



# *Programación de Sistemas*

## *Programación Orientada a Objetos*

Julio Villena Román  
<jvillena@it.uc3m.es>

MATERIALES BASADOS EN EL TRABAJO DE DIFERENTES AUTORES:  
M.Carmen Fernández Panadero, Raquel M. Crespo García  
Carlos Delgado Kloos, Natividad Martínez Madrid





# *Programación de Sistemas*

## *Programación BASADA en Objetos*

Julio Villena Román  
<jvillena@it.uc3m.es>

MATERIALES BASADOS EN EL TRABAJO DE DIFERENTES AUTORES:  
M.Carmen Fernández Panadero, Raquel M. Crespo García  
Carlos Delgado Kloos, Natividad Martínez Madrid



# Escenario IV:

## Declarar e implementar una clase

- Ahora que ya sabes interpretar código e implementar tus propios métodos te encargan el diseño de una clase completa para crear un nuevo tipo de datos con sus características y comportamiento.

- **Objetivo:**
  - Ser capaz de **declarar una clase** con un conjunto de características (**atributos**) y comportamientos (**métodos**)
  - Ser capaz de **crear objetos** de una clase dada y modificar o restringir el acceso a su estado y su comportamiento
- **Plan de trabajo:**
  - Memorizar la **nomenclatura** básica de la programación orientada a objetos
  - Practicar el **modelado** de objetos con ejemplos sencillos para distinguir entre una clase, un objeto, su estado y su comportamiento
  - Repasar la **sintaxis** java para declarar **clases, atributos, constructores y métodos**
  - Recordar el mecanismo y la sintaxis para **paso de mensajes** entre objetos



# Objetivos



- Definir los **conceptos básicos** de la programación **basada** en objetos
  - Clases, objetos
  - Miembros (atributos, métodos)
  - Abstracción y ocultación de información
- Describir **relación** entre objeto y clase
- **Crear** un objeto sencillo y **modelar**
  - sus características (por medio de atributos)
  - su comportamiento (por medio de métodos)

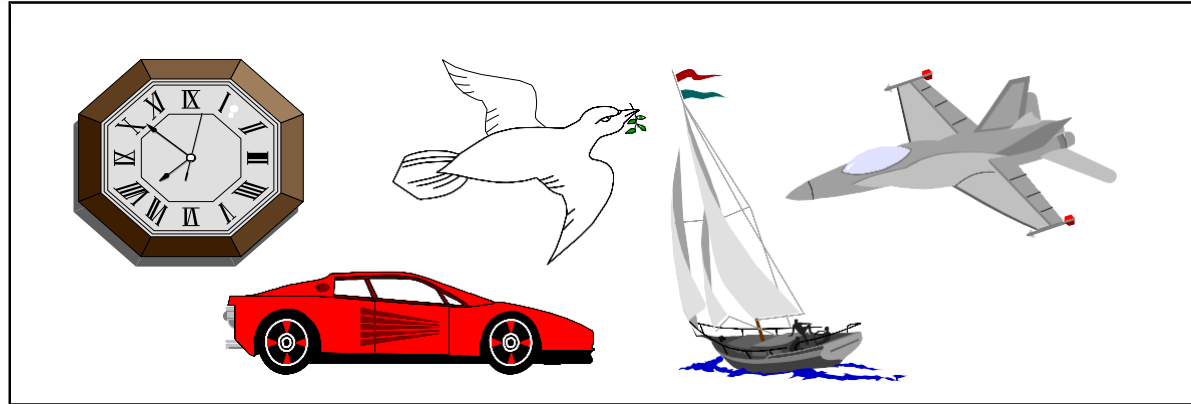




- ❖ Clases y objetos
- ❖ Encapsulación de objetos
  - ❖ Abstracción funcional
  - ❖ Abstracción de datos
- ❖ Miembros de una clase (atributos y métodos)
- ❖ Paso de mensajes
- ❖ Sobrecarga de métodos
- ❖ Constructores
- ❖ Modificadores y acceso

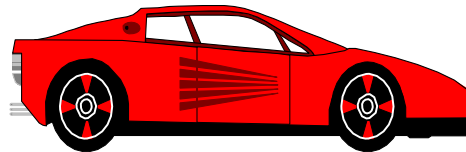


# ¿Qué es un objeto?



- Los **objetos** son representaciones (simples/complejas) (reales/imaginarias) de cosas: *reloj, avión, coche*
- No todo puede ser considerado como un objeto, algunas cosas son simplemente características o **atributos** de los objetos: *color, velocidad, nombre*

# ¿Qué es un objeto?



- **Abstracción funcional**

- Hay cosas que sabemos que los coches hacen pero no cómo lo hacen:

- avanzar
    - parar
    - girar a la derecha
    - girar a la izquierda

- **Abstracción de datos**

- Un coche tiene además ciertos atributos:

- color
    - velocidad
    - tamaño
    - etc.

# ¿Qué es un objeto?



- Es una forma de agrupar un conjunto de datos (**estado**) y de funcionalidad (**comportamiento**) en un mismo bloque de código que luego puede ser referenciado desde otras partes de un programa
- La **clase** a la que pertenece el objeto puede considerarse como un nuevo **tipo de datos**





# Ejemplo



## Clase

## Objetos

```
public class Coche {  
    private String color;  
    private int velocidad;  
    private float tamaño;  
  
    public Coche (String color, int velocidad, float tamaño){  
        this.color = color;  
        this.velocidad = velocidad;  
        this.tamaño = tamaño;  
    }  
  
    public void avanzar(){}  
    public void parar(){}  
    public void girarIzquierda(){}  
    public void girarDerecha(){}  
}
```

Estado

Constructor

Comportamiento

```
public static void main (String[] args){  
    Coche miCoche = new Coche ("verde", 80, 3.2f);  
    Coche tuCoche = new Coche ("rojo", 120, 4.1f);  
    Coche suCoche = new Coche ("amarillo", 100, 3.4f);  
}
```



- **this** referencia al objeto de la clase actual

# Ejercicio 1



- Implementa la clase **Bicicleta**, que tiene tres atributos, **velocidadActual**, **platoActual** y **piñonActual**, de tipo entero y cuatro métodos **acelerar()**, **frenar()**, **cambiarPlato(int plato)**, y **cambiarPiñon(int piñon)**, donde el primero dobla la velocidad actual, el segundo reduce a la mitad la velocidad actual, y el tercero y cuarto ajustan el plato y el piñón actual respectivamente según los parámetros recibidos. La clase debe tener además un constructor que inicialice todos los atributos.
- Crea dos objetos de la clase bicicleta: **miBicicleta** y **tuBicicleta**



# Encapsulación de objetos



- **Encapsulación:** describe la vinculación de un comportamiento y un estado a un objeto en particular.
- **Ocultación de información:** Permite definir qué partes del objeto son visibles (el interfaz público) que partes son ocultas (privadas)

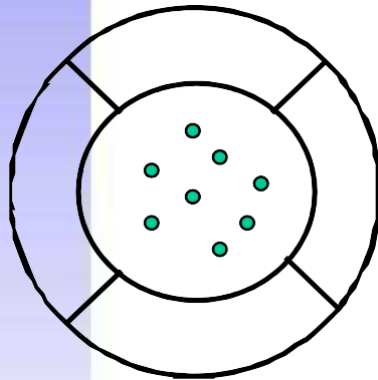


- La llave de contacto es un interfaz público del mecanismo de arranque de un coche
- La implementación de cómo arranca realmente es privada y sobre ella sólo puede actuar la llave de contacto

**ventajas**

El objeto puede cambiar y su interfaz pública ser compatible con el original: esto facilita reutilización de código

# Encapsulación de objetos



## MIEMBROS DE UNA CLASE

Los objetos encapsulan atributos permitiendo acceso a ellos únicamente a través de los métodos

- ▶ **Atributos (Variables):** Contenedores de valores
- ▶ **Métodos:** Contenedores de funciones

Un objeto tiene

- ▶ **Estado:** representado por el contenido de sus atributos
- ▶ **Comportamiento:** definido por sus métodos



Normalmente:

- ▶ Los métodos son públicos
- ▶ Los atributos son privados
- ▶ Puede haber métodos privados
- ▶ Es peligroso tener atributos públicos

# Definición de objetos



## *Miembros públicos*

- los miembros públicos describen **qué** pueden hacer los objetos de esa clase
  - Qué pueden hacer los objetos (métodos)
  - Qué son los objetos (su abstracción)

## *Miembros privados*

- describen la implementación de **cómo** lo hacen.
  - Ejemplo: el objeto contacto interacciona con el circuito eléctrico del vehículo, este con el motor, etc.
  - ***En sistemas orientados a objetos puros todo el estado es privado y sólo se puede cambiar a través del interfaz público.***
  - Ej: El método público frenar puede cambiar el valor del atributo privado velocidad.



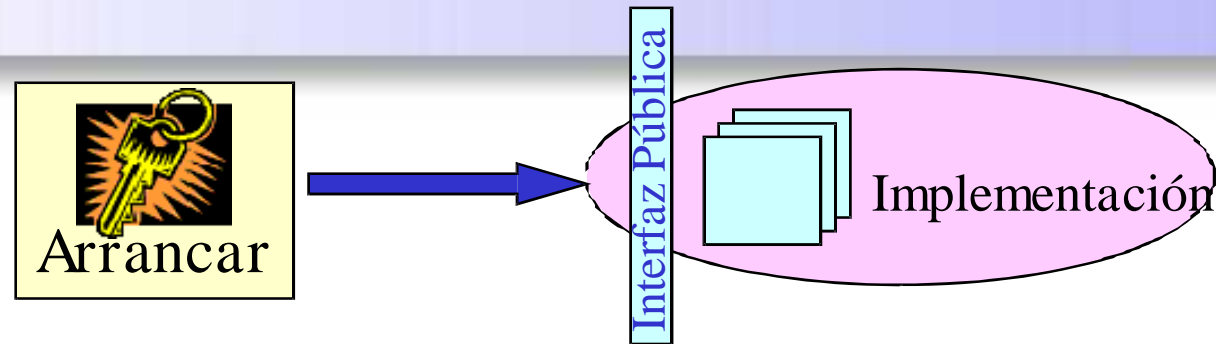
# Interacciones entre objetos



- El ***modelado de objetos*** modela:
  - Los objetos y
  - Sus interrelaciones
- Para realizar su tarea el objeto puede ***delegar*** trabajos en otro que puede ser parte de él mismo o de cualquier otro objeto del sistema.
- Los objetos interaccionan entre sí enviándose ***mensajes***



# Paso de Mensajes



- Un objeto envía un *mensaje* a otro
  - Esto lo hace mediante una **llamada** a sus atributos o métodos
- Los mensajes son tratados por la **interfaz pública** del objeto que los recibe
  - Eso quiere decir que sólo podemos hacer llamadas a aquellos atributos o métodos de otro objeto que sean **públicos o accesibles** desde el objeto que hace la llamada
- El objeto receptor reaccionará
  - **Cambiando su estado**: es decir modificando sus atributos
  - **Enviando otros mensajes**: es decir llamando a otros atributos o métodos del mismo objeto (públicos o privados) o de otros objetos (públicos o accesibles desde ese objeto)

# Ejemplo



## Clase Coche

```
public class Coche {  
  
    private String color;  
    private int velocidad;  
    private float tamaño;  
    private Rueda[] ruedas;  
    private Motor motor;  
  
    public Coche (String color, int velocidad,  
                  float tamaño, Rueda[] ruedas,  
                  Motor motor){  
        this.color = color;  
        this.velocidad = velocidad;  
        this.tamaño = tamaño;  
        this.ruedas = ruedas;  
        this.motor = motor;  
    }  
  
    public void avanzar(){  
        motor.inyectarCarburante();  
        for (int i=0; i < ruedas.length; i++){  
            ruedas[i].girar();  
        }  
    }  
  
    public static void main (String[] args){  
  
        Rueda[] ruedas = {new Rueda(20,"Dunlop"),  
                           new Rueda(20,"Dunlop"),  
                           new Rueda(22, "Dunlop"),  
                           new Rueda(22, "Dunlop")};  
        Coche miCoche = new Coche ("verde", 80,3.2f,  
                                     ruedas, new Motor("Diesel",100));  
    }  
}
```

Paso de  
mensajes

## Clase Motor

```
public class Motor {  
  
    private String tipo;  
    private int caballos;  
  
    public Motor(String tipo, int caballos){  
        this.tipo = tipo;  
        this.caballos = caballos;  
    }  
  
    public void inyectarCarburante(){...}  
}
```

## Clase Rueda

```
public class Rueda {  
  
    private double diametro;  
    private String fabricante;  
  
    public Rueda (double diametro, String fabricante){  
        this.diametro = diametro;  
        this.fabricante = fabricante;  
    }  
  
    public void girar(){...}  
}
```





# Clasificación de objetos



- **Clase:** Conjunto de objetos con estados y comportamientos similares
  - Podemos referirnos a la clase “Coche” (cualquier instancia de la clasificación coche)
- “Mi coche” es un **objeto**, es decir una **instancia** particular de la clase coche
- La clasificación depende del problema a resolver

# Objetos vs. Clases

Una **clase** es una entidad abstracta

- Es un tipo de clasificación de datos
- Define el comportamiento y atributos de un grupo de estructura y comportamiento similar

**Clase Coche**

Métodos: arrancar, avanzar, parar, ...

Atributos: color, velocidad, etc.

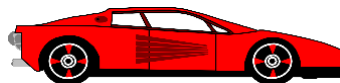
—————→ Nombre de la clase  
—————→ Métodos (funciones)  
—————→ Atributos (datos)

Un **objeto** es una instancia de una clase

- Un objeto se distingue de otros miembros de la clase por sus atributos

**Objeto Ferrari**

Perteneciente a la  
clase coche



Nombre: Ferrari  
Métodos: arrancar, avanzar, parar, ...  
Atributos: color = "rojo";  
velocidad 300Km/h

- Una **clase** se declara, un **objeto** además se crea

# Sobrecarga (Overloading)

¿Qué es?



- Podemos definir una clase con dos métodos con el mismo nombre si los argumentos son distintos.
- Se utiliza mucho para los constructores.
- Sabemos cual de los dos métodos tenemos que ejecutar por los parámetros que le pasamos cuando le llamamos.



# Sobrecarga (Overloading)

## ¿Para qué sirve?



### Clase

### Objetos

```
public class Coche {  
  
    private String color;  
    private int velocidad;  
    private float tamaño;  
  
    public Coche (String color, int velocidad, float tamaño){  
        this.color = color;  
        this.velocidad = velocidad;  
        this.tamaño = tamaño;  
    }  
  
    public void avanzar(){}  
    public void avanzar(int metros){}  
    public void avanzar(int metros, int velocidad){}  
  
    public void parar(){}  
    public void girarIzquierda(){}  
    public void girarDerecha(){}  
  
}
```

} Sobrecarga

```
public static void main (String[] args){  
    Coche miCoche = new Coche ("verde", 80, 3.2f);  
    Coche tuCoche = new Coche ("rojo", 120, 4.1f);  
    Coche suCoche = new Coche ("amarillo", 100, 3.4f);  
  
    miCoche.avanzar();  
    tuCoche.avanzar(1000);  
    suCoche.avanzar(1000,120);  
}
```

Son métodos distintos porque aunque tengan el mismo nombre tienen distintos argumentos. Tienen distinta funcionalidad

# Ejercicio 2



- Sobre la clase **Bicicleta**, implementa los método sobrecargados **cambiarPlato()**, y **cambiarPiñon()**, que no reciben argumentos y que cambian el plato actual y el piñón actual a un valor por defecto, en concreto, 1.



# Constructores



- Cuando se crea un objeto sus miembros se **inician** con un método constructor
- Los constructores:
  - llevan el **mismo nombre** que la clase
  - **No** tienen **tipo** de resultado (ni siquiera void)
- Conviene que haya al menos 1 constructor
- Pueden existir varios que se distinguirán por los parámetros que aceptan (**sobrecarga**)
- Si no existen se crea un **constructor por defecto** sin parámetros que inicializa las variables a su valor por defecto.
- Si la clase tiene algún constructor, el constructor por defecto deja de existir. En ese caso, si queremos que haya un constructor sin parámetros tendremos que declararlo explícitamente.



# Constructores



```
public class Coche {  
  
    private String color;  
    private int velocidad;  
    private float tamaño;  
  
    public Coche (){}  
  
    public Coche (String color){  
        this.color = color;  
    }  
  
    public Coche (String color, int velocidad){  
        this.color = color;  
        this.velocidad = velocidad;  
    }  
  
    public Coche (String color, int velocidad, float tamaño){  
        this.color = color;  
        this.velocidad = velocidad;  
        this.tamaño = tamaño;  
    }  
  
    public void avanzar(){}  
    public void parar(){}  
    public void girarIzquierda(){}  
    public void girarDerecha(){}  
}
```

Array de objetos de la clase Coche

```
public static void main (String[] args){  
  
    Coche[] concesionario = {new Coche("verde"), new Coche("rojo",120),  
                             new Coche("amarillo",100,3.2f)};  
  
}
```

Sobrecarga de constructores



# Ejercicio 3



- Sobre la clase **Bicicleta**, implementa un constructor adicional que no recibe parámetros y que inicializa la velocidad actual a 0, y el plato actual y el piñón actual a 1.





# Modificadores y acceso

## Static (miembros estáticos)



- Modificador **static**
- Sólo existen **una vez por clase**, independientemente del número de instancias (objetos) de la clase que hayamos creado y aunque no exista ninguna.
- El método o el atributo **se comportan siempre de la misma manera**
- Se puede acceder a los miembros estáticos utilizando el **nombre de la clase**.
- Un método estático **no** puede acceder a miembros no estáticos directamente, tiene que crear primero un objeto



# Modificadores y acceso

## Static (miembros estáticos)



### Atributo estático

```
public class Coche {  
    private String color;  
    private int velocidad;  
    private float tamaño;  
    private static int numeroRuedas = 4; }  
...  
    public static void main (String[] args){  
        System.out.println(Coche.numeroRuedas);  
    }  
}
```

Atributo estático

### Otros ejemplos

```
int radium = 3;  
double areaCircle = Math.PI * radium * radium;  
  
int minValue = Integer.MIN_VALUE; => -231  
int maxValue = Integer.MAX_VALUE; => 231-1
```

**Método estático:** Tiene acceso a atributos estáticos  
No necesitamos crear instancias

```
public static void main(String args[]) {  
    int x1 = Integer.parseInt(args[0]);  
    double y1 = Double.parseDouble(args[1]);  
}
```

```
Math.sqrt(100);  
Math.cos(76);
```

<http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>



# Modificadores y acceso

## Static. Algunas reglas



- Los miembros **estáticos** se invocan con:

```
NombreClase.metodoEstatico();  
NombreClase.atributoEstatico;
```

- Para acceder a los miembros no estáticos necesitamos disponer de una instancia (objeto) de la clase

```
NombreClase nombreObjeto = new NombreClase();
```

- Los miembros **no** estáticos se invocan con:

```
nombreObjeto.metodoNormal();  
nombreObjeto.atributoNormal;
```

- Cuando invocación (llamada) a un miembro estático de la clase se realiza dentro de la propia clase se puede omitir el nombre de la misma. Es decir podemos escribir:

```
metodoEstatico();  
atributoEstatico;
```

en lugar  
de:

```
NombreClase.metodoEstatico();  
NombreClase.atributoEstatico;
```



# Acceso

## Métodos get() y set()



- Los atributos de una clase son generalmente privados para evitar que puedan ser accesibles / modificables desde **cualquier otra clase**.
- A veces nos interesa que algunas clases determinadas sí puedan acceder a los atributos.
- Uso de métodos **get()** y **set()**


```
public class Coche {  
  
    private String color;  
    private int velocidad;  
  
    public void setColor(String color){  
        this.color = color;  
    }  
  
    public String getColor(){  
        return this.color;  
    }  
  
    public void setVelocidad(int velocidad){  
        this.velocidad = velocidad;  
    }  
  
    public int getVelocidad(){  
        return this.velocidad;  
    }  
}
```

# Ejercicio 4



- Sobre la clase **Bicicleta**, implementa los métodos **get()** y **set()** necesarios para poder acceder y modificar todos los atributos.



MODIFICADORES		clase	metodo	atributo
acceso	public	Accesible desde cualquier otra clase		
	(friendly)	Accesible sólo desde clases de su propio <b>paquete</b>		
	private		Accesibles sólo dentro de la clase	
otros	static	Clase de nivel máximo. Se aplica a clases internas	Es el mismo para todos los objetos de la clase.  Se utiliza: NombreClase.metodo();	Es la misma para todos los objetos de la clase.  Se utiliza: NombreClase.atributo;

# Paquetes



- Un *paquete* agrupa *clases* (e *interfaces*)
- Las jerarquías de un paquete se corresponden con las jerarquías de directorios
- Para referirse a miembros y clases de un paquete se utiliza la notación de separarlos por puntos.
  - Ej: Cuando importamos paquetes de clases matemáticas

```
import java.math.BigDecimal;
```

- La clase `BigDecimal` está en el directorio `java/math` dentro del JDK
- No es necesario importar todas las clases: paquete `java/lang`
  - `String`
  - `Integer`
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`



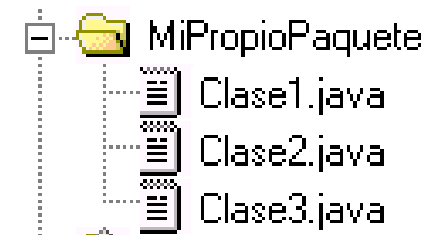
# Paquetes



- ***¿Cómo crear mis propios paquetes?***

- Almaceno mis clases en un directorio con el nombre del paquete
- Pongo al principio de todas las clases que pertenezcan al paquete la instrucción

```
package MiPropioPaquete;
```



- Si quiero importar las clases de ese paquete desde otras clases y/o proyectos, pongo al principio de cada clase

```
import MiPropioPaquete.Clase1
```





# *Programación de Sistemas*

## *Programación ORIENTADA a Objetos*

Julio Villena Román  
<jvillena@it.uc3m.es>

MATERIALES BASADOS EN EL TRABAJO DE DIFERENTES AUTORES:  
M.Carmen Fernández Panadero, Raquel M. Crespo García  
Carlos Delgado Kloos, Natividad Martínez Madrid



# Escenario V:

## Reutilizar código. Herencia

- Una vez que eres capaz de crear tus propias clases estás preparado para trabajar en equipo y reutilizar código de tus compañeros. Tu equipo te proporciona un conjunto de clases y te pide que crees especializaciones o generalizaciones de las mismas
- **Objetivo:**
  - Ser capaz de crear una **clase derivada** añadiendo algunas características (atributos) y comportamiento (métodos) a una clase existente.
  - Ser capaz de extraer todo el código común de un conjunto de clases similares para agruparlo en una nueva **clase padre** para que sea más fácil de mantener.
  - Ser capaz de **crear objetos**, y **referenciar** y **acceder** a sus atributos y métodos dependiendo de su posición en la jerarquía de herencia y sus modificadores
- **Plan de trabajo:**
  - Memorizar la **nomenclatura** relacionada con herencia
  - Memorizar la **sintaxis** de java relacionada con la herencia (**extends**), con la referencia (**super**) y con modificadores de acceso avanzados (**protected**)
  - Conocer mecanismos básicos de herencia como **ocultación** de atributos, **sobreescritura** de métodos ,saber para qué sirven y cómo se usan



# Contenidos

- ▶ Conceptos básicos de herencia
- ▶ Jerarquía de herencia
- ▶ Reescritura I: Ocultación de atributos
- ▶ Reescritura II: Redefinición de métodos
- ▶ Constructores de clases derivadas
- ▶ Polimorfismo
- ▶ El modificador final



# Herencia

## ¿Qué es? ¿Para qué sirve?



- Es un mecanismo para la **reutilización de software**
- Permite definir a partir de una clase otras clases relacionadas que supongan una:
  - **Especialización** (la clase Coche como especialización de Vehículo)
    - **Escenario:** Tenemos que desarrollar una nueva clase que se parece mucho a una que tenemos pero necesita información (características y comportamiento) adicional.
    - **Solución (subclase, clase hija o clase derivada):** Creamos una clase derivada y añadimos nueva funcionalidad sin tener que reescribir el código común.
  - **Generalización** (la clase Vehículo como generalización de coche).
    - **Escenario:** Tenemos un conjunto numeroso de clases muy similares con código que se repite y es difícil de actualizar y mantener (ejemplo hay que añadir una letra al número de serie)
    - **Solución (superclase, clase padre o clase base):** Movemos el código que se repite a un único sitio



# Herencia

## ¿Para qué sirve?



**Sin herencia**

Recurso

- nombre
- descripcion
- decirNombre()
- decirDescripcion()

Aula

- nombre
- descripcion
- localizacion
- decirNombre()
- decirDescripcion()
- decirLocalizacion()

Ordenador

- nombre
- descripcion
- sistemaOperativo
- decirNombre()
- decirDescripcion()
- decirSistemaOp()

**Con herencia**

Recurso

- nombre
- descripcion
- decirNombre()
- decirDescripcion()

Aula

- localizacion
- decirLocalizacion()

Ordenador

- sistemaOperativo
- decirSistemaOp()

```
public class Aula extends Recurso
public class Ordenador extends Recurso
```

Los atributos y métodos que aparecen en azul en la clase padre se repiten en las clases hijas [Izquierda]

No es necesario repetir el código, basta con decir que una clase **extiende** a la otra o **hereda** de ella. [Derecha]



# Herencia

## Nomenclatura



- Si definimos la clase Coche a partir de la clase Vehículo se dice que:
  - "Coche" **hereda** las variables y métodos de "Vehículo"
  - "Coche" **extiende** de "Vehículo"
  - "Coche" es **subclase** de "Vehículo"
  - clase **derivada**
  - clase **hija**
  - "Vehículo" es **superclase** de "Coche"
  - clase **base**
  - clase **padre**
- La herencia realiza la relación **es-un**
  - Un coche **es-un** vehículo; un perro **es-un** mamífero, etc.



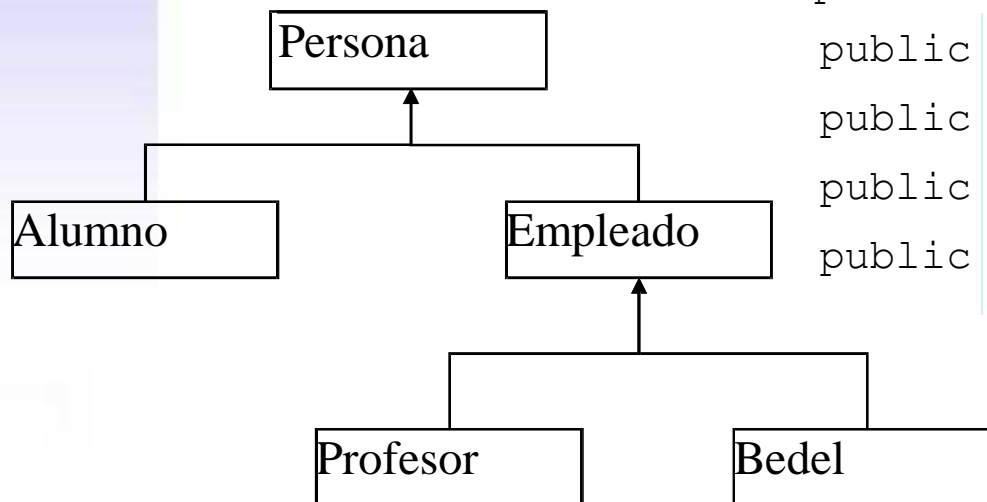
# Herencia

## Declaración de clases derivadas



- La sintaxis para declarar clases derivadas es:

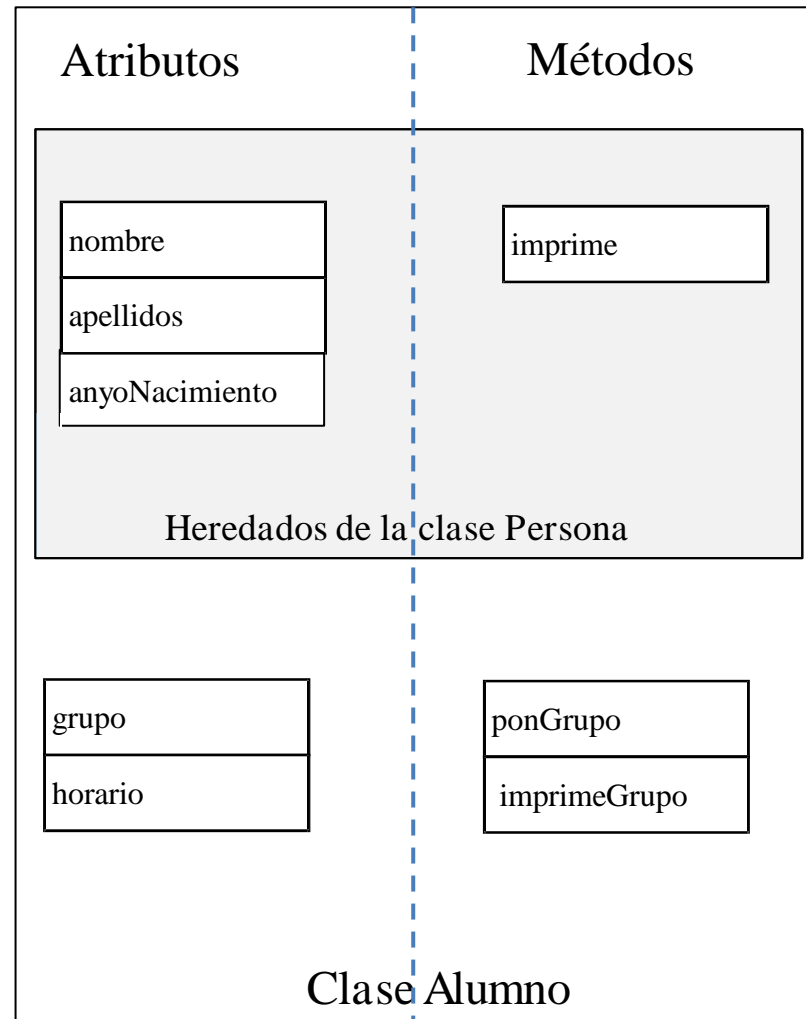
```
class ClaseDerivada extends ClaseBase { ... }
```



```
public class Persona { ... }
public class Alumno extends Persona { ... }
public class Empleado extends Persona { ... }
public class Profesor extends Empleado { ... }
public class Bedel extends Empleado { ... }
```

# Herencia

## Clase derivada (subclase)



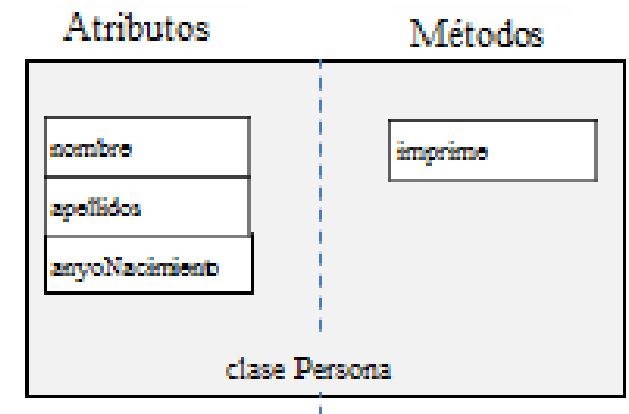


# Herencia

## ¿ Cómo se usa? Ej.: Persona.java

```
public class Persona {  
  
    protected String nombre;  
    protected String apellidos;  
    protected int anyoNacimiento;  
  
    public Persona () {  
  
    }  
  
    public Persona (String nombre, String apellidos,  
                    int anyoNacimiento){  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.anyoNacimiento = anyoNacimiento;  
    }  
  
    public void imprime(){  
        System.out.println("Datos Personales: " + nombre  
                            + " " + apellidos + " (" +  
                            anyoNacimiento + ")");  
    }  
}
```

- **protected** accesible desde las subclases

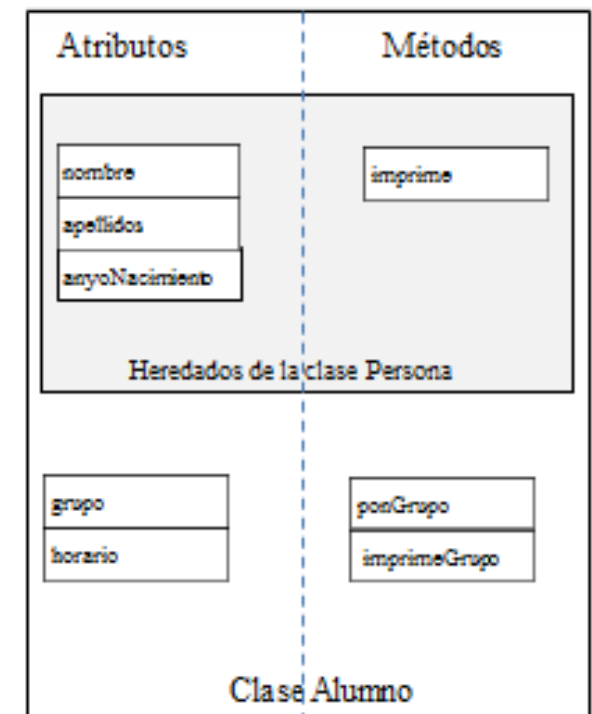


# Herencia

## ¿ Cómo se usa? Ej.: Alumno.java

```
public class Alumno extends Persona {  
  
    protected String grupo;  
    protected char horario;  
  
    public Alumno() {  
    }  
  
    public Alumno (String nombre, String apellidos,  
                    int anyoNacimiento) {  
        super(nombre, apellidos, anyoNacimiento);  
    }  
  
    public void ponGrupo(String grupo, char horario) {  
        if (grupo == null || grupo.length() == 0){  
            System.out.println("Grupo no valido");  
        }  
        else if (horario != 'M' && horario != 'T'){  
            System.out.println("Horario no valido");  
        }  
        this.grupo = grupo;  
        this.horario = horario;  
    }  
  
    public void imprimeGrupo(){  
        System.out.println(" Grupo " + grupo + horario);  
    }  
}
```

- **super** acceder a atributos o métodos de la superclase



# Herencia

## ¿ Cómo se usa? Ej.: Prueba.java

```
public class Prueba {  
    public static void main (String[] args){  
        Persona vecina = new Persona ("Luisa", "Asenjo Martínez", 1978);  
        Alumno unAlumno = new Alumno ("Juan", "Ugarte López", 1985);  
        unAlumno.ponGrupo("66", 'M');  
        vecina.imprime();  
        unAlumno.imprime();  
        unAlumno.imprimeGrupo();  
    }  
}
```

### Salida por pantalla

Datos Personales: Luisa Asenjo Martínez (1978)

Datos Personales: Juan Ugarte López (1985)

Grupo 66M



# Herencia

## ¿ Qué pasa si...?



- Defino el atributo **nombre** de **Persona** como **private**.
  - Se hereda, pero no podemos acceder a él, salvo que implementemos métodos para ello (p.ej. **getNombre()**)
- Implemento el constructor de la subclase asignando los valores a los atributos directamente en lugar de llamar a **super**
  - No aprovecho la potencia de la reutilización de código
  - En este caso es viable porque los atributos se han definido como **protected** (¡¡no siempre será así!!)

```
public Alumno (String nombre, String apellidos,  
               int anyoNacimiento) {  
    this.nombre = nombre;  
    this.apellidos = apellidos;  
    this.anyoNacimiento = anyoNacimiento;  
}
```

```
public Alumno (String nombre, String apellidos,  
               int anyoNacimiento) {  
    super(nombre, apellidos, anyoNacimiento);  
}
```

OK

# Herencia

## Consecuencias de la extensión de clases

- Herencia de la interfaz
  - La parte pública de la clase derivada contiene la parte pública de la clase base. La clase **Alumno** contiene el método **imprime()**
- Herencia de la implementación
  - La implementación de la clase derivada contiene la de la clase base. Al invocar los métodos de la clase base sobre el objeto de la clase derivada (**unAlumno.imprime()**) se produce el comportamiento esperado



# Ejercicio 5



- Partiendo de la clase `Bicicleta`, la cual tiene tres atributos, `velocidadActual`, `platoActual` y `piñonActual`, de tipo entero y cuatro métodos `acelerar()`, `frenar()`, `cambiarPlato(int plato)`, y `cambiarPiñon(int piñon)`, implementa las clases `BicicletaMontaña` y `BicicletaTandem`.
- `BicicletaMontaña` tiene un atributo `suspension` de tipo entero y un método `cambiarSuspension(int suspension)`
- `BicicletaTandem` tiene un atributo `numAsientos` de tipo entero.
- Crear los constructores de estas clases para inicializar todos sus atributos, haciendo uso de `super`

# Herencia

## Jerarquía de herencia en Java

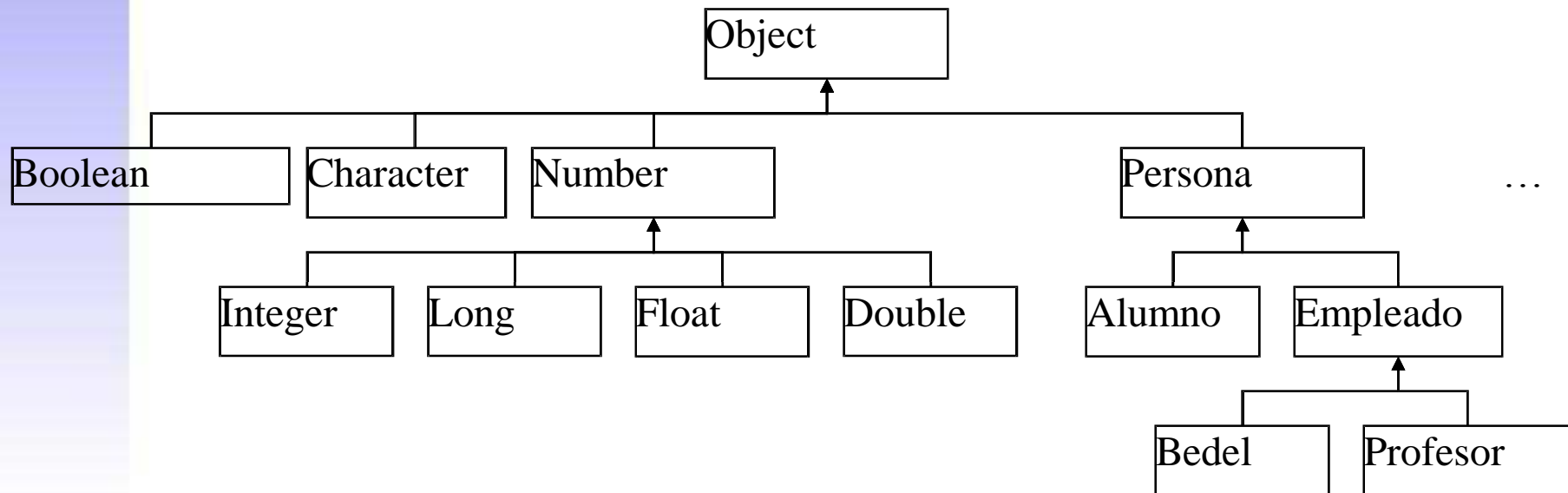


- En Java, todas las clases están relacionadas en **una única jerarquía** de herencia
- Una clase puede:
  - heredar explícitamente de otra clase
  - o bien heredar implícitamente de la clase `Object` (definida en el núcleo de Java)
- Esto se cumple tanto para las clases predefinidas como para las clases definidas por el usuario



# Herencia

## Jerarquía de herencia en Java





# Herencia

## Reescritura (o sobrescritura)



- Modificación de los elementos de la clase base dentro de la clase derivada
- La clase derivada puede definir:
  - Un atributo con el mismo nombre que uno de la clase base → **Ocultación de atributos**
  - Un método con la misma signatura que uno de la clase base → **Redefinición de métodos**
- Lo más usual cuando se produce reescritura es que se reescriba un método



# Reescritura I (Shadowing)

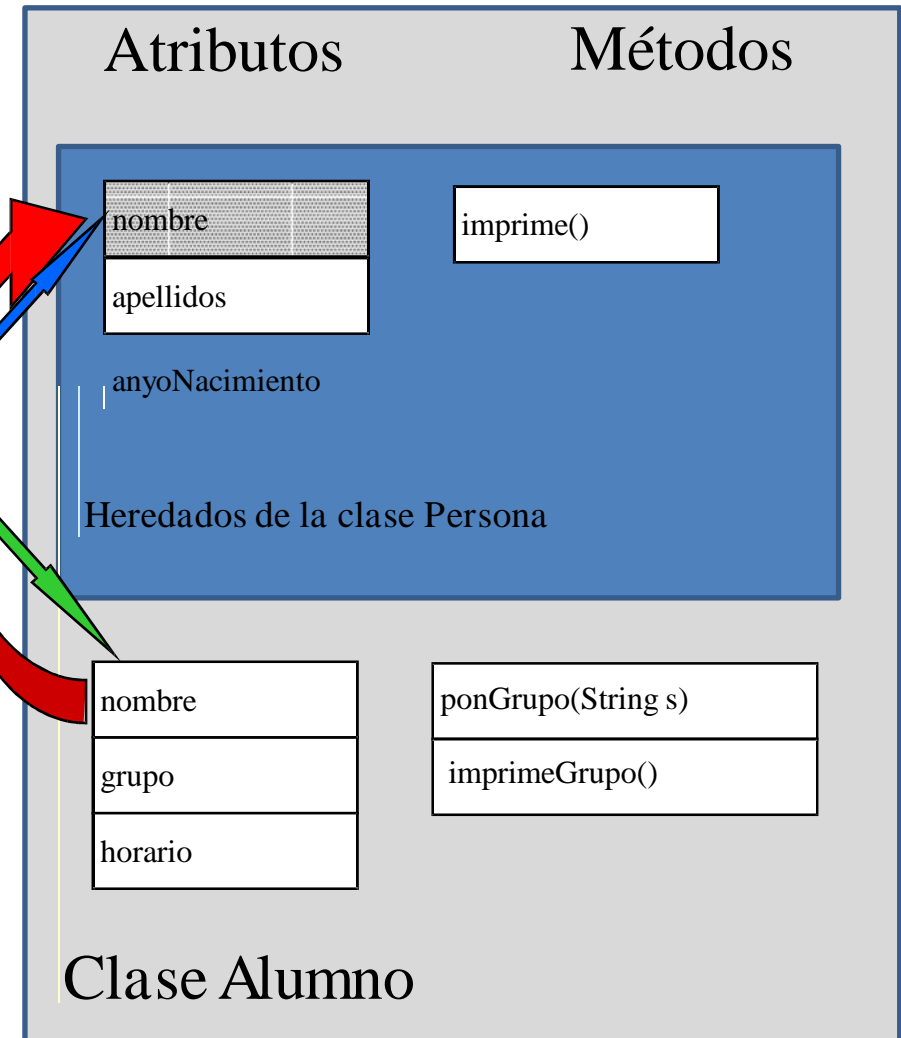
## Ocultación de atributos



```
Alumno a = new Alumno(...);  
System.out.println(a.nombre);
```

```
Persona p = a;  
System.out.println(p.nombre);
```

- Mismo nombre pero el tipo puede ser distinto



# Reescritura I (Shadowing)

## Ocultación de atributos. Ejemplo

```
class Persona {  
    public String nombre = "Juan";  
}
```

```
class Alumno extends Persona {  
    public int nombre = 10003041;  
}
```

```
class Test {  
    public static void main (String[] args) {  
  
        Alumno a = new Alumno ();  
        Persona p = a;  
        System.out.println(p.nombre);  
        System.out.println(a.nombre);  
        ;  
    }  
}
```

Imprime "Juan"

Imprime 10003041

# Reescritura I (Shadowing)

## Ocultación de atributos

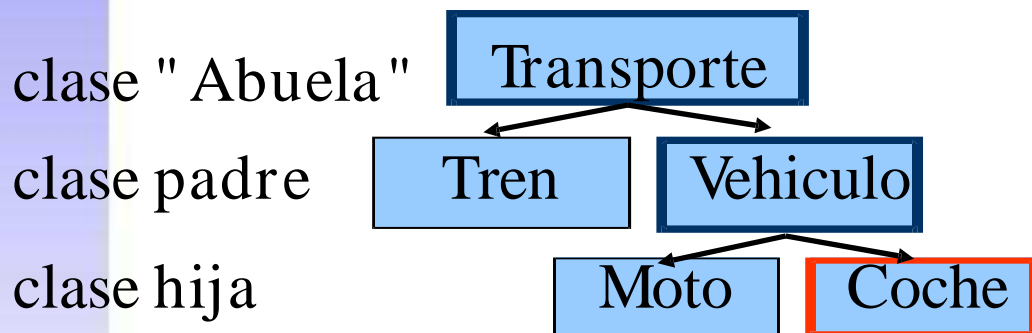


- Si definimos en una subclase un atributo del mismo nombre y tipo que en la superclase, la de la superclase queda oculta.
- Podemos acceder a la variable de la subclase o de la superclase utilizando **this** y **super**.



# Reescritura I (Shadowing)

## Ocultación de atributos



*String nombre* = "terrestre"

*String nombre* = "turismo"

*String nombre* = "Ferrari"

- ¿Cómo acceder a variables ocultas (desde la clase hija)?
  - *nombre* (nombre del coche)
  - *this.nombre* (nombre del coche)
  - *super.nombre* (nombre del vehículo)
  - *((Vehiculo)this).nombre* (nombre del vehículo)
  - ~~*super.super.nombre* (Mal)~~
  - *((Transporte)this).nombre* (nombre del transporte)

variables  
clase hija:  
visibles

Variables  
clases padre  
ocultas

# Reescritura II (Overriding)

## Redefinición de métodos. ¿Qué es?



- La reescritura de métodos es útil para
  - Ampliar la funcionalidad de un método
  - Particularizar la funcionalidad de un método a la clase derivada
- Si definimos en una subclase un **método** con la misma signatura (nombre + tipo y número de parámetros) que en la superclase el de la superclase queda oculto.
- ¿Cómo acceder a métodos ocultos?
  - `arrancar()` (ejecuta el método arrancar del coche)
  - `this.arrancar()` (ejecuta el método arrancar del coche)
  - `super.arrancar()` ( método arrancar del vehículo)
  - ~~`super.super.nombre`~~ (Mal)

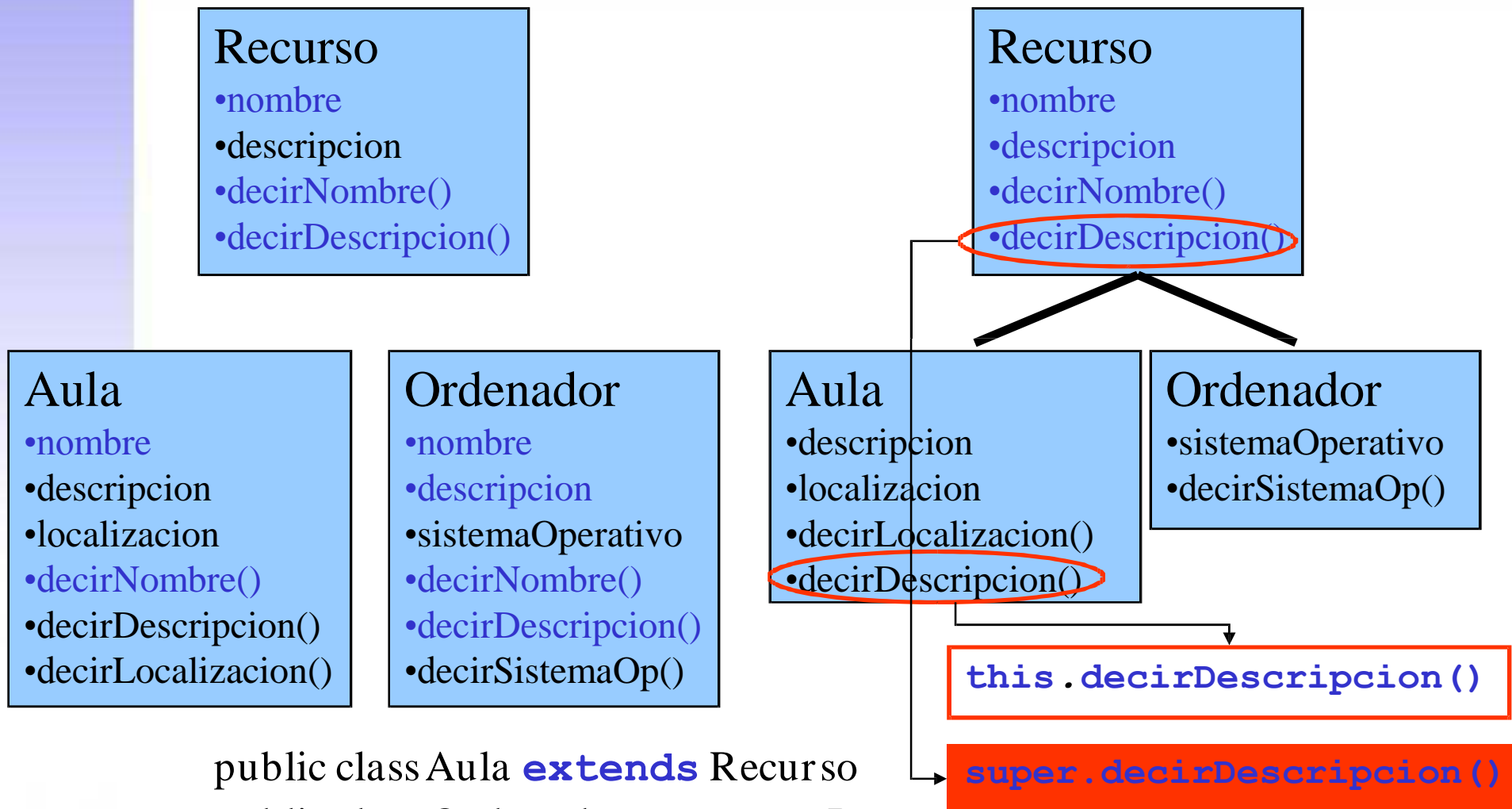
} Métodos  
clase hija:  
visibles

} métodos  
clases padre:  
ocultos



# Reescritura II (Overriding)

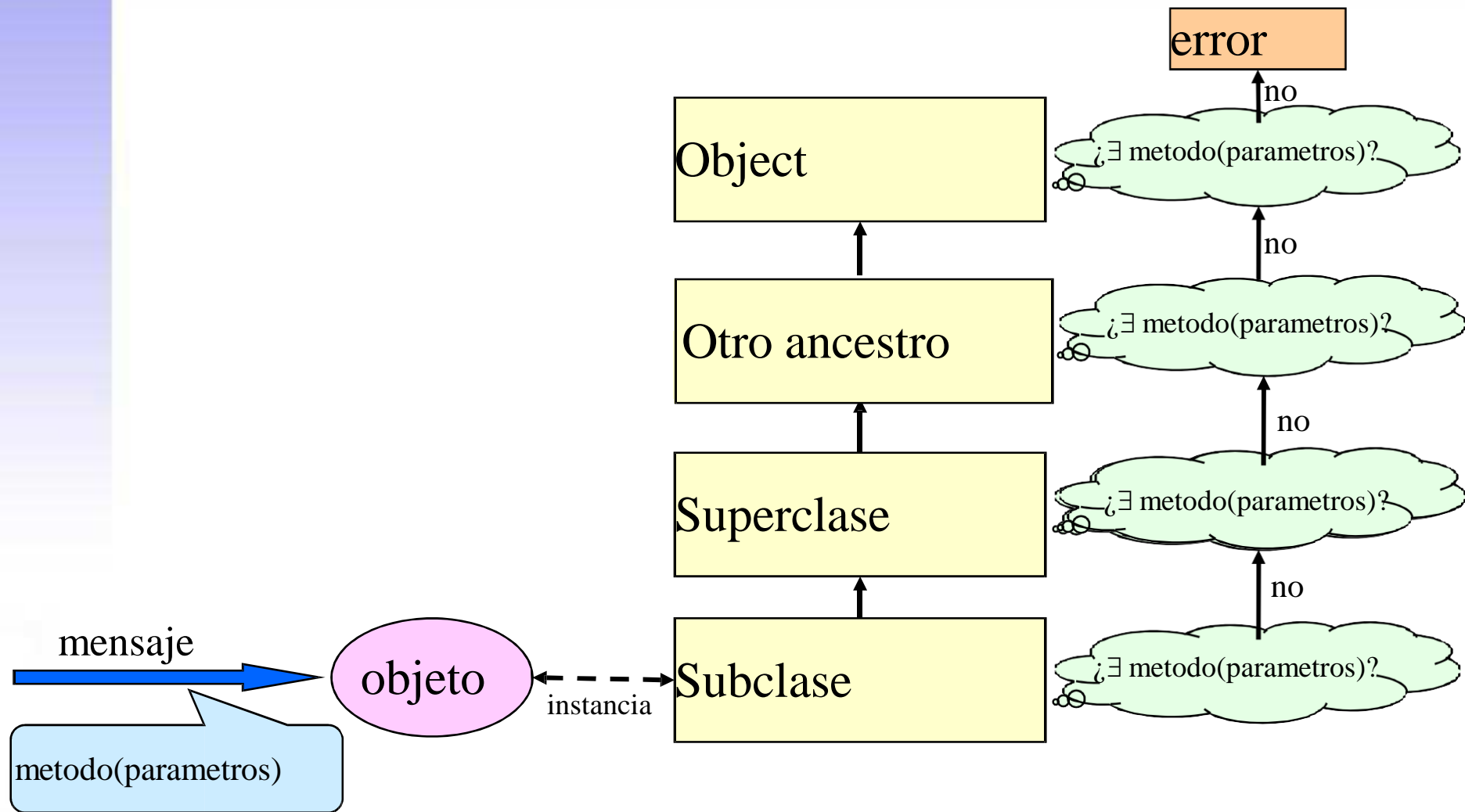
Redefinición de métodos ¿Para qué sirve?



```
public class Aula extends Recurso
public class Ordenador extends Recurso
```

# Reescritura II (Overriding)

## Redefinición de métodos





# Reescritura II (Overriding)

## Redefinición de métodos



- Al mandar un mensaje a un objeto, el método seleccionado:
  - Depende de la clase real de la que el objeto es una instancia
  - No de la clase de referencia a la que esté asignado, como en el caso de los atributos



# Reescritura II (Overriding)

## Redefinición de métodos. Ejemplo

```
class Persona {  
    public String nombre = "Juan";  
    public void imprimir() {  
        System.out.println("Persona: " + nombre);  
    }  
}  
class Alumno extends Persona {  
    public String nombre = "JuanGarcía";  
    public void imprimir() {  
        System.out.println("Alumno: " + nombre);  
    }  
}  
class Test2 {  
    public static void main (String[] args) {  
        Alumno a = new Alumno();  
        Persona p = a;  
        a.imprimir();  
        p.imprimir();  
    }  
}
```

Ambas imprimen:  
"Alumno: JuanGarcia"

# Ejercicio 6



- Sobreescribe el método `acelerar()`, de `Bicicleta`, en las subclases `BicicletaMontaña` y `BicicletaTandem`, de tal forma que en la primera `acelerar` suponga triplicar la velocidad actual y en la segunda cuadruplicar la velocidad actual.
- Crea dos objetos de las clases `BicicletaMontaña` y `BicicletaTandem` e invoca sobre ellos el método `acelerar()`, ¿cuál es el resultado?
- Desde estos objetos que has creado, ¿cómo accederías a la implementación del método `acelerar()`, en la clase `Bicicleta`?



# Reescritura vs. sobrecarga



- **Reescritura:** La subclase sustituye la implementación de un método de la superclase
  - Ambos métodos tienen que tener la misma signatura
- **Sobrecarga:** Existe más de un método con el mismo nombre pero distinta signatura
  - Los métodos sobrecargados pueden definirse en la misma clase o en distintas clases de la jerarquía de herencia



# Constructores y herencia



- Para la creación de un objeto:
  1. Se crea su parte base
  2. Se añade su parte derivada
    - Si la clase base del objeto hereda a su vez de otra, en el paso 1 se aplica el mismo orden de creación, hasta llegar a **Object**
- En la creación de un objeto **Alumno** que hereda de **Persona**, los pasos son:
  1. Se crea la parte correspondiente a **Persona**. Para ello
    1. Se crea la parte correspondiente a **Object**
    2. Se añaden los elementos de **Persona**
  2. Se añaden los elementos de **Alumno**



# Constructores y herencia



- En el constructor de la clase derivada se realiza siempre una **llamada al constructor de la clase base**
- Ésta es la primera acción del constructor (aparece en la primera línea)
- Hay dos posibilidades:
  - No indicarlo explícitamente
  - Indicarlo explícitamente (**obligatoriamente en la primera línea**)



# Constructores y herencia



1. Si no se indica explícitamente, Java inserta automáticamente una llamada a **super()** en la primera línea del constructor de la clase derivada

```
public Alumno (String nombre, String apellidos,  
               int anyoNacimiento, String grupo,  
               char horario) {  
    // aquí inserta Java una llamada (invisible) a super()  
    this.nombre = nombre;  
    this.apellidos = apellidos;  
    this.anyoNacimiento = anyoNacimiento;  
    this.grupo = grupo;  
    this.horario = horario;  
}
```

# Constructores y herencia



## 2. Indicándolo explícitamente

```
public Alumno (String nombre, String apellidos,  
               int anyoNacimiento, String grupo,  
               char horario) {  
    super(nombre, apellidos, anyNacimiento);  
    this.grupo = grupo;  
    this.horario = horario;  
}
```





# Más sobre **super**



- **super**

- referencia al objeto actual como si fuera una instancia de su superclase
- A través de la referencia a **super** se puede acceder explícitamente a métodos de la superclase
- Para reescribir métodos (no sólo el constructor), puede ser útil usar la referencia a **super**

```
public class Alumno extends Persona {  
    // el resto permanece igual  
    public void imprime(){  
        super.imprime();  
        System.out.print(" Grupo " + grupo + horario);  
    }  
}
```



# Polimorfismo



- Capacidad de un objeto de decidir qué método aplicar, dependiendo de la clase a la que pertenece
  - Una llamada a un método sobre una referencia de un tipo genérico (clase base) ejecuta la implementación correspondiente del método dependiendo de la clase del objeto que se creó
- Permite diseñar e implementar sistemas extensibles
  - Los programas pueden procesar objetos genéricos (descritos por referencias de la superclase)
  - El comportamiento concreto depende de las subclasses



# Polimorfismo

## Ejemplo



- **Alumno**, **Profesor** y **Bedel**.
- Creamos un array de **Persona** donde incluimos objetos de **Alumno**, **Profesor** y **Bedel**.
- Al invocar al método **imprimir()**, sobrescrito en las clases **Alumno**, **Profesor** y **Bedel**, sobre el array de **Persona**, cada objeto utilizará su propia implementación del método

```
Persona[] grupo = {new Alumno(...), new
    Profesor(...), new Bedel(...), new Alumno(...)};

for (int i=0; i<grupo.length; i++){
    grupo[i].imprimir();
}
```

# Polimorfismo

## Ligadura dinámica



- Se llama al método correcto, aunque nos estemos refiriendo al objeto de la subclase a través de una referencia a la superclase
- Este mecanismo se llama “***ligadura dinámica***”
  - permite detectar **en tiempo de ejecución** cuál es el método adecuado para llamar
- El compilador no genera el código para llamar al método en tiempo de compilación
  - Genera código para calcular qué método llamar



# Ejercicio 7



- Crea un array de la clase `Bicicleta`, que contenga objetos de las clases `Bicicleta`, `BicicletaMontaña` y `BicicletaTandem`
- Invoca el método `acelerar()` sobre cada uno de los objetos aprovechando las propiedades de polimorfismo y ligadura dinámica

# Modificadores y acceso

## Final



- Si no se quiere que las clases derivadas sean capaces de modificar un método o un atributo de la clase base, se añade a ese método o atributo la palabra reservada **final**

# Modificadores y acceso

## Final



- El modificador **final** se puede aplicar a:
  - Parámetros: Indica que dentro del método no podemos cambiar el valor de dicho parámetro

```
public void miMetodo(final int p1, int p2){} //no podemos cambiar valor p1
```

- Atributos: Indica que dentro de la clase no podemos cambiar el valor de dicho atributo. Se utiliza para definir constantes junto con **static**

```
public static final double PI = 3.14; //no podemos cambiar el valor
```

- Métodos: Indica que las clases que hereden de estas no pueden sobrescribir dicho método.

```
public final void myMethod(){ } //no podemos sobrescribir myMethod
```

- Clases: Impide la extensión de clases. No se puede “heredar de ella”

```
public final class myClass(){ } //no podemos extender myClass
```



MODIFICADORES		<i>clase</i>	<i>metodo</i>	<i>atributo</i>
acceso	public	Accesible desde cualquier otra clase		
	(friendly)	Accesible sólo desde clases de su propio paquete		
	protected		Accesible desde la clase y sus subclases	
	private		Accesibles sólo dentro de la clase	
otros	final	No se puede heredar de ellas. Es la hoja en el árbol de herencia	No se puede ocultar Es <b>cte</b> y no puede ser modificado en las clases hijas	No se puede cambiar su valor, es <b>cte</b> . Se suele utilizar en combinación con static
	static	Clase de nivel máximo. Se aplica a classes internas	Es el mismo para todos los objetos de la clase. Se utiliza: NombreClase.metodo();	Es la misma para todos los objetos de la clase. Se utiliza: NombreClase.atributo;







# *Programación de Sistemas*

## *Programación ORIENTADA a Objetos (II)*

Julio Villena Román  
<jvillena@it.uc3m.es>

MATERIALES BASADOS EN EL TRABAJO DE DIFERENTES AUTORES:  
M.Carmen Fernández Panadero, Raquel M. Crespo García  
Carlos Delgado Kloos, Natividad Martínez Madrid



# Contenidos

- ▶ Casting. Compatibilidad de tipos
- ▶ Clases y métodos abstractos
- ▶ Interfaces



# Casting (conversión)

## Sintaxis y tipos



- Sintaxis: (tipo) identificador
- Dos tipos de casting:
  - *Widening o upcasting*: Una subclase se utiliza como instancia de la superclase. **Es implícito.**
  - *Narrowing o downcasting*: La superclase se utiliza como instancia de una subclase.
- **Conversión explícita.**
- Sólo se puede hacer casting entre clases padre e hija no entre clases hermanas

# Casting (conversión)

## Widening o upcasting



### 1. Compatibilidad hacia arriba (*upcasting*)

- Un objeto de la clase derivada siempre se podrá usar en el lugar de un objeto de la clase base (ya que se cumple la relación “es-un”)

```
Persona p = new Alumno() ;
```



# Casting (conversión)

## Narrowing o downcasting



### 2. Compatibilidad hacia abajo (*downcasting*)

- No se produce por defecto, ya que un objeto de la clase base no siempre es un objeto de la clase derivada

```
Alumno a = new Persona(); // error
```

- Sólo es posible en los casos en los que el objeto de la clase base realmente sea un objeto de la clase derivada
- Estos casos se tendrán que indicar explícitamente con un *casting* (con una asignación explícita de la clase).

# Casting (conversión)

## Ejemplo



```
public class Prueba2 {  
    public static void main (String[] args) {  
        Persona p1;  
  
        //conversión ascendente implícita - funciona  
        Alumno a1 = new Alumno();  
        p1 = a1;  
  
        Alumno a2;  
  
        //conversión descendente implícita - No funciona  
        a2 = p1; //error porque no hago conversión explícita  
  
        //conversión descendente explícita - funciona  
        a2 = (Alumno) p1; //p1 referencia una instancia  
                           //de Alumno  
    }  
}
```

Un alumno  
siempre es una  
persona  
(**implícito**)

Una persona no  
siempre es un  
alumno

Si alguien además de persona es alumno (no siempre ocurre) podemos pedirle cosas de alumno pero tendremos que decirle **explícitamente** que le trataremos como alumno.

# Casting (conversión)

## Ejemplo



```
Persona p2 = new Persona();  
Alumno a3;
```

```
//conversión descendente implícita - no funciona  
a3 = p2; //da error de compilación
```

Una persona no siempre es un alumno. No podemos asumir **implicitamente** que lo sea

```
//conversión descendente explícita - no funciona a veces  
//lanzará la excepción ClassCastException  
//porque p2 no es de la clase Alumno  
a3 = (Alumno) p2; //error
```

Una persona a veces es un alumno pero si no lo es (no lo hemos creado como tal) no podemos tratarlo como si lo fuera, ni siquiera aunque se lo digamos **explicitamente**

```
//conversión descendente implícita - no funciona  
Alumno a4 = new Persona(); //error
```

Una persona no siempre es un alumno. No podemos asumir **implicitamente** que lo sea

# Casting (conversión)

## El operador instanceof



- **Sintaxis:**

**objeto instanceof clase**

- Comprueba si un objeto es realmente de la clase derivada

- **Ejemplo:**

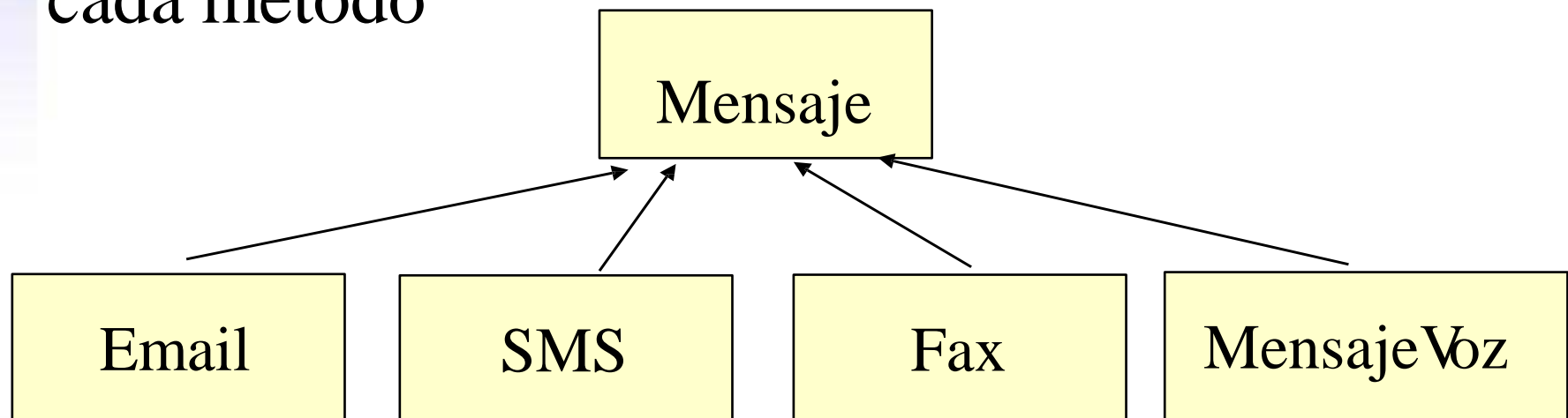
```
public Alumno comprueba (Persona p) {  
    Alumno a = null;  
    if (p instanceof Alumno)  
        a = (Alumno) p;  
    return a;  
}
```



# Clases abstractas

## ¿Qué son?

- Aquellas que tienen *al menos un método abstracto* (sin implementar, sin código).
- Declara la *estructura* de una determinada *abstracción*, sin implementar completamente cada método



# Clases abstractas

## Características

- Las clases y métodos abstractos se definen con la palabra clave *abstract*

```
public abstract class Figura {...}
```

- No pueden llevar el modificador abstract:
  - los **constructores**
  - los métodos **estáticos**
  - los métodos **privados**

# Clases abstractas

## Características

- *No podemos crear objetos* de una clase abstracta
  - Pueden existir referencias a clases abstractas
  - Pero apuntarán a objetos de clases derivadas de la clase abstracta.

```
Figura fig = new Rectangulo(2,3);
```

- *Sí podemos heredar* de una clase abstracta
- En una clase abstracta puede haber
  - Metodos **abstractos**
  - Métodos **no abstractos**



# Clases abstractas

## ¿Para qué sirven?: Implementaciones parciales

- Las clases abstractas suelen usarse para representar clases con **implementaciones parciales**
  - Algunos métodos no están implementados pero sí declarados
- El objetivo de las implementaciones parciales es dar una **interfaz común** a todas las clases derivadas de una clase base abstracta
  - Incluso en los casos en los que la clase base no tiene la suficiente información como para implementar el método



# Clases abstractas

## Métodos abstractos

- Métodos declarados pero no implementados en las clases abstractas

```
public abstract tipoDevuelto nombre (listaParametros);
```

- Se declaran con la palabra reservada **abstract**
- Las clases que hereden de la clase abstracta deberán implementar los métodos abstractos de la superclase
  - O serán abstractas ellas también

**NOTA: No hay llaves!!** No están implementados: después de la declaración se pone solo un ;

# Clases abstractas

## ¿Cómo se usan? Ejemplo

```
public abstract class Figura {  
    protected double dim1;  
    protected double dim2;  
  
    public Figura(double dim1, double dim2) {  
        this.dim1 = dim1;  
        this.dim2 = dim2;  
    }  
    public abstract double area();  
}
```

```
public class Rectangulo extends Figura {  
    public Rectangulo(double dim1, double dim2) {  
        super(dim1, dim2);  
    }  
    public double area() {  
        return dim1*dim2;  
    }  
}
```

# Ejercicio 8



- Partiendo de la clase `Bicicleta`, la cual tiene tres atributos, `velocidadActual`, `platoActual` y `piñonActual`, de tipo entero y cuatro métodos `acelerar()`, `frenar()`, `cambiarPlato(int plato)`, y `cambiarPiñon(int piñon)`, implementa la clase abstracta `Vehículo`, la cual será una superclase de `Bicicleta`
- Piensa qué atributos y métodos serán propios de `Bicicleta` y cuáles pueden heredarse de `Vehículo`, teniendo en cuenta que además de `Bicicleta` existirán las subclases `Coche` y `Camión`
- Piensa qué métodos deben ser abstractos y cuáles no en la clase `Vehículo`



# Clases abstractas

## Polimorfismo

El array es de objetos de tipo **Figura** (abstracto)

```
public static void main(String args[]){  
    Figura[] misFiguras = new Figura[3];  
    misFiguras[0] = new Rectangulo(1,3);  
    misFiguras[1] = new Triangulo(2,5);  
    misFiguras[2] = new Cuadrado (3);
```

Los elementos del array son de un tipo concreto (**Rectangulo**, **Triangulo**, **Cuadrado...**)

```
    for (int i=0; i<misFiguras.length; i++){  
        System.out.println(misFiguras[i].area());  
    }
```

Llamamos a area() sobre objetos de tipo Figura  
Y en tiempo de ejecución mira a ver qué tipo de objeto contiene, (**Ligadura dinámica**)



# Interfaces

## ¿Qué son?



- Los interfaces son colecciones de métodos (y constantes)
  - **Todos los métodos de un interfaz son abstractos**
- El acceso a un interfaz es *público*
  - Los atributos son public, static y final
  - Los métodos son public
- Los interfaces son *implementados* por clases
  - una *clase* implementa un interfaz definiendo los cuerpos de **todos** los métodos de la interfaz
  - una *clase abstracta* implementa un interfaz definiendo los cuerpos de **todos** los métodos de la interfaz o declarando alguno como abstracto
  - una *clase* (abstracta o no) puede implementar uno o más interfaces

# Interfaces

## ¿Qué son?



- Una **interfaz** es un elemento puramente de **diseño**
  - ¿Qué se quiere hacer?
- Una **clase** (incluidas las abstractas) es una mezcla de **diseño e implementación**
  - ¿Qué se quiere hacer y **cómo** se hace?
- Distintas clases pueden implementar la interfaz de distintas formas

# Interfaces

## Declaración



- Sintaxis:

```
public interface nombreInterfaz {  
    static final tipo CONSTANTE = valor;  
    tipoDevuelto nombreMetodo(listaParam) ;  
}
```

NOTA 1: **No hay llaves!!** No está implementado después de la declaración se pone sólo un ;

NOTA 2: Las constantes y métodos en las interfaces son siempre públicos (no hay necesidad de hacerlo explícito)

# Interfaces

## Implementación

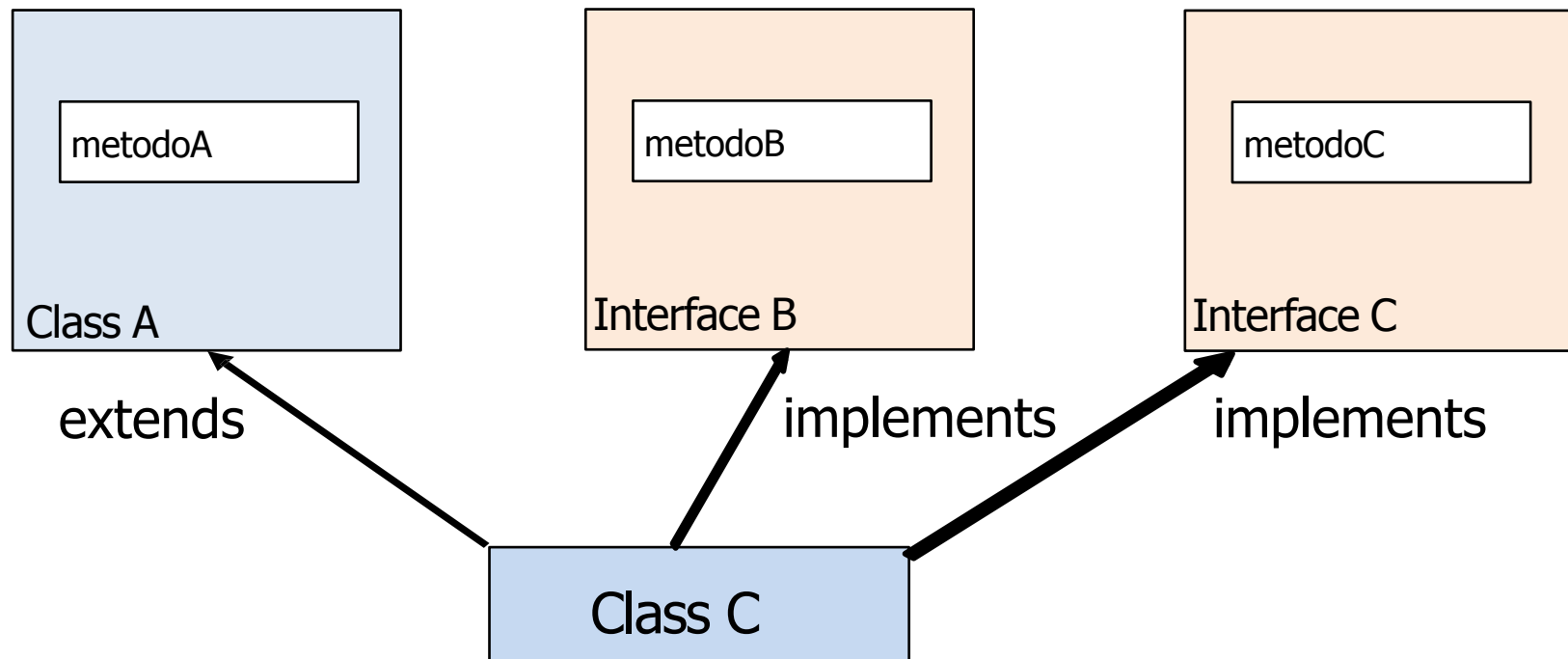


- Si una clase implementa una interfaz, quiere decir que implementa todos los métodos abstractos de esa interfaz
- Esto se representa con la palabra reservada **implements**:

```
public class Clase implements Interfaz {...}
```

# Interfaces

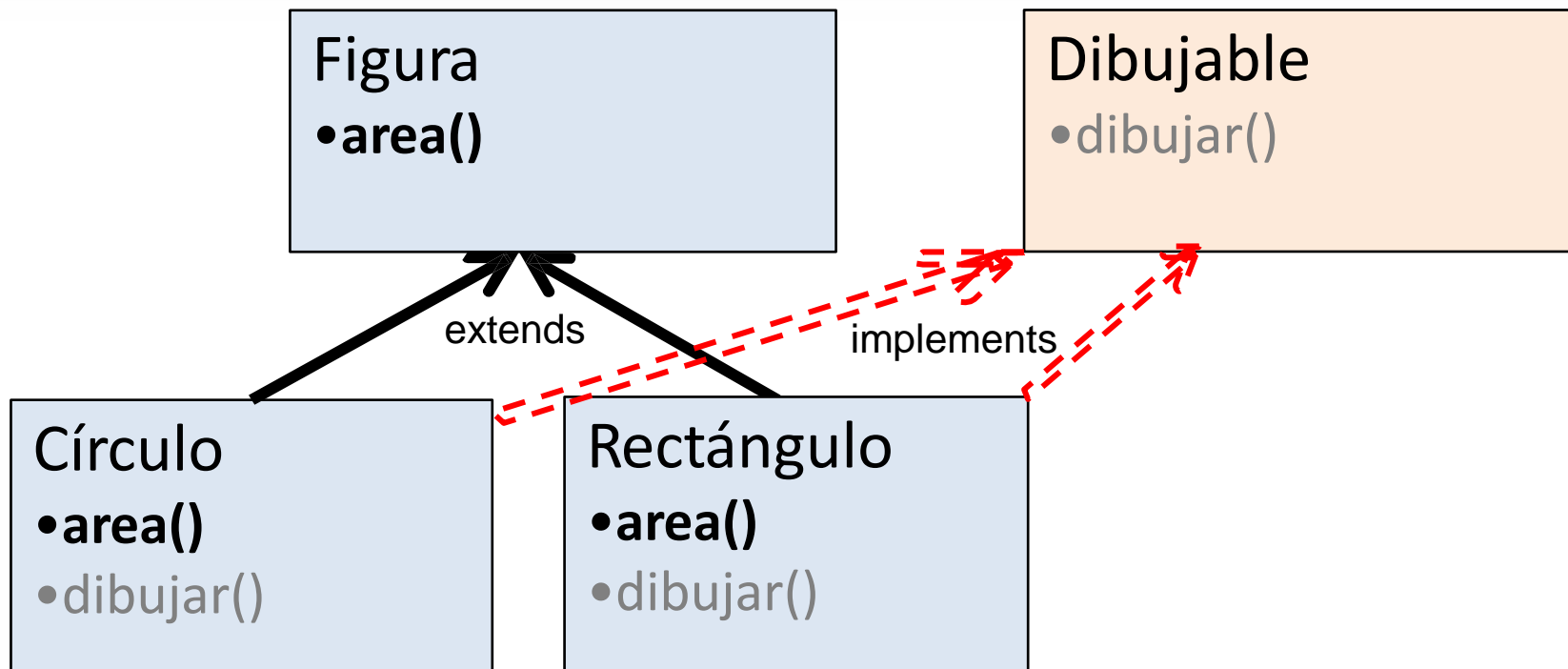
## ¿Para qué sirven? Herencia múltiple



- En Java una clase hereda de una única superclase
  - **No** existe la herencia múltiple
- Pero puede implementar varios interfaces

# Interfaces

## ¿Cómo se usan?



```
public abstract class Figura {...}
```

```
public interface Dibujable {...}
```

```
public class Circulo extends Figura implements Dibujable
```

```
public class Rectangulo extends Figura implements Dibujable
```

# Ejercicio 9



- Implementa la interfaz `Imprimible`, la cual contiene el método `imprime()` que no devuelve ningún valor.
- La clase `Bicicleta` implementa la interfaz `Imprimible` de tal forma que se imprima por pantalla la velocidad actual el piñón actual y el plato actual.

# Interfaces

## Herencia de interfaces y polimorfismo

- Las interfaces también pueden tener una jerarquía de herencia
- Los métodos que deberán incluir las clases que implementen las interfaces se van acumulando siguiendo la jerarquía
- Las interfaces también dan soporte a la resolución dinámica de métodos durante la ejecución (ligadura dinámica)





# Ejercicio 10



- Implementa la interfaz `Definir`, la cual contiene el método `getAtributos()` que devuelve el valor de los atributos de un objeto. La interfaz `Imprimible` hereda de `Definir`.
- ¿Qué cambios hay que hacer en la interfaz `Imprimible`? ¿Y en la clase `Bicicleta`?

# Resumen Orientación a objetos




- *Clase* (concreta)
  - *Todos* los métodos implementados
- *Clase abstracta*
  - *Al menos un* método no implementado, (sólo declarado)
  - modificador `abstract`
- *Interfaz*
  - *Nada* de implementación
  - palabra reservada: `interface`



# Resumen Orientación a objetos

- *Clase* (concreta o abstracta)
  - puede *extender* (`extends`) a *una* sola clase (herencia simple)
  - puede *implementar* (`implements`) *uno o más* interfaces (herencia múltiple)
- *Interfaz*
  - puede extender (`extends`) a *uno o más* interfaces



MODIFICADORES		<i>clase</i>	<i>metodo</i>	<i>atributo</i>
acceso	public	Accesible desde cualquier otra clase		
	(friendly)	Accesible sólo desde clases de su propio paquete		
	protected		Accesible desde la clase y sus subclases	
	private		Accesibles sólo dentro de la clase	
otros	abstract	No se pueden instanciar Son <b>para heredar</b> de ellas Al menos 1 método abstracto	No tiene código Se implementa en las subclases o clases hijas	
	final	No se puede heredar de ellas. Es la hoja en el árbol de herencia	No se puede ocultar Es <b>cte</b> y no puede ser modificado en las clases hijas	No se puede cambiar su valor, es <b>cte</b> . Se suele utilizar en combinación con static
	static	Clase de nivel máximo. Se aplica a classes internas	Es el mismo para todos los objetos de la clase. Se utiliza: NombreClase.metodo();	Es la misma para todos los objetos de la clase. Se utiliza: NombreClase.atributo; 100