Programación orientada a objetos

Curso INEM. Programación en Java
Santiago Muelas Pascual
smuelas@fi.upm.es

Repaso

- Herencia
- Superclase/subclase
- super
- Modificador protected
- Redefinicion de métodos
- Modificador final (asociado a herencia)
- Clase Object

Polimorfismo

- Consiste en la posibilidad de que una referencia a un objeto pueda conectarse también con objetos descendientes de este
- Puede tratarse cualquier subclase de animal como si fuese un objeto de la clase Animal
- En otras palabras "Me da igual el tipo de subclase que seas siempre que cumplas con la declaración de Animal"
- Tiene sentido por la existencia de la herencia

Ejemplo

```
class Animal {
  void mover() { …
  }
}
class Perro extends Animal {}
class Pez extends Animal {}
public static void main(String[] args) {
  Animal a = new Perro();
  a = new Pez();
  a.mover();
}
```

Polimorfismo

- Sirve para generalizar, olvidándose de los detalles y buscando puntos comunes
- Se usa, sobre todo, en la construcción de estructuras de datos
- Escenario:
  - Necesitamos mantener un array de perros y gatos de una clínica veterinaria
  - ¿Cómo definimos dicho array? ¿Usamos uno o dos?
- También se puede realizar de manera implícita en el paso de argumentos
  - `public void metodoquesea (Mamifero m)`

Ejercicio

- Crear un array de la clase Animal de tamaño 2
- Añadir a la primera posición un objeto de la clase Rana
- Añadir a la segunda posición un objeto de la clase Pez
- Recorrer el array llamando al método mover de cada objeto

Polimorfismo

- Nos permite crear sistemas que sean fácilmente extensibles sin apenas tener que modificarlos
  - Nuevas subclases pueden ser añadidas sin apenas tener que modificar el código existente
- ¡OJO! No podemos tratar a un objeto de una superclase como si fuese de una subclase
  - Ej: Animal a;
    - a puede ser un objeto de la clase Perro, Pez, … pero no sabemos cuál.

Polimorfismo

- ¿Qué hacemos para convertir una referencia de una superclase a una subclase?
  - Casting Ej Perro p = (Perro) a;
- ¿Y si nos confundimos?
  - Ej
    - Perro p = new Perro();
    - String s = (String) p;
- ¿Qué hacemos para detectar si el casting es el adecuado?

Operador instanceof

- Para comprobar si una referencia es una instancia (clase o subclase) de la clase indicada

```
Animal h = new Animal();
if (h instanceof Animal) System.out.println("Es un animal");

Caballo h = new Caballo();
if (h instanceof Animal) System.out.println("Es un animal");
```

Clases Abstractas

- Es una clase tan genérica, que nunca se creará una instanciación en objetos
  - Se produce un error de compilación si se intenta crear un objeto
- Sirve sólo como superclase, y, por tanto, para definir subclases
- Cuando uno de los métodos no tiene implementación, estamos ante una clase **abstracta**
- El método sin implementación será un método **abstracto**

```
public abstract class Figura {
  public abstract double area();
}
```

Ejemplo abstract

- Una figura con el método área es una clase abstracta típica ¿Por qué?
- No se puede calcular el área de una figura
- Es necesario saber qué figura es la subclase (círculo, cuadrado, …)
- Y redefinir el método para calcular el área dependiendo de la figura correcta

Ejercicio

- Modificar la clase Animal del ejercicio anterior para que el método mover sea abstracto

Herencia múltiple

- Ya hemos dicho que no existe herencia múltiple entre clases
  - ¿Por qué?
- ¿Y si necesitamos tener herencia múltiple?

Interfaces

- Similares a las clases abstractas
- En donde todos los métodos son abstractos
- Define el comportamiento pero no la implementación
  - El qué pero no el cómo
- Las clases pueden implementar una interfaz
  - Pero tienen que dar una definición de todos sus métodos
  - O ser clases abstractas
    - Ya que no estamos dando una implementación de todos sus métodos

Ejemplo

```
public interface Figura {
  double area();
}
public class Circulo implements Figura {
  double area() {
    …
  }
}
public static void main(String[] args) {
  Figura f = new Circulo(); // Similar a una clase abstracta
}
```

Interfaces

- No contienen a variables de instancia
- Sólo se pueden declarar constantes de tipo public
  - **public static final**
  - Y tienen que inicializarse
  - Si no se especifica public static final, el compilador lo añade automáticamente
- **Los métodos tienen visibilidad public**
  - Aunque no se especifique

Ejercicio

- Escribir la interfaz Animal que contenga al método mover
- Escribir las clases Pez y Caballo que implementen el método mover mostrando un mensaje de texto por la salida. E.g. "nadando"/"corriendo"
- Crear un array de objetos que cumplen la interfaz Animal y añadir dos objetos de la clase Pez y Caballo
- Recorrer el array llamando al método mover.

Ejemplo

```
public interface Figura {
  double PI = 3.14;
  double area();
}
public class Circulo implements Figura {
  double area() {
    …
  }
}
public static void main(String[] args) {
  Figura f = new Circulo(); // Similar a una clase abstracta
}
```

Ejemplo

```
public interface Figura {
  double PI = 3.14;
  double area();
}
public abstract class Circulo implements Figura {
}
public static void main(String[] args) {
  Figura f = new Circulo(); //Error
}
```

Herencia de interfaces

- Una interfaz puede extender a otro al igual que sucedía con las clases

```
public interface ObjetoGrafico {
  public void pintar();
}
public interface FiguraGrafica extends ObjetoGrafico {
  public void area();
}
public class CuadradoGrafico implements FiguraGrafica {
  public void area() { … }
  public void pintar() { …}
}
```

Herencia múltiple

- Ya hemos dicho que no existe herencia múltiple entre clases
- Una clase puede implementar más de una interfaz

Herencia múltiple

```
interface Volador {
  void volar();
}

interface Corredor{
  void correr();
}

class Pegaso implements Volador, Corredor {
  public void volar() {…}
  public void correr() {…}
}
```

Herencia múltiple

```
interface Volador {
  void volar();
}

interface Corredor{
  void correr();
}

class Animal{
  void comer();
}

class Pegaso extends Animal implements Volador, Corredor {
  public void volar() {…}
  public void correr() {…}
  public void comer() {…}
}
```

Clase Object

- Realmente todas las clases que creemos son subclases…
- … de la clase Object
- Esta clase tiene métodos habituales que están disponibles para cualquier clase que creemos
- Los más habituales son:
  - equals
  - toString
  - clone
  - hashCode
    - Los veremos más adelante

Método toString

- Devuelve un representación en un String de un objeto
- Por defecto devuelve el nombre del paquete, el de la clase y una representación hexadecimal correspondiente al valor devuelto por el método hashCode

String toString() {…}

Método equals

- == devuelve verdadero si las variables contienen la misma referencia
- El método equals se utiliza para comparar el contenido
- Por defecto, devuelve la comparación ==
- La mayoría de las clases de biblioteca tienen redefinido el método equals

Método equals

```
public boolean equals(Object o) {
  return (o instanceof MiClase &&
         ((MiClase) o).getMiAtributo == this.miAtributo);
}
```

Método clone

- Para realizar una copia del objeto desde donde se llama
- Para implementarlo hay que implementar la interfaz Clonable
- Por defecto, se copian el valor directo de las variables
  - ¿Qué pasa con las referencias?
- Si no se implementa la interfaz, se genera una excepción (error en ejecución como se verá más adelante)

Ejemplo

```
class Perro implements Cloneable{
 String nombre;
 public Perro(String nombre) {
  this.nombre = nombre;
 }
 public Object clone() {
  Perro p = new Perro(this.nombre);
  return p;
 }
public static void main(String[] args) {
 Perro p = new Perro("Pancho");
 Perro q = (Perro) p.clone();
}
}
```