

# Herencia, interfaces y polimorfismo

Programación I


Grado en Ingeniería Informática

MDR, JCRdP y JDGD

# Definición de herencia

- ▶ Definir clases nuevas a partir de otras ya establecidas
- ▶ Con las mismas características que las originales, más otras nuevas
- ▶ Se puede ver como una **extensión** o como una **especialización**
- ▶ Terminología:
  - La nueva se denomina subclase o clase derivada
  - La original se denomina clase base o superclase

# Cambios en la nueva clase

- ▶ Se pueden añadir nuevos atributos y nuevos servicios
  - ▶ Se pueden cambiar los servicios heredados
  - ▶ Cuando una clase deriva de varias a la vez, se le denomina herencia múltiple (no posible en Java)
  - ▶ La herencia tiene múltiples problemas, por ejemplo de colisión de nombres
- 

# Cuándo aplicar herencia

- ▶ Como forma de **representar la jerarquía natural** de las clases según una taxonomía apropiada (especialización)
  - Por ejemplo: en el juego del ajedrez, peones, alfiles, rey, reina, caballos y torres son piezas del juego
- ▶ Para **reutilizar el código existente**, adaptándolo a las nuevas necesidades
  - Por ejemplo: especializar clases que representan elementos de interfaz de usuario: botones, entrada de texto, etc.

# Clientela entre clases

- ▶ Clientela es el uso de un objeto de otra clase como atributo
- ▶ Las relaciones de herencia o clientela por su parecido podrían usarse erróneamente
- ▶ **Herencia:** los objetos de la clase derivada **deben poder ser** objetos de la clase base
- ▶ **Clientela:** los objetos de la clase cliente **tienen** (como atributos) uno o varios objetos de la clase usada

# Herencia en Java

- ▶ Todas las clases de Java derivan de otra. Si no se especifica derivan de la clase **Object** del paquete `java.lang`
- ▶ **Object** es la clase raíz de toda la jerarquía de clases
- ▶ No es posible heredar de una clase con el modificador **final**
- ▶ No es posible que una clase herede de varias a la vez (herencia múltiple)

# Herencia en Java

- ▶ Formato de herencia:

```
[...] class SubClase extends SuperClase {  
    ...  
}
```

- ▶ Ejemplo:

```
public class Alumno extends Persona {  
    private int NumExpediente;  
    public Alumno(...){  
        ...  
    }  
}
```

# La subclase

- ▶ Hereda todos los métodos y atributos (los privados no se pueden usar)
- ▶ Puede redefinir (reemplazar) los métodos heredados (exceptuando los **final** o **static**)
- ▶ Puede añadir nuevos métodos y atributos
- ▶ Los métodos redefinidos pueden aumentar su accesibilidad pero no restringirla. Por ejemplo: un método **protected** puede hacerse **public** pero no **private**



# La subclase

- ▶ Los constructores **no se heredan**
- ▶ En los constructores se puede llamar a los de la clase base con **super**(parm1, parm2, ...)
- ▶ Puede llamar a los métodos redefinidos de la superclase con **super.método** (parámetros ...)

# Ejemplo de uso de super

```
public class Alumno extends Persona {  
    private int NumExpediente;  
    public Alumno(String nombre,  
                   String telefono, int NumExpediente){  
        super(nombre, telefono);  
        this.NumExpediente= NumExpediente;  
    }  
    public String toString(){  
        String res = super.toString();  
        res += " (" + NumExpediente + ")";  
        return res;  
    }  
}
```

# Asignando referencias (subclase => superclase)

- ▶ Es posible asignar referencias de distintas clases
- ▶ Estas asignaciones **no cambian los objetos** referenciados, sólo la forma en que los usamos
- ▶ **Siempre** es posible asignar una referencia de una clase A a otra de una clase B si B es super clase directa o indirectamente de A
- ▶ Ejemplo:

```
Base refB;  
Derivada refA; // Derivada es subclase de Base  
...  
refB = refA;   // Asignación segura
```

# Asignando referencias (superclase => subclase) (1 / 2)

- ▶ Es posible realizar asignaciones en el sentido subclase a subclase siempre que el objeto implicado pertenezca a la clase destino o sea subclase de ésta
- ▶ El requisito anterior no puede ser garantizado durante la compilación como ocurre en la asignación de referencias de subclase a superclase
- ▶ Si al ejecutar la asignación no se cumple el requisito anterior se lanzará una excepción

# Asignando referencias (superclase => subclase) (2/2)

- ▶ Para evitar asignaciones de este tipo por error el compilador **nos exige** que usemos un cast
- ▶ Es frecuente usar **instanceof** para comprobar que la operación tendrá el resultado esperado
- ▶ Ejemplo:

```
CBase refB;  
CDerivada refA; // CDerivada es subclase de CBase  
...  
[if (refB instanceof CDerivada) {]  
refA = (CDerivada)refB; // Puede lanzar excepción
```

# Polimorfismo

- ▶ Nos permite manejar objetos de distintas clases de manera uniforme y simple
- ▶ El polimorfismo permite que en una llamada a un método se ejecute el **método** de la clase a la que pertenece el **objeto**, no el de la clase a la que pertenece la **referencia**
- ▶ Junto con la posibilidad de referenciar objetos de distinta clase se usa como herramienta potente de abstracción

# Clases abstractas (1 / 2)

- ▶ Cuando una clase se crea con la idea de que sea clase base de otras, sus métodos hacen de interfaz común de las derivadas
- ▶ En este caso es posible que se desee tener métodos sin implementación. La implementación la harán las clases derivadas.
- ▶ Estos métodos se declaran con el modificador **abstract**

# Clases abstractas (2 / 2)

- ▶ Una clase es abstracta y se le debe aplicar el modificador **abstract** cuando tiene algún método abstracto
- ▶ No es posible crear objetos de clases abstractas pero sí referencias a éstas
- ▶ Las clases derivadas seguirán siendo abstractas mientras no implementen todos los métodos abstractos de la clase base



# Ejemplo de herencia (1 / 4)

```
public abstract class FiguraGeométrica {  
    protected float altura;  
    public FiguraGeométrica(float altura) {  
        this.altura = altura;  
    }  
    public abstract float perímetroBase();  
    public abstract float superficieBase();  
    public abstract String nombre();  
    public float volumen() {  
        return altura * superficieBase();  
    }  
}
```

# Ejemplo de herencia (2/4)

```
public class Cilindro extends FiguraGeométrica {  
    protected float radio;  
    public Cilindro(float radio, float altura) {  
        super(altura);  
        this.radio = radio;  
    }  
    public float perímetroBase() {  
        return 2 * (float) Math.PI * radio;  
    }  
    public float superficieBase() {  
        return (float) Math.PI * radio * radio;  
    }  
    public String nombre() {  
        return "Cilindro";  
    }  
}
```

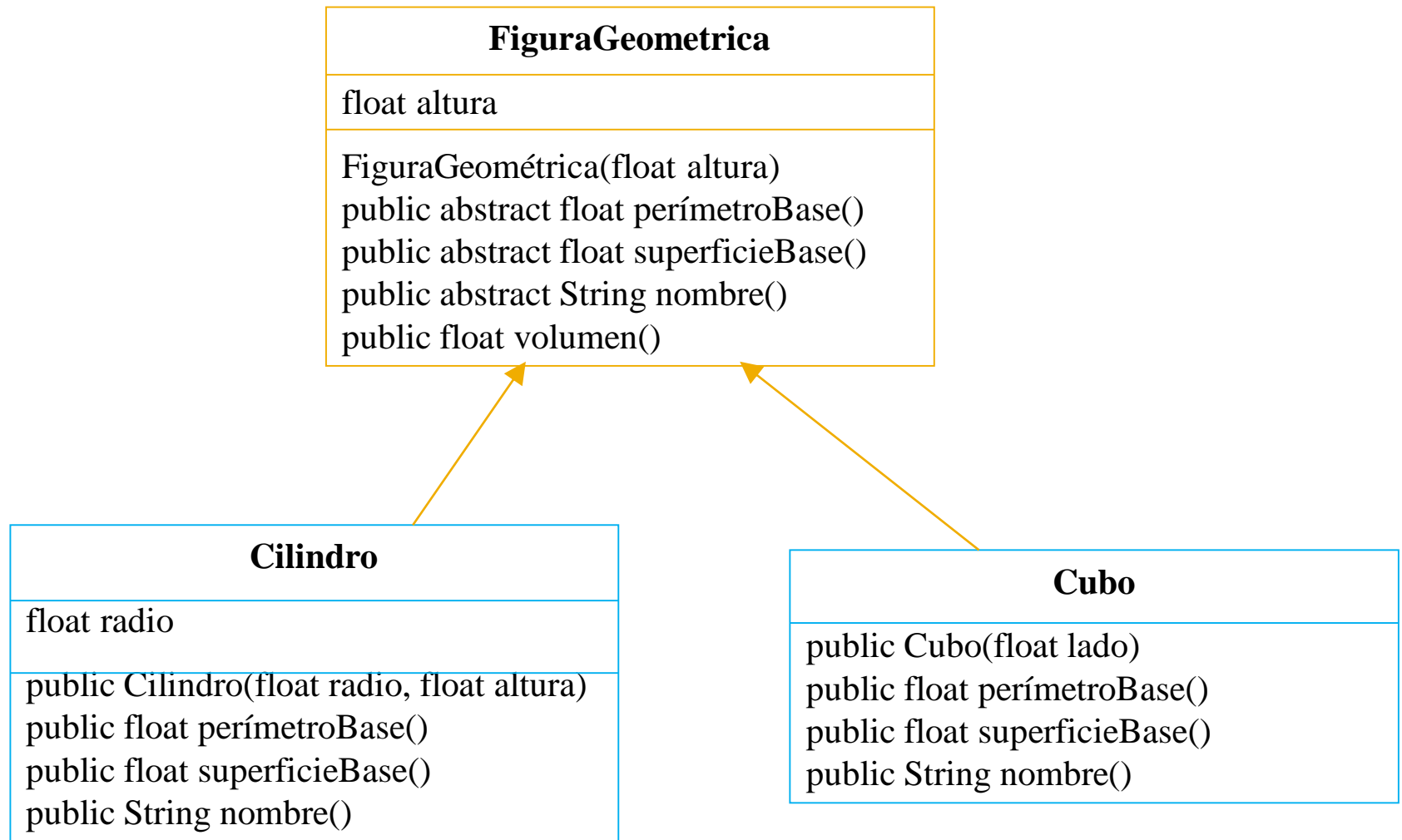
# Ejemplo de herencia (3 / 4)

```
public class Cubo extends FiguraGeométrica {  
    public Cubo(float lado) {  
        super(lado);  
    }  
    public float perímetroBase() {  
        return 4 * altura;  
    }  
    public float superficieBase() {  
        return altura * altura;  
    }  
    public String nombre() {  
        return "Cubo";  
    }  
}
```

# Ejemplo de herencia (4/4)

```
public class Herencia {  
    public static void main(String[] args) {  
        FiguraGeométrica f;  
        if (Math.random() < 0.5) {  
            f = new Cilindro(1, 2);  
        } else {  
            f = new Cubo(2);  
        }  
        System.out.println("El volumen del " + f.nombre() + " es "  
                            + f.volumen());  
    }  
}
```

# Diagrama de clases



# Interfaces

- ▶ Una **interface** es una clase especial que no tiene atributos y no implementa ningún método (excepto los **static**)
- ▶ A los efectos es una clase abstracta sin atributos y con todos sus métodos abstractos
- ▶ Se usa para establecer qué se debe hacer pero no cómo (implementación)
- ▶ Las clases que implementan la interfaz son las responsables de definir la implementación

# Interfaces

- ▶ Una ventaja de la interfaces es que una clase puede heredar de otra clase e implementar una o varias interfaces permitiendo una pseudo herencia múltiple
- ▶ Se pueden tener referencias a interfaces pero no objetos
- ▶ Estas referencias de interfaces permiten acceder a objetos de cualquier clase que la implemente

# Definiendo una interface

- ▶ Una interface se define igual que una clase pero con **interface** en vez de **class**
- ▶ Los métodos no tienen bloque de ejecución, se escriben terminando con ";"

▶ Formato:

```
[...] interface NombreInterface {  
    [public] Tipo método1(parm ...);  
    [public] Tipo método1(parm ...);  
    ...  
}
```



# Implementando una interfaz

Formato de herencia completo en la definición de una nueva clase:

```
TipoAccesibilidad class NombreClase  
    [extends ClaseBase]  
    [implements Interface1, Interface2, ...]{  
        ...  
}
```

- ▶ Se deben implementar todos los métodos de las interfaces, si no, la clase será abstracta

# Ejemplo interfaz (1 / 3)

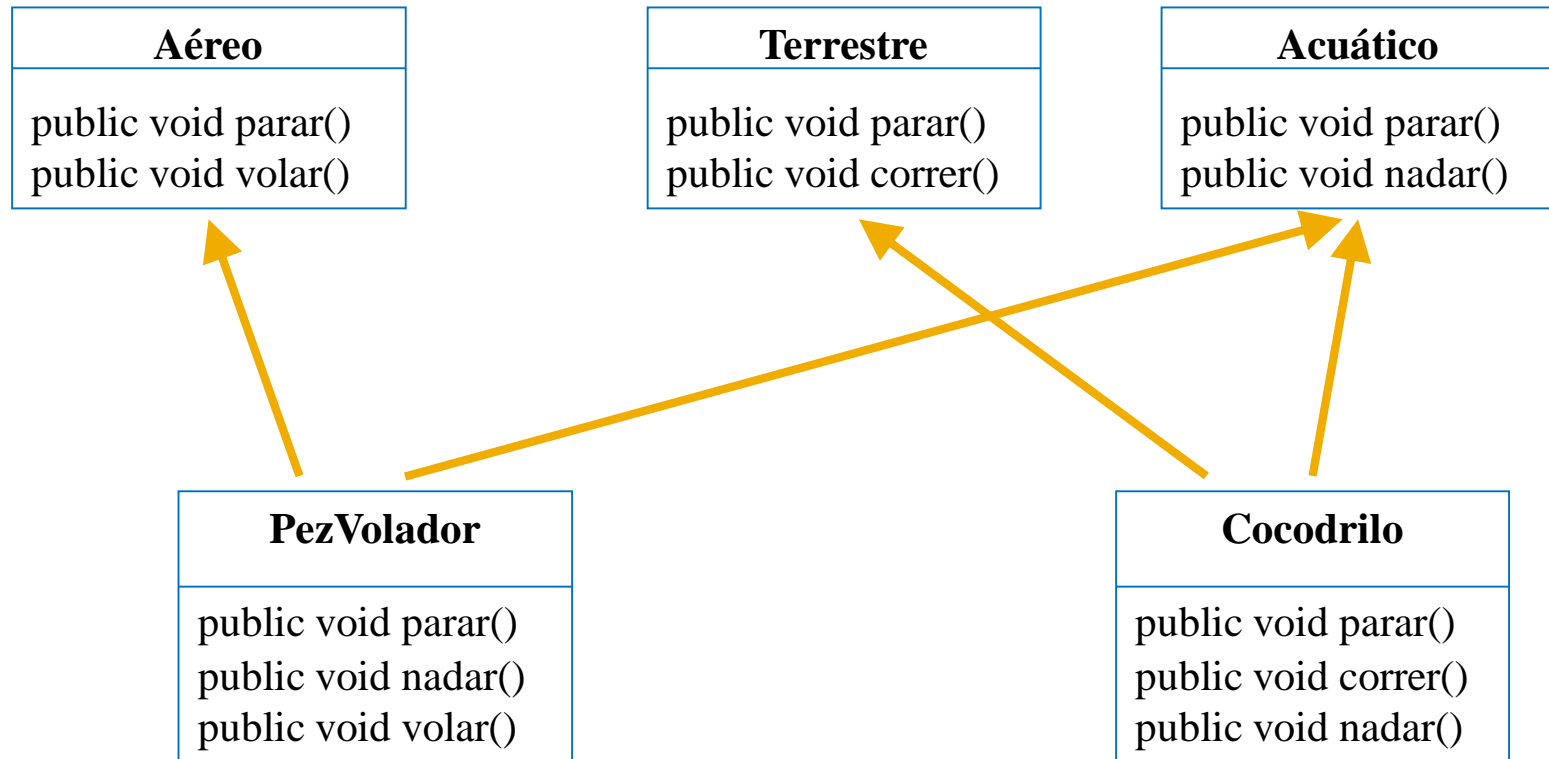
```
interface Aéreo {  
    public void parar();  
    public void volar();  
}  
interface Terrestre {  
    public void parar();  
    public void correr();  
}  
interface Acuático {  
    public void parar();  
    public void nadar();  
}
```

# Ejemplo interfaz (2 / 3)

```
class PezVolador implements Aéreo, Acuático{  
    private final static String ini="El pez volador";  
    public void parar() { System.out.println(ini+" para");}  
    public void volar() { System.out.println(ini+" vuela");}  
    public void nadar() { System.out.println(ini+" nada");}  
}
```

```
class Cocodrilo implements Acuático, Terrestre{  
    private final static String ini="El pez terrestre";  
    public void parar() { System.out.println(ini+" para"); }  
    public void nadar() { System.out.println(ini+" nada"); }  
    public void correr() { System.out.println(ini+" corre"); }  
}
```

# Diagrama de interfaces/clases



# Ejemplo interfaz (3 / 3)

```
public class Interfaz {  
    public static void main(String[] args) {  
        Cocodrilo juancho= new Cocodrilo();  
        PezVolador nemo= new PezVolador();  
        Acuático pez1, pez2;  
        Aéreo pájaro;  
        juancho.parar(); juancho.nadar(); juancho.correr();  
        nemo.parar(); nemo.nadar(); nemo.volar();  
        pez1 = nemo; pez1.parar(); pez1.nadar();  
        pez2 = juancho; pez2.parar(); pez2.nadar();  
        pájaro =(Aéreo) pez1;  
        pájaro.volar();  
        pájaro=(Aéreo) pez2;  
    }  
}
```