Programación de sistemas

Orientación a Objetos en Java

I. Programación ***Basada*** en objetos

II. Programación ***orientada*** a objetos

Ingeniería Telemática

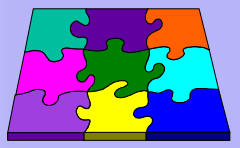
M. Carmen Fernández Panadero

<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

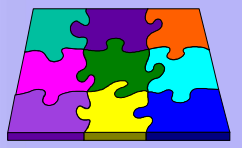
Contenidos



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

Herencia

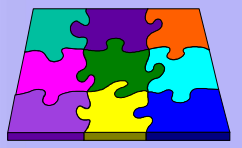
¿Qué es?¿Para qué sirve?



- 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?



Recurso
•nombre
•descripcion
•decirNombre()
•decirDescripcion()

Aula
•nombre
•descripcion
•localizacion
•decirNombre()
•decirDescripcion()
•decirLocalizacion()

Ordenador
•nombre
•descripcion
•sistemaOperativo
•decirNombre()
•decirDescripcion()
•decirSistemaOp()

Recurso
•nombre
•descripcion
•decirNombre()
•decirDescripcion()

Aula
•localizacion
•decirLocalizacion()

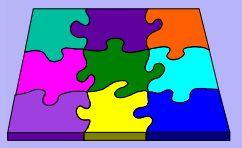
Ordenador
•sistemaOperativo
•decirSistemaOp()

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. (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

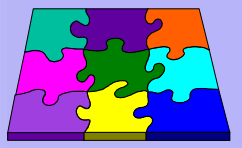
Nomenclatura



- Si definimos la clase coche a partir de la clase vehículo se dice que:
  - "coche" ***hereda*** las variables y métodos de "vehiculo"
    "coche" ***extiende*** de "vehiculo"
  - "coche" es ***subclase*** de "vehiculo"
    clase ***derivada***
    clase ***hija***
  - "vehiculo" 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.

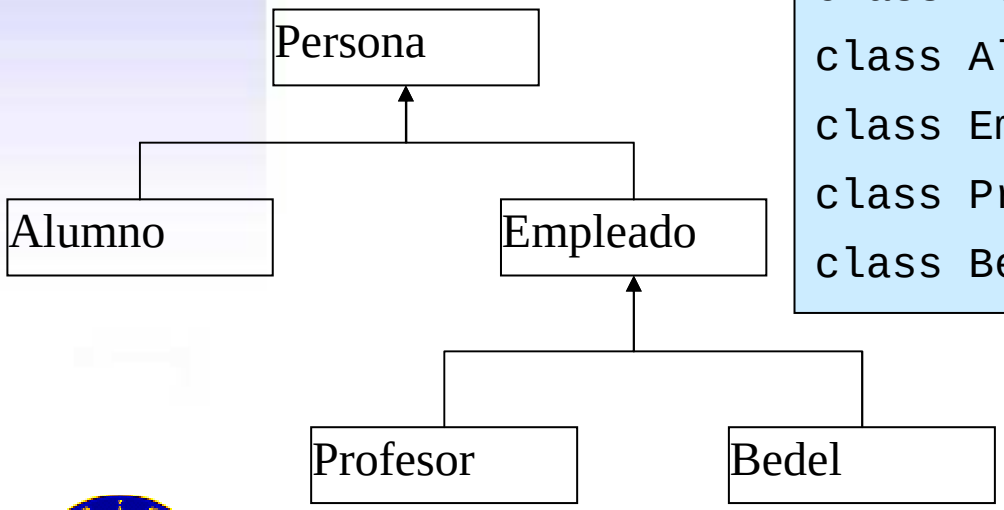
Herencia

Declaración de clases derivadas



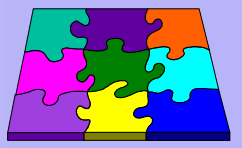
- La sintaxis para declarar clases derivadas es:

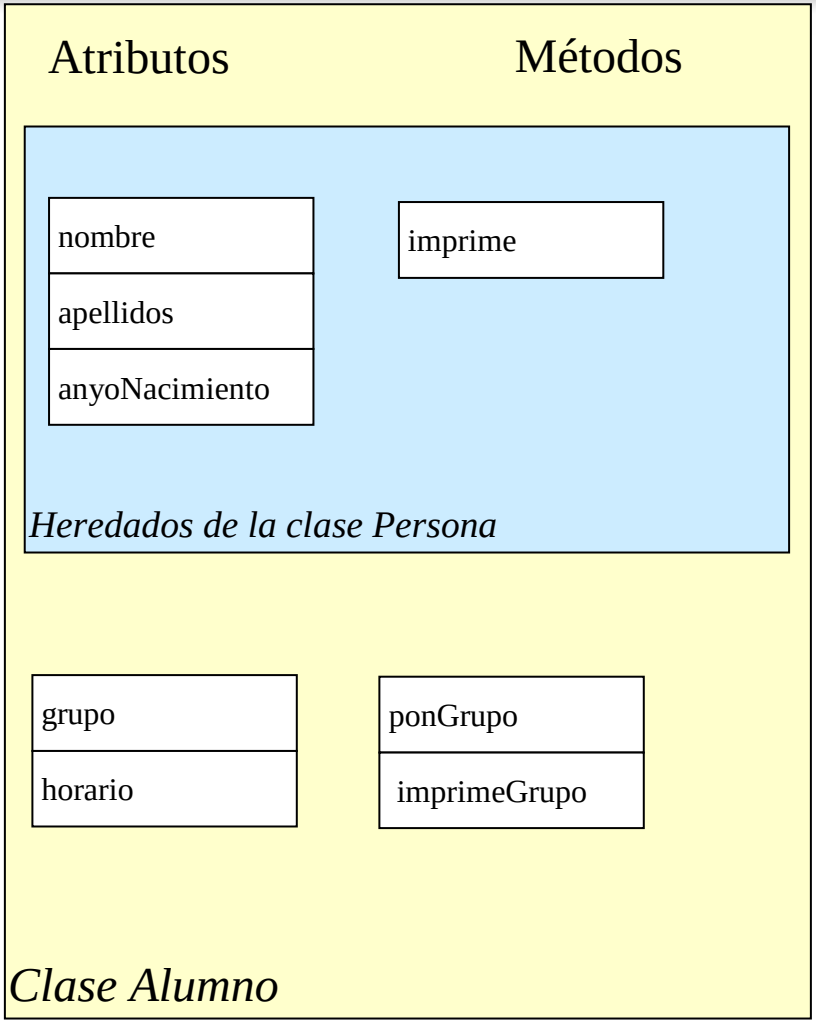
```
class ClaseDerivada extends ClaseBase { ... }
```




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

Herencia
Clase derivada





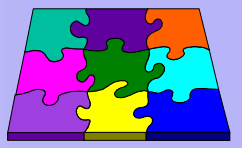
Herencia

¿ 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

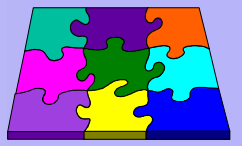
¿ Cómo se usa? Ej.: Alumno.java



```
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

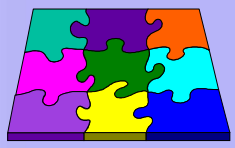
¿ Cómo se usa? Ej.: Prueba.java



```
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

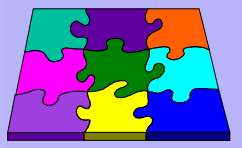
Consecuencias de la extensión de clases



- 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

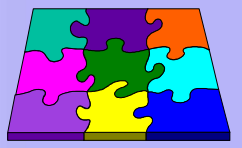
Jerarquía de herencia en Java

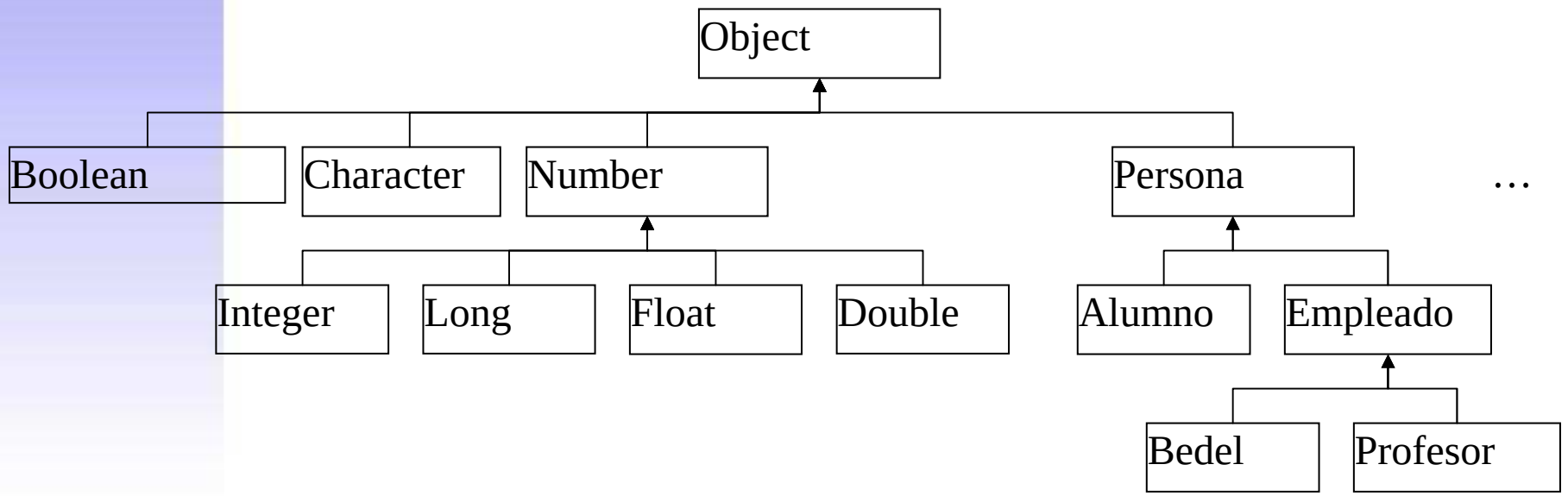


- En Java, todas las clases están relacionadas en **una única jerarquía** de herencia
- Una clase puede:
  - heredar explícitamente de otra clase
  - o bien heredar implícitamente de la clase Object (definida en el núcleo de Java)
- Esto se cumple tanto para las clases predefinidas como para las clases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

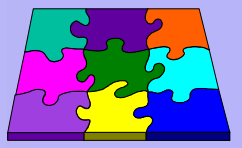
Herencia

Jerarquía de herencia en Java



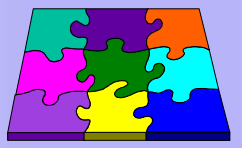


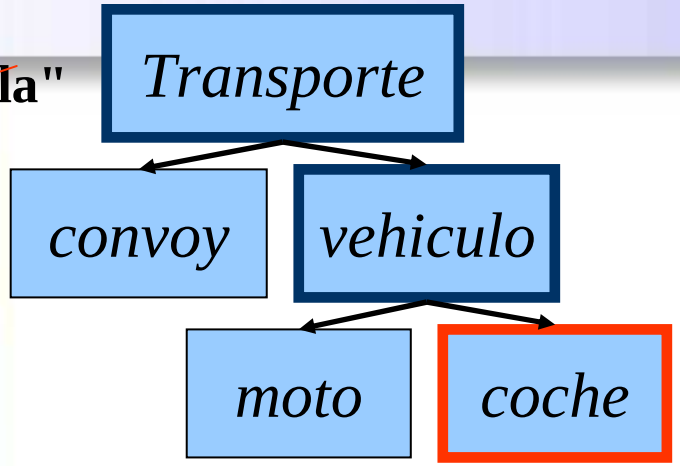
Herencia
Reescritura



- 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



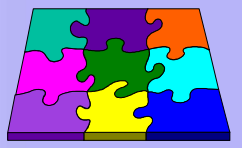


- ¿Cómo acceder a variables ocultas?
  - ***nombre*** (nombre del coche)
  - ***this.nombre*** (nombre del coche)
  - ***super.nombre*** (nombre del vehículo)
  - ***((vehiculo)this).nombre*** (nombre del vehículo)
  - ~~***super.super.nombre***~~ (Mal)
  - ***((transporte)this).nombre*** (nombre del transporte)

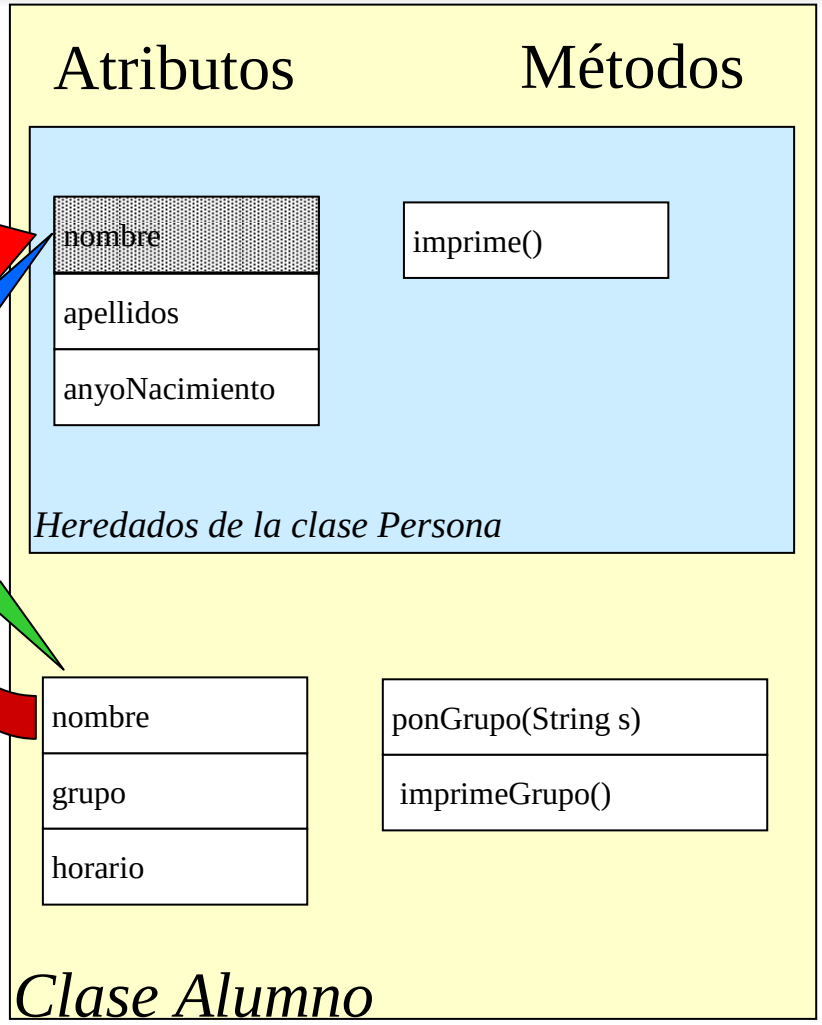
variables clase hija: visibles

Variables clases padre ocultas

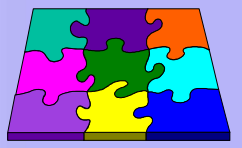
Reescritura I (Shadowing)
Ocultación de atributos



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

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

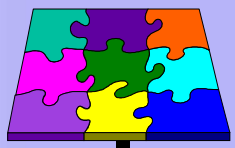


Reescritura I (Shadowing)
Ocultación de atributos



- 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)



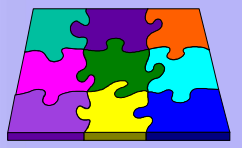
Ocultación de atributos. Ejemplo

```
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

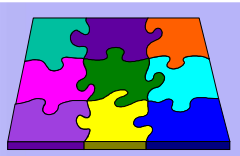
Reescritura I (Shadowing)
Ocultación de atributos



- 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?



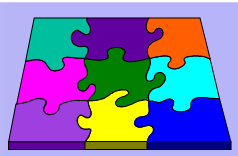
- Si definimos en una subclase un ***método*** con la misma **signatura** (nombre + tipo y número de parámetros) que en la superclase el de la superclase queda oculto.
- Si ponemos el modificador ***final*** a un método no lo podemos redefinir.
- ¿Cómo acceder a métodos ocultos?:
  - ***arrancar()*** (ejecuta el método arrancar del coche)
  - ***this.arrancar()*** (ejecuta el método arrancar del coche)
  - ***super.arrancar()*** (método arrancar del vehículo)
  - ~~***super.super.nombre***~~ (Mal)

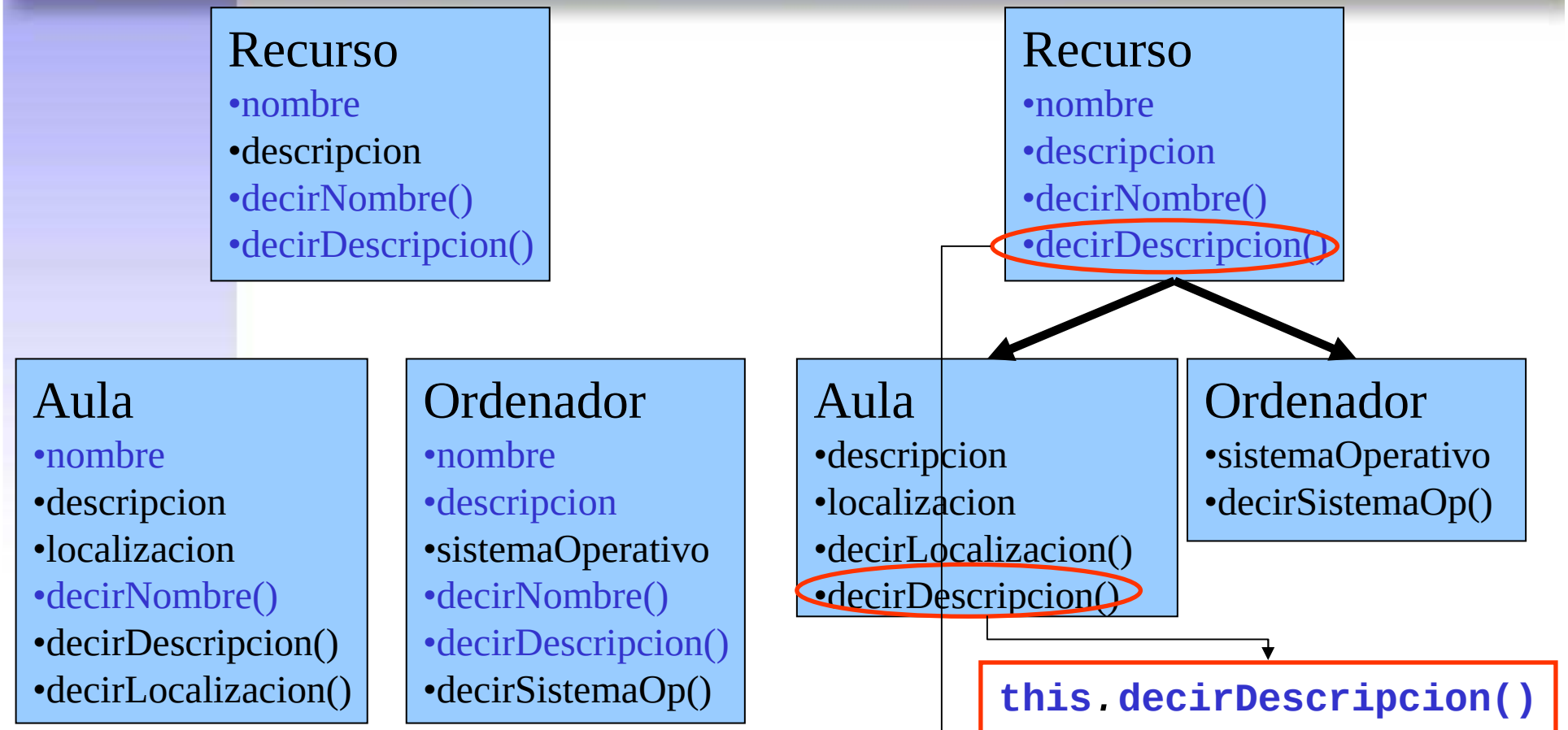
métodos clase hija: visibles

métodos clases padre: ocultos

Reescritura II (Overriding)

Redefinición de métodos ¿Para qué sirve?



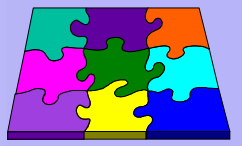


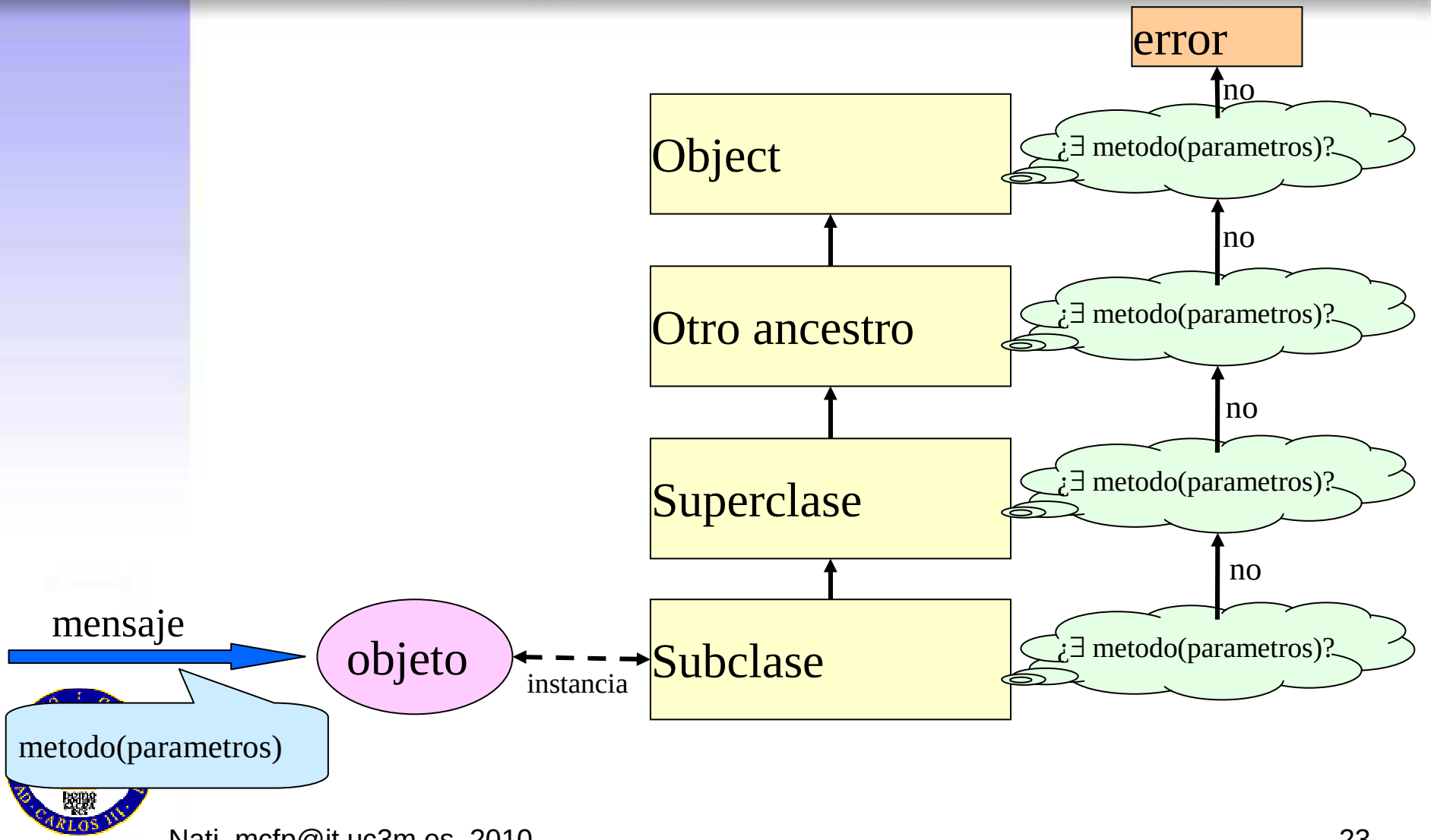
public class Aula `extends` Recurso

public class Ordenador `extends` Recurso

Reescritura II (Overriding)

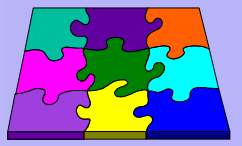
Redefinición de métodos





Reescritura II (Overriding)

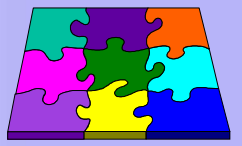
Redefinición de métodos



- 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)

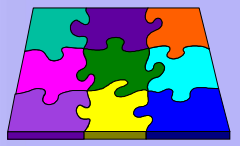
Redefinición de métodos. Ejemplo



```
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();
  }
}
```

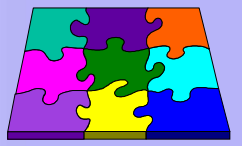
Ambas imprimen:
"Extend.show: ExtendStr"

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`**

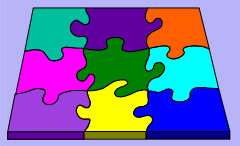
Referencias this y super



- **`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`**

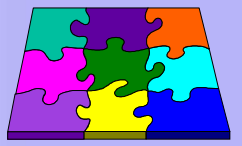
```
public class Alumno extends Persona {
 // el resto permanence igual
  public void imprime(){
    super.imprime();
    System.out.print(" Grupo " + grupo + horario);
  }
}
```

Reescritura vs. sobrecarga



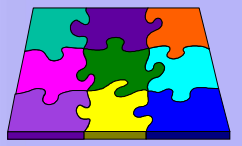
- **Reescritura**: La subclase sustituye la implementación de un método de la superclase
  - Ambos métodos tienen que tener la misma signatura
- **Sobrecarga**: Existe más de un método con el mismo nombre pero distinta signatura
  - Los métodos sobrecargados pueden definirse en la misma clase o en distintas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

Constructores y herencia



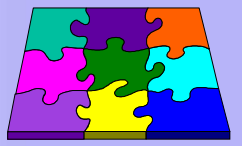
- Para la creación de un objeto se siguen los siguientes pasos:
  1. Se crea su parte base
  2. Se añade su parte derivada
  – Si la clase base del objeto hereda a su vez de otra, en el paso 1 se aplica el mismo orden de creación, hasta llegar a **`Object`**
- Por ejemplo, en la creación de un objeto **`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)

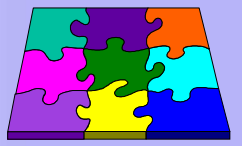
Constructores y herencia



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;
}
```

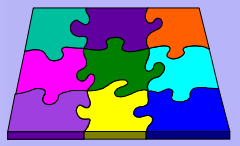
Constructores y herencia



2. Indicándolo explícitamente

```
public Alumno (String nombre, String apellidos,
        int anyoNacimiento, String grupo, char
  horario) {
  super(nombre, apellidos, anyNacimiento);
  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 sobreescribir dicho método.

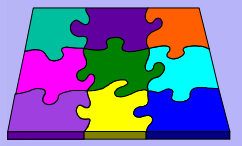
```
public final void myMethod(){} //no podemos sobreescribir myMethod
```

  - **Clases**: Impide la extensión de clases. No se puede "heredar de ella"

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

Modificadores y acceso

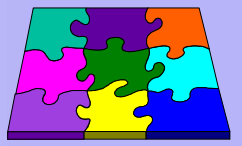
Static (miembros estáticos)



- 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		*clase*	*metodo*	*atributo*
acceso	public	Accesible desde cualquier otra clase		
	(friendly)	Accesible sólo desde clases de su propio paquete		
	protected	Se aplica a clases internas	Accesible desde la clase y sus subclases	
	private	Se aplica a clases internas	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.

(*) http://www.coderanch.com/t/527185/java/java/Understanding-Modifiers-classes