

# Taller de Lenguajes II

Temas de hoy:

- Cadena de invocación de constructores
- Clases abstractas y métodos abstractos
  - Definición
  - La palabra clave abstract
- Polimorfismo
  - Definición
  - *Binding Dinámico*

# Cadena de Invocación de Constructores

## ¿Cómo es creado el objeto por la clase derivada?

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

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

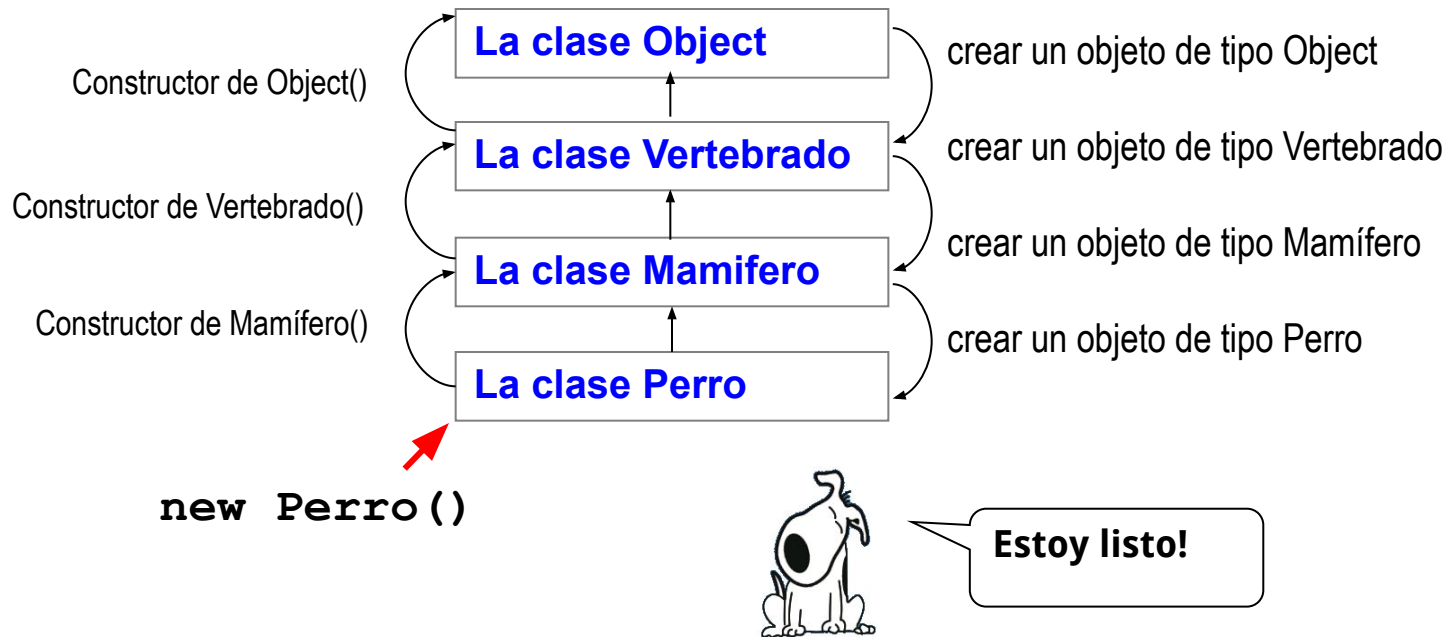
La interface de la clase base no se copia en la clase derivada; el objeto de la clase derivada contiene un sub-objeto de la clase base.

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, debe existir una llamada a un constructor de la superclase.

# Cadena de invocación a constructores

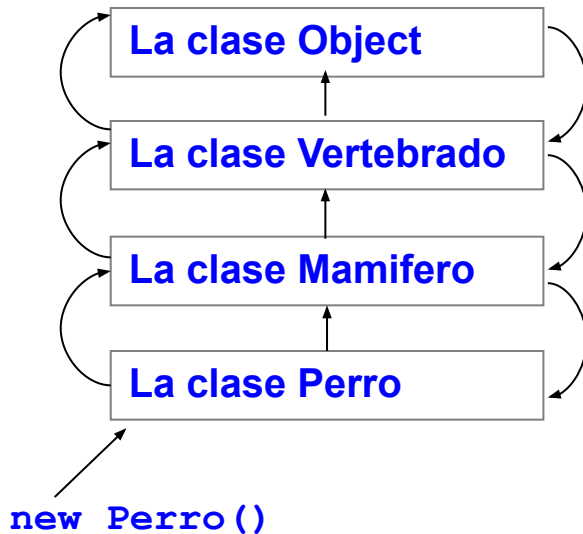
El compilador Java, automáticamente invoca en cada constructor de una clase derivada, al constructor nulo de su clase base, si no se invocó ninguno explícitamente.

```
public class Perro extends Mamifero{

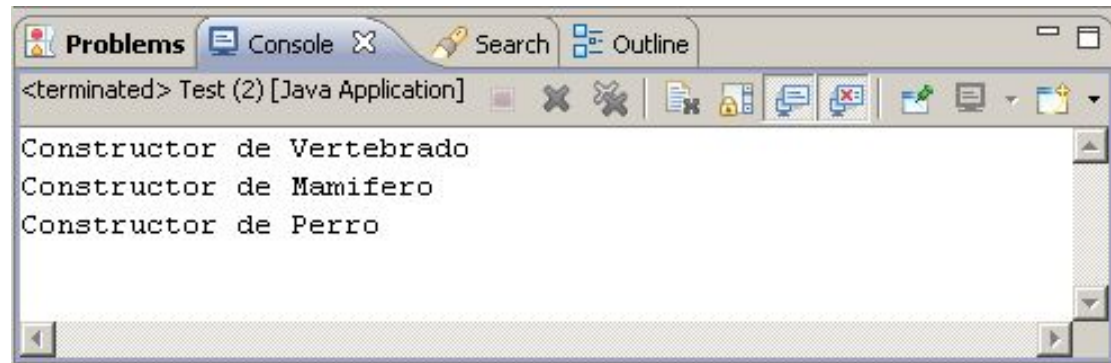
    public Perro(){
        super(); // si no se pone, igual se invoca
        System.out.println("Constructor
    }
    public void comer(){
    }
```

```
public class Mamifero extends Vertebrado {
    public Mamifero() {
        super(); // si no se pone, igual se invoca
        System.out.println("Constructor de Mamifero");
    }
    public void comer(){
    }
}
```

```
public class Vertebrado {
    public Vertebrado () {
        super(); // si no se pone, igual se invoca
        System.out.println("Constructor de Vertebrado");
    }
}
```



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

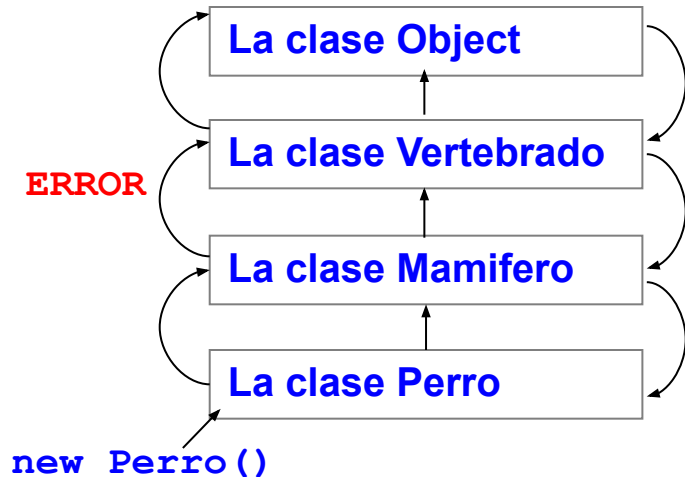


# Cadena de invocación a constructores

¿Qué pasa si Vertebrado tiene un constructor con argumentos y no tiene el constructor sin argumentos (default)?

```
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()  
        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");  
    }  
}
```

## ¿Cómo hacemos?

Desde el constructor de **Mamífero** se debe invocar a alguno de los constructores existentes en la superclase **Vertebrado** usando la palabra clave **super(...)** y la lista de argumentos apropiada.

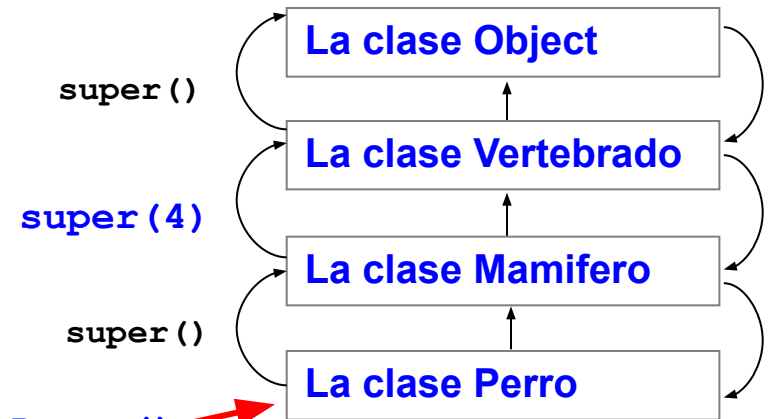
# Cadena de invocación a constructores

¿Qué pasa si Vertebrado tiene un constructor con argumentos y no tiene el constructor sin argumentos (*default*)? continuación

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



**Perro p = new Perro()** →

Si un constructor no invoca a ningún constructor de la clase base, el compilador inserta la invocación al **constructor nulo (super())**.

Si un constructor invoca explícitamente a un constructor de la superclase, debe hacerlo en la primera línea de dicho constructor.

# super() y super

## super()

**super()** invoca a un constructor de la superclase y debe estar en la primer línea de código del constructor.

JAVA garantiza la correcta creación de los objetos ya 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 código del constructor **Perro()** espera hasta que el padre se inicialice para continuar con su código.

## super

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

# Clases Concretas & Abstractas

- En JAVA podemos definir **clases concretas** y **clases abstractas**.
- A partir de una **clase concreta** se crean instancias. Todos los métodos de una clase concreta tienen implementación (cuerpo).
- Una **clase abstracta** NO puede ser instanciada, sólo puede ser extendida. Sus métodos pueden ser abstractos y concretos.
- Una clase abstracta es ideal para representar un concepto, no es apropiada para crear instancias.



# Clases Abstractas

Una clase abstracta representa un **concepto abstracto** que no es instanciable, expresa la **interfaz de comportamiento de un objeto** y **NO una implementación** particular.

Las clases abstractas permiten manipular un conjunto de clases a través de una **interfaz de comportamiento común**.

Se espera que las **clases abstractas** sean extendidas por **clases concretas**.

Una clase abstracta puede contener **métodos abstractos** y **métodos concretos**.

**¿Qué es un método abstracto?** Es un método que NO tiene implementación. Se define el nombre, el tipo de retorno, la lista de argumentos, termina con “;”. Se antepone la palabra clave **abstract** al tipo de datos de retorno/void.

**abstract void dibujar();**

**¿Para qué sirve un método sin código?**



Para definir un comportamiento común para todos los objetos de las subclases concretas

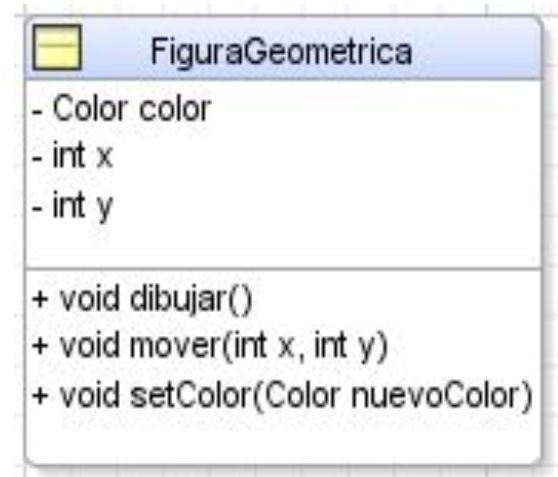
# Clases Abstractas

En JAVA podemos modelar un **CONCEPTO** mediante una **clase abstracta**

Pensemos en una aplicación que manipula figuras geométricas en la pantalla. Podríamos dibujar **círculos**, **rectángulos**, **triángulos**, **líneas rectas**, etc.

Todas las figuras geométricas pueden **cambiar de color**, **dibujarse** en la pantalla, **moverse**, calcular la **superficie**, el **perímetro**, etc., pero cada una lo hace de una manera particular.

Una figura geométrica es un concepto abstracto: representa el comportamiento común de las todas las figuras geométricas, sin embargo no es posible dibujar o redimensionar una figura geométrica genérica, sólo sabemos que todas las figuras geométricas concretas, como los círculos, rectángulos, triángulos tienen esas capacidades.



# Clases Abstractas

```
public class FiguraGeometrica {  
    private Color color;  
    private int x;  
    private int y;  
  
    public void mover(int x, int y){  
        this.x = x;  
        this.y = y;  
    }  
    public void setColor(Color nuevoColor) {  
        color = nuevoColor;  
        dibujar();  
    }  
    public void dibujar() {  
        ??  
    }  
  
    public int area(){  
        ??  
    }  
}
```

La clase **FiguraGeometrica** NO representa una figura real y por lo tanto NO puede definir implementaciones para todos sus métodos.  
**¿Qué hacemos?** La declaramos abstracta

# Clases Abstractas

Para declarar una **clase abstracta** se antepone la palabra clave **abstract** a la palabra clave **class**. Una clase abstracta es una clase que solamente puede ser EXTENDIDA, no puede ser INSTANCIADA. El compilador garantiza esta característica.

```
public abstract class FiguraGeometrica {
```

```
    private Color color;  
    private int x;  
    private int y;
```

```
    public void mover(int x, int y) { }
```

```
        this.x=x;
```

```
        this.y=y;
```

```
    }
```

```
    public void setColor(Color nuevoColor) {
```

```
        color = nuevoColor;
```

```
        dibujar();
```

```
    }
```

```
    public abstract void dibujar();
```

```
    public abstract void area();
```

```
}
```

Los **métodos abstractos** no tiene cuerpo, su declaración termina con “;”

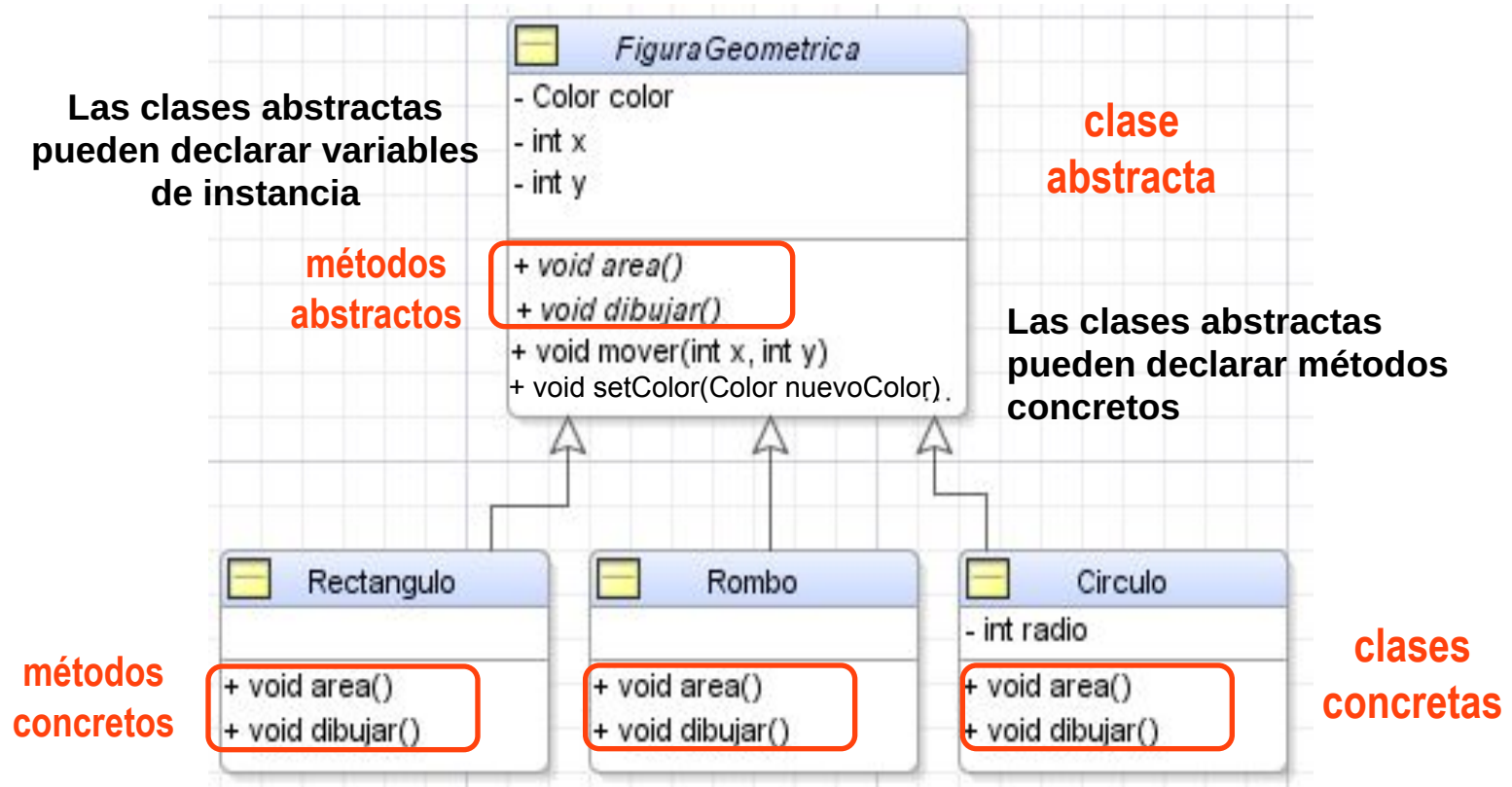
```
public class TestFiguras {  
    public static void main(String args[]) {  
        new FigurasGeometricas();  
    }
```



Si se intenta crear objetos de una **clase abstracta**, fallará la compilación. El compilador NO permite crear instancias de clases abstractas.

Una **clase abstracta** puede contener **métodos abstractos** y **métodos concretos**.

# Clases Abstractas y Herencia



**FiguraGeometrica** es una clase abstracta y los métodos **area()** y **dibujar()** son abstractos.

Para que las subclases **Rectangulo**, **Rombo** y **Circulo** sean concretas, deben proveer una implementación de cada uno de los métodos abstractos de la clase **FiguraGeometrica**.

La clase **FiguraGeometrica** está parcialmente implementada.

# Clases Abstractas y Herencia

```
public abstract class FiguraGeometrica {  
    private Color color;  
    private int x;  
    private int y;  
  
    public void mover(int x, int y){  
        this.x=x;  
        this.y=y;  
    }  
    public void setColor(Color nuevoColor){  
        color = nuevoColor;  
        dibujar();  
    }  
    public abstract void dibujar();  
    public abstract void area();  
}
```

```
public class Rombo extends FiguraGeometrica{  
    public void dibujar() {  
        System.out.println("dibujar() de Rombo");  
    }  
    public void area() {  
        System.out.println("area() de Rombo");  
    }  
}
```

```
public class Circulo extends FiguraGeometrica{  
    public void dibujar() {  
        System.out.println("dibujar() de Circulo");  
    }  
    public void area() {  
        System.out.println("area() de Circulo");  
    }  
}
```

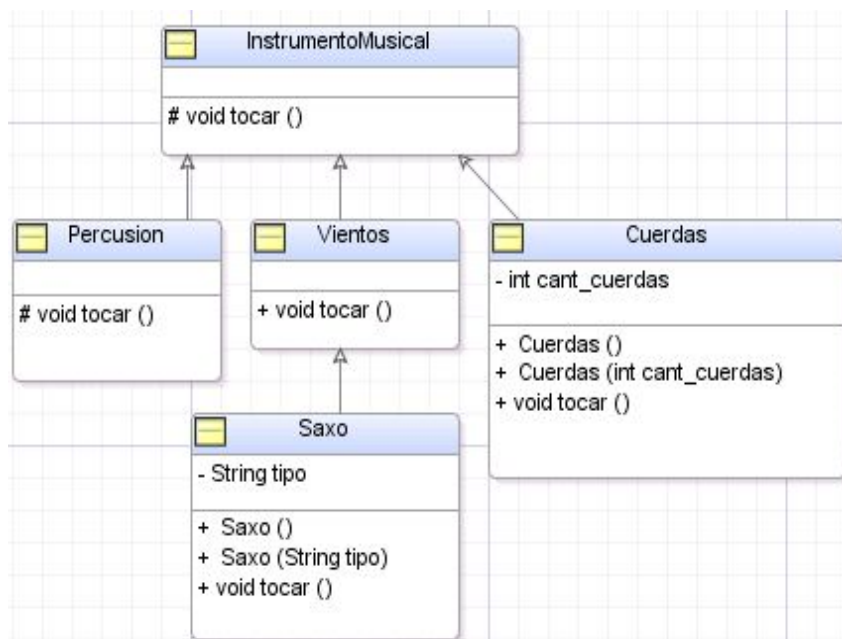
```
public class Rectangulo extends FiguraGeometrica{  
    public void dibujar() {  
        System.out.println("dibujar() de Rectangulo");  
    }  
    public void area() {  
        System.out.println("area() de Rectangulo");  
    }  
}
```

# Polimorfismo

## Herencia y Upcasting

Se tiene esta jerarquía de instrumentos musicales y una clase CancionSimple, con un método **sonar()** sobrecargado -uno con argumento de tipo Vientos y otro de tipo Cuerdas-. ¿Es una buena solución?

```
public abstract class InstrumentoMusical {  
    public abstract void tocar(Nota n);  
}
```



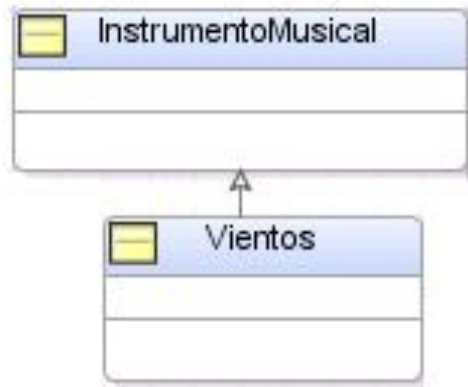
```
public class CancionSimple {  
  
    Nota[] pentagrama = new Nota[100];  
  
    public void sonar(Vientos i){  
        for (Nota n:pentagrama)  
            i.tocar(n);  
    }  
    public void sonar(Cuerdas i){  
        for (Nota n:pentagrama)  
            i.tocar(n);  
    }  
  
    public static void main(String[] args){  
        CancionSimple cs = new CancionSimple();  
        Vientos flauta = new Vientos();  
        cs.sonar(flauta);  
  
        Cuerdas piano = new Cuerdas();  
        cs.sonar(piano);  
    }  
}
```

# Polimorfismo

## Herencia y Upcasting

Lo más interesante de la herencia es la relación entre la clase derivada y la clase base: “la clase derivada **es un tipo** de la clase base” (es-un o es-como-un).

**Upcasting** es la conversión de una referencia de un objeto de la clase derivada a una referencia de la clase base.



```
public abstract class InstrumentoMusical {
    public abstract void tocar(Nota n);
}
```

```
public class Vientos extends InstrumentoMusical {
    public void tocar(Nota n) {
        System.out.print("Vientos.tocar(): "+n);
    }
}
```

```
public class CancionSimple {
    Nota[] pentagrama = new Nota[100];
    public void sonar(InstrumentoMusical i) {
        for (Nota n:pentagrama)
            i.tocar(n);
    }
    public static void main(String[] args) {
        CancionSimple cs = new CancionSimple();
        Vientos viento = new Vientos();
        cs.sonar(viento);
    }
}
```

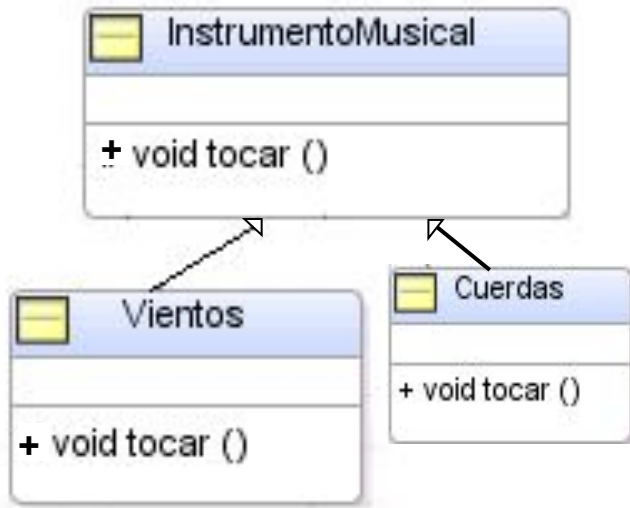
Este método puede aceptar como parámetro una referencia a un objeto InstrumentoMusical o a cualquier derivado de Instrumento

Se pasa como parámetro una referencia a un objeto Vientos sin necesidad de hacer **casting**, se hace un **upcasting** automático.



# Polimorfismo

Esta solución es extensible. Se escribió código que habla con la clase `InstrumentoMusical` y con cualquiera de sus derivadas.



```
public class CancionSimple {
    Nota[] pentagrama = new Nota[100];

    public void sonar(InstrumentoMusical i){
        for (Nota n:pentagrama)
            i.tocar(n);
    }

    public static void main(String[] args) {
        CancionSimple cs = new CancionSimple();
        Vientos flauta=new Vientos();
        cs.sonar(flauta);
        Cuerdas guitarra=new Cuerdas();
        cs.sonar(guitarra);
    }
}
```

se hace un **upcasting** automático

El **polimorfismo** permite escribir código que “hable” con la clase base y las clases derivadas.

# Polimorfismo y Binding Dinámico

¿Puede el compilador JAVA saber que el objeto `InstrumentoMusical` pasado como parámetro en `sonar()` es una referencia a un objeto `Vientos` o `Cuerdas`?

```
public class CancionSimple {
    Nota[] pentagrama = new Nota[100];
    public void sonar(InstrumentoMusical i){
        for (Nota n:pentagrama)
            i.tocar(n);
    }
    public static void main(String[] args) {
        CancionSimple cs = new CancionSimple();
        if (args[0].equals("V"))
            Vientos ins=new Vientos();
        else
            Cuerdas ins=new Cuerdas();
        cs.sonar(ins);
    }}

```

No hay manera de saber, mirando el código, que tipo de instrumento ejecutará el `tocar()`, eso depende del valor que tome la variable `i` en ejecución.

Conectar la invocación de un método con el cuerpo del método, se llama **binding**. Si el binding, se hace en compilación, se llama **binding temprano (early binding)** y si se hace en ejecución **binding tardío (early binding) o binding dinámico (dynamic binding)**.

## Binding temprano

En compilación se resuelven todas las invocaciones a métodos

### Código fuente

```
m1 () {}
m2 () {}
m3 () {}
main () {
    m1 ();
    m2 ();
    . . .
}
```

**compilador**

### Código compilado

```
. . .
. . .
llamada a m1 ()
llamada a m2 ()
. . .
. . .
```

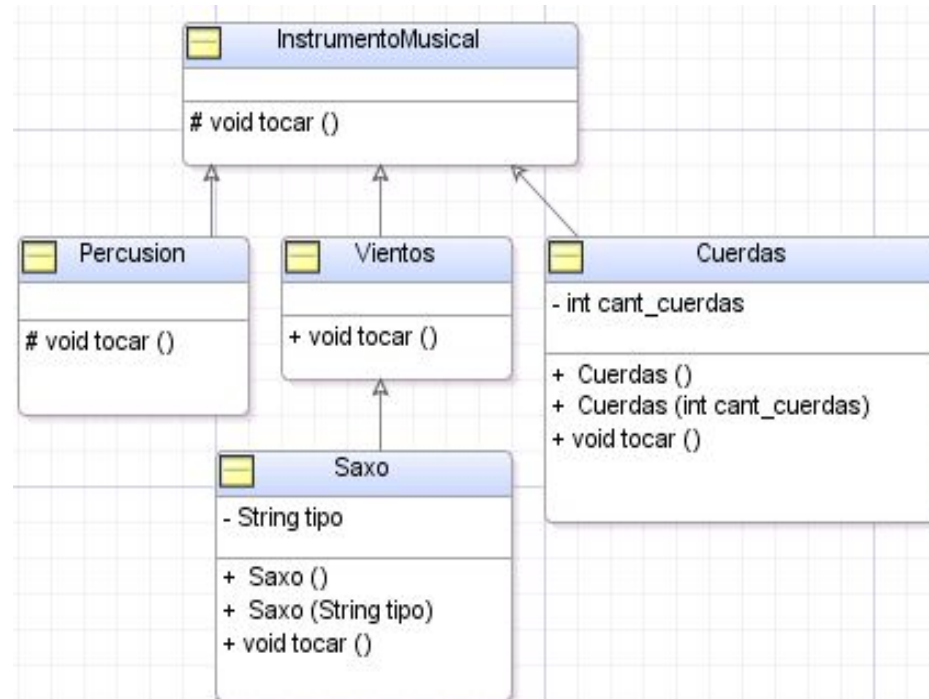
```
m1 ()
.....
.....
m2 ()
.....
.....
```

## Binding Dinámico

El compilador no conoce qué método invocará cuando recibe la referencia a un objeto `InstrumentoMusical`. El binding se resuelve en ejecución.

# Polimorfismo y Binding Dinámico


- JAVA usa **binding dinámico**: la resolución de las invocaciones a métodos, se hace en ejecución (run-time), basándose en el tipo del objeto receptor.
- JAVA provee un mecanismo para **determinar en ejecución el tipo del objeto receptor** e invocar al método apropiado.
- Aprovechando que JAVA usa binding dinámico, el programador puede escribir **código que hable con la clase base**, conociendo que las clases derivadas funcionarán correctamente usando el mismo código.
- El polimorfismo promueve la construcción de código **extensible**: un programador podría agregar subclasses a `InstrumentoMusical` sin tocar el método `sonar(InstrumentoMusical i)`.



# Polimorfismo y Binding Dinámico

```
package polimorfismo;

public class CancionSimple {
    private Nota[] pentagrama = new Nota[100];
    public void sonar(InstrumentoMusical i) {
        for (Nota n: pentagrama)
            i.tocar(n);
    }
    public static void main(String[] args) {
        CancionSimple cs = new CancionSimple();
        InstrumentoMusical[] m={new Viento(),
                                new Cuerdas(),new Percusion(),new Saxo()};
        for (int i=0;i<m.length;i++)
            cs.sonar(m[i]);
    }
}
```



**upcasting** automático a **InstrumentoMusical**.

El **binding dinámico** resuelve a qué método invocar, cuando más de una clase en una jerarquía de herencia implementó un método (en este caso **tocar()**).

La JVM busca la implementación de los métodos invocados de acuerdo al tipo del objeto receptor en tiempo de ejecución. *Binding* dinámico.

La declaración del tipo de la variable (**InstrumentoMusical**) es utilizada por el compilador de java para chequeo de errores durante la fase de compilación, pero el compilador no puede resolver qué método que se ejecutará.

Si se agregan nuevas clases a la jerarquía de herencia, el método **sonar(InstrumentoMusical i)**, no se ve afectado — esto es lo que provee el polimorfismo.

El **polimorfismo** significa “diferentes formas”. En POO se tiene una interfaz común en la clase base y diferentes versiones de métodos en las subclases que se ligan dinámicamente: **binding dinámico**.