


Métodos Avanzados de Programación Científica y Computación

M^a Luisa Díez Platas

Tema 2. Relaciones entre clases (I)

¿Cómo estudiar este tema?

| IDEAS CLAVE | LO + RECOMENDADO | + INFORMACIÓN | TEST |
|---------------------------------|--|--|---|
| ¿Cómo estudiar este tema? | No dejes de leer... | A fondo |  |
| Abstracción y herencia | Herencia | Ejemplo de herencia en Java | |
| Conceptos avanzados de herencia | Paquetes en Java | Ejemplo de polimorfismo en Java | |
| Polimorfismo | Herencia en Java | UML Distilled | |
| Composición y agregación | No dejes de ver... | Programación orientada a objetos usando Java | |
| This y super |  Proyecto en ArgoUML | | |
| Complejidad de un algoritmo |  Herencia en Java | | |
| |  Polimorfismo en Java | | |

- Abstracción y herencia
- Polimorfismo
- Agregación
- Conceptos relacionados con los métodos

Características de los métodos de las clases. this

- **this**, argumento oculto de todo métodos no estático de una clase
- Es una referencia al objeto que va a soportar la operación

```
class Persona{  
    //definicion de atributos  
    private int edad;  
    //definicion de métodos  
    public Persona(){this.edad=0;}  
    public Persona(int ed;){this.edad=ed;}  
    public getEdad(){return this.edad;}  
    public setEdad(int ed){this.edad=ed;}  
}
```

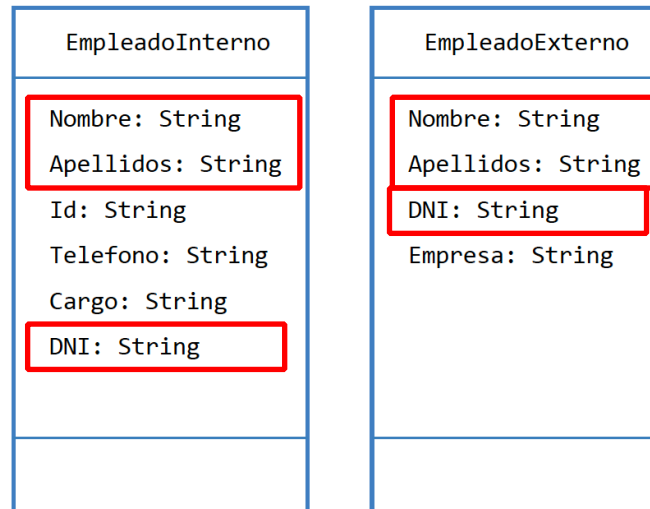
public setEdad(int edad){this.edad=edad;}

Persona p1=new Persona(32);
p1.getEdad();

Necesario resolver

Herencia

- La herencia implica un proceso de abstracción
- Generalización/especialización

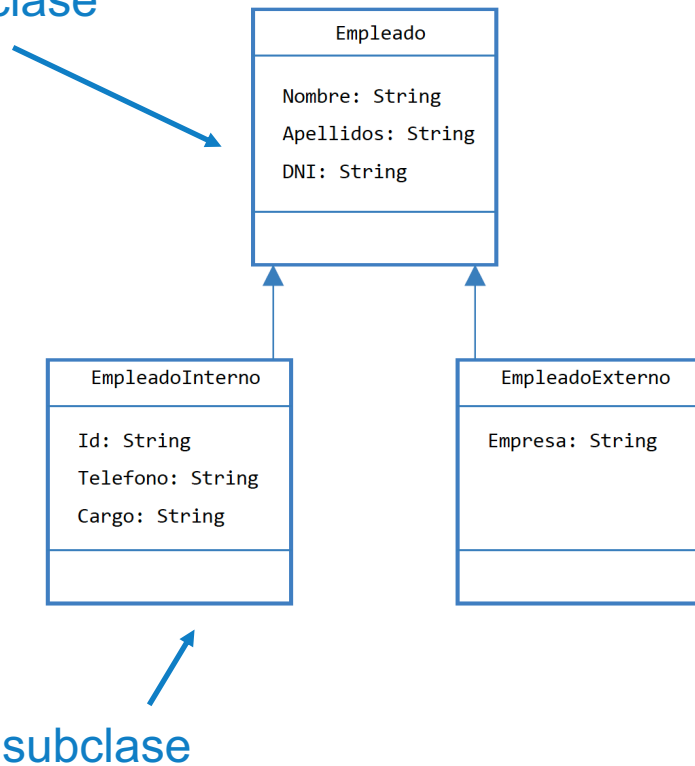


Característica para definir relaciones jerárquicas → proceso por el que una subclase recibe todos los atributos y métodos de una clase superior

Herencia.

- La clase base contiene los atributos comunes a las clases derivadas

superclase



```
class Empleado{
    private String Nombre;
    private String Apellidos;
    private String DNI;
}

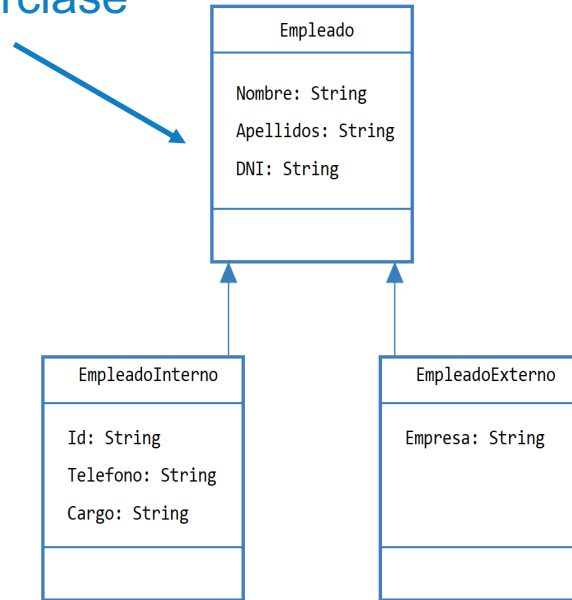
class EmpleadoInterno extends Empleado{
    private String Id;
    private String Telefono;
    private String Cargo;
}

class EmpleadoExterno extends Empleado{
    private String Empresa;
}
```

Herencia. private

- La clase base contiene los atributos comunes a las clases derivadas

superclass



subclase

```
EmpleadoInterno ei=new EmpleadoInterno(...);
String nombre=ei.getNombre(); ✓ Hereda el método
nombre=ei.Nombre; X no puede acceder es privado
ei.imprimirDatos();
```

```
class Empleado{
    private String Nombre;
    private String Apellidos;
    private String DNI;
    public String getNombre(){return Nombre;}
    Public String getApellidos(){return Apellidos;}
}
```

```
class EmpleadoInterno extends Empleado{
    private String Id;
    private String Telefono;
    private String Cargo;
    public String obtenerNombre(){return getNombre();}
    public void imprimirDatos(){
        System.out.println("nombre y apellidos: "+getNombre()+" "+getApellidos());
    }
}
class EmpleadoExterno extends Empleado{
    private String Empresa;
}
```

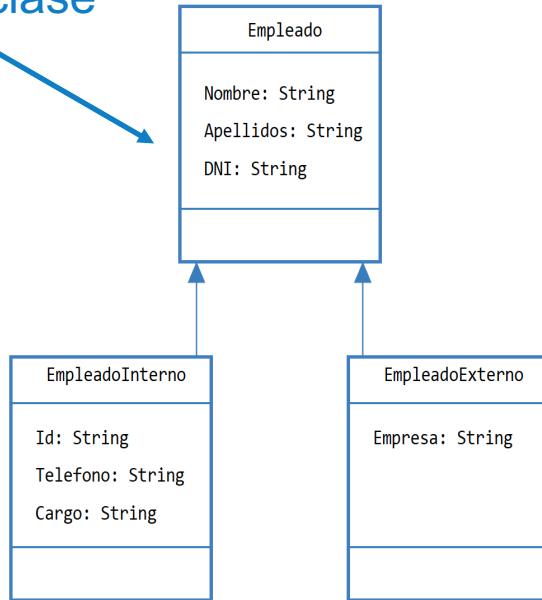
No es necesario este método pero no puede acceder al Nombre

Acceso a través del método de la superclase

System.out.println("nombre y apellidos: "+Nombre
"+Apellidos); X privados para la subclase

Herencia. protected

superclass



subclase

```
class Empleado{
    protected String Nombre;
    private String Apellidos;
    private String DNI;
    public String getNombre(){return Nombre;}
    Public String getApellidos(){return Apellidos;}
}
```

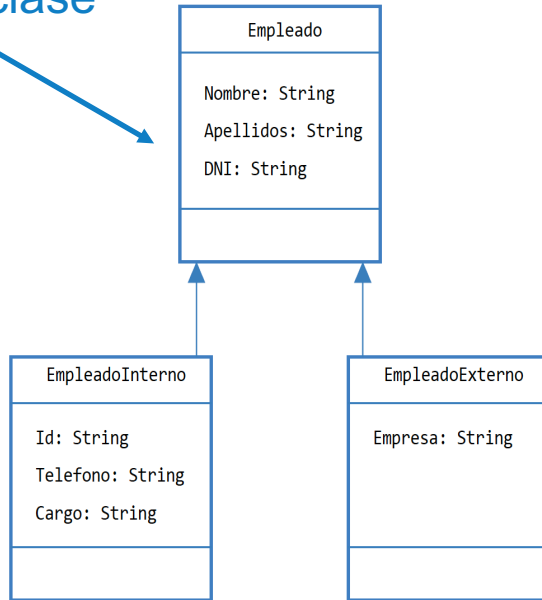
```
class EmpleadoInterno extends Empleado{
    private String Id;
    private String Telefono;
    private String Cargo;
    public void imprimirDatos(){
        System.out.println("nombre y apellidos: "+Nombre+" "+getApellidos());
    }
}
class EmpleadoExterno extends Empleado{
    private String Empresa;
}
```

Es accesible por la subclase

```
EmpleadoInterno ei=new EmpleadoInterno(...);
String nombre=ei.getNombre(); ✓ Hereda el método
nombre=ei.Nombre; X no puede acceder es protected (solo accesible en la clase en la que se define o en las subclases)
ei.imprimirDatos();
```

Herencia. public

superclass



subclase

```
class Empleado{
    publica String Nombre;
    private String Apellidos;
    private String DNI;
    public String getNombre(){return Nombre;}
    Public String getApellidos(){return Apellidos;}
}
```

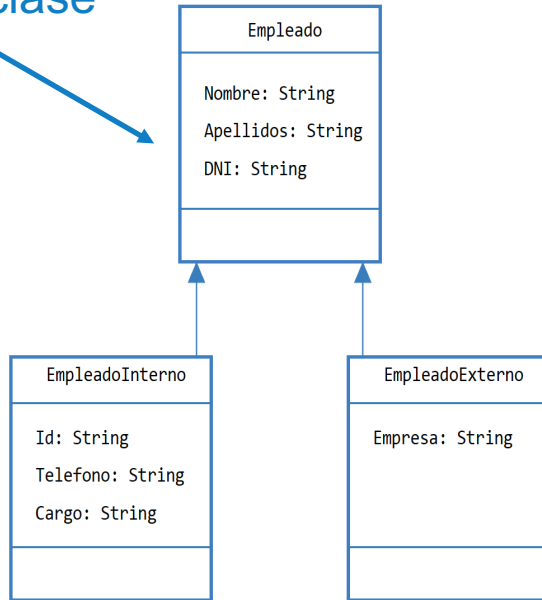
```
class EmpleadoInterno extends Empleado{
    private String Id;
    private String Telefono;
    private String Cargo;
    public void imprimirDatos(){
        System.out.println("nombre y apellidos: "+Nombre+" "+getApellidos());
    }
}
class EmpleadoExterno extends Empleado{
    private String Empresa;
}
```

Es accesible por la subclase

```
EmpleadoInterno ei=new EmpleadoInterno(...);
String nombre=ei.getNombre(); ✓ Hereda el método
nombre=ei.Nombre; ✓ puede acceder es public
ei.imprimirDatos();
```


Herencia. super()

superclass



subclass

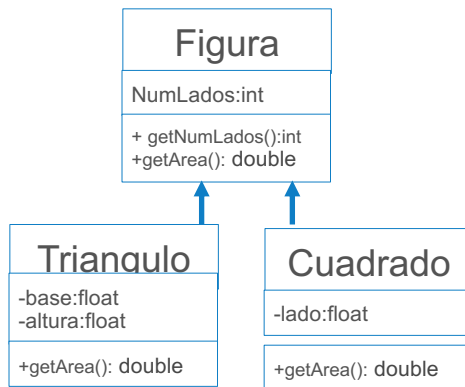
```
class Empleado{
    private String Nombre;
    private String Apellidos;
    private String DNI;
    public Empleado(String n, String a; String d){
        Nombre=n;
        Apellidos=a;
        DNI=d;
    }
    public String getNombre(){return Nombre;}
}

class EmpleadoExterno extends Empleado{
    String Empresa;
    public EmpleadoExterno(String n, String a; String d,String e){
        super(n,a,d);
        Empresa=e;
    }
}
```

```
EmpleadoExterno ee1=new EmpleadoExterno ("Jose","Garcia","22222X","Aentar");
```

Herencia. Clases abstractas

No tiene implementado alguno de los métodos
No se pueden crear ejemplares → no hacer new de la clase



```
abstract class Figura{
    private int NumLados;
    public int getNumLados(){return NumLados;}
    public abstract double getArea();
}

class Triangulo extends Figura{
    private double base;
    private double altura;
    public double getArea(){return base*altura/2;}
}
```

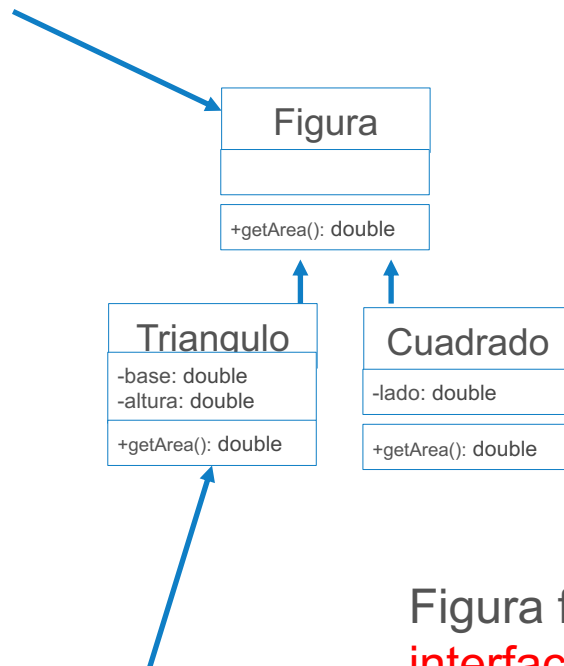
`Figura f=new Figura();` **X no se puede, la clase Figura es abstracta**

`Triangulo t=new Triangulo();` **✓ La clase Triangulo no es abstracta porque se ha implementado el método abstracto**

Interfaces

No tiene implementado ninguno de los métodos (abstracta pura)
No se pueden crear ejemplares → no hacer new de la clase

interface



Implementan
la interface

```
interface Figura{

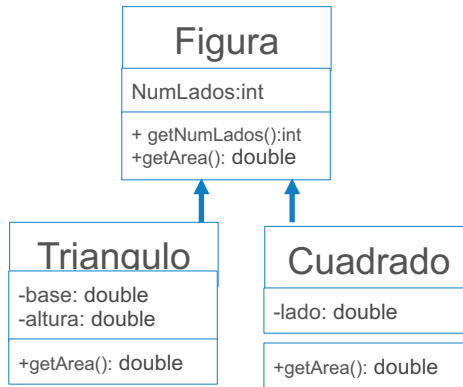
    public double getArea();
}

class Triangulo implements Figura{
    private double base;
    private double altura;
    public float getArea(){return base*altura/2;}
}
```

`Figura f=new Figura();` X no se puede, Figura es una interface

`Triangulo t=new Triangulo();` ✓ La clase Triangulo implementa la interfaz

Herencia. Polimorfismo



capacidad de ejecutar un método que cambia de comportamiento dependiendo de como ha sido instanciado el objeto con el que se invoca

```
abstract class Figura{
    private int NumLados;
    public int getNumLados(){return NumLados;}
    public abstract double getArea();
}
class Triangulo extends Figura{
    private double base;
    private double altura;
    public Triangulo(double b, double a,int n){
        super(n);
        base=b;altura=a;
    }
    public double getArea(){return base*altura/2;}
}
```

```
class Cuadrado extends Figura{
    public double lado;
    public Cuadrado(double l, int n){
        super(n);
        lado=l;
    }
    public double getArea(){return lado*lado;}
}
```

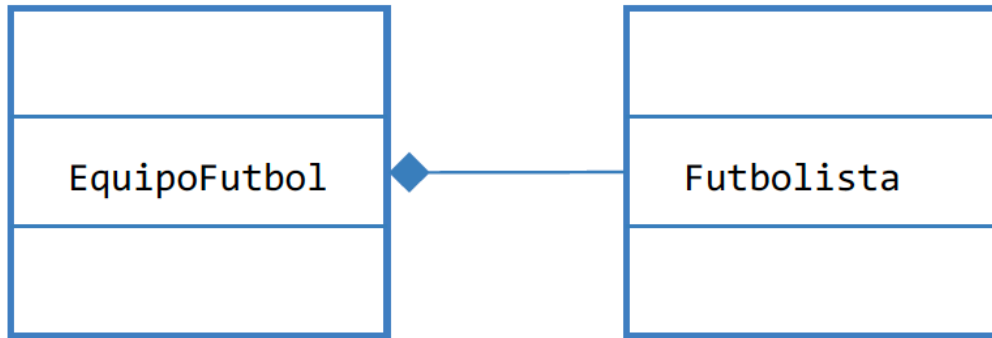
```
Figura f= new Triangulo(2.0,3.0,3);
f.getArea(); //invoca al método de triangulo (enlace en tiempo de ejecución)
f= new Cuadrado(2.0,4);
f.getArea(); //invoca al método de cuadrado (enlace en tiempo de ejecución)
```

Ligadura tardía

Ligadura temprana

```
Triangulo t=new Triangulo(2.0,3.0,3);
t.getArea(); //invoca al método de triangulo , enlace en tiempo de compilación
```

Agregación



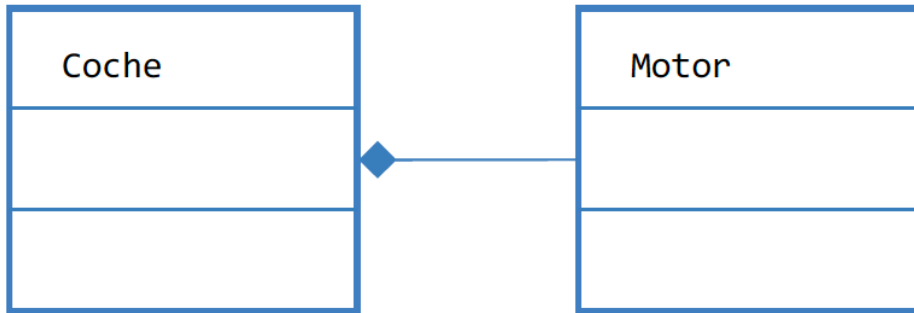
```
class Futbolista{
//....
public Futbolista(.....){//.....}
class EquipoFutbol {
    private Futbolista[] futbolistas;
public EquipoFutbol(.....){
    futbolistas=new Futbolista[20];
}
}
```

Se crea el array pero no cada una de los futbolistas

- Una clase esta formada por objetos de otra
- Los objetos contenidos no **se crean necesariamente al crear el objeto de la contenedora**
- La destrucción del objeto de la contenedora **no implica la destrucción de los objetos contenidos**

Composición débil

Composición



```
class Motor{
//...
public Motor(.....){//.....}
}

class Coche {
    private Motor elMotor;
public Coche(.....){
    elMotor=new Motor(...);
}
}
```

An arrow points from the text 'Se crea el objeto contenido al mismo tiempo que el contenedor' to the line `elMotor=new Motor(...);` in the Coche class definition.

Se crea el objeto contenido al mismo tiempo que el contenedor

- Una clase esta formada por objetos de otra
- Los objetos **contenidos se crean al crear el objeto de la contenedora**
- La destrucción del objeto de la contenedora **implica la destrucción de los objetos contenidos**

UNIVERSIDAD
INTERNACIONAL
DE LA RIOJA

unir

www.unir.net