

# Constructores

- **Inicialización de Objetos – Constructores**
  - ¿Qué son los Constructores?
  - Constructor *default*.
  - Constructores con argumentos
  - Sobrecarga de constructores
  - Cadena de invocación a constructores
- **Usos y diferencias entre:**
  - `this` y `this ()`
  - `super` y `super()`

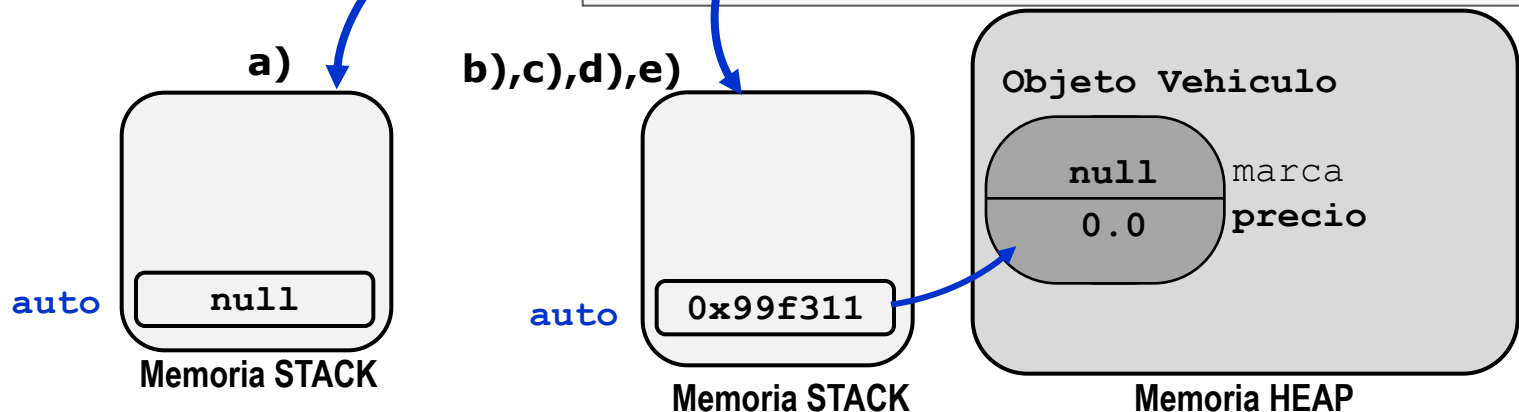
# Constructores

Para crear un objeto se utiliza el operador **new**. La creación e inicialización de un objeto involucra los siguientes pasos:

- Se aloca espacio para la variable
- Se aloca espacio para el objeto en la HEAP y se inicializan los atributos con valores por defecto.
- Se inicializan explícitamente los atributos del objeto.
- Se ejecuta el constructor (parecido a un método y tiene el mismo nombre de la clase)**
- Se asigna la referencia del nuevo objeto a la variable.

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

```
public class Test {  
    public static void main(String args[]) {  
        Vehiculo auto;  
        auto = new Vehiculo();  
    }  
}
```



# ¿Qué son los Constructores?

Los constructores son piezas de código -sintácticamente similares a los métodos- que permiten definir un estado inicial específico de un objeto en el momento de su creación.

Se diferencian de los métodos tradicionales porque:

- Deben tener el mismo nombre que la clase. La regla de que el nombre de los métodos debe comenzar con minúscula, no se aplica a los constructores.
- No retornan un valor.
- Son invocados automáticamente.

```
public class Vehiculo {  
    private String marca;  
    private double precio;  
  
    public Vehiculo() {  
    }  
}
```

NO retorna valor

La inicialización está garantizada: cuando un objeto es creado, se aloca almacenamiento en la memoria HEAP y se invoca al constructor.

```
Vehiculo v = new Vehiculo();
```

El operador **new()** se puede utilizar en cualquier lugar del código.

- La expresión **new** retorna una referencia al objeto creado recientemente, pero el constructor no retorna un valor.
- Java siempre llama automáticamente a un constructor cuando crea un objeto (antes de que el objeto sea usado). De esta forma la inicialización del objeto está garantizada.

# Constructor sin argumentos

Un constructor sin argumento o constructor *Default*, es usado para crear un objeto básico.

- Si una clase NO tiene constructores, el compilador inserta automáticamente un constructor default, con cuerpo vacío:

```
public class Vehiculo {  
    private String marca;  
    private double precio;  
  
    //métodos  
}
```

Cuando se compila

```
public Vehiculo() {  
}
```

Cuando se crea un objeto de la clase Vehiculo, con `new Vehiculo()`, se invocará el constructor por defecto, aún cuando no se encuentre explícitamente en la clase.

- Si la clase tiene al menos un constructor, con o sin argumentos, el compilador NO insertará nada.

# Constructores con argumentos

En general los constructores son usados para inicializar los valores del objeto que se está creando. *¿Cómo especificar los valores para la inicialización?* Los constructores pueden tener parámetros para la inicialización de un objeto.


```
public class Vehiculo {  
    private String marca;  
    private double precio;  
  
    public Vehiculo(String mar, double pre) {  
        marca = mar;  
        precio = pre;  
    }  
}
```

## Codificaciones equivalentes



```
public Vehiculo(String marca,  
                double precio) {  
    this.marca = marca;  
    this.precio = precio;  
}
```

Si este constructor es el único de la clase, el compilador no permitirá crear un objeto **Vehiculo** de otra manera que no sea usando este constructor.



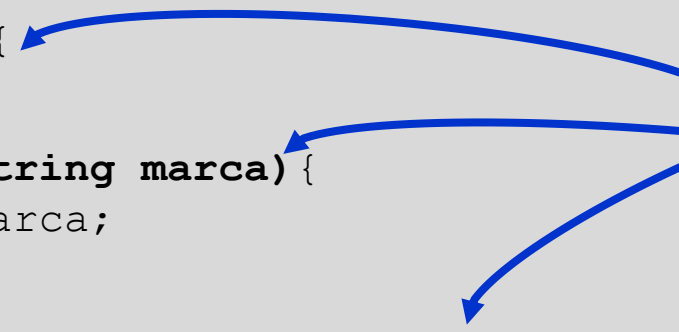
```
public class Automotores {  
    public static void main(String[] args) {  
        Vehiculo auto1 = new Vehiculo("CITROEN", 13500.00);  
        Vehiculo auto2 = new Vehiculo("HONDA", 12400.50);  
    }  
}
```

# Sobrecarga de Constructores

¿Qué pasa si se quiere construir un objeto Vehiculo de distintas maneras?

Se escriben en la clase más de un constructor ➡ sobrecarga de constructores.

```
public class Vehiculo {  
    private String nroPatente="";  
    private String propietario="SinDueño";  
  
    public Vehiculo() {  
    }  
  
    public Vehiculo(String marca) {  
        this.marca = marca;  
    }  
  
    public Vehiculo(String marca, double precio) {  
        this.marca = marca;  
        this.precio = precio;  
    }  
}
```



La **sobrecarga de métodos** permite que el mismo nombre de método sea usado con distintos tipos y cantidad de argumentos.

```
public class Botanico {  
    public static void main(String[] args) {  
        Vehiculo a1=new Vehiculo();  
        Vehiculo a2=new Vehiculo("HONDA");  
        Vehiculo a3=new Vehiculo("HONDA",12300.50);  
    }  
}
```

# this() y this

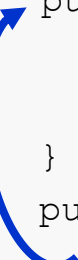
## this()

Cuando dentro de una clase, hay más de un constructor, puede surgir la necesidad de llamarse entre ellos para evitar duplicar código. Para hacer esto puede usarse `this()`, el cual hace una llamada al constructor de la misma clase que coincida con la lista de argumentos.

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

    public Vehiculo(String marca) {
        // podría tener más código
        this.marca = marca;
    }

    public Vehiculo(String marca, double precio) {
        this(marca); // debe estar en la 1ª línea
        this.precio = precio;
    }
}
```

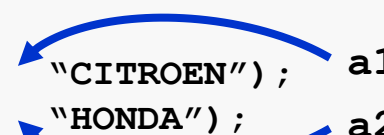


## this

Si tenemos 2 objetos de la clase **Vehiculo**, llamados a1 y a2 e invocamos al método **setNroPatente()** sobre ambos objetos, **¿cómo sabe el método para que objetos se llama?**

```
public class Vehiculo {
    private String marca;
    private String precio;
    . . .
    public setMarca(String marca) {
        this.marca = marca;
    }
}
```

```
Vehiculo a1 = new Vehiculo();
Vehiculo a2 = new Vehiculo();
. . .
a1.setMarca("CITROEN"); a1
a2.setMarca("HONDA"); a2
```

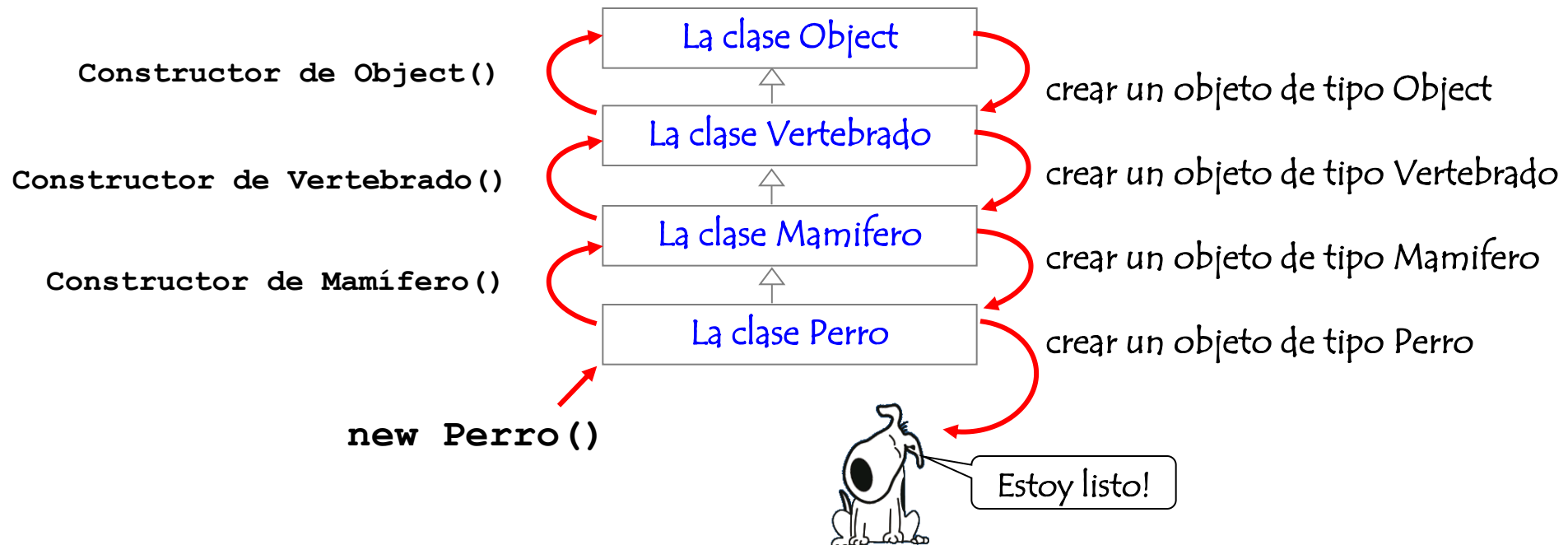


El compilador agrega como 1º argumento de cada método una referencia al objeto que está siendo manipulado.

# 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.



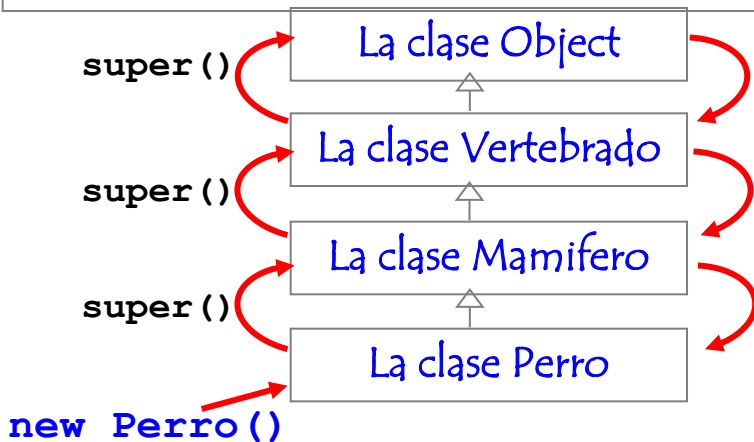
# Cadena de invocación a constructores

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.

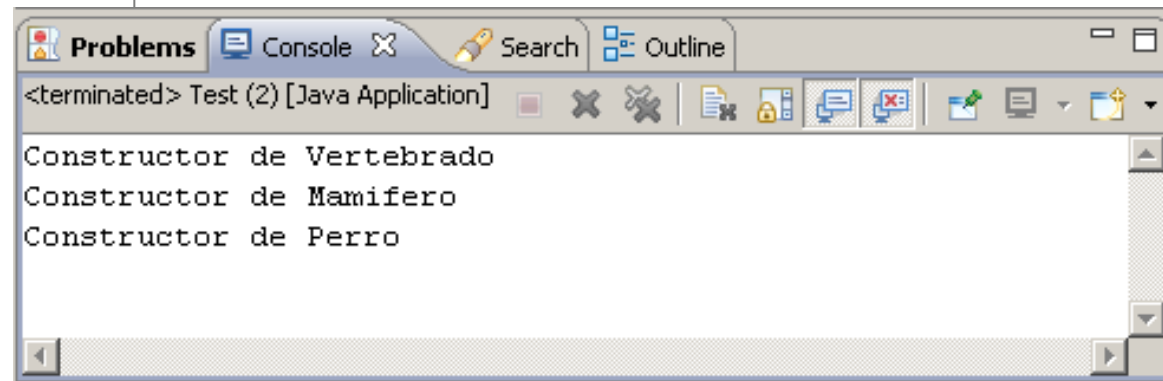
```
public class Perro extends Mamifero{
    . . .
    public Perro(){
        super(); // si no se pone, igual se invoca
        System.out.println("Constructor de Perro");
    }
    public void comer(){ }
}
```

```
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()`?

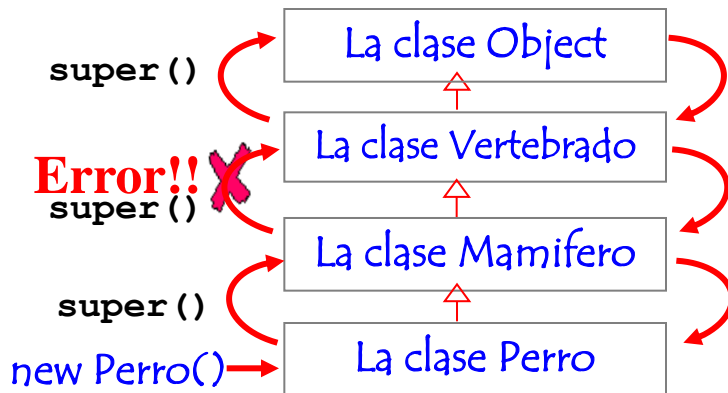


# Cadena de invocación a constructores

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

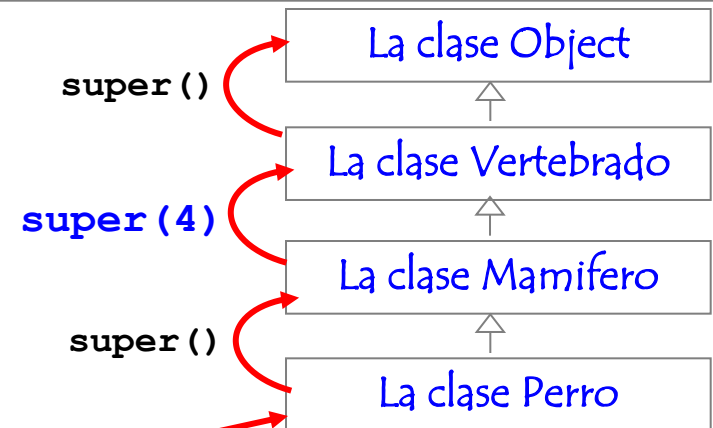
# Cadena de invocación a constructores

¿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 {
    private int cantpatas;
    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()**).

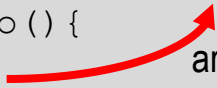
Si un constructor invoca explícitamente a un constructor de la superclase, debe hacerlo en la primera línea de dicho constructor.

# 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);  se invoca al constructor de Mamifero con  
        sonido=new String("guau");  
        . . .  
    }  
}
```

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.