

# **Tema 7 Utilización avanzada de clases**

---

7.1.- La herencia.

- Restricciones y constructores.

7.2.- Atributos de clases.

7.3.- Clases y métodos abstractos y finales.

7.4.-Tipos de atributos y modificadores de ámbito.

7.5.- Interfaces.

7.6.-Librería de clases.



# Herencia de clases

---

- La herencia permite que una **clase herede las características y el comportamiento** de otra clase (**atributos y métodos**), y a continuación modificar este comportamiento según lo necesite.
- Para heredar una clase utilizamos la palabra reservada **extends** seguida del nombre de la clase de la que hereda.



# Restricciones de la herencia.

---

- No se heredan, los atributos y métodos con modo de acceso **private**.
- No se hereda un atributo de la clase padre si en la clase hija se define un atributo con el mismo nombre que en la clase padre.
- No se hereda un método si este está sobrecargado.



# Restricciones de la herencia.

---

- Se heredan los atributos y métodos con modo de acceso `public` y `protected`.
- Se heredan los atributos y métodos con modo de acceso `package` (por defecto sin especificar nada) si la clase padre e hija pertenecen al mismo paquete.



# Constructores

---

- Los constructores de la clase padre **no se heredan** pero si se pueden invocar desde la subclase.
- Exista o no constructor en la subclase, se hace una llamada al constructor por defecto de la superclase.



# Constructores

---

- *Si la primera instrucción de un constructor de una subclase no es una invocación a otro constructor con **this** o **super**, Java añade de forma invisible e implícita una llamada **super()** con la que invoca al constructor por defecto de la superclase.*
- *Luego continúa con las instrucciones de inicio de las variables de la subclase y luego sigue con la ejecución normal. **Si en la superclase no hay constructor por defecto ocurrirá un error.***



# Constructores

---

- *Si se invoca a constructores de superclases mediante **super(...)** en la primera instrucción, entonces se llama al constructor seleccionado de la superclase, luego inicia las propiedades de la subclase y luego sigue con el resto de sentencias del constructor.*



# Constructores

---

- *Finalmente, si esa primera instrucción es una invocación a otro constructor de la clase con **this(...)**, entonces se llama al constructor seleccionado por medio de **this** y realiza sus instrucciones, y después continúa con las sentencias del constructor.*
- *La inicialización de variables la habrá realizado el constructor al que se llamó mediante **this(..)**.*



# Casting de clases

---

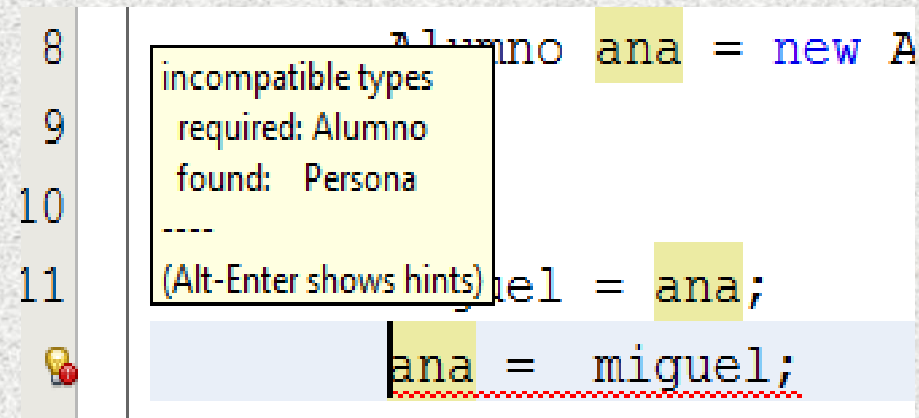
- El operador de casting o moldeado se puede utilizar con objetos de clases, igual que los utilizamos con los tipos básicos.
- Los objetos no se convierten de una clase a otra, igual que un tipo float no se convierte a double simplemente utilizando el operador (double), lo que podemos hacer es convertir referencias para indicar la subclase concreta a la que pertenece esa referencia



# Casting de clases

- La razón de los casting está en que es posible asignar referencias de una superclase a una subclase, pero no al revés.

```
public class PruebaInstituto {  
    public static void main(String arg[]) {  
        Persona miguel = new Persona("Miguel", "24781195");  
        Alumno ana = new Alumno("Ana", "22435688");  
        miguel = ana;  
        ana = (Alumno) miguel;  
        System.out.println(miguel);  
        System.out.println(ana);  
    }  
}
```





# Casting de clases

---

- Aunque el objeto miguel (Persona) reciba la referencia de ana (Alumno) no podrá acceder a los atributos y métodos de la clase Alumno, pero es necesario que contenga una referencia a Alumno para poder hacer el casting a Alumno.
- Un error de tipos incompatibles provoca una excepción del tipo **ClassCastException**.



# Instanceof

---

**Instanceof** nos permite comprobar si un objeto pertenece a una determinada clase.

**objeto instanceof clase**

Devuelve true si el Objeto pertenece a la clase.



# Las clases en JAVA

---

## ■ Tipos *de clases* :

***Superclase:*** Es la clase de la cual otra clase hereda todos sus atributos y métodos.

***Ejemplo.*** Class Nif extends Dni

*Declara una clase Nif que hereda todos los atributos y métodos de la clase Dni.*



# Modificadores de clase

---

*modificador* **class** NombreClase [**extends** NombreSuperclase] [**implements** listaDeInterfaces ]

- Si no se especifica ningún modificador , la clase será visible en todas las declaradas en el mismo paquete.
- Si no se especifica ningún paquete, se considera que pertenece a un paquete por defecto al que pertenecen todas las clases que no declaran explícitamente el paquete al que pertenecen.



# Modificadores de clase : tipos

---

- Son palabras reservadas que se anteponen a la declaración de la clase.
- Los modificadores posibles son:
  - **public**
  - **abstract**
  - **final**



# public

---

- Cuando se crean varias clases que se agrupan formando un paquete (package), **sólo** las declaradas **public** pueden ser **accedidas** desde otro **paquete**.
- Toda clase **public**, debe ser declarada en un **fichero fuente** con el **nombre** de esa clase pública: **NombreClase.java**
- *En un fichero fuente puede haber más de una clase, pero **sólo una** con el modificador **public**.*



# abstract

---

- Las clases abstract no pueden ser instanciadas.
- Sirven para declarar subclases que deben redefinir los métodos declarados abstract.
- Los métodos de una clase abstract pueden no ser abstract, pero tampoco se podrán instanciar objetos de la clase.
- Cuando existe algún método abstract, la clase debe ser declarada abstract , en caso contrario el compilador dará un error.



# Ejemplo: abstract

```
abstract class Animal {
```

```
    String nombre;
```

```
    int patas;
```

```
    public Animal (){}  
    ↗
```

Constructores

```
    public Animal (String n, int p ) {
```

```
        nombre= n;
```

```
        patas = p;
```

```
    }
```

```
    abstract void habla ();
```

```
} //Fin de Animal  
↘
```

//método abstracto que debe ser redefinido por las subclases



## Ejemplo: abstract

```
class Perro extends Animal {
```

```
// La clase Perro es una subclase de la clase abstract Animal
```

```
String raza ;
```

```
public Perro (String n, int p, String r ) {
```

```
    super (n, p );
```

```
    raza = r; }
```

**Llama al constructor de la Superclase**

```
public void habla () {
```

```
System.out.println ("\\Me llamo " +nombre);
```

```
System.out.println ("Mi raza es " +raza ) ;
```

```
} //End habla
```

```
} //End Animal
```

***Este método es necesario redefinirlo para poder instanciar objetos de la clase Perro***



## Ejemplo: abstract

```
class Gallo extends Animal {
```

```
// La clase Gallo es una subclase de la clase abstract  
Animal
```

```
    public Gallo (String n, int p) {  
        super (n, p );  
    }
```

```
// Redefinimos habla para poder instanciar objetos de  
la clase Gallo.
```

```
    public void habla () {  
        System.out.println("\nSoy un gallo, me llamo  
"+nombre );  
    }
```

```
} //End Animal
```



## Ejemplo: abstract

```
class EjemAbstracta {  
    public static void main(String argm[]) {  
        Perro toby; //objeto toby de la clase Perro  
        toby = new Perro("Toby", 4, "Bernardo");  
        Gallo kiko = new Gallo("Kiko", 2);  
        kiko.habla();  
        toby.habla();  
    }  
}
```



**Ejercicio propuesto** Define la clase abstract para instanciar objetos alumnos y profesores.

---

```
Abstract class Persona {  
    static int numPersonas = 0 ;    // atributo de clase  
    String nombre;                  // atributo de objeto  
    public Persona (String n) {    // constructor  
        nombre =n;  
        numPersonas++; }  
  
    Abstract void categoria();  
    public void muestra () {  
        System.out.println("Soy " + nombre + " somos " +  
            (numPersonas - 1) + "personas más");  
    }  
}  
}  
//End clase Persona
```



## class Alumno

```
class Alumno extends Persona {  
    int nmatri;  
    String curso;  
    static int totalAlum ;  
    public Alumno (String n, String nif, String cur, int matri ) {  
        super(n,nif);  
        nmatri=matri;  
        curso= cur;  
        totalAlum ++;  
    }  
    public void Categoria() { // redefinimos el método abstract  
        System.out.println("Soy " +nombre+ " de " + curso + "somos en  
        clase : "+totalAlum + " Mi DNI es " +dni );  
    }  
}
```



## Class Profe

```
class Profe extends Persona {
```

```
    String espec;
```

```
    public Profe (String n, String nif, String espe) {  
        super (n,nif);  
        espec= espe;  
    }
```

```
    public void Categoria () {
```

```
        System.out.println ("Soy profesor de" +espec + "me  
        llamo " +nombre);
```

```
    }
```

```
}
```



## Ejemplo:

```
class Instituto {  
    public static void main ( String argm [ ] ) {  
        Alumno al1= new Alumno ("Lucas","9924", "1º ASIR ", 835 );  
        Alumno al2 =new Alumno ("Dani", "5444", "1º DAW ",956);  
        Profe prof1 = new Profe("Javi","8869","Informática") ;  
        prof1.Categoria();  
        al1.Categoria();  
        System.out.println(al2.nombre);  
    }  
}
```



# final

---

- Una clase declarada **final** impide que pueda ser **superclase** de otras clases. Ninguna clase puede heredar de una clase **final**.
- A diferencia del abstract, pueden existir en la clase métodos **final**, sin que la clase que los contiene sea final.
- Una clase puede ser a la vez:
  - **public abstract**
  - **public final**



# Resumen

- Declaración de una clase

Modificadores `class` `NombreClase`

- Modificadores de clase

**public**      Fichero fuente con el nombre de la clase.

**abstract**    No se podrán crear objetos de esta clase.

**final**      No puede ser superclase de otras. No se puede heredar.



# Declaración de atributos

---

- Los atributos sirven para almacenar valores de los objetos que se instancian a partir de una clase.

- Sintaxis

[modifiDeAmbito ] [static ] [final][transient] [volatile] tipo  
nombreAtributo



# Tipos de atributos

---

- ☒ **static**
- ☒ **final**
- ☒ **transient**
- ☒ **volatile**



# Atributos *static*

- Ejemplo -

**static tipo nombreAtributo**

```
class Persona {  
    static int numPersonas = 0 ;    // atributo de clase  
    String nombre ;                // atributo de objeto  
    public Persona (String n) {    // constructor  
        nombre =n;  
        numPersonas++; }  
  
    public void muestra () {  
        System.out.print ("Soy "+nombre);  
        System.out.print("pero hay "+(numPersonas-1) + "personas  
        más"); }  
}
```



# Atributos de clase - *static*

---

```
class Grupo {  
    public static void main(String args[ ] ) {  
        Persona p1, p2, p3;  
        // se crean tres instancias del atributo nombre y sólo  
        // una del atributo numPersonas  
        p1 = new Persona ("Pedro");  
        p2 = new Persona ("Juan");  
        p3 = new Persona ("Susana");  
        p2.muestra() ;  
        p1.muestra();  
    }  
}
```



# Métodos static

---

- Los métodos **static** son métodos de clase (no de objeto) y por tanto, no necesitan instanciar la clase (crear un objeto de esa clase) para poder llamar a ese método.
- Se ha estado utilizando hasta ahora siempre que se declaraba una clase ejecutable, ya que para poder ejecutar el método **main()** no se declara ningún objeto de esa clase.
- Los métodos de clase (**static**) únicamente pueden acceder a sus atributos de clase (**static**) y nunca a los atributos de objeto (**no static**).



# Ejemplo:

```
class EnteroX {  
    int x;  
    static int  getx() {  
        return x; }  
    static void setX(int nuevaX) {  
  
        x = nuevaX; }  
} //End class
```

Mostraría el siguiente mensaje de error por parte del compilador:

MetodoStatic1.java:4: Can't make a static reference to nonstatic variable x in class EnteroX.  
return x;

MetodoStatic1.java:7: Can't make a static reference to nonstatic variable x in class EnteroX.  
x = nuevaX;  
2 errors.



## Ejemplo:

Sí que sería correcto:

```
class EnteroX {  
    static int x;  
    static int getx() {  
        return x; }  
    static void setX(int nuevaX) {  
        x = nuevaX; }  
}  
  
class AccedeMetodoStatic {  
    public static void main(String argumentos[ ]) {  
        EnteroX.setX(4);  
        System.out.println(EnteroX.getx()); }  
}
```

Al ser los métodos **static**,  
puede accederse a ellos sin  
tener que crear un objeto.

**Entero X:**



# Atributos **-final**

---

- La palabra **final** calificando a un atributo, sirve para declarar constantes.
- Si además es **static**, se puede acceder a dicha constante, anteponiendo el nombre de la clase, sin necesidad de instanciarla o crear un objeto de la misma.
- El valor de un atributo final debe ser asignado en la declaración.



## Atributos - *final* - ejemplo

---

```
class Circulo {  
    final double PI=3.14159265;  
    int radio;  
    Circulo (int n) { radio= n; }  
    public double area ( ) { return PI*radio*radio ; }  
}  
  
class AtributoFin {  
    public static void main (String args [ ] ) {  
        Circulo c = new Circulo(15) ;  
        System.out.println( c.area() ) ; }  
}
```



# Atributos transient

---

- ◆ Los atributos de un objeto, por defecto se consideran persistentes. Esto significa que a la hora de almacenarlos, por ejemplo en un fichero, los valores de los atributos deben almacenarse también.
- ◆ Aquellos atributos que no forman parte del estado persistente del objeto, porque almacenan estados transitorios o puntuales del objeto se declaran **transient**



## Atributos **transient** - ejemplo -

---

```
class Atributo3 {  
    int var1, var2;  
    transient int numVecesModificado=0;  
    void modifica(int v1, int v2) {  
        var1=v1;  
        var2=v2;  
        numVecesModificado++;  
    }  
}
```



# Atributos **volatile**

---

- Si una clase contiene atributos de objeto que son modificados asíncronamente por distintos **threads** que se ejecutan concurrentemente, se pueden utilizar atributos **volatile**, para indicarle a la MVJ este hecho, y cargar el atributo desde memoria antes de utilizarlo, y volver a almacenarlo en memoria después, para que cada thread pueda “verlo” en un estado coherente.



# modificadores de ámbito de atributos

---

- ☒ private
- ☒ public
- ☒ protected
- ☒ El ambito por efecto



# Ámbito de atributos - private -

---

- El modificador de ámbito **private** es el más restrictivo de todos. Todo atributo **private** es **visible únicamente dentro de la clase** en la que se declara.
- No existe ninguna forma de acceder al mismo si no es a través de algún método (no **private**) que devuelva o modifique su valor.
- Una buena metodología de diseño de clases es declarar los **atributos private** siempre que sea posible, ya que esto evita que algún objeto pueda modificar su valor.



# Ámbito de atributos - public -

- Es el menos restrictivo de todos. Un atributo **public** será visible en cualquier clase que desee acceder a él, simplemente anteponiendo el nombre de la clase.
- Las aplicaciones bien diseñadas minimizan el uso de los atributos **public** y maximizan el uso de atributos **private**. La forma apropiada de acceder y modificar atributos de objetos es a través de métodos que accedan a los mismos.



```
final class Empleado {  
    public String nombre;  
    public String dirección;  
    private int sueldo;  
}
```

Los atributos nombre y dirección podrán ser modificados por cualquier clase. Ejemplo:

```
emple1.nombre="Pedro López";
```

Mientras que el sueldo no puede ser modificado directamente por otra clase que no sea Empleado.

## *Ejemplo : public*

---

Para que la clase estuviera bien diseñada, se deberían haber declarado `private` los tres atributos y declarar métodos para modificarlos .

De estos métodos, el que modifica el atributo sueldo es de tipo `private` para que no pudiera ser utilizado por otra clase distinta de Empleado.



# Ámbito de atributos - **protected**

---

- Los atributos **protected** pueden ser accedidos por las clases y subclases del mismo paquete (*package*) pero no pueden ser accedidos por subclases de otro paquete.

```
package PProtegido;

public class Protegida {
    protected int valorProtegido;
    public Protegida(int v) {
        valorProtegido=v;
    }
}
```

```
package PProtegido;

public class Protegida2 {
    public static void main(String args[]) {
        Protegida p1= new Protegida(0);
        p1.valorProtegido = 4;
        System.out.println (
            p1.valorProtegido); }
}
```



# Ámbito de atributos - **protected**

---

- **Protegida2** pertenece al mismo paquete que la clase **Protegida**, y puede acceder a su atributo **protected**, **valorProtegido**.

```
package OtroPaquete;
import PProtegido.*;

public class Protegida3 {
    public static void main(String args[ ] ) {
        Protegida p1= new Protegida(0);
        p1.valorProtegido = 4;      Error, valor protegido
        System.out.println(p1.valorProtegido);}
    }
Error valor protegido
```



## Ámbito de atributos - **protected**


- En este caso, se importa el paquete **PProtegido** para poder acceder a la clase **Protegida**, pero el paquete en el que se declara **Protegida3** es distinto al que contiene **Protegida**, por lo que no se puede acceder a sus atributos **protected**.



# Ámbito de atributos - **protected**

---

```
package OtroPaquete;  
import PProtegido.*;  
  
public class Protegida4 extends Protegida {  
    public Protegida4(int v) {  
        super(v);    }  
    public void modifica(int v) {  
        valorProtegido=v; }  
    }  
}
```



Protegida4 puede acceder a valorProtegido, a través del método modifica, porque es heredada de Protegida



# Ámbito de atributos - **protected**

---

**En resumen:** Un atributo protegido **sólo** puede ser modificado por clases del mismo paquete, ahora bien, si se declara una subclase, entonces esa subclase es la encargada de proporcionar los medios para acceder al atributo protegido



## Ámbito **por defecto** de los atributos

---

- ***El ámbito por defecto de los atributos.***

Los atributos que no llevan ningún modificador de ámbito, pueden ser accedidos desde las clases del mismo paquete, pero no desde otros paquetes.



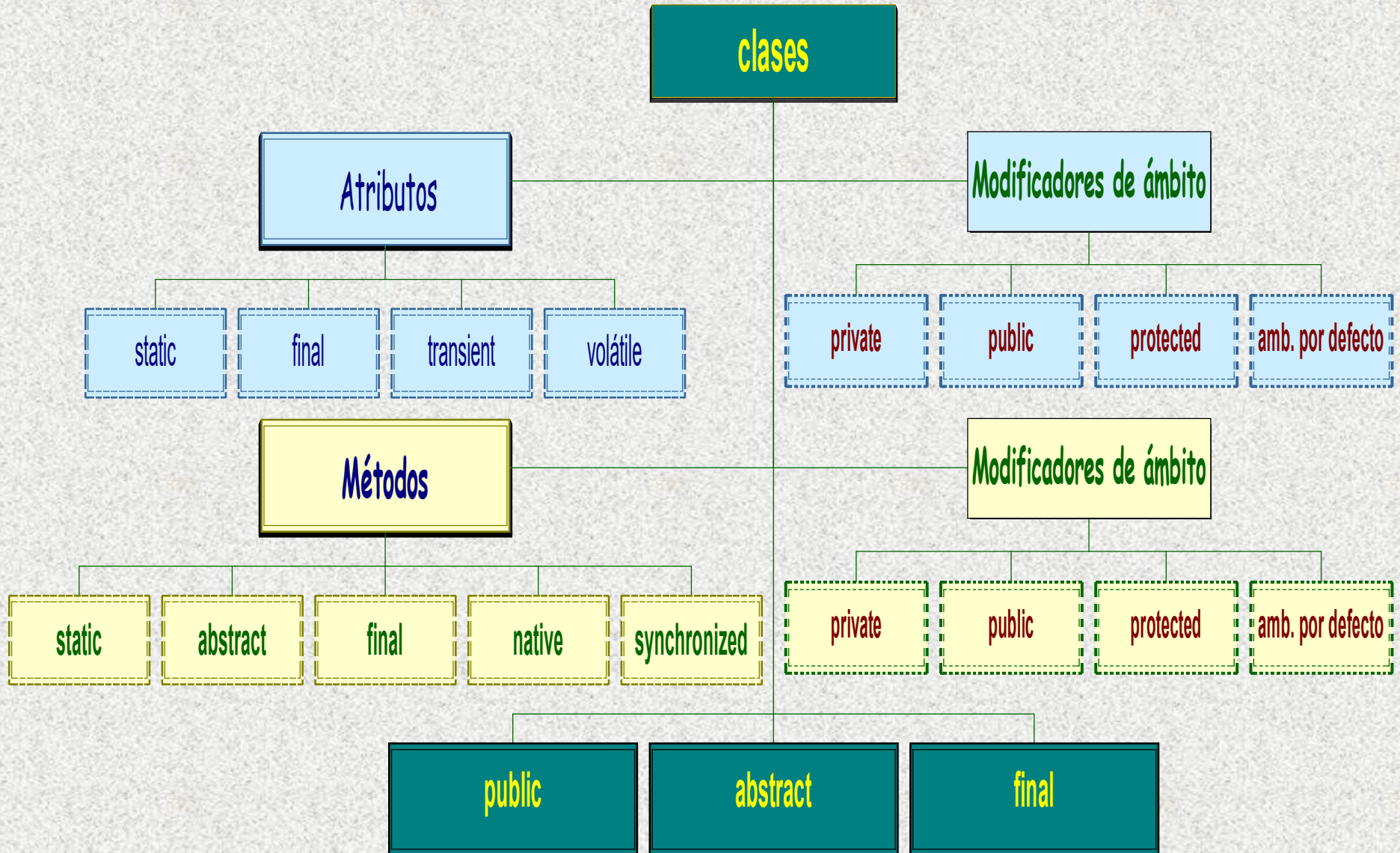
## Ejemplo de Alumnos y Profesores.

---

```
abstract class Persona {  
    final String nombre ;  
    String dni ;  
    public Persona(String n, String nif) {  
        nombre = n;  
        dni= nif;  
    }  
    abstract void Categoria ();  
}
```



# Resumen





# Interface

---

- Es un conjunto de constantes y métodos, pero de éstos últimos sólo el formato, no su implementación.
- Cuando una clase declara una lista de interfaces, asume que se van a redefinir todos los métodos definidos en la interface.

*class NombreClase implements Interface1, Interface2..  
InterfaceN.*

*Class Nif extends Dni implement OperacionesAritmeticas,  
OperacionesLogicas.*



# Interfaces.

---

- En Java no está permitida la herencia múltiple (más de una superclase). Una aproximación a este concepto es la utilización de una sola superclase y una o varias interfaces.
- Una interfaz es un conjunto de constantes y métodos abstractos.



# Interfaces

---

- Cuando una clase declara **una lista de interfaces** mediante la **cláusula implements** hereda todas las constantes definidas en el interface y se compromete a redefinir todos los métodos del interface .
- Mediante el uso de interfaces se consigue que distintas clases implementen métodos con el mismo nombre, y se comprometen a implementarlas
- Los interfaces permiten al programador, definir un conjunto de funcionalidades sin tener “ni idea” de cómo serán implementadas



# Interfaces

- Por ejemplo, el interface **java.lang.Runnable** especifica que cualquier clase que implemente este interface deberá redefinir el método `run()`. Así, la MVJ puede realizar llamadas a este método (sabe que existe) aunque no sepa qué hace realmente.
- Un interface se declara de forma similar a las clases.

*Sintaxis: Declaración de interface {  
Cuerpo de interface*

```
interface Padre {  
    int MAXIMO=1000;  
    void metodo1(){}  
}
```

}



# Interfaces

---

- En Java se usan interfaces para utilizar los métodos de la interfaz sobre objetos de los que no conocemos la clase (uno de los usos más interesantes de las interfaces).
- Las interfaces también se pueden heredar.



## *Ejemplo*

```
interface Mercancia{  
    public double getPrecio();  
    public String getDescripcion();  
}
```

```
interface MercanciaViva extends Mercancia{  
    public boolean necesitaComida();  
    public boolean necesitaRiego();  
}
```



```
public class PlantaJardin implements MercanciaViva{  
    double precio;  
    boolean estaRegada;  
    String descripcion;  
    PlantaJardin(double precio,String  descripcion{  
        this.precio = precio;  
        this.descripcion = descripcion;  }  
    public boolean necesitaComida(){  
        return false;  }  
    public double getPrecio(){  
        return precio;  }  
    public String getDescripcion(){  
        return descripcion;  }  
    public boolean necesitaRiego(){  
        return estaRegada;  }  
}
```



# La librería de clases en Java

---

`java.lang.Object`

`java.awt.Component`

`java.awt.Container`

`java.awt.Windows`

`java.awt.Frame`

Cada una de las clases `Object`, `Component`, `Container`, `Windows` y `Frame` heredan en sus definiciones los atributos y métodos de las clases precedentes, sobreescribiendolos cuando lo necesitan.

Los nombres están precedidos por las etiquetas **`java.lang`** y **`java.awt`** que no son nombres de clases, sino un mecanismo de agregación de clases, **propio de java**, denominado paquete (**Package**)



# Uso de Packages

---

`java.lang.Object`

`java.awt.Component`

`java.awt.Container`

`java.awt.Windows`

`java.awt.Frame`

Los paquetes se utilizan para agrupar clases que tienen funcionalidades comunes.

Existen un conjunto de paquetes predefinidos, donde están englobados todas las clases de la librería del lenguaje.

Para hacer referencia a la clase Frame:  
**`java.awt.Frame`**

Es más simple utilizar una directiva de importación : **`import java.awt.* ;`**

Importa todas las clases del paquete **`java.awt`**



# La clase Object

---

- Todas las clases de Java poseen una superclase común, es la clase **Object**.
- Al ser superclase de todas las clases de Java, todos los objetos Java en definitiva son de tipo Object, lo que permite crear **métodos genéricos**.

```
public class C{  
    public static void f(Object o){  
    ...  
    }  
}
```

```
public class D{  
    public static void main(String args[] ){  
        //todas las líneas son válidas  
        C.f(new String());  
        C.f(new int[7]);  
        C.f(new C());  
        ...  
    }  
}
```



# La clase Object

---

- Object proporciona métodos que son heredados por todas las clases.
- La idea es que todas las clases utilicen el mismo nombre y prototipo de método para hacer operaciones comunes como comparar, clonar, escribir,... y para ello habrá que redefinir esos métodos a fin de que se ajusten a las necesidades particulares de cada clase.

```
public String toString(){}  
.
```



# La clase Object - equals -

---

```
Coche uno=new Coche("Renault","Megane","P4324K");  
Coche dos=uno;           //dos y uno son referencias al mismo coche  
boolean resultado=(uno.equals(dos));           //Resultado valdrá true  
resultado=(uno==dos);           //Resultado también valdrá true  
dos=new Coche("Renault","Megane","P4324K");           //los mismos datos  
resultado=(uno.equals(dos));           //Resultado valdrá true  
resultado=(uno==dos);           //Resultado ahora valdrá false
```

En el ejemplo , **equals devuelve true** si los dos coches **tienen el mismo** modelo, marca y matrícula.

El operador **"=="** devuelve **true** si las dos referencias que se comparan apuntan al mismo objeto.



# Redefinición del método equals

---

```
public class Coche extends Vehículo{  
    ...  
    public boolean equals (Object o){  
        if ((o!=null) && (o instanceof Coche)){  
            if (((Coche)o).matricula==matricula && ((Coche)o).marca==marca &&  
                ((Coche)o).modelo==modelo))  
                return true;  
            else  
                return false  
        }  
        else  
            return false; //Si no se cumple todo lo anterior  
    }  
}
```



# La clase Object -- metodo clone --

---

- El método **clone** realiza una copia idéntica de un objeto.
- Es un método **protected** por lo que sólo podrá ser usado por la **propia clase** y sus descendientes, salvo que se le redefina con **public**.
- **La copia realizada por** clone es un nuevo objeto y por lo tanto tendrá una nueva referencia; es decir modificar el objeto clonado no afecta al original.
- El método clone en la clase **Object** **duplica literalmente todas las propiedades; cuando estas son tipos** primitivos no hay problemas, pero cuando son referencias a objetos, entonces el clonado fallará porque no duplica los objetos, sino que duplicará sólo las referencias.



# metodo : public Object clone()

---

- Para poder redefinir el método clone, la clase debe implementar la interfaz **Cloneable**, sino ocurriría una excepción del tipo:

## **CloneNotSupportedException.**

```
public class Coche extends Vehiculo implements Arrancable, Cloneable {  
    public Object clone(){  
        try{  
            return (super.clone());  
        }catch(CloneNotSupportedException cnse){  
            return null;  
        }  
    } .... //Clonación  
    Coche uno=new Coche();  
    Coche dos=(Coche)uno.clone();
```



# metodo : public Object clone()

---

Esta es una definición vaga del método clone ya que utiliza el clone de la propia clase **Object**. **No es lo habitual. Lo habitual es arreglar los problemas** que provoca clone con las referencias a objetos internas:



# metodo : public Object clone()

```
public class Coche extends Vehiculo implements Arrancable,  
Cloneable{
```

```
....
```

```
public Object clone(){
```

```
try{
```

```
Coche c=(Coche) super.clone();
```

```
c.motor=(Motor) motor.clone();
```

```
return c;
```

```
}catch(CloneNotSupportedException cnse){
```

```
return null;
```

```
}
```

```
}
```



## metodo : public Object clone()

---

```
public class Coche extends Vehiculo implements  
Arrancable,Cloneable{
```

```
....
```

```
public Object clone(){
```

```
try{
```

```
Coche c=(Coche) super.clone();
```

```
c.motor=(Motor) motor.clone();
```

```
return c;
```

```
}catch(CloneNotSupportedException cnse){
```

```
return null;
```

```
}
```

```
}
```



# metodo : public Object clone()

---

- Suponiendo que los coches constan de una propiedad llamada ***motor de clase Motor, clase que a su vez es clonable;***
- ***El código anterior sería el correcto*** para clonar, de otro modo con el código anterior a éste, los dos coches, el original y el clonado, compartirían motor.



## Resumen - Herencia -

- Uno de los objetivos fundamentales de la programación orientada a objetos es la de facilitar la *reutilización del código*. Ello permite volver a utilizar elementos que al haber sido ya realizados son bien conocidos y están, posiblemente, probados de forma exhaustiva.
- La herencia se emplea en el propio lenguaje, a lo largo del conjunto de librerías que posee.
- El lenguaje, da soporte a la definición de nuevas clases heredadas de las características ya definidas.
- La clase derivada o subclase hereda todos los atributos y métodos de la superclase o clase base.



## Resumen – Herencia -

En Java, la relación entre los elementos de una subclase y los de la superclase se pueden resumir de la forma siguiente:

- Todo atributo o método de la superclase que no aparezca en la definición de la subclase es heredado sin modificación alguna, exceptuando **los constructores**, ya que en cada nueva clase, incluso las derivadas, deben **definir los suyos propios**.
- Si no se implementa ningún constructor, entonces se genera uno sin argumentos por defecto.
- Los métodos de la superclase que se definan de nuevo en la clase derivada se sobrescriben. Esta definición se aplicará a los objetos de la clase derivada.
- En la clase derivada pueden añadirse atributos (que generalmente serán privados) y métodos adicionales.



## Ejemplo

- En el nivel más alto de la jerarquía tenemos la superclase (**PERSONA**), y descendiendo en la jerarquía vamos obteniendo especializaciones de la superclase, las subclases (**ESTUDIANTE y EMPLEADO**), o clases descendientes de la superclase.
- A su vez, las subclases pueden asumir el papel de superclase, si poseen clases descendientes en la estructura de clases.
- Éste sería el caso de la clase empleado si diferenciáramos entre *personal docente (PD)* y *personal administrativo y servicios (PAS)*.
- En la siguiente figura se ilustra el nuevo nivel de la jerarquía:



