILITY = 0.08;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 2;
    // The food value of a single rabbit. In effect, this is the
    // number of steps a fox can go before it has to eat again.
    private static final int RABBIT_FOOD_VALUE = 9;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).

    // 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;
```

Rabbit

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
public class Rabbit
{
    // Characteristics shared by all rabbits (class variables).

    // The age at which a rabbit can start to breed.
    private static final int BREEDING_AGE = 5;
    // The age to which a rabbit can live.
    private static final int MAX_AGE = 40;
    // The likelihood of a rabbit breeding.
    private static final double BREEDING_PROBABILITY = 0.12;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 4;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // 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;
```

Fox

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
public void hunt(List<Fox> newFoxes)
{
    incrementAge();
    incrementHunger();
    if(alive) {
        giveBirth(newFoxes);
        // Move towards a source of food if found.
        Location newLocation = findFood();
        if(newLocation == null) {
            // No food found - try to move to a free location.
            newLocation = field.freeAdjacentLocation(location);
        }
        // See if it was possible to move.
        if(newLocation != null) {
            setLocation(newLocation);
        }
        else {
            // Overcrowding.
            setDead();
        }
    }
}
```

Rabbit

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
public void run(List<Rabbit> newRabbits)
{
    incrementAge();
    if(alive) {
        giveBirth(newRabbits);
        // Try to move into a free location.
        Location newLocation = field.freeAdjacentLocation(loc
        if(newLocation != null) {
            setLocation(newLocation);
        }
        else {
            // Overcrowding.
            setDead();
        }
    }
}
```

Fox

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar...

```
public Location getLocation()
{
    return location;
}

/**
 * Place the fox at the new location in
 * @param newLocation The fox's new loca
 */
private void setLocation(Location newLoc
{
    if(location != null) {
        field.clear(location);
    }
    location = newLocation;
    field.place(this, newLocation);
}

/**
 * Increase the age. This could result i
 */
private void incrementAge()
{
    age++;
    if(age > MAX_AGE) {
        setDead();
    }
}
```

Rabbit

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Cód

```
public Location getLocation()
{
    return location;
}

/**
 * Place the rabbit at the new location in the give
 * @param newLocation The rabbit's new location.
 */
private void setLocation(Location newLocation)
{
    if(location != null) {
        field.clear(location);
    }
    location = newLocation;
    field.place(this, newLocation);
}

/**
 * Increase the age. This could result in the rabbi
 */
private void incrementAge()
{
    age++;
    if(age > MAX_AGE) {
        setDead();
    }
}
```

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Ce

```java
public abstract class Animal
{
    // Whether the animal is alive or not.
    private boolean alive;
    // The animal's field.
    private Field field;
    // The animal's position in the field.
    private Location location;
```

Rabbit ×

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar

```java
public class Rabbit extends Animal
{
    // Characteristics shared by all rabbits (class variables).

    // The age at which a rabbit can start to breed.
    private static final int BREEDING_AGE = 5;
    // The age to which a rabbit can live.
    private static final int MAX_AGE = 40;
    // The likelihood of a rabbit breeding.
    private static final double BREEDING_PROBABILITY = 0.12;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 4;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).

    // The rabbit's age.
    private int age;
```

Fox ×

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```java
public class Fox extends Animal
{
    // Characteristics shared by all foxes (class variables).

    // The age at which a fox can start to breed.
    private static final int BREEDING_AGE = 15;
    // The age to which a fox can live.
    private static final int MAX_AGE = 150;
    // The likelihood of a fox breeding.
    private static final double BREEDING_PROBABILITY = 0.08;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 2;
    // The food value of a single rabbit. In effect, this is the
    // number of steps a fox can go before it has to eat again.
    private static final int RABBIT_FOOD_VALUE = 9;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).
    // The fox's age.
    private int age;
    // The fox's food level, which is increased by eating rabbits.
    private int foodLevel;
```

Fox

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
public Fox(boolean randomAge, Field field, Location location)
{
    age = 0;
    alive = true;
    this.field = field;
    setLocation(location);
    if(randomAge) {
        age = rand.nextInt(MAX_AGE);
        foodLevel = rand.nextInt(RABBIT_FOOD_VALUE);
    }
    else {
        // leave age at 0
        foodLevel = rand.nextInt(RABBIT_FOOD_VALUE);
    }
}
```

Rabbit

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
public Rabbit(boolean randomAge, Field field, Location location)
{
    age = 0;
    alive = true;
    this.field = field;
    setLocation(location);
    if(randomAge) {
        age = rand.nextInt(MAX_AGE);
    }
}
```

Animal

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar

```
public Animal(Field field, Location location)
{
    alive = true;
    this.field = field;
    setLocation(location);
}
```

Fox

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
public Fox(boolean randomAge, Field field, Location location)
{
    super(field, location);
    if(randomAge) {
        age = rand.nextInt(MAX_AGE);
        foodLevel = rand.nextInt(RABBIT_FOOD_VALUE);
    }
    else {
        age = 0;
        foodLevel = RABBIT_FOOD_VALUE;
    }
}
```

Rabbit

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar

```
public Rabbit(boolean randomAge, Field field, Location location)
{
    super(field, location);
    age = 0;
    if(randomAge) {
        age = rand.nextInt(MAX_AGE);
    }
}
```

Animal

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuent

```java
/**
 * Make this animal act - that is: make it do
 * whatever it wants/needs to do.
 * @param newAnimals A list to receive newly born animals.
 */
abstract public void act(List<Animal> newAnimals);

/**
 * Check whether the animal is alive or not.
 * @return true if the animal is still alive.
 */
protected boolean isAlive()
```

Fox

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```java
public void act(List<Animal> newFoxes)
{
    incrementAge();
    incrementHunger();
    if(isAlive()) {
        giveBirth(newFoxes);
        // Move towards a source of food if found.
        Location newLocation = findFood();
        if(newLocation == null) {
            // No food found - try to move to a free location.
            newLocation = getField().freeAdjacentLocation(getLocation());
        }
        // See if it was possible to move.
        if(newLocation != null) {
            setLocation(newLocation);
        }
        else {
            // Overcrowding.
            setDead();
        }
```

Rabbit

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```java
public void act(List<Animal> newRabbits)
{
    incrementAge();
    if(isAlive()) {
        giveBirth(newRabbits);
        // Try to move into a free location.
        Location newLocation = getField().freeAdjacentLocation(getLocation());
        if(newLocation != null) {
            setLocation(newLocation);
        }
        else {
            // Overcrowding.
            setDead();
        }
    }
}
```

SimulatorView

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2016.03.18
 */
public interface SimulatorView
{
    /**
     * Define a color to be used for a given class of animal.
     * @param animalClass The animal's Class object.
     * @param color The color to be used for the given class.
     */
    void setColor(Class<?> animalClass, Color color);

    /**
     * Determine whether the simulation should continue to run.
     * @return true If there is more than one species alive.
     */
    boolean isViable(Field field);

    /**
     * Show the current status of the field.
     * @param step Which iteration step it is.
     * @param field The field whose status is to be displayed.
     */
    void showStatus(int step, Field field);

    /**
     * Prepare for a new run.
     */
    void reset();
}
```

GraphView

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar

```
public class GraphView implements SimulatorView
{
    private static final Color LIGHT_GRAY = new Color(0,

    private static JFrame frame;
    private static GraphPanel graph;
    private static JLabel stepLabel;
    private static JLabel countLabel;

    // The classes being tracked by this view
    private Set<Class<?>> classes;
    // A map for storing colors for participants in the
    private Map<Class<?>, Color> colors;
    // A statistics object computing and storing simulat
    private FieldStats stats;

    /**
     * Constructor.
     *
     * @param width The width of the plotter window (in
     * @param height The height of the plotter window (i
     * @param startMax The initial maximum value for the
     */
    public GraphView(int width, int height, int startMax
    {
        stats = new FieldStats();
        classes = new HashSet<>();
        colors = new HashMap<>();

        if (frame == null) {
            frame = makeFrame(width, height, startMax);
        }
        else {
            graph.newRun();
        }

        //showStatus(0, null);
    }
```

GraphView

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar

```
     */
    public void setColor(Class<?> animalClass, Color
    {
        colors.put(animalClass, color);
        classes = colors.keySet();
    }

    /**
     * Show the current status of the field. The statu
     * two classes in the field. This view currently
     * two classes. If the field contains more than tw
     * will be plotted.
     *
     * @param step Which iteration step it is.
     * @param field The field whose status is to be di
     */
    public void showStatus(int step, Field field)
    {
        graph.update(step, field, stats);
    }

    /**
     * Determine whether the simulation should continu
     * @return true If there is more than one species
     */
    public boolean isViable(Field field)
    {
        return stats.isViable(field);
    }

    /**
     * Prepare for a new run.
     */
    public void reset()
    {
        stats.reset();
        graph.newRun();
    }
```

SimulatorView

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
 *
 * @author Michael Kölling and David J. Barnes
 * @version 2016.03.18
 */
public interface SimulatorView
{
    /**
     * Define a color to be used for a given class of animal.
     * @param animalClass The animal's Class object.
     * @param color The color to be used for the given class.
     */
    void setColor(Class<?> animalClass, Color color);

    /**
     * Determine whether the simulation should continue to run.
     * @return true If there is more than one species alive.
     */
    boolean isViable(Field field);

    /**
     * Show the current status of the field.
     * @param step Which iteration step it is.
     * @param field The field whose status is to be displayed.
     */
    void showStatus(int step, Field field);

    /**
     * Prepare for a new run.
     */
    void reset();
}
```

GridView

Compilar | Deshacer | Cortar | Copiar | Pegar | Buscar... | Cerrar | Código Fuente

```
public class GridView extends JFrame implements SimulatorView
{
    // Colors used for empty locations.
    private static final Color EMPTY_COLOR = Color.white;

    // Color used for objects that have no defined color.
    private static final Color UNKNOWN_COLOR = Color.gray;

    private final String STEP_PREFIX = "Step: ";
    private final String POPULATION_PREFIX = "Population: ";
    private JLabel stepLabel, population;
    private FieldView fieldView;

    // A map for storing colors for participants in the simulation
    private Map<Class<?>, Color> colors;
    // A statistics object computing and storing simulation inform
    private FieldStats stats;

    /**
     * Create a view of the given width and height.
     * @param height The simulation's height.
     * @param width  The simulation's width.
     */
    public GridView(int height, int width)
    {
        stats = new FieldStats();
        colors = new HashMap<>();

        setTitle("Fox and Rabbit Simulation");
        stepLabel = new JLabel(STEP_PREFIX, JLabel.CENTER);
        population = new JLabel(POPULATION_PREFIX, JLabel.CENTER);
```