

Relaciones entre Objetos...



Un conjunto de objetos aislados tiene escasa capacidad para resolver un problema. En una aplicación real los objetos colaboran e intercambian información, existiendo distintos tipos de relaciones entre ellos.

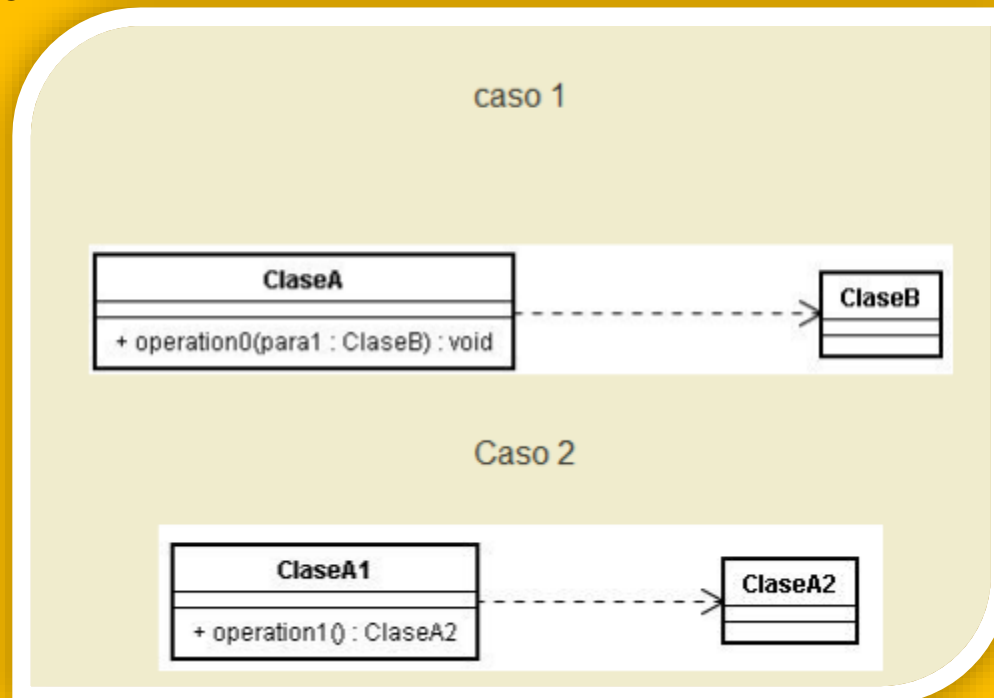
A nivel de diseño, podemos distinguir entre 5 tipos de relaciones básicas entre clases de objetos:

**dependencia, asociación, agregación,
composición y herencia**

Dependencia

En el mundo real la dependencia significa la necesidad de tener elementos acoplados en los cuales unos necesitan de otros para su funcionamiento, los sistemas deben ser diseñados con bajos niveles de acoplamiento.

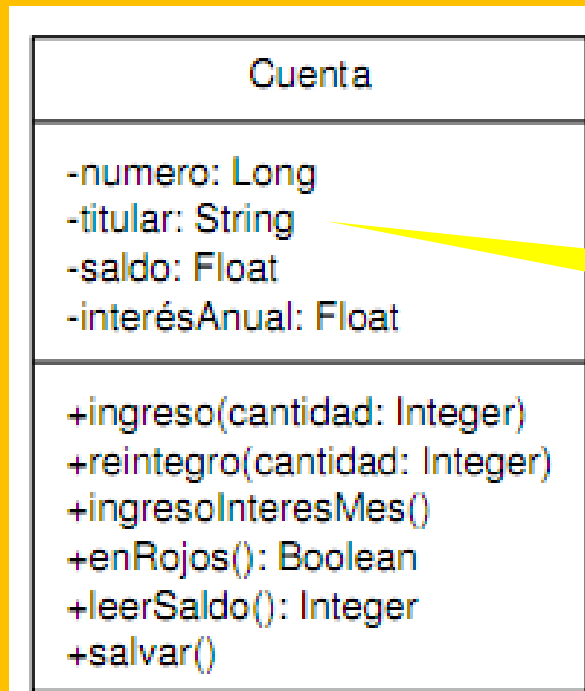
Una clase depende de otra, cuando: uno de los parámetros o el tipo de retorno de cualquiera de los métodos de la clase dependiente es del tipo de la clase independiente.



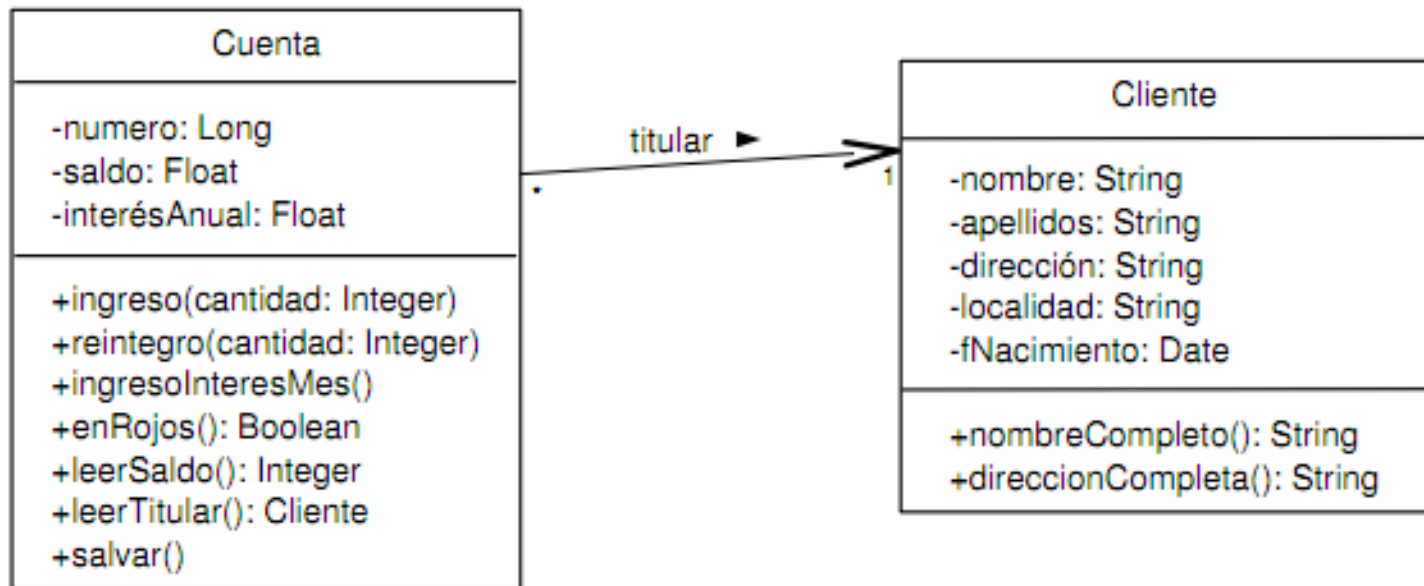
Asociación

La asociación es la relación más importante y común. Refleja una relación entre dos clases independientes que se mantiene durante la vida de los objetos de dichas clases o al menos durante un tiempo prolongado.

En UML suele indicarse el nombre de la relación, el sentido de dicha relación y las cardinalidades en los dos extremos.



Vamos a sustituir el atributo titular por una asociación con una nueva clase Cliente completa



Una asociación se implementa en Java, introduciendo referencias a objetos de una clase como atributos en la otra.

Si la relación tiene una cardinalidad superior a uno entonces será necesario utilizar un array de referencias. También es posible utilizar una estructura de datos dinámica del paquete `java.util` como `Vector` ó `LinkedList` para almacenar las referencias.

Normalmente la conexión entre los objetos se realiza recibiendo la referencia de uno de ellos en el constructor o una operación ordinaria del otro.

```

public class Cliente {
    private String nombre, apellidos;
    private String direccion, localidad;
    private Date fNacimiento;

    Cliente(String aNombre, String aApellidos, String aDireccion,
            String aLocalidad, Date aFNacimiento) {
        nombre = aNombre;
        apellidos = aApellidos;
        direccion = aDireccion;
        localidad = aLocalidad;
        fNacimiento = aFNacimiento;
    }

    String nombreCompleto() { return nombre + " " + apellidos; }
    String direccionCompleta() { return direccion + ", " + localidad; }
}

```

```

public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo;
    private float interesAnual;

    // Constructor general
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero;
        titular = aTitular;
        saldo = 0;
        interesAnual = aInteresAnual;
    }

    Cliente leerTitular() { return titular; }

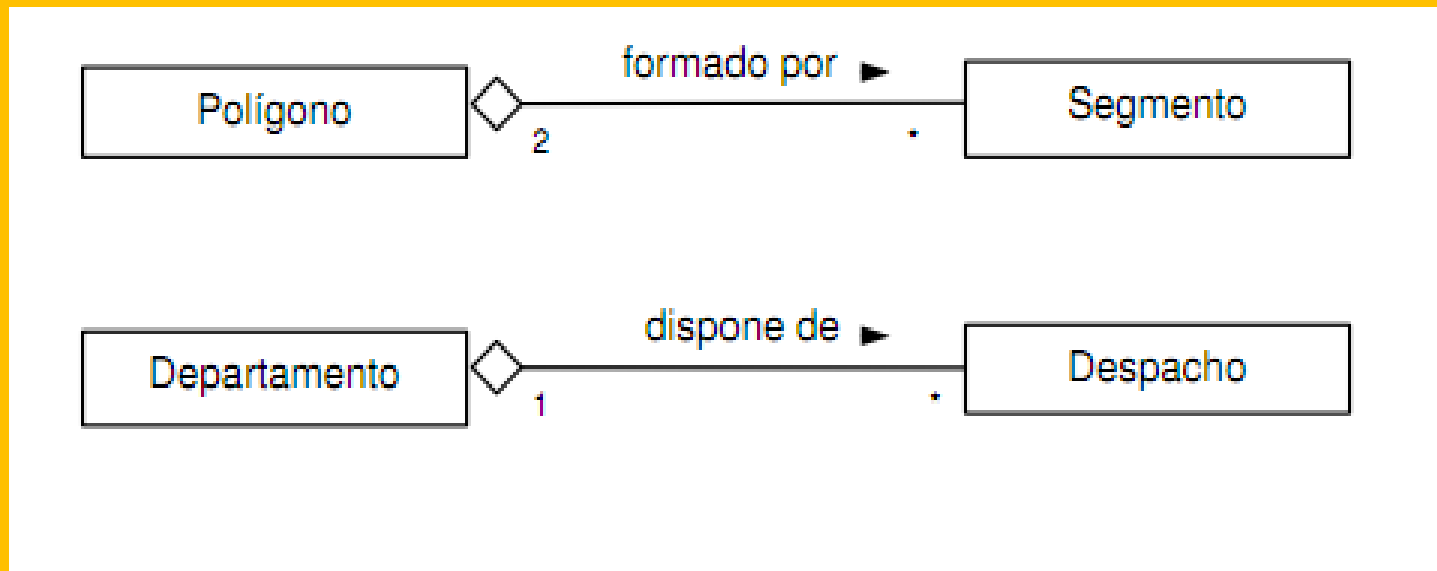
    // Resto de operaciones de la clase Cuenta a partir de aquí

```

Agregación

Es un tipo especial de asociación donde se añade el matiz semántico de que la clase de donde parte la relación representa el “todo” y las clases relacionadas “las partes”.

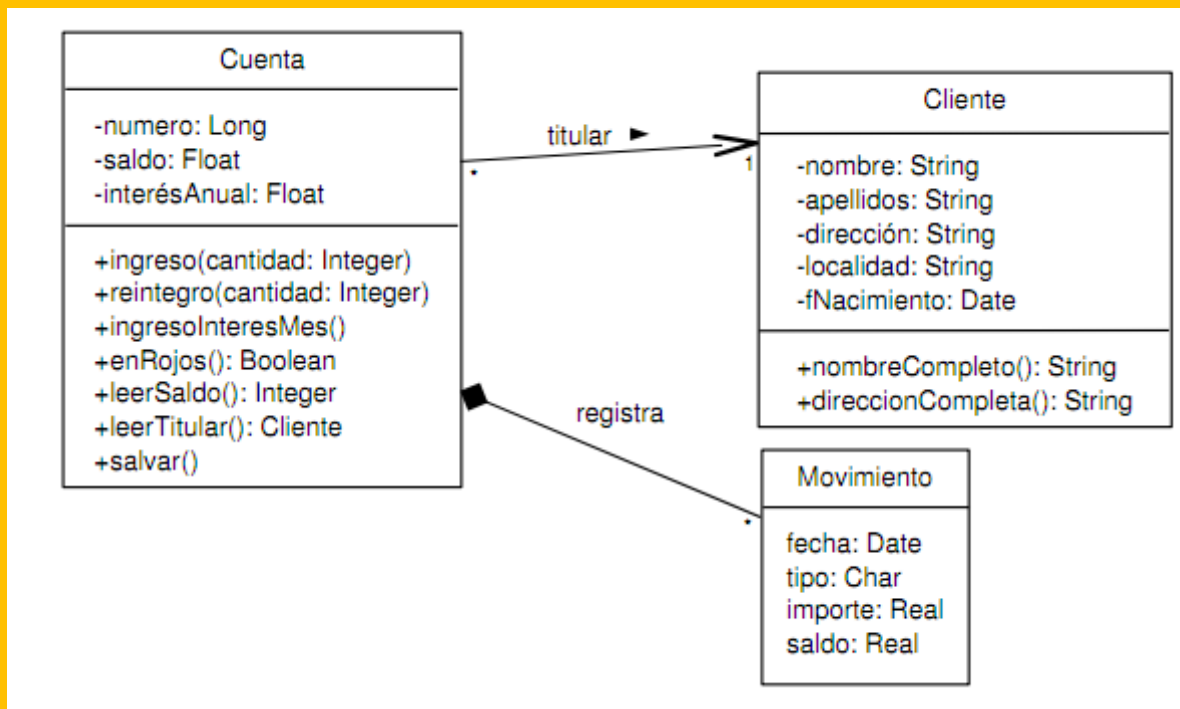
Realmente Java y la mayoría de los lenguajes orientados a objetos no disponen de una implementación especial para este tipo de relaciones. Básicamente se tratan como las asociaciones ordinarias.



Composición

Es un tipo de agregación que añade el matiz de que la clase “todo” controla la existencia de las clases “parte”. Es decir, normalmente la clase “todo” creará al principio las clases “parte” y al final se encargará de su destrucción.

Supongamos que añadimos un registro de movimientos a la clase Cuenta, de forma que quede constancia tras cada ingreso o reintegro.



Las composiciones tienen una implementación similar a las asociaciones, con la diferencia de que el objeto principal realizará en algún momento la construcción de los objetos compuestos.

```
import java.util.Date

class Movimiento {
    Date fecha;
    char tipo;
    float importe;
    float saldo;

    public Movimiento(Date aFecha, char aTipo, float aImporte, float aSaldo) {
        fecha = aFecha;
        tipo = aTipo;
        importe = aImporte;
        saldo = aSaldo;
    }
}
```

```
public class Cuenta {
    private long numero;
    private Cliente titular;
    private float saldo;
    private float interesAnual;
    private LinkedList movimientos; // Lista de movimientos

    // Constructor general
    public Cuenta(long aNumero, Cliente aTitular, float aInteresAnual) {
        numero = aNumero; titular = aTitular; saldo = 0; interesAnual = aInteresAnual;
        movimientos = new LinkedList();
    }

    // Nueva implementación de ingreso y reintegro
    public void ingreso(float cantidad) {
        movimientos.add(new Movimiento(new Date(), 'I', cantidad, saldo += cantidad));
    }

    public void reintegro(float cantidad) {
        movimientos.add(new Movimiento(new Date(), 'R', cantidad, saldo -= cantidad));
    }

    public void ingresoInteresMes() { ingreso(interresAnual * saldo / 1200); }

    // Resto de operaciones de la clase Cuenta a partir de aquí
}
```

Nota: también sería necesario modificar el otro constructor y la operación salvar para tener en cuenta la lista de movimientos a la hora de leer/escribir la información de la Cuenta en disco

Herencia

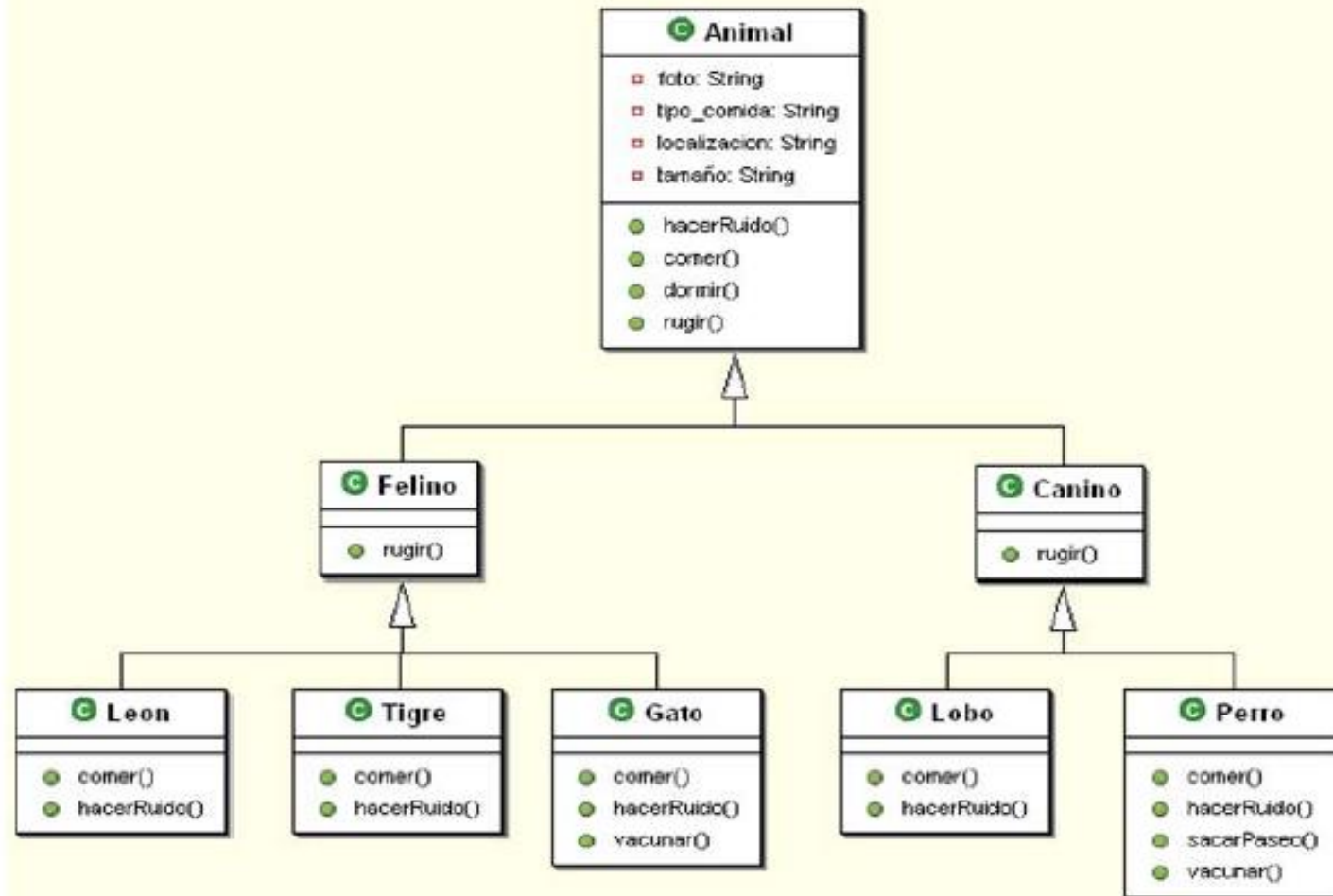
Relación de herencia

- Se basa en la existencia de relaciones de generalización/especialización entre clases.
- Las clases se disponen en una jerarquía, donde una clase hereda los atributos y métodos de las clases superiores en la jerarquía.
- Una clase puede tener sus propios atributos y métodos adicionales a lo heredado.
- Una clase puede modificar los atributos y métodos heredados.

Relación de herencia

- Las clases por encima en la jerarquía a una clase dada, se denominan superclases.
- Las clases por debajo en la jerarquía a una clase dada, se denominan subclases.
- Una clase puede ser superclase y subclase al mismo tiempo.
- Tipos de herencia:
 - Simple.
 - Múltiple (no soportada en Java)

Ejemplo



Herencia

- La implementación de la herencia se realiza mediante la *keyword*: `extends`.

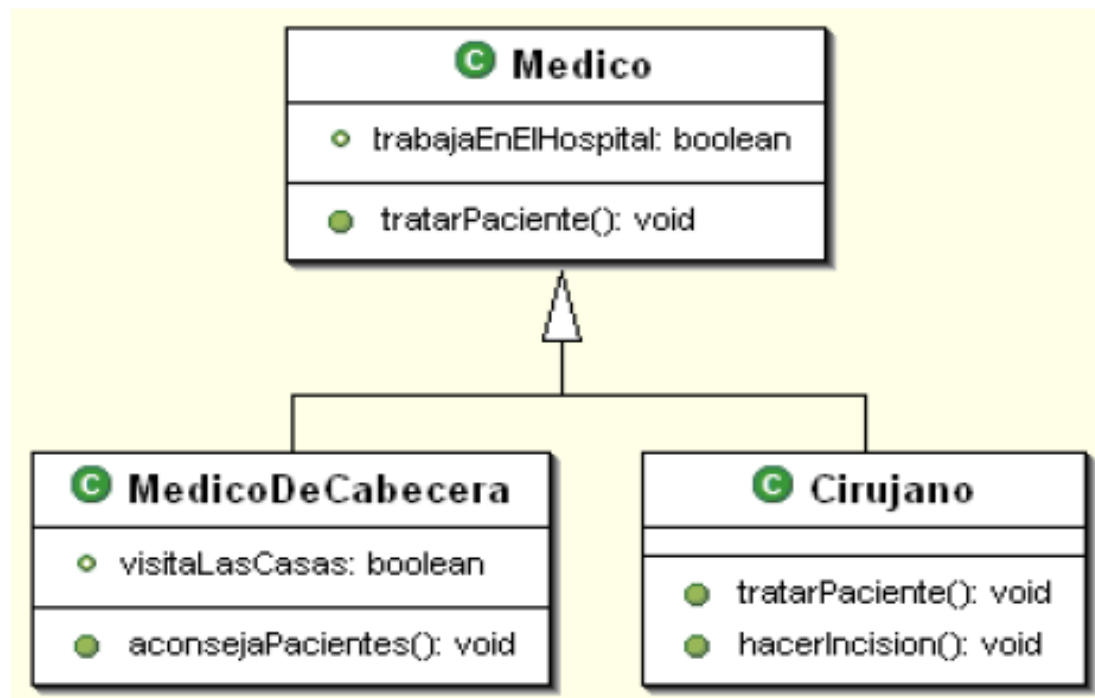
- Declaración de la herencia:

- ```
modificador_acceso class nom_clase extends nom_clase
{
}
```

- Ejemplo:

- ```
public class MiClase extends OtraClase  
{  
}
```

Ejemplo



Ejemplo

```
public class Medico
{
    public boolean trabajaEnHospital;

    public void tratarPaciente()
    {
        //Realizar un chequeo.
    }
}
```

```
public class MedicoDeCabecera extends Medico
{
    public boolean visitaLasCasas;

    public void aconsejaPacientes()
    {
        //Ofrecer remedios caseros.
    }
}
```

```
public class Cirujano extends Medico
{
    public void tratarPaciente()
    {
        //Realizar una operación.
    }

    public void hacerIncision()
    {
        //Realizar la incisión (jouch!).
    }
}
```


Ejercicio



Contesta a las siguientes preguntas basándote en el ejemplo anterior:



¿Cuántos atributos tiene la clase Cirujano?:__.



¿Cuántos atributos tiene la clase MedicoDeCabecera?:__.



¿Cuántos métodos tiene la clase Medico?:__.



¿Cuántos métodos tiene la clase Cirujano?:__.



¿Cuántos métodos tiene la clase MedicoDeCabecera?:__.



¿Puede un MedicoDeCabecera tratar pacientes?:__.



¿Puede un MedicoDeCabecera hacer incisiones?:__.

Ejercicio (solución)



Contesta a las siguientes preguntas basándote en el ejemplo anterior:



¿Cuántos atributos tiene la clase Cirujano?: 1.



¿Cuántos atributos tiene la clase MedicoDeCabecera?: 2.



¿Cuántos métodos tiene la clase Medico?: 1.



¿Cuántos métodos tiene la clase Cirujano?: 2.



¿Cuántos métodos tiene la clase MedicoDeCabecera?: 2.



¿Puede un MedicoDeCabecera tratar pacientes?: Si.



¿Puede un MedicoDeCabecera hacer incisiones?: No.

La clase Object



En Java todas las clases heredan de otra clase:



Si lo especificamos en el código con la *keyword* `extends`, nuestra clase heredará de la clase especificada.



Si no lo especificamos en el código, el compilador hace que nuestra clase herede de la clase `Object` (raíz de la jerarquía de clases en Java).



Ejemplo:




```
public class MiClase extends Object
{
    // Es redundante escribirlo puesto que el
    // compilador lo hará por nosotros.
}
```

La clase Object

- Esto significa que nuestras clases siempre van a contar con los atributos y métodos de la clase Object.
- Algunos de sus métodos mas importantes son:
 - **public boolean** equals(Object o);
Compara dos objetos y dice si son iguales.
 - **public** String toString();
Devuelve la representación visual de un objeto.
 - **public** Class getClass();
Devuelve la clase de la cual es instancia el objeto.

La clase Object (cont.)

A vertical line on the left side of the slide, composed of five circles connected by a line. The third circle from the top is highlighted with a yellow and orange gradient, while the others are white with purple outlines.

public int hashCode();

Devuelve un identificador unívoco después de aplicarle un algoritmo hash.

public Object clone();

Devuelve una copia del objeto.

Otros métodos:

public void finalize();

Un método llamado por el Garbage Collector.

public void wait(); **public void** notify();

public void notifyAll();

Tienen que ver con el manejo de threads.

Ejemplo

```
public class MiClase
```

```
{  
}
```

```
public class TestMiClase
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        MiClase mc = new MiClase();
```

```
        System.out.println("mc: " + mc);
```

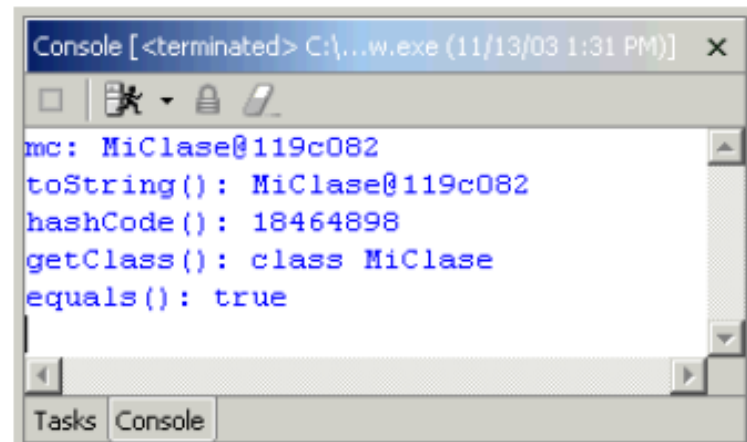
```
        System.out.println("toString(): " + mc.toString());
```

```
        System.out.println("hashCode(): " + mc.hashCode());
```

```
        System.out.println("getClass(): " + mc.getClass());
```

```
        System.out.println("equals(): " + mc.equals(mc));
```

```
    }  
}
```



The screenshot shows a console window titled "Console [<terminated> C:\...w.exe (11/13/03 1:31 PM)]". The output text is as follows:

```
mc: MiClase@119c082  
toString(): MiClase@119c082  
hashCode(): 18464898  
getClass(): class MiClase  
equals(): true
```

At the bottom of the window, there are two tabs: "Tasks" and "Console", with "Console" being the active tab.

Castings

- El casting es una forma de realizar conversiones de tipos.
- Hay dos clases de casting:
 - UpCasting: conversión de un tipo en otro superior en la jerarquía de clases. No hace falta especificarlo.
 - DownCasting: conversión de un tipo en otro inferior en la jerarquía de clases.
- Se especifica precediendo al objeto a convertir con el nuevo tipo entre paréntesis.

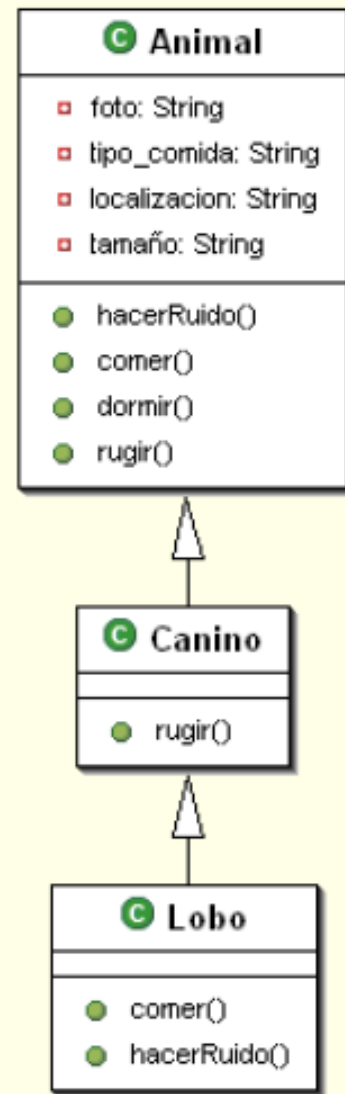
Ejemplo

```
public class Test
{
    public static void main (String[] args)
    {
        Lobo lobo = new Lobo();

        // UpCastings
        Canino canino = lobo;
        Object animal = new Lobo();
        animal.comer();

        // DownCastings
        lobo = (Lobo)animal;
        lobo.comer();
        Lobo otroLobo = (Lobo)canino;
        Lobo error = (Lobo) new Canino();
    }
}
```

No compila. No puedes llamar al método comer() sobre un Object. No puedes convertir un Canino en un Lobo.



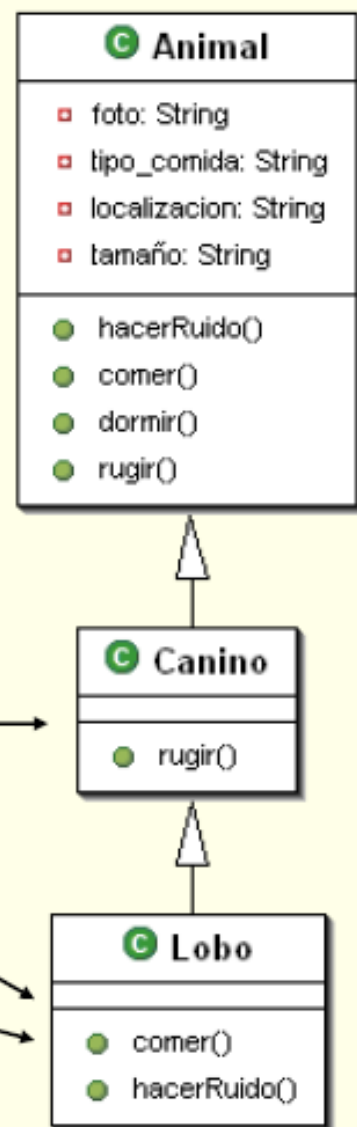
Sobrescribir un método

- Sobrescribir un método significa que una subclase reimplementa un método heredado.
- Para sobrescribir un método hay que respetar totalmente la declaración del método:
 - El nombre ha de ser el mismo.
 - Los parámetros y tipo de retorno han de ser los mismos.
 - El modificador de acceso no puede ser mas restrictivo.
- Al ejecutar un método, se busca su implementación de abajo hacia arriba en la jerarquía de clases.

Ejemplo

```
public class Test
{
    public static void main (String[] args)
    {
        Lobo lobo = new Lobo();

        lobo.hacerRuido();
        lobo.rugir();
        lobo.comer();
        lobo.dormir();
    }
}
```

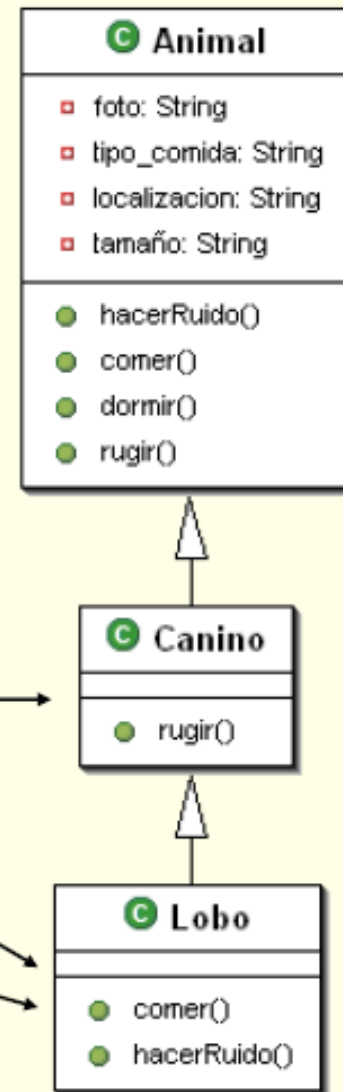


Ejemplo

```
public class Test
{
    public static void main (String[] args)
    {
        Animal animal = new Lobo();

        animal.hacerRuido();
        animal.rugir();
        animal.comer();
        animal.dormir();
    }
}
```

Los castings no modifican al objeto. Solo su tipo, por lo que se siguen ejecutando sobre el mismo objeto.



Sobrescribir vs. Sobrecargar

- Sobrecargar un método es un concepto distinto a sobrescribir un método.
- La sobrecarga de un método significa tener varias implementaciones del mismo método con parámetros distintos:
 - El nombre ha de ser el mismo.
 - El tipo de retorno puede ser distinto
 - Los parámetros tienen que ser distintos.
 - El modificador de acceso puede ser distinto.

Sobrescribir vs. Sobrecargar

- Habrá que tener muy en cuenta los parámetros que se envían y las conversiones por defecto para saber qué método se ejecuta.

- Por ejemplo:

- Tenemos un método que recibe un float.

```
public void miMetodo(float param) { }
```

- miObjeto.miMetodo(1.3); llamará sin problemas al método.

- Sobrecargamos el método para que reciba un double.

```
public void miMetodo(double param) { }
```

Sobrescribir vs. Sobrecargar



Continuación del ejemplo:



`miObjeto.miMetodo(1.3);` ya no llama al método con float.



Recordemos que un número real por defecto es double.



Para seguir llamando al método con float debemos especificarlo implícitamente:

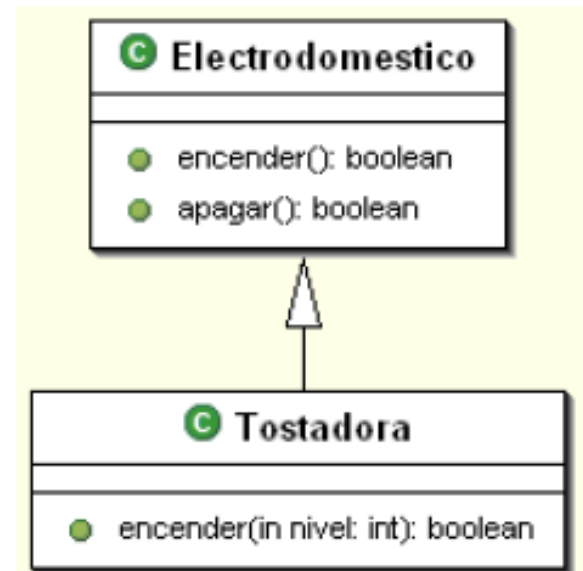


`miObjeto.miMetodo(1.3F);` o `miObjeto.miMetodo((float)1.3);`

Ejemplo

```
public class Electrodomestico
{
    public boolean encender()
    {
        //Hacer algo.
    }
    public boolean apagar()
    {
        //Hacer algo.
    }
}
```

```
public class Tostadora extends Electrodomestico
{
    public boolean encender(int nivel)
    {
        //Hacer algo.
    }
}
```

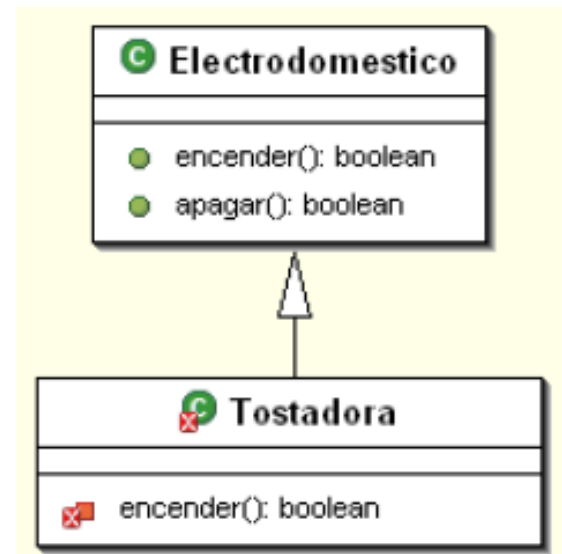


No es sobrescritura. Los parámetros son distintos. Es sobrecarga.

Ejemplo

```
public class Electrodomestico
{
    public boolean encender()
    {
        //Hacer algo.
    }
    public boolean apagar()
    {
        //Hacer algo.
    }
}
```

```
public class Tostadora extends Electrodomestico
{
    private boolean encender()
    {
        //Hacer algo.
    }
}
```



No compila. Es sobrescritura restringiendo el acceso.

SOBREESCRITURA DE MÉTODOS



- J2SE 5.0 añade una novedad al respecto.
- Se permite la **SOBREESCRITURA** de métodos cambiando también el tipo de retorno, pero siempre que:
 - El método que se está **SOBREESCRITO** sea de una clase padre (de la que heredamos directa o indirectamente).
 - El nuevo tipo de retorno sea hijo del tipo de retorno del método original (es decir, que herede de él directa o indirectamente).
- Por tanto, no es válido para tipos primitivos.

El uso de la Herencia

- Debemos usar herencia cuando hay una clase de un tipo mas específico que una superclase. Es decir, se trata de una especialización.
- Lobo es mas específico que Canino. Luego tiene sentido que Lobo herede de Canino.
- Debemos usar herencia cuando tengamos un comportamiento que se puede reutilizar entre varias otras clases del mismo tipo genérico.
- Las clases Cuadrado, Circulo y Triangulo tiene que calcular su área y perímetro luego tiene sentido poner esa funcionalidad en una clase genérica como Figura.

El uso de la Herencia



No debemos usar herencia solo por el hecho de reutilizar código. Nunca debemos romper las dos primeras reglas.



Podemos tener el comportamiento cerrar en Puerta. Pero aunque necesitemos ese mismo comportamiento en Coche no vamos a hacer que Coche herede de Puerta. En todo caso, coche tendrá un atributo del tipo Puerta.



No debemos usar herencia cuando no se cumpla la regla: Es-un (Is-a).



Refresco es una Bebida. La herencia puede tener sentido. Bebida es un Refresco. ¿? No encaja luego la herencia no tiene sentido.

Ejercicio

```
public class A
{
    int ivar = 7;
    public void m1()
    {
        System.out.println("A's m1, ");
    }
    public void m2()
    {
        System.out.println("A's m2, ");
    }
    public void m3()
    {
        System.out.println("A's m3, ");
    }
}
```

```
public class B extends A
{
    public void m1()
    {
        System.out.println("B's m1, ");
    }
}
```

```
public class C extends B
{
    public void m3()
    {
        System.out.println("C's m3, " + (ivar + 6));
    }
}
```

```
public class Mix
{
    public static void main(String[] args)
    {
        A a = new A();
        B b = new B();
        C c = new C();
        A a2 = new C();
```

¿?

```
}
```

Ejercicio

- ¿Qué salida produce la inclusión en el programa anterior de estas tres líneas de código en el recuadro vacío?

```
b.m1();  
c.m2();  
a.m3();
```

A's m1, A's m2, C's m2, 6

```
c.m1();  
c.m2();  
c.m3();
```

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

```
a.m1();  
b.m2();  
c.m3();
```

B's m1, A's m2, C's m3, 13

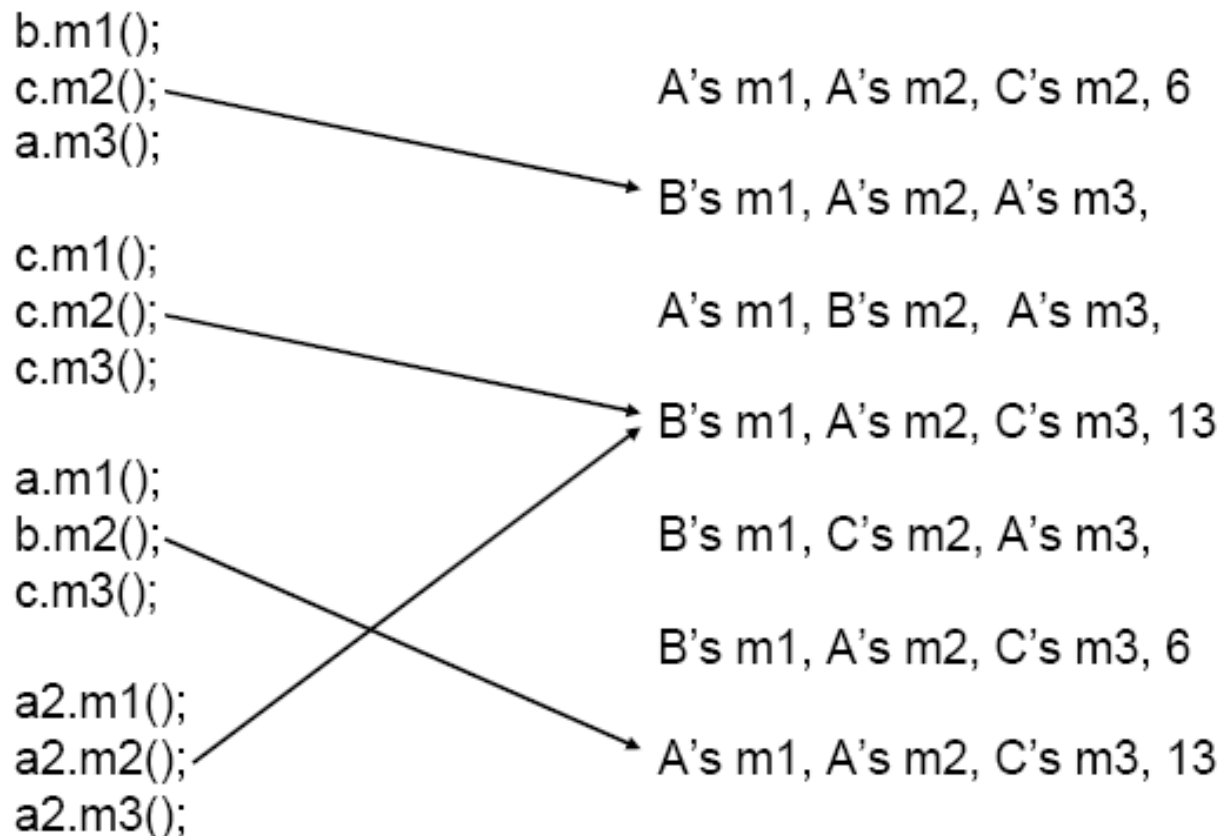
B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

```
a2.m1();  
a2.m2();  
a2.m3();
```

A's m1, A's m2, C's m3, 13

Solución



super y this

- super y this son dos *keywords* de Java.
- super es una referencia al objeto actual pero apuntando al padre.
- super se utiliza para acceder desde un objeto a atributos y métodos (incluyendo constructores) del padre.
- Cuando el atributo o método al que accedemos no ha sido sobrescrito en la subclase, el uso de super es redundante.
- Los constructores de las subclases incluyen una llamada a super() si no existe un super o un this.

super y this



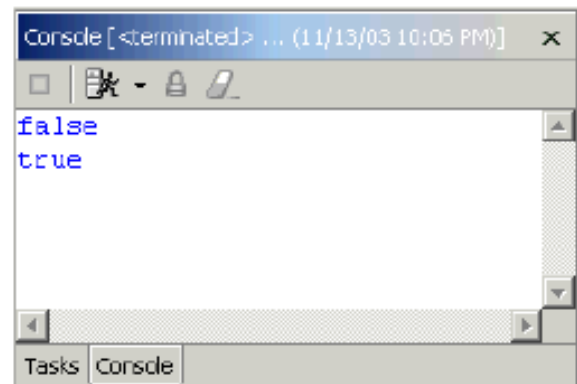
Ejemplo de acceso a un atributo:



```
public class ClasePadre  
{  
    public boolean atributo = true;  
}
```



```
public class ClaseHija extends ClasePadre  
{  
    public boolean atributo = false;  
    public void imprimir()  
    {  
        System.out.println(atributo);  
        System.out.println(super.atributo);  
    }  
}
```



super y this



Ejemplo de acceso a un constructor:

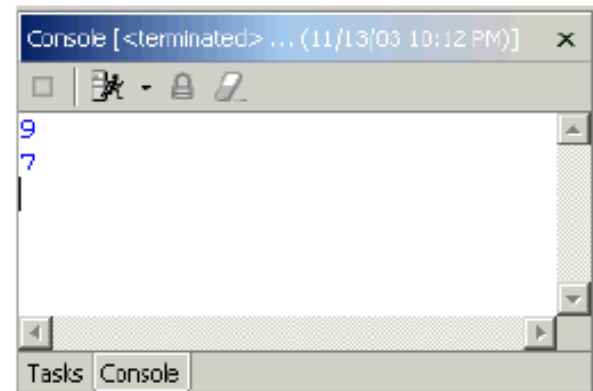


```
public class ClasePadre
{
    public ClasePadre(int param)
    {
        System.out.println(param);
    }
}
```




```
public class ClaseHija extends ClasePadre
{
    public ClaseHija(int param)
    {
        super(param + 2);
        System.out.println(param);
    }
}
```

Nota: tiene que ser la primera línea del constructor y solo puede usarse una vez por constructor.



super y this

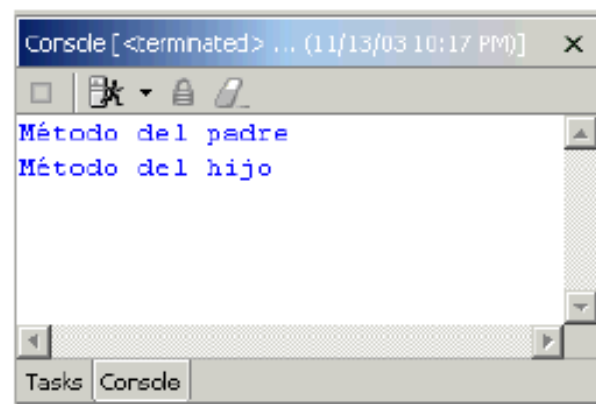
 Ejemplo de acceso a un método:

 **public class** ClasePadre

```
{  
    public void imprimir()  
    {  
        System.out.println("Método del padre");  
    }  
}
```

 **public class** ClaseHija **extends** ClasePadre

```
{  
    public void imprimir()  
    {  
        super.imprimir();  
        System.out.println("Método del hijo");  
    }  
}
```



super y this

- this es una referencia al objeto actual.
- this se utiliza para acceder desde un objeto a atributos y métodos (incluyendo constructores) del propio objeto.
- Existen dos ocasiones en las que su uso no es redundante:
 - Acceso a un constructor desde otro constructor.
 - Acceso a un atributo desde un método donde hay definida una variable local con el mismo nombre que el atributo.

super y this



Ejemplo de acceso a un atributo:



```
public class MiClase
{
    private int x = 5;
    public void setX(int x)
    {
        System.out.println("x local vale: " + x);
        System.out.println("x atributo vale: " + this.x);
        this.x = x;
        System.out.println("x atributo vale: "
                           + this.x);
    }
}
```

super y this



Ejemplo de acceso a un constructor:

```
public class MiClase
{
    public MiClase()
    {
        this(2);
        System.out.println("Constructor sin");
    }
    public MiClase(int param)
    {
        System.out.println("Constructor con");
    }
}
```

Nota: tiene que ser la primera línea del constructor y solo puede usarse una vez por constructor.

