Taller de Lenguajes II

Temas de hoy:

Herencia en JAVA

- Herencia en OO
- La cláusula **extends**
- Sobreescritura
- La palabra clave **super**
- La clase Object: los métodos **equals(), toString() y hashCode().**
- Implementación de herencia: cadena de invocación de constructores

Herencia

"es un rasgo o rasgos morales, científicos, ideológicos, etc., que, habiendo caracterizado a alguien, continúan advirtiéndose en sus descendientes o continuadores"

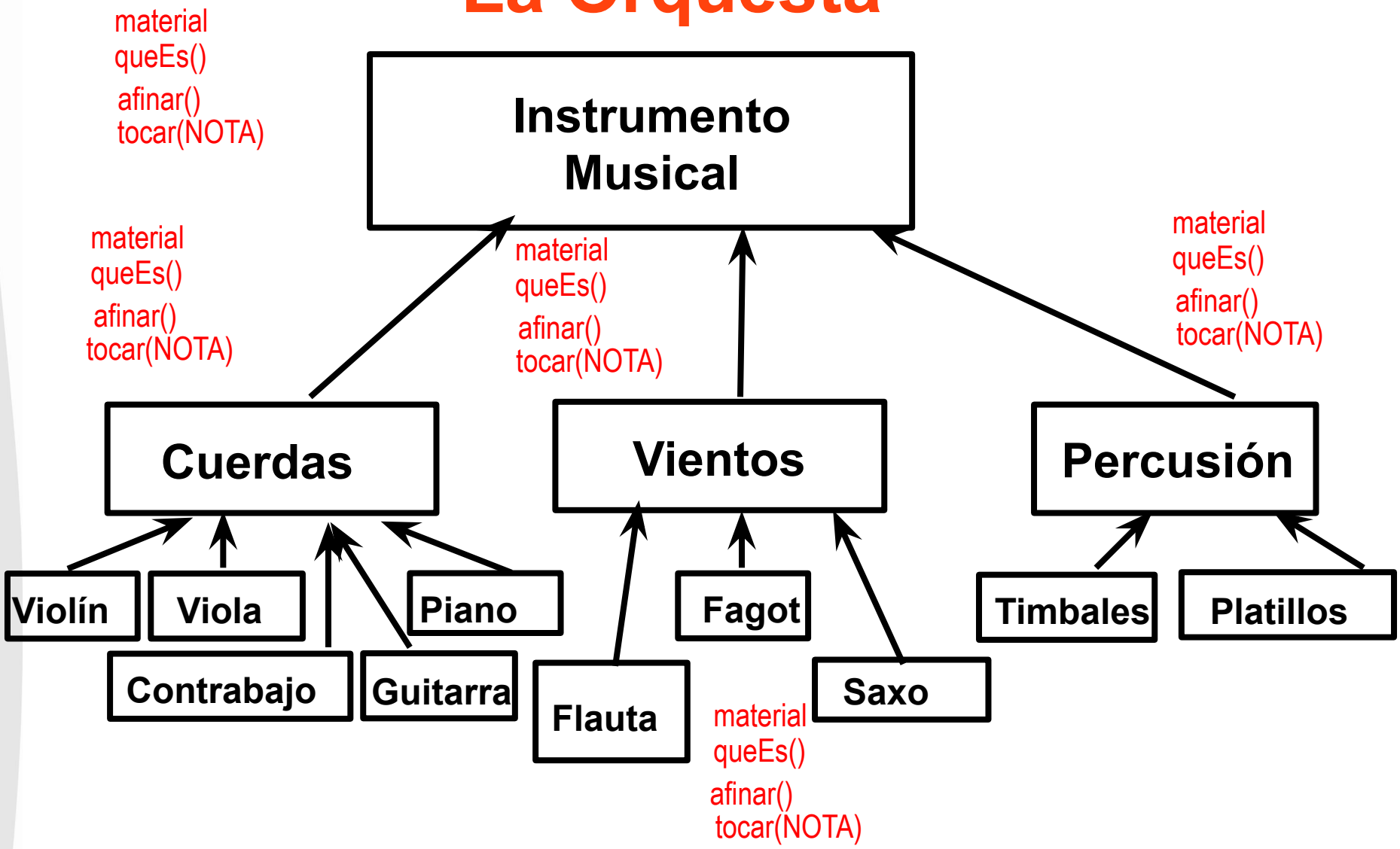
(Diccionario de la Real Academia Española)

Según el paradigma de objetos "la herencia es el mecanismo que permite definir una clase de objetos tomando como base la definición de otra clase"

Herencia
La Orquesta Sinfónica

qué Es()
afinar()
material
ejecutar()

Herencia
La Orquesta



Herencia
La Relación ES-UN
Generalización-Especialización

Un **Violín ES-UN** instrumento de **Cuerdas.**
Un **Violonchelo ES-UN** instrumento de **Cuerdas.**
Una **Viola ES-UN** instrumento de **Cuerdas.**
Una **Guitarra ES-UN** instrumento de **Cuerdas.**
Un **Piano ES-UN** instrumento de **Cuerdas.**

Una **Flauta ES-UN** instrumento de **Vientos.**
Una **Fagot ES-UN** instrumento de **Vientos.**
Un **Saxo ES-UN** instrumento de **Vientos.**

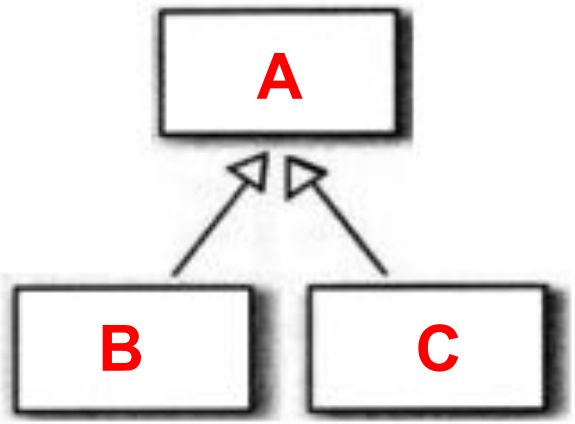
Un **Timbal ES-UN** instrumento de **Percusión.**
Un **Platillo ES-UN** instrumento de **Percusión.**

La consecuencia más importante de la relación **ES-UN** es la **herencia**.

Herencia

La programación orientada a objetos permite a las **clases expresar similitudes** entre objetos que tienen **características** y **comportamiento común**. Estas similitudes pueden expresarse usando **herencia**. El término **herencia** se refiere al hecho que una clase **hereda** los atributos (variables) y el comportamiento (métodos) de otra clase.

La clase **A** es la **superclase** y las clases **B** y **C** las **subclases.**



Una de las contribuciones más interesantes de la herencia es la flexibilidad para capturar y aprovechar al máximo las características comunes de diferentes clases de objetos.

La clase que hereda se llama **subclase** o **clase derivada** y a la clase de partida se la llama **superclase** o **clase base.**

"Sin la herencia, cada clase sería una unidad independiente, cada una desarrollada desde cero. Diferentes clases no tendrían ninguna relación entre sí, el desarrollador de cada clase proporciona los métodos de la manera que desea/elija. Cualquier consistencia entre clases es el resultado de la disciplina de los programadores. La herencia hace que sea posible definir un nuevo software de la misma manera que se introduce un concepto a un recién llegado, al compararlo con algo que ya es familiar "

(Brad Cox, es el creador de Objetive-C, el lenguaje adoptado por las plataformas OS X y iOS de Apple)

Herencia Simple y Herencia Múltiple

La **herencia simple** es aquella en la que cada **subclase hereda** de una **única clase**. En herencia simple cada **clase tiene un solo ascendiente y múltiples descendientes**.

La **herencia múltiple** es aquella en la que **una subclase tiene múltiples superclases**.

JAVA soporta herencia simple.

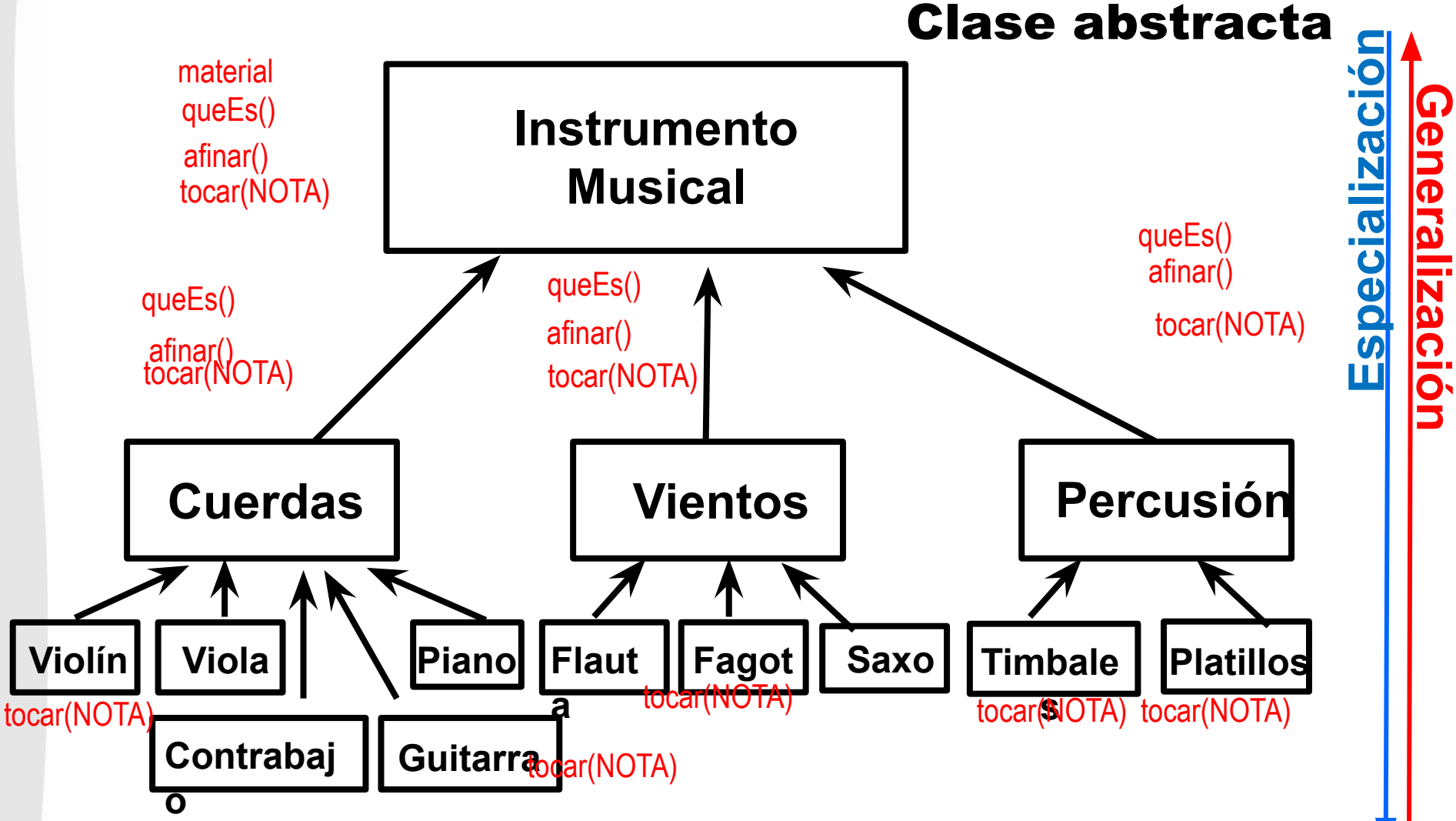
Especialización
Sobreescritura & Extensión

Si una **clase define** un **método** de instancia con el **mismo nombre, tipo de retorno y parámetros** que uno de los métodos definidos en su **superclase**, dicho método **sobrescribe** al de la superclase.

La **sobreescritura de métodos** o **reemplazo** es un concepto íntimamente ligado a la herencia. La **sobreescritura** permite **especializar el comportamiento.**

Cuando agregamos nuevos métodos en la subclase, también estamos **especializando comportamiento**, este mecanismo es de **extensión**.

Herencia y Especialización por Sobreescritura



¿Hay una manera general de **afinar** un instrumento y de **ejecutar una nota**?

En JAVA....

```java
package concurso;

public abstract class  InstrumentoMusical {
public abstract void afinar();

public abstract void tocar(Nota n);

public abstract String queEs();

}
```

```java
package concurso;

public class Cuerdas extends InstrumentoMusical {
    public void afinar() {
        System.out.println("Cuerdas.afinar()");
    }

    public void tocar(Nota n) {
        System.out.println("Cuerdas.tocar(NOTA.LA)");
    }

    public String queEs() {
        System.out.println("Cuerdas.queEs()");
        return "Cuerdas.queEs()";
    }
}
```

```java
package concurso;

public class Guitarra extends Cuerdas {

    @Override
    public void tocar(Nota n) {
        System.out.println("Guitarra.tocar(NOTA.FA)");
    }

}
```

Sobreescritura

```java
package concurso;

public class Violin extends Cuerdas {

    @Override
    public void afinar() {
        System.out.println("Violin.afinar()");
    }

}
```

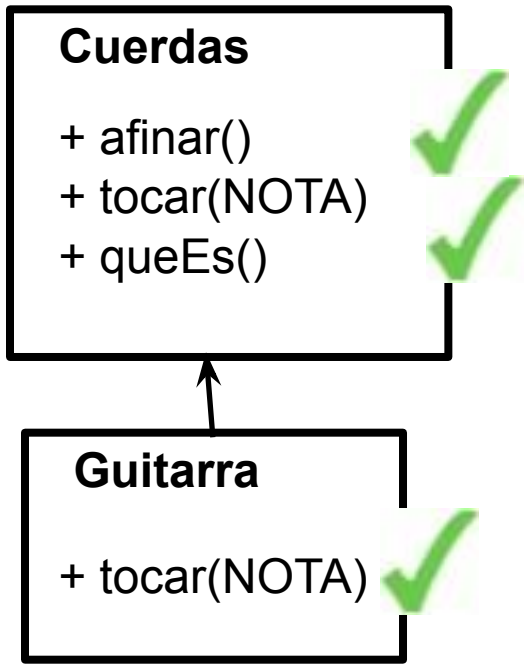
Sobreescritura

Automáticamente las subclases heredan las variables y métodos de la superclase

Métodos propios y heredados

¿Qué puedo hacer ? ¿Qué método se ejecuta?

```
Guitarra x= new Guitarra();
x.tocar(Nota.LA);
x.afinar();
x.queEs();
```



¿Está bien? ¿Qué método se ejecuta?

```
Cuerdas x= new Guitarra();
x.tocar(Nota.FA);
```

SI!!!

Sobreescritura y la palabra clave super

```java
package concurso;

public class Cuerdas extends InstrumentoMusical {
    public void afinar() {
        System.out.println("Cuerdas.afinar()");
    }

    public void tocar(Nota n) {
        System.out.println("Cuerdas.tocar(NOTA.LA)");
    }

    public String queEs() {
        System.out.println("Cuerdas.queEs()");
        return "Cuerdas.queEs()";
    }
}
```

¿Es posible invocar al método **afinar()** de la **clase Cuerdas** desde un método de la **clase Violin**?

SI!!!!!

```java
package concurso;

public class Violin extends Cuerdas {

    @Override
    public void afinar() {
        super.afinar();
        System.out.println("Violin.afinar()");
    }
}
```

¿Cómo?

Usando la palabra clave super

Ejemplo de sobreescritura y extensión

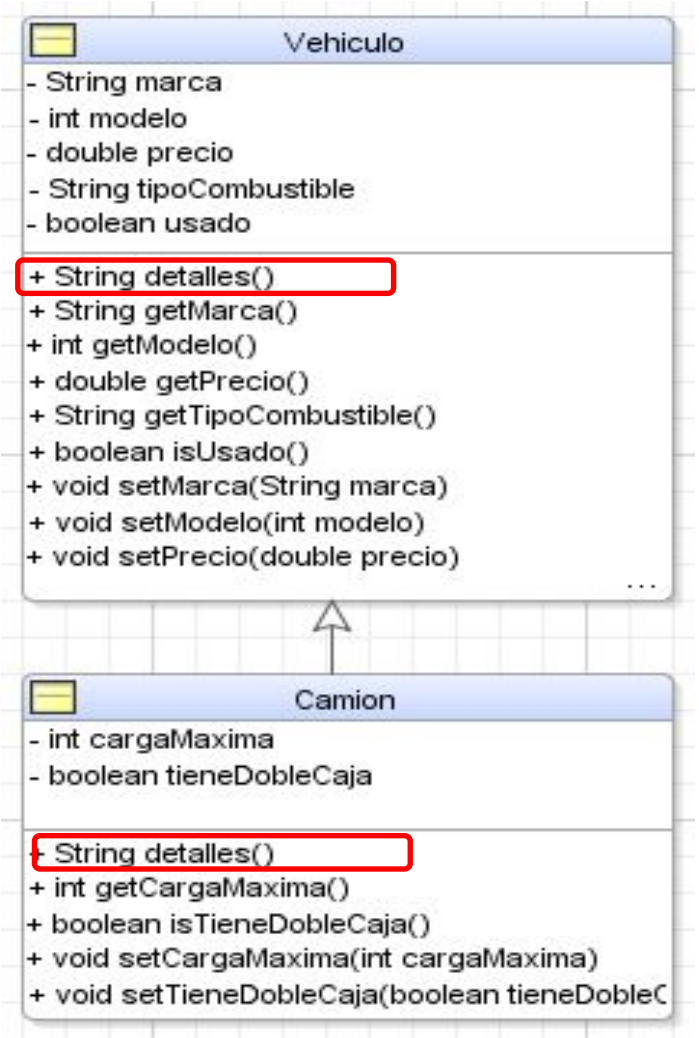
La clase **Camion** hereda todos los atributos y métodos de **Vehiculo** y:

Agrega dos atributos **cargaMaxima** y **tieneDobleCaja.**

Agrega los métodos:

getCargaMaxima(), **setCargaMaxima(int)**, **isTieneDobleCaja()** y **setTieneDobleCaja(boolean)**.

Sobreescribe el método **detalles()**



Continúa el ejemplo….

```
public class Vehiculo {
  private String marca;
  private double precio;
   //más atributos

  public String detalles() {
 return "Vehiculo marca: "+getMarca()
+ "\n"+ "Precio: "+ getPrecio();
   }

   // getters y setters
}
```

```
public class Camion extends Vehiculo {
   private boolean tieneDobleCaja;
   private int cargaMaxima;

   public String detalles() {
   return "Vehiculo marca: "+getMarca()
+ "\n"+"Precio: "+getPrecio()
+"\n"+"carga máxima:"+getCargaMaxima();
   }

   // getters y setters
}
```

El método detalles() definido en la clase Vehiculo se reemplazó o sobrescribió en la subclase Camion.

```
public class Test {
 public static void main(String args[]){
  Vehiculo v = new Vehiculo();
  v.setMarca("Ford");
  v.setPrecio(12000.4);
  System.out.println(v.detalles());

  Camion c = new Camion();
  c.setMarca("Scania");
  c.setPrecio(35120.4);
  c.setCargaMaxima(3000);
  System.out.println (c.detalles());
 }
}
```

Ejecutan métodos diferentes !!

SALIDA

```
Vehiculo marca: Ford
Precio: 12000.4
Vehiculo marca: Scania
Precio: 35120.4
carga máxima:3000
```

Continúa el ejemplo….

```
public class Vehiculo {
 private String marca;
 private double precio;

 public String detalles() {
  return "Vehiculo marca: "+getMarca()
+"\n" + "Precio: "+ getPrecio();
 }

}
```

¿Es posible invocar al método **detalles()** de la **clase Vehiculo** desde un método de la **clase Camion**?

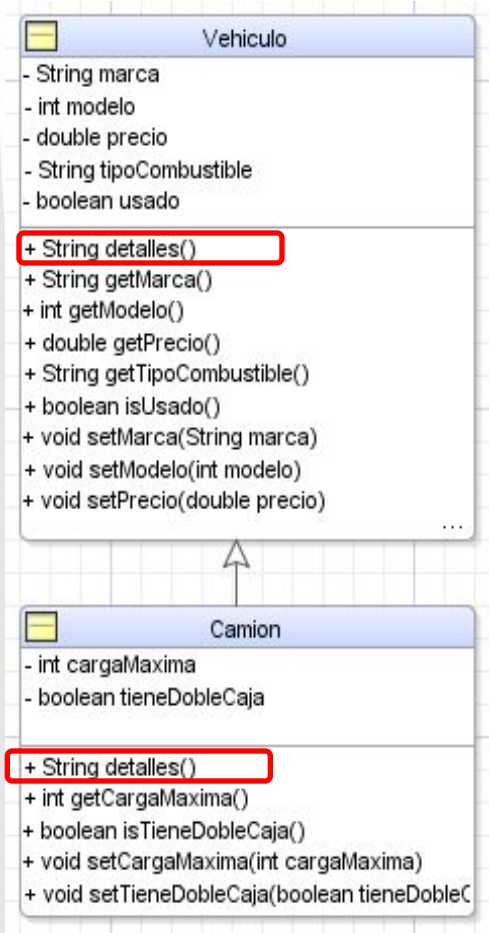
SI!!!!!

```
public class Camion extends Vehiculo {
 private boolean tieneDobleCaja;
 private int cargaMaxima;

 public String detalles() {
  return super.detalles()
 + "\n"
 + "carga máxima:"+getCargaMaxima();

 }
```

¿Cómo?

Usando la palabra clave super

Continúa el ejemplo…




```
Vehiculo vc = new Camion();
vc.detalles();
```

→ **upcasting**

¿Qué método detalles() se ejecuta?

El asociado con el objeto "real"; al que hace referencia la variable en ejecución, un objeto **Camion**. Esta característica se llama **binding dinámico** y es propia de los lenguajes OO.

¿Funciona?

```
Vehiculo vc = new Camion();          ✓
vc.setMarca("Mercedes Benz");        ✓
vc.setPrecio(35120.4);               ✓
vc.setCargaMaxima(3000);             ✗  No está disponible para Vehiculo
System.out.println(vc.detalles());
```

NO Compila

El **upcasting** es seguro, la **clase base tiene una interface pública** que es **igual o es un subconjunto de la clase derivada.**

La Orquesta Sinfónica

```
package concurso;

public class Orquesta {

    public static void main(String[] args) {
        InstrumentoMusical[] instrumentos=new InstrumentoMusical[5];

        instrumentos[0]=new Flauta();
        instrumentos[1]=new Guitarra();
        instrumentos[2]=new Violin();
        instrumentos[3]=new Percusion();
        instrumentos[4]=new Vientos();
        new Orquesta().afinarInstrumentos(instrumentos);

    }

    public  void afinarInstrumentos(InstrumentoMusical[] inst) {

        for (int i = 0; i < inst.length; i++) {
            inst[i].afinar();
        }

    }
}
```

InstrumentoMusical es una **clase abstracta**, sin embargo es un **tipo de datos**. Podemos declarar variables de ese tipo.

En un arreglo de **InstrumentoMusical** podemos guardar objetos de cualquiera de sus subclases (generalización, ES-UN)

¿Puede el compilador conocer a qué método afinar() se está invocando? **NO!!!**

Dinámicamente se resuelve qué método afinar() se invocará- BINDING DINÁMICO.
afinar() es polimórfico: tiene “múltiples formas” dependiendo del objeto receptor.

La clase Object

La clase **Object** es la superclase de todas las clases JAVA y está ubicada en el paquete **java.lang**

Cuando se declara una clase sin usar la palabra clave **extends** el compilador JAVA implícitamente agrega el código **extends Object** a la declaración de la clase.

Es equivalente a:

```
public class Vehiculo {
   private String marca;
   private double precio;
   //más propiedades

   public String detalles() {
    return "Vehiculo marca: "+getMarca()
      + "\n"+"Precio: "+ getPrecio();
   }

   //continúa la clase Vehiculo
}
```

```
public class Vehiculo extends Object{
   private String marca;
   private double precio;
   //más propiedades

   public String detalles() {
    return "Vehiculo marca: "+getMarca()
      + "\n"+"Precio: "+ getPrecio();
   }

 //continúa la clase Vehiculo
}
```

De esta manera estamos habilitados para **sobreescribir** los métodos **heredados de Object**.

La clase Object
El método equals()

Definición del método equals() de la clase Object:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

Plaza plaza1 = **new** Plaza(“Plaza Rocha”);
Plaza plaza2 = **new** Plaza(“Plaza Rocha”);
System.*out*.println("Las plazas son iguales? "+ plaza1.equals(plaza2));

¿Qué se imprime en pantalla?

Las plazas son iguales?
false

¿Es el resultado esperado?

NO!! Esperamos true, dado que los valores de los objetos son iguales, ambos objetos representan a la plaza Rocha

¿Cuándo es apropiado sobrescribir el método ***equals()*** de **Object**?

Cuando las instancias de una clase tienen una noción de igualdad lógica que difiere de la identidad o referencias de esas instancias. Este es generalmente el caso de clases que representan valores, como por ejemplo las clases Integer, Double, String, de la API de JAVA.

La clase Object
El método toString()

El método **public String toString(){}** retorna un string formado por el nombre de la clase, seguido del símbolo @ más la representación hexadecimal del código hash del objeto sobre el que se invoca al método toString().

```
Plaza plaza1 = new Plaza("Plaza Rocha");
System.out.println(plaza1);
```

capitulo2.laplata.Plaza@addbf1

El objetivo del método toString() es producir una representación textual, concisa, legible y expresiva del contenido del objeto.

¿Cuándo se invoca al método toString()?

Cuando se pasa un objeto como parámetro en los métodos print(), println(), printf(), println(). Internamente las distintas versiones del método print() invocan al método toString().

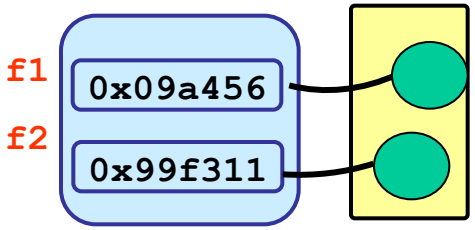
El método toString() también es usado por el Debugger, lo que facilita la interpretación de los pasos de la ejecución de un programa.

Es una buena práctica sobreesscribir el método toString() para producir un resultado amigable que permite informar mejor sobre el objeto en cuestión.

La clase Object
Los métodos equals() y toString()
TODO JUNTO

```
public class Fecha {
private int dia= 1;
private int mes= 1;
private int año=2023;
 // métodos de instancia
}
```

```
Fecha f1 = new Fecha();
Fecha f2 = new Fecha();
f1.equals(f2);  → false
f1==f2;         → false
f1.toString();  → Fecha@19821f
```



En este ejemplo el objetivo del método equals(Object o) es comparar el contenido de dos objetos y la del método toString() es producir una representación textual, concisa, legible y expresiva del contenido del objeto.

La clase Object
Los métodos equals() y toString()

```
public class Fecha {
 private int dia = 1;
 private int mes = 1;
 private int año = 2023;

public boolean equals(Object o){
  boolean result=false;
  if ((o!=null) && (o.getClass()==Fecha)) {
   Fecha f=(Fecha)o;
   if ((f.getDia()==this.getDia())
   &&(f.getMes()==this.getMes())&&
              (f.getAño()==this.getAño()))
result=true;
  }
  return result;
 }

 public String toString(){
                                    return
getDia()+"-"+getMes()+"-"+getAño();
 }

 public static void main(String args[]){
   Fecha f1, f2;
   f1 = new Fecha();
   f2 = new Fecha();
   System.out.println(f1==f2);
   System.out.println(f1.equals(f2));
   System.out.println(f1.toString());
```

Es un operador que permite determinar la clase real del objeto

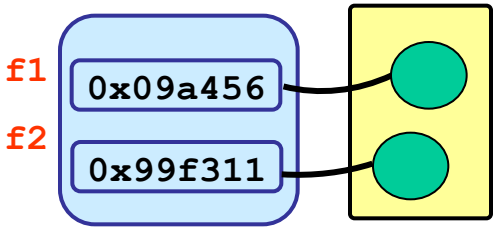
Ahora en la clase **Fecha**:
El método **equals(Object o)** cumple su objetivo: **comparar el contenido** de dos objetos de tipo **Fecha**. Es por esta razón que frecuentemente se lo **sobrescribe**.

El método **toString()** retorna un **String** con datos del objeto Fecha en una representación ***legible***.

¿Por qué usamos el operador == para comparar el día, mes y año de una fecha?

La salida es:

```
false
true
1-1-2023
```



La clase Object
El método hashCode()

El método hashCode() de la clase Object:

La sobrescritura del hashCode() está íntimamente relacionada con el framework de colecciones de java.

Toda clase que sobrescribe el **método equals()** debe **sobrescribir el hashCode()** para asegurar un funcionamiento apropiado de sus objetos en contenedores basados en hashing, por ejemplo HashSet, HashMap.

Una correcta sobrescritura del método hashCode() asegura que dos instancias lógicamente iguales tengan el mismo hashcode y en consecuencia las estructuras de datos basadas en hashing que almacenan y recuperan estos objetos, funcionarán correcta y eficientemente.

Es una buena práctica sobrescribir el método hashcode() cuando se sobreescribe el equals() y de esta manera se respeta el siguiente contrato de la clase **Object**: "si dos objetos son iguales según el método **equals**() entonces invocar al método **hashCode()** sobre cada uno de estos objetos, debe producir el mismo valor".

Resumiendo....

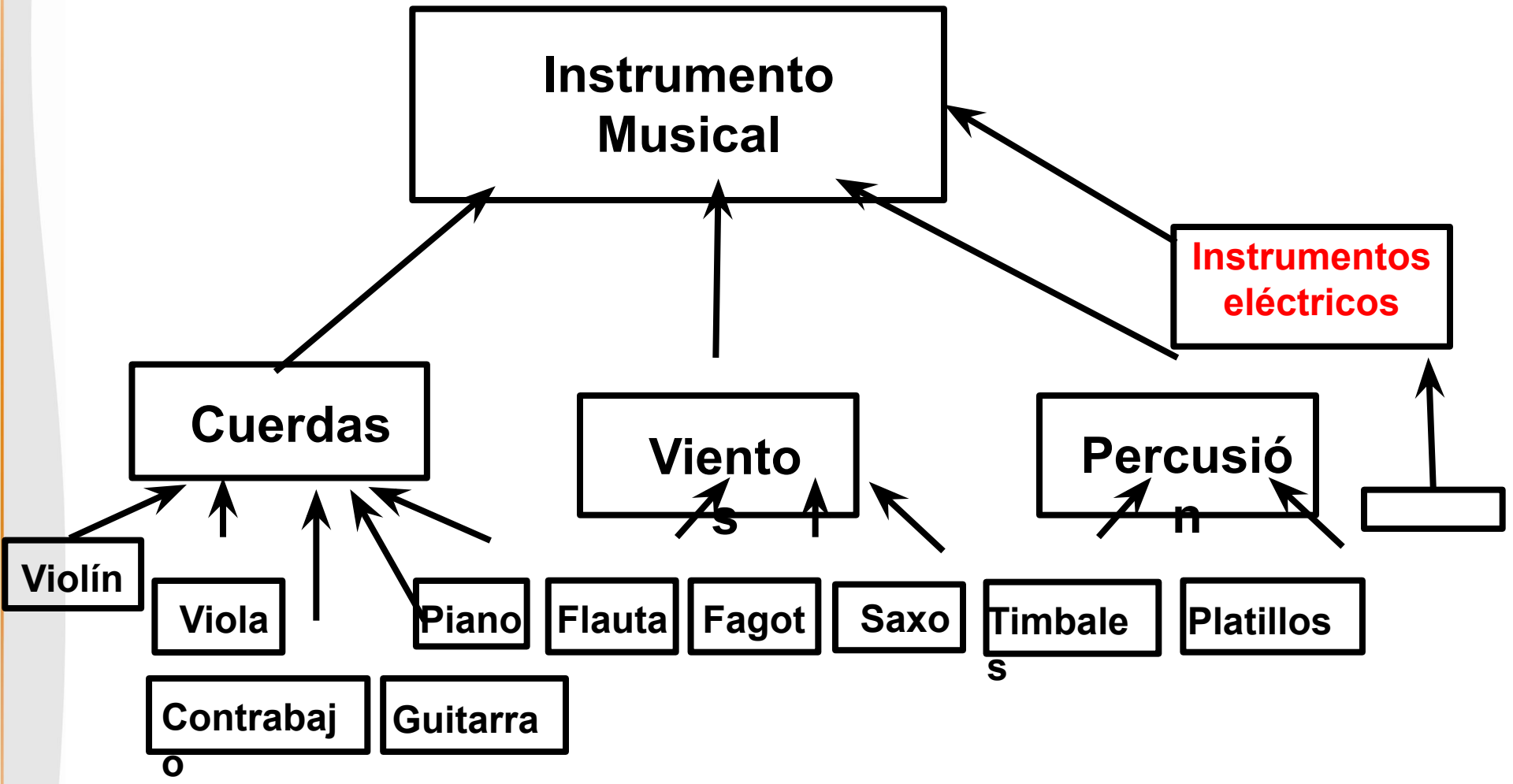
La **herencia** es un mecanismo propio de los lenguajes orientados a objetos, que:

Permite recoger los aspectos comunes de dos o más clases de objetos con el máximo nivel de detalle (a nivel de atributo y de método).

Promueve la reutilización de código, **herencia de implementación**, una subclase puede heredar comportamiento de una clase base, por tanto el código no necesita volver a ser escrito en la clase derivada.

Ofrece la posibilidad de establecer tantos niveles de **generalización** o de **especialización** como sean necesarios para reflejar nuestro modelo de la realidad. Surge el concepto de **jerarquía de clases**.

Resumiendo....



Implementación de herencia en JAVA
Cadena de Invocación de Constructores

¿Cómo se crea el objeto de la clase derivada?

La **herencia** involucra una **clase base** y **clase derivada**.

La herencia determina que la **clase derivada** tiene la **misma interface** que la **clase base** y quizás algunos métodos y atributos adicionales.

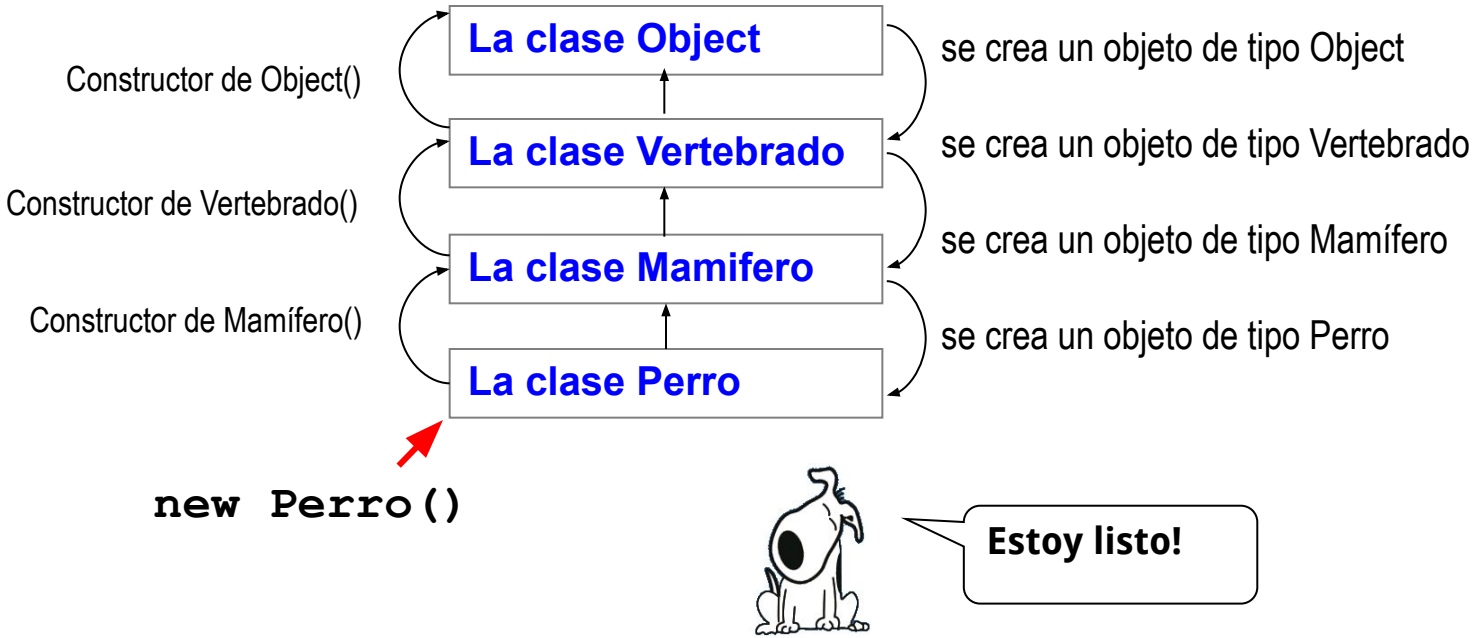
La **interface de la clase base no se copia en la clase derivada**: el **objeto de la clase derivada** contiene un **sub-objeto de la clase base**.

La **inicialización** del sub-objeto de la clase base **comienza en el constructor de la clase derivada** invocando al constructor de la clase base.

Cadena de invocación a constructores

¿Cómo se construye un objeto?

Recorriendo la **jerarquía de herencia en forma ascendente** e invocando al **constructor de la superclase** desde cada constructor, en cada nivel de la jerarquía de clases:



En cada constructor de una clase derivada se invoca al constructor de la superclase.

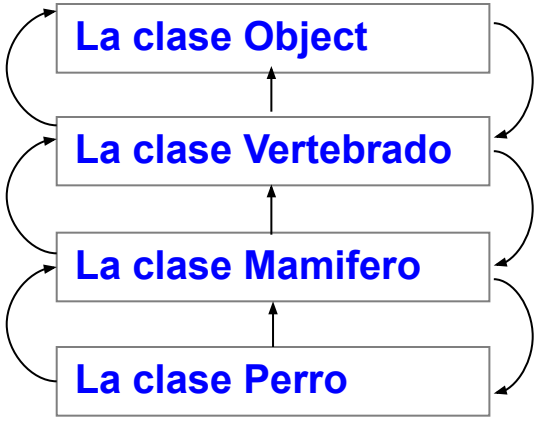
Cadena de invocación a constructores

El compilador Java automáticamente invoca al constructor por defecto (sin argumentos o constructor nulo) de la superclase, si no se lo invocó explícitamente.

```
public class Perro extends Mamifero{

 public Perro(){
   super(); //el compilador lo agrega,si no se escribe
   System.out.println("Constructor de Perro");
 }
 public void comer(){}
}
```

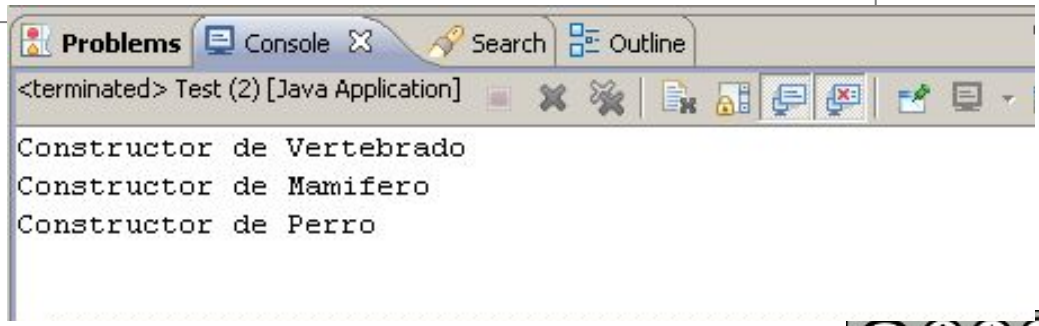
```
public class Mamifero extends Vertebrado {
 public Mamifero(){
     super();//el compilador lo agrega,si no se escribe
     System.out.println("Constructor de Mamifero");
 }
 public void comer(){}
}
```

```
public class Vertebrado {
 public Vertebrado (){
   super(); //el compilador lo agrega,si no se escribe
   System.out.println("Constructor de Vertebrado");
 }
public void comer(){}
}
```



`new Perro()`

¿Cuál sería la salida de la ejecución de `new Perro()`?

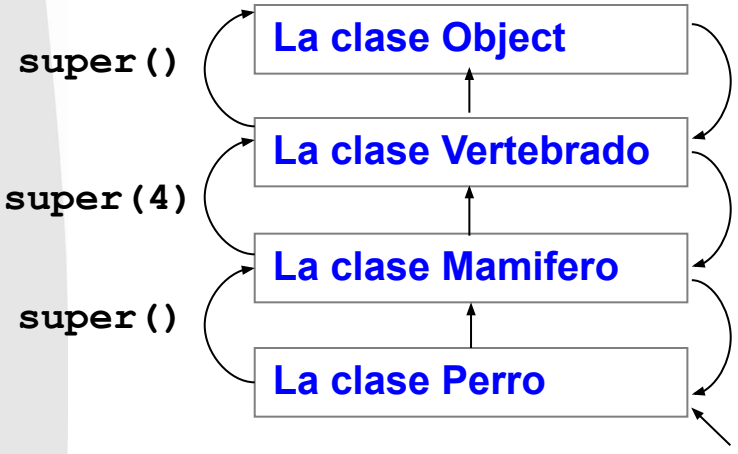


Cadena de invocación a constructores

¿Qué pasaría si Vertebrado tiene un constructor con argumentos y no tiene el constructor sin argumentos (default)? ¿Cómo se construye un objeto Perro?

```
public class Perro extends Mamifero{
 public Perro(){
  super();
  System.out.println("Constructor de Perro");
 }
 public void comer(){ }
}
```

```
public class Mamifero extends Vertebrado {
      public Mamifero(){
      super(4);
      System.out.println("Constructor de Mamifero");
      }
      public void comer(){ }
}
```




```
public class Vertebrado {
 private int cantpatas;
 public Vertebrado (int i){
   super();
   cantpatas = c;
      System.out.println("Constructor de Vertebrado");
 }
}
```

El constructor de **Mamífero** debe invocar explícitamente al constructor de la superclase **Vertebrado** usando la palabra clave **super(…)** y la lista de argumentos apropiada.

El compilador inserta "silenciosamente" la invocación al **constructor nulo (super())**, si no se escribe.

La invocación al constructor de la superclase debe hacerse en la primera línea del constructor de la clase derivada.

Las palabras claves super() y super

super()

super() invoca a un **constructor de la superclase** y debe escribirse en la primer línea de código del constructor.
JAVA garantiza la correcta creación de los objetos dado que los constructores siempre invocan a los constructores de la superclase. De esta manera todo objeto contiene una referencia al objeto de la superclase habilitando la herencia de estado y comportamiento.

```
public class Perro extends Mamifero {
private String sonido;

 public Perro(){
   super(4);
   sonido=new String("guau");
 }

}
```

se invoca al constructor de **Mamifero** con argumento de tipo entero.

En este ejemplo, el constructor de Perro() finaliza una vez que finalizó el constructor de Mamífero, y así sucesivamente, es decir un objeto Perro estará construído una vez que los objetos de las superclases se hayan construído.

super

Todos los métodos de instancia disponen de la variable **super** (además de **this**), la cual contiene una referencia al objeto de la superclase. La palabra clave **super**, permite invocar desde la subclase a métodos de la superclase.