



Programación de sistemas

Orientación a Objetos en Java

I. Programación **Basada** en objetos

II. Programación **orientada** a objetos

Ingeniería Telemática

M. Carmen Fernández Panadero

<mcfp@it.uc3m.es>





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





Contenidos



- ⌘ Clases y objetos
- ⌘ Encapsulación de objetos
 - ⌘ Abstracción funcional
 - ⌘ Abstracción de datos
- ⌘ Miembros de una clase (atributos y métodos)
- ⌘ Paso de mensajes





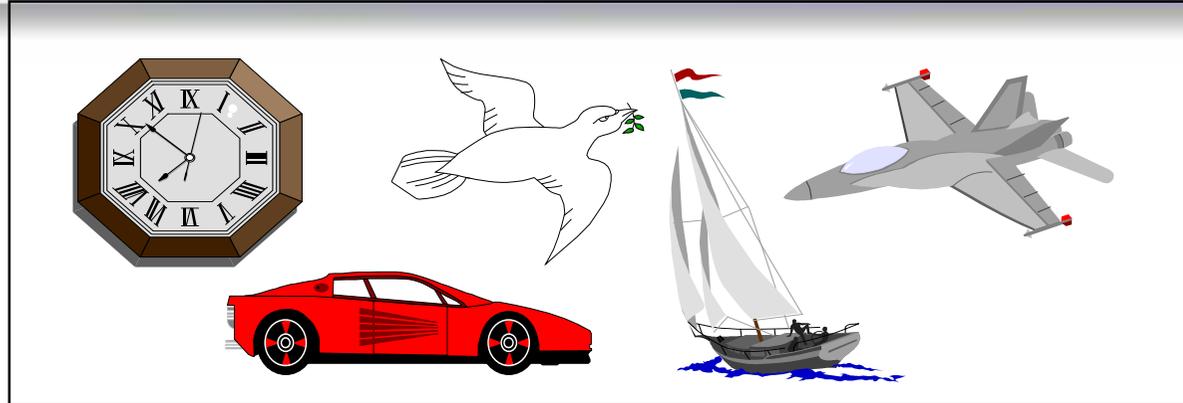
Objetivos



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



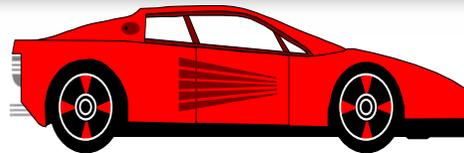
¿Qué es un objeto?



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



¿Qué es un objeto?



- *Abstracción funcional*

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

- avanzar
- parar
- girar a la dcha
- girar a la izda

- *Abstracción de datos*

- Un coche tiene además ciertos atributos:

- color
- velocidad
- tamaño
- etc..

- La forma en que se definen los atributos no tiene importancia para el diseño





¿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**.





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)



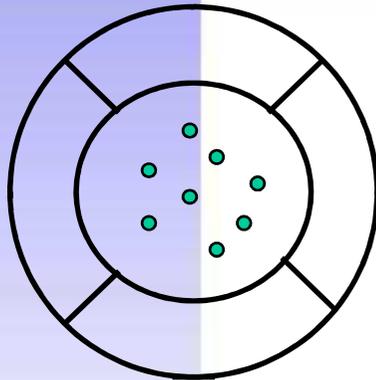
- *La llave de contacto es un **interfaz público** del mecanismo de arranque de un coche*
- *La implementación de cómo arranca realmente es **privada** y sobre ella sólo puede actuar la llave de contacto*

ventajas

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



Encapsulación de objetos



MIEMBROS DE UNA CLASE

Los objetos encapsulan atributos permitiendo acceso a ellas 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** puede hacer los objetos de esa clase)
 - Qué puede hacer el objeto (métodos)
 - Qué es el objeto (su abstracción)

Miembros privados

- describen la implementación de **cómo** lo hace.
 - Ejemplo el objeto contacto interactúa 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

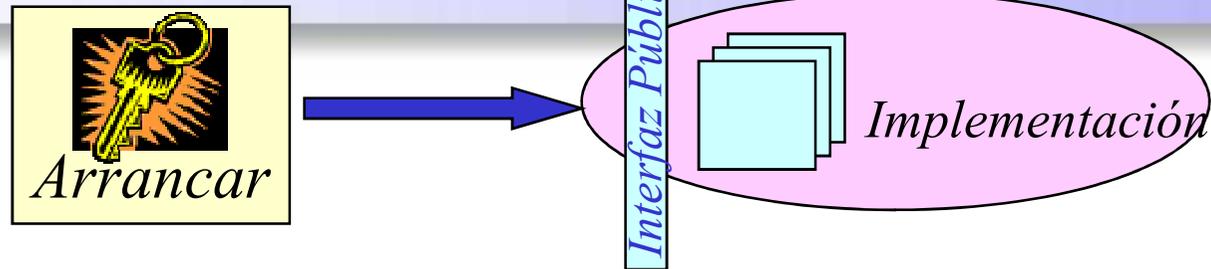


- El ***modelado de objetos*** modela:
 - los objetos y
 - sus interrelaciones
- Para realizar su tarea el objeto puede ***delegar*** trabajos en otro que puede ser parte de él mismo o de cualquier otro objeto del sistema.
- Los objetos interaccionan entre sí enviándose ***mensajes***





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))



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.

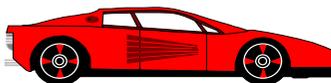
- Nombre de la clase
- Métodos (funciones)
- Atributos (datos)

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

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

Objeto Ferrari

Pertenece 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





Programación de sistemas

Orientación a Objetos en Java

I. Programación **Basada** en objetos

II. Programación **orientada** a objetos

Ingeniería Telemática

M. Carmen Fernández Panadero

<mcfp@it.uc3m.es>





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**) y con la referencia (**super**, **this**) y acceso (**modificadores**) a los distintos miembros
- Conocer mecanismos básicos de herencia como **ocultación** de atributos, **sobreescritura** de métodos y **sobrecarga** de constructores y saber para qué sirven y cómo se usan





Contenidos

- ⌘ Conceptos básicos de herencia
- ⌘ Jerarquía de herencia
- ⌘ ReescrituraI: Ocultación de atributos
- ⌘ ReescrituraII: Redefinición de métodos
- ⌘ Sobrecarga de métodos
- ⌘ Constructores de clases derivadas
- ⌘ Los modificadores static y final
- ⌘ Ámbitos y acceso





Herencia



¿Qué es? ¿Para qué sirve?

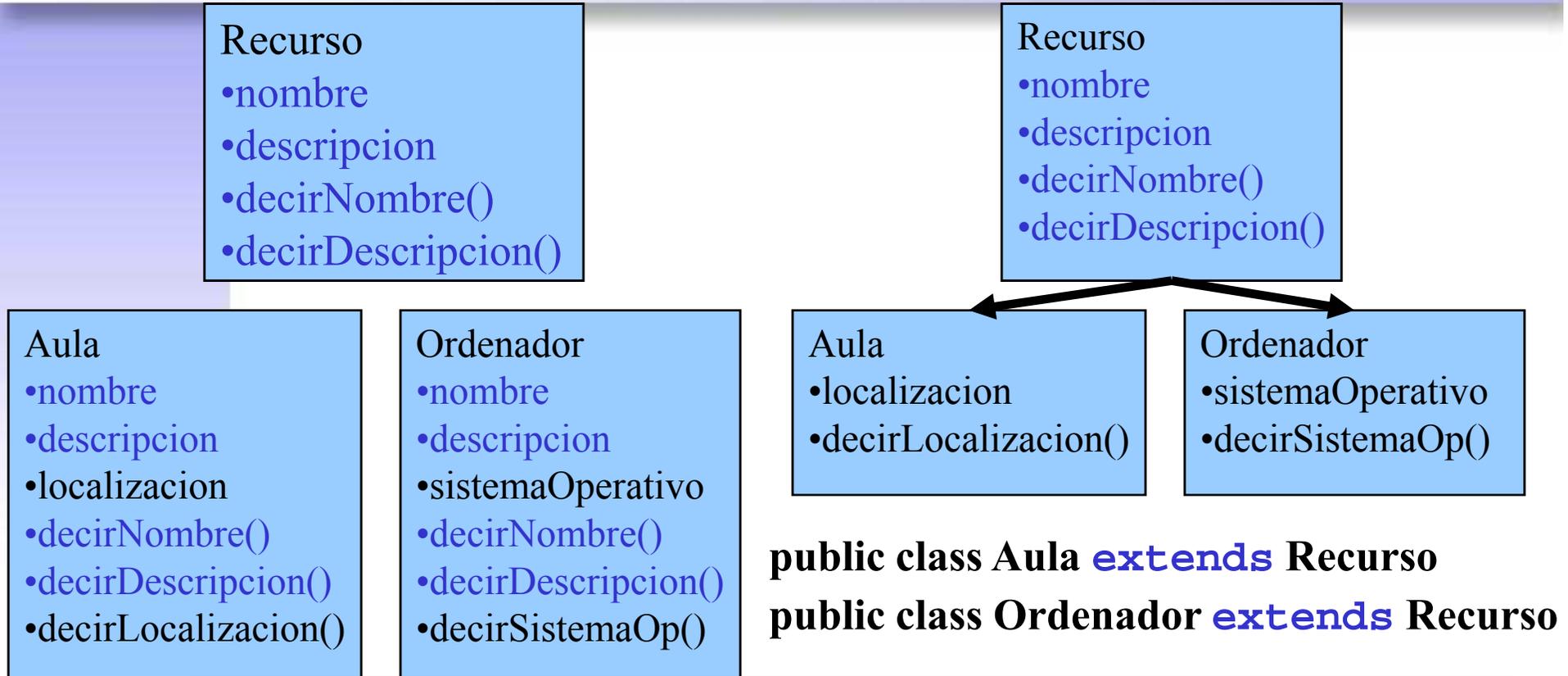
- Es un mecanismo para la reutilización de software
- Permite definir a partir de una clase otras clases relacionadas que supongan una:
 - **Especialización** de la clase dada.(ej la clase Coche es una especialización de la clase Vehículo)
 - **Escenario:** Tenemos que desarrollar una nueva clase que se parece mucho a una que ya tenemos pero necesita información (características y comportamiento) adicional.
 - **Solución:** Creamos una clase derivada de la antigua y añadimos nueva funcionalidad sin tener que reescribir el código común
 - **Generalización** de la clase dada.(La clase vehículo es una generalización de la clase 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:** Movemos el código que se repite a un único sitio (la clase padre)





Herencia

¿Para qué sirve?



Los atributos y métodos que aparecen en azul en la clase padre se repiten en las clases hijas. (Dibujo izda)

No es necesario repetir el código, basta con decir que una clase **extiende** a la otra o **hereda** de ella. (Dibujo dcha)





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.





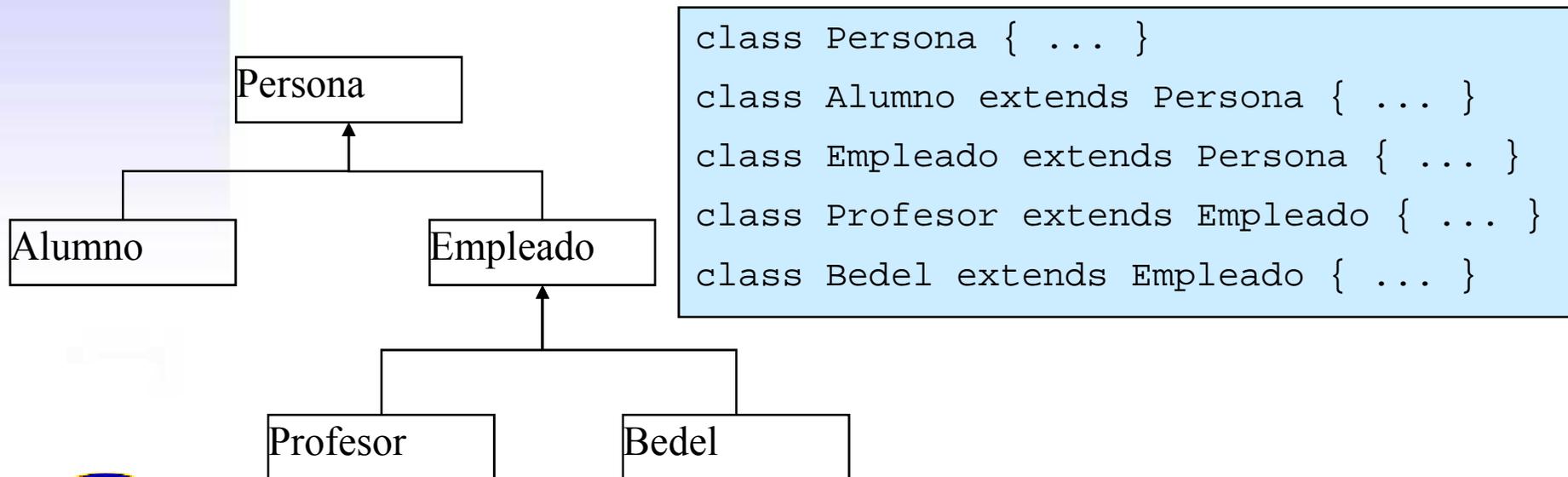
Herencia

Declaración de clases derivadas



- La sintaxis para declarar clases derivadas es:

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



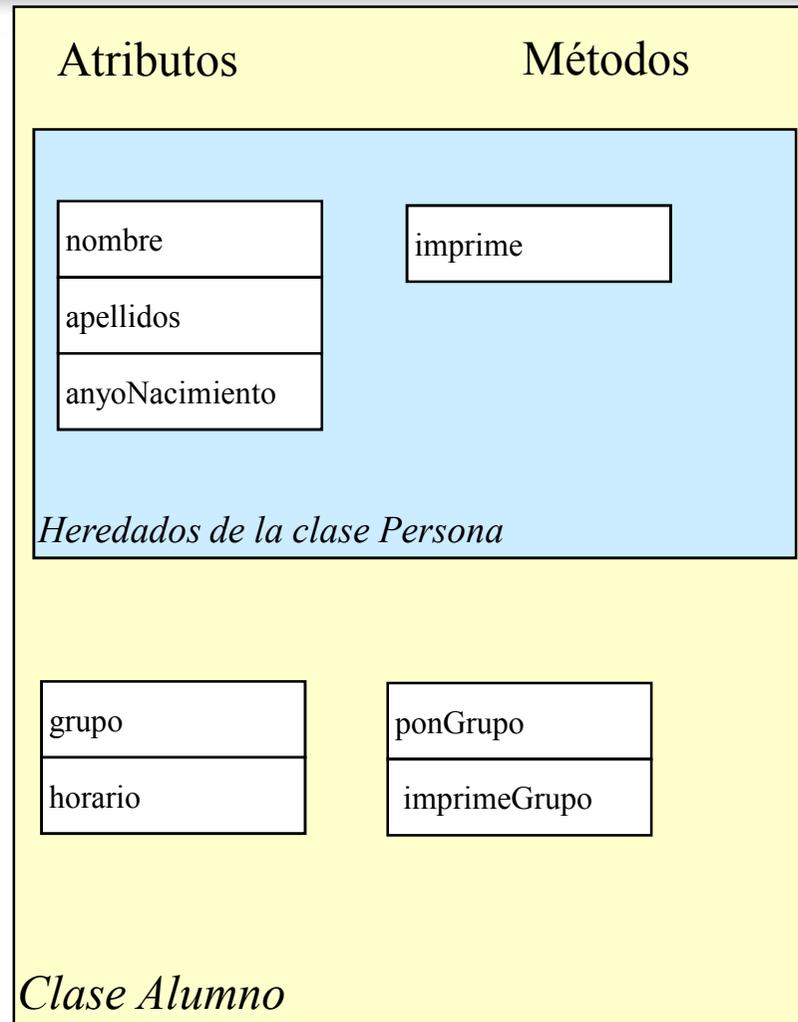
```
class Persona { ... }
class Alumno extends Persona { ... }
class Empleado extends Persona { ... }
class Profesor extends Empleado { ... }
class Bedel extends Empleado { ... }
```





Herencia

Clase derivada





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.print("Datos Personales: " + nombre  
                          + " " + apellidos + " (" +  
                          + anyoNacimiento + ")");  
    }  
}
```





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)
        throws Exception {
        if (grupo == null || grupo.length() == 0)
            throw new Exception ("Grupo no válido");
        if (horario != 'M' && horario != 'T')
            throw new Exception ("Horario no válido");

        this.grupo = grupo;
        this.horario = horario;
    }

    public void imprimeGrupo(){
        System.out.print(" Grupo " + grupo + horario);
    }
}
```





Herencia

¿ Cómo se usa? Ej.: Prueba.java



```
public class Prueba {
    public static void main (String[] args) throws Exception{

        Persona vecina = new Persona ("Luisa", "Asenjo Martínez", 1978);
        Alumno unAlumno = new Alumno ("Juan", "Ugarte López", 1985);
        unAlumno.ponGrupo("66", 'M');

        vecina.imprime();

        System.out.println();

        unAlumno.imprime();
        unAlumno.imprimeGrupo();
    }
}
```





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





Herencia

Jerarquía de herencia en Java



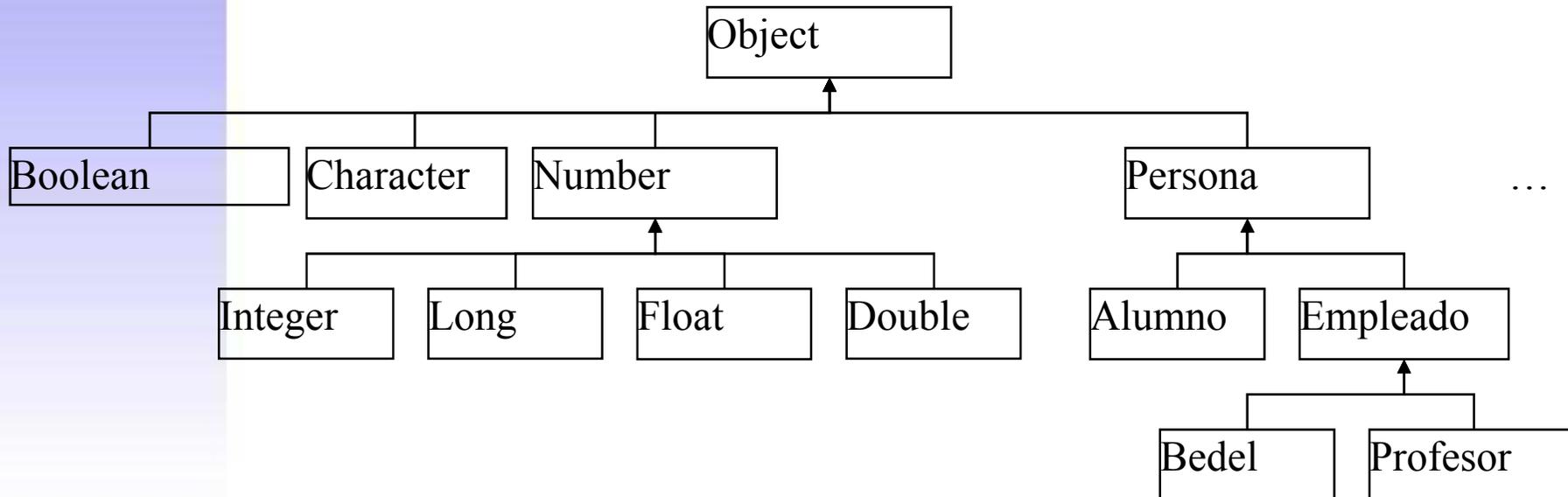
- En Java, todas las clases están relacionadas en **una única jerarquía** de herencia
- Una clase puede:
 - heredar explícitamente de otra clase
 - o bien heredar implícitamente de la clase Object (definida en el núcleo de Java)
- Esto se cumple tanto para las clases predefinidas como para las clases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





Herencia

Jerarquía de herencia en Java





Herencia

Reescritura



- 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



clase "Abuela" *Transporte*

String nombre = "terrestre"

clase padre *convoy*

vehiculo

String nombre = "turismo"

clase hija

moto

coche

String nombre = "Ferrari"

- ¿Cómo acceder a variables ocultas?

- *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





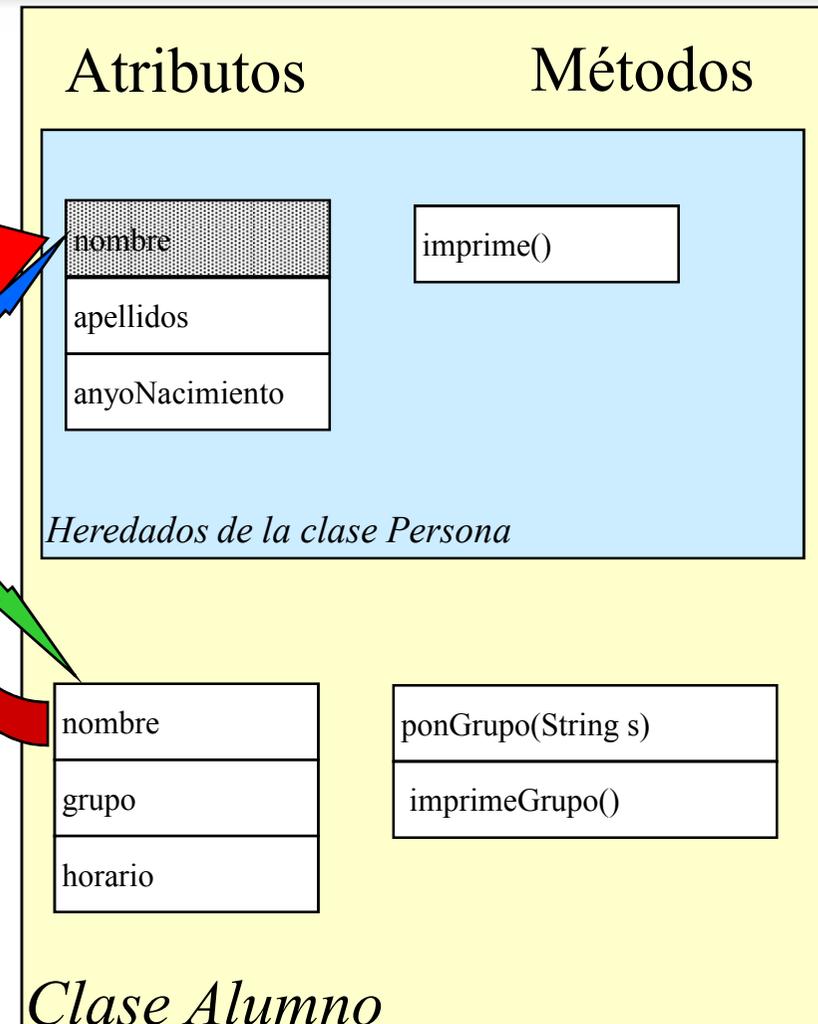
Reescritura I (Shadowing)

Ocultación de atributos



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

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





Reescritura I (Shadowing)

Ocultación de atributos



- Cuando se accede a un atributo, se usa el tipo de la referencia para decidir a qué valor se accede
- El atributo en la subclase tiene que tener el mismo nombre que en la superclase
 - Pero no necesariamente el mismo tipo
- No tiene mucha aplicación práctica
 - Permite a las superclases definir nuevos atributos sin afectar a las subclases





Reescritura I (Shadowing)



Ocultación de atributos. Ejemplo

```
class SuperShow {  
    public String str = "SuperStr";  
}
```

```
class ExtendShow extends SuperShow {  
    public int str = 7;  
}
```

```
class Show {  
    public static void main (String[] args)  
{  
        ExtendShow ext = new ExtendShow();  
        SuperShow sup = ext;  
        System.out.println(sup.str);  
        System.out.println(ext.str);  
    }  
}
```

Imprime SuperStr

Imprime 7





Reescritura I (Shadowing)

Ocultación de atributos



- Si definimos en una subclase una *variable* 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*.
 - Ej: "coche" *extiende* de "vehiculo" y "vehiculo" *extiende* de transporte.
 - Definimos en las tres clases la variable String nombre.
 - ¿Cómo sabemos si nos estamos refiriendo al nombre del transporte, al nombre del vehiculo o al nombre del coche?





Reescritura II (Overriding)

Redefinición de métodos. ¿Qué es?



- 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.
- Si ponemos el modificador *final* a un método no lo podemos redefinir.
- ¿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

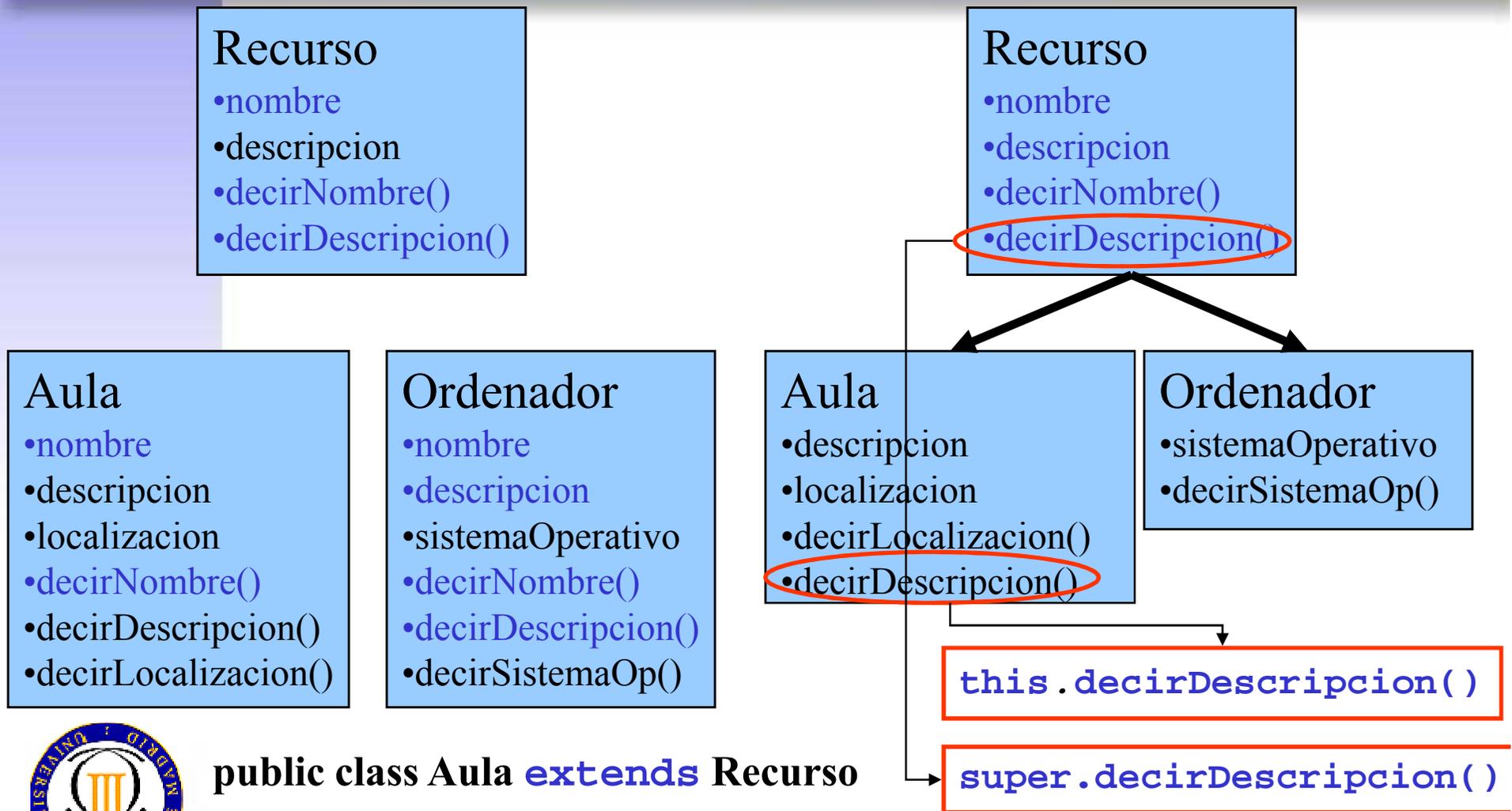




Reescritura II (Overriding)



Redefinición de métodos ¿Para qué sirve?

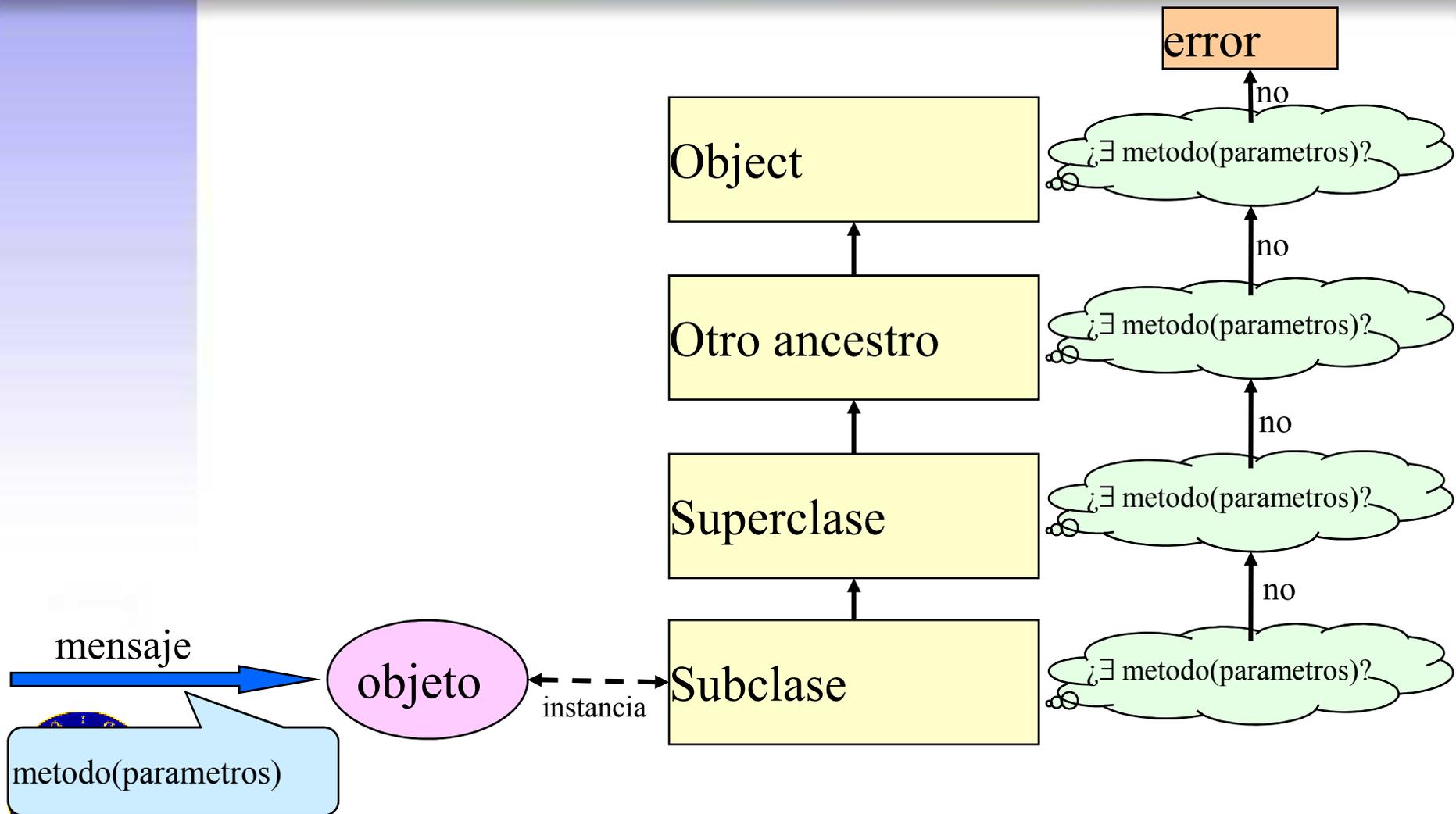


```
public class Aula extends Recurso
public class Ordenador extends Recurso
```



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 SuperShow {  
    public String str = "SuperStr";  
    public void show() {  
        System.out.println("Super.show: " + str);  
    }  
}  
class ExtendShow extends SuperShow {  
    public String str = "ExtendStr";  
    public void show() {  
        System.out.println("Extend.show: " + str);  
    }  
}  
class Show2 {  
    public static void main (String[] args) {  
        ExtendShow ext = new ExtendShow();  
        SuperShow sup = ext;  
        sup.show();  
        ext.show();  
    }  
}
```

Ambas imprimen:
"Extend.show: ExtendStr"





Reescritura II (Overriding)

Métodos finales



- La reescritura de métodos es útil para
 - Ampliar la funcionalidad de un método
 - Particularizar la funcionalidad de un método a la clase derivada
- 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**





Referencias `this` y `super`



- `this` referencia al objeto de la clase actual
- `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 clase base
 - Para reescribir métodos, 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);  
    }  
}
```





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.
- En este caso *no hay ocultación* de la información, se puede acceder a los dos métodos.





Sobrecarga (Overloading)

¿Para qué sirve?



Aula

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

*Son dos métodos distintos porque aunque tengan el mismo nombre tienen **distintos argumentos***

*Tienen **distinta funcionalidad** como ocurre en el caso que se muestra en el ejemplo*

describe el aula en general

describe el mueble que se encuentra en el aula y que le pasamos como parámetro





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

Notas a recordar



- Cuando se crea un objeto sus miembros se *inicializan* 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 y herencia



- Para la creación de un objeto se siguen los siguientes pasos:
 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**
- Por ejemplo, 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 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, anyoNacimiento);  
    this.grupo = grupo;  
    this.horario = horario;  
}
```





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
```

- **Metodos:** 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 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.
- 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. 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;
```



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 para heredar 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 cte y no puede ser modificado en las clases hijas	No se puede cambiar su valor, es cte . Se suele utilizar en combinación con static
	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.

