

*PROGRAMACIÓN

SEGUNDO TRIMESTRE
2ª TUTORÍA COLECTIVA

HERENCIA

- * Hay ocasiones en las que unas clases son más generales que otras o bien, más específicas.

Por ejemplo, la clase “Cuenta Naranja” es más general que la clase Cuenta Bancaria.

- * En un caso así, las características de la clase más general, seguro que también están en la más específica.

Todas las cuentas bancarias tienen un saldo, titular y en ellas se pueden hacer ingresos o reintegros, por lo tanto las “Cuentas Naranja” también.

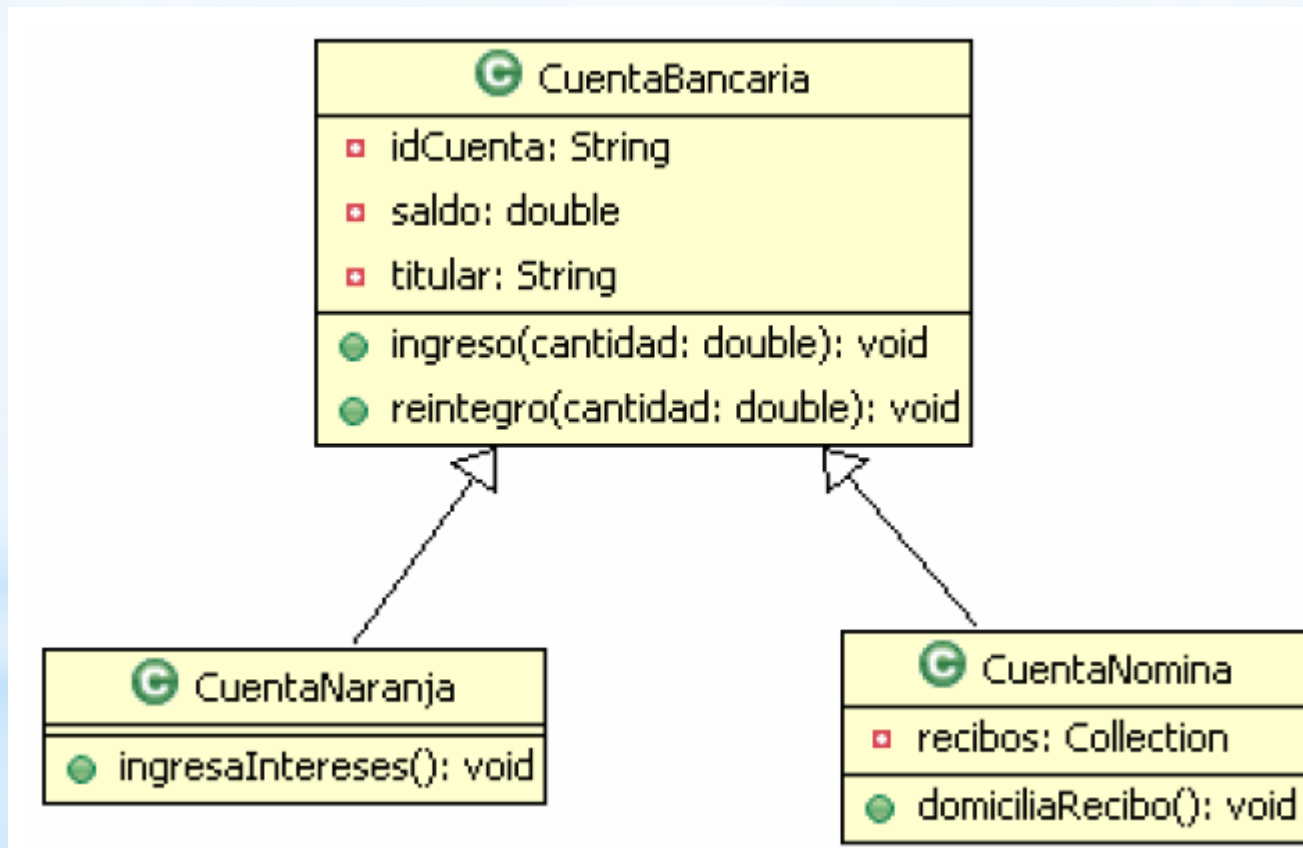
- * Si ocurre esto, podemos decir que una clase **hereda** de otra todas sus propiedades y métodos, a parte de tener los suyos propios.

En la “Cuenta Nómina” se pueden domiciliar recibos, pero en la “Cuenta Naranja” no.

- * A las clases más generales se les llama **Superclases** y a las más particulares **Subclases**.

HERENCIA en UML

- * En UML la herencia se indica por flechas como las que aparecen en el siguiente diagrama.



HERENCIA EN JAVA

- * Se utiliza la palabra reservada **extends** a continuación del nombre de la clase, para indicar de qué clase se hereda.

Superclase

```
public class CuentaBancaria {  
    private String idCuenta;  
    private double saldo;  
    private String titular;  
  
    public void ingreso(double cantidad){  
        //Hacer ingreso  
    }  
  
    public void reintegro(double cantidad){  
        //Hacer reintegro  
    }  
}
```

Subclases

```
public class CuentaNomina extends CuentaBancaria {  
  
    private Collection recibos;  
  
    public void domiciliaRecibo(){  
        //Domiciliar un recibo  
    }  
}
```

```
public class CuentaNaranja extends CuentaBancaria {  
    public void ingresaIntereses(){  
    }  
}
```

LA CLASE Object

- * En Java, todas las clases heredan de otra clase, lo indiquemos o no.
- * Si lo especificamos en el código con la palabra `extends`, heredarán de la clase indicada.
- * Si no lo especificamos heredan del padre de todas las clases que es la clase **Object**.
- * Esto significa que **todas nuestras clases heredarán las propiedades y los métodos de la clase Object**.

MÉTODOS DE LA CLASE Object

Algunos de sus métodos de esta clase son:

- * `public boolean equals (Object o)`
Comprueba si el objeto es igual a otro.
- * `public String toString()`
Devuelve la representación visual de un objeto.
- * `public Class getClass()`
Devuelve la clase a la que pertenece el objeto.
- * `public Object clone()`
Devuelve una copia del objeto.

Ejemplo clase Object

```
public class Punto
{
    public int x = 0;
    public int y = 0;

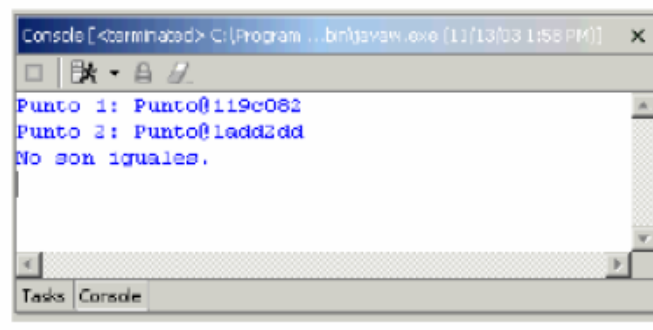
    public Punto(int param1, int param2)
    {
        x = param1;
        y = param2;
    }
}

public class Test
{
    public static void main(String[] args)
    {
        Punto pun1 = new Punto(1,2);
        Punto pun2 = new Punto(1,2);
        System.out.println("Punto 1: " + pun1);
        System.out.println("Punto 2: " + pun2);
        if(pun1.equals(pun2))
            System.out.println("Son iguales.");
    }
}
```

Como Punto hereda de la clase Object implícitamente, podremos utilizar el método ***equals***.

Sin embargo, este método falla porque en ningún sitio está indicado cuándo dos puntos son iguales.

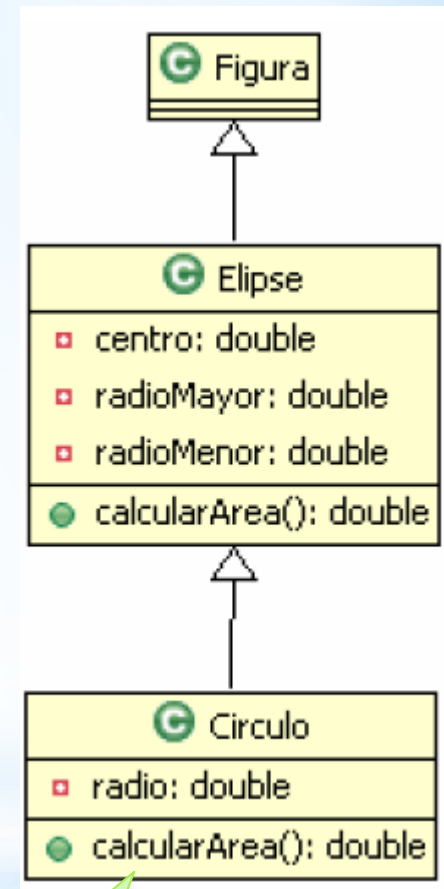
La solución es redefinirlo para esta clase indicando que dos puntos son iguales cuando sus dos coordenadas son iguales (*sobreescribir u override*)



Llamada al método equals, de la clase Object

SOBRESCRIBIR UN MÉTODO

- * Una clase **sobrescribe** un método cuando vuelve a definir las instrucciones de un método heredado.
- * Para sobrescribir un método hay que respetar totalmente la cabecera del método heredado:
 - El nombre ha de ser el mismo.
 - Los parámetros y el tipo de retorno han de ser los mismos.
 - El modificador de acceso no puede ser más restrictivo.
- * Cuando un objeto ejecuta un método, éste se busca por la jerarquía de clases de abajo a arriba.



Método sobrescrito

USO DE LA HERENCIA

- * Debemos usar la herencia cuando haya una clase que sea más específica que otra.
- * Debemos utilizar la herencia cuando tengamos un comportamiento que se puede reutilizar entre otras varias clases del mismo tipo genérico.
Por ejemplo, las clases Triángulo, Círculo y Cuadrado, tienen que calcular su área y perímetro, luego tiene sentido colocar esa funcionalidad en una clase genérica que podría llamarse Figura.
- * No debemos utilizar la herencia solamente por el hecho de reutilizar código.
- * Si no se cumple la regla **Es-un**, entonces no debemos aplicar la herencia.
 - Una rosa es una flor, luego la herencia tiene sentido.
 - Una flor es una rosa????, la herencia no tiene sentido.

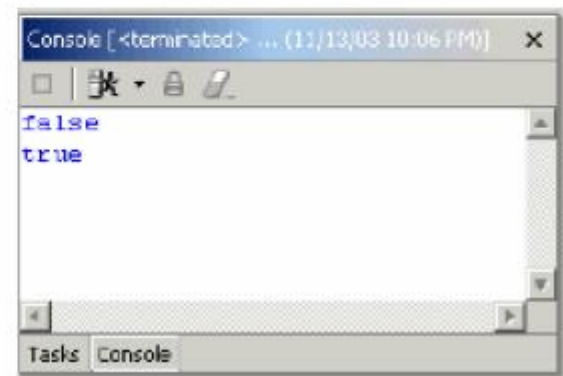
ACCESO A MÉTODOS Y ATRIBUTOS DE LA CLASE PADRE

- * En una clase, se puede utilizar la palabra **super**, para poder acceder a las propiedades y métodos (incluyendo constructores) de la clase padre del objeto.

```
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);
    }
}
```

Atributo de la
Superclase



Herencia y ocultación de información: control de acceso en Java.

Los principales modificadores de acceso en Java, que proporcionan mecanismos para la ocultación de información son `public`, `private` y `protected`. Veamos cómo afectan estos modificadores de acceso a las subclases:

- Cualquier miembro de la superclase cuyo acceso sea de tipo `public` podrá ser accedido desde la subclase, pues recordemos que este modificador indica que al atributo o método que vaya precedido por él se podrá acceder desde cualquier otra clase incluidas, evidentemente, las subclases.

-

Los miembros de la superclase cuyo acceso sea de tipo `private` no podrán ser accedidos desde la subclase bajo ningún concepto, pues recordemos que este modificador indica que al atributo o método que vaya precedido por él sólo se podrá acceder desde la propia clase. Por lo tanto, al contrario de lo que ocurría en el código de la clase `Trabajador`, en el código de las subclases `TrabajadorFijo` o `TrabajadorTemporal` no podremos hacer referencia a estos miembros privados heredados, pues nos daría un error de sintaxis.

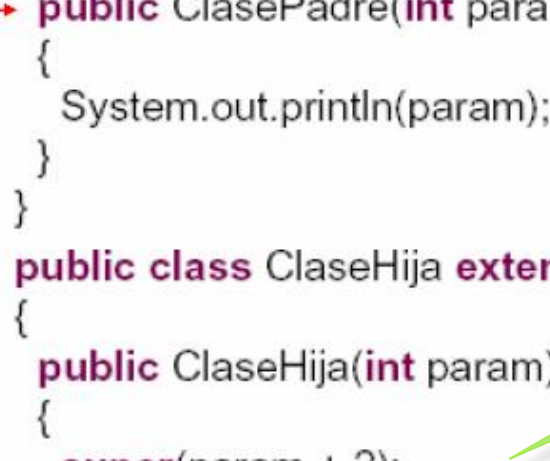
Si una subclase pudiese acceder a los datos privados de su superclase, las clases que heredasen de esa subclase también podrían acceder a ellos y, en definitiva, se propagaría el acceso a las partes declaradas como privadas, perdiéndose consecuentemente los beneficios del ocultamiento de la información.

- Por su parte, los miembros de la superclase cuyo acceso sea de tipo `protected` podrán ser accedidos desde todas sus subclases, pertenezcan éstas o no al mismo paquete, pues indica que al atributo o método que vaya precedido por él sólo se podrá acceder desde la propia clase, desde sus subclases y desde las clases que pertenezcan a su mismo paquete.

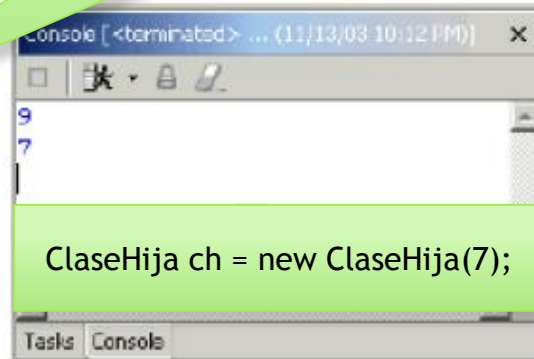
ACCESO AL CONSTRUCTOR DE UNA SUPERCLASE.- Ejemplo

```
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);
    }
}
```



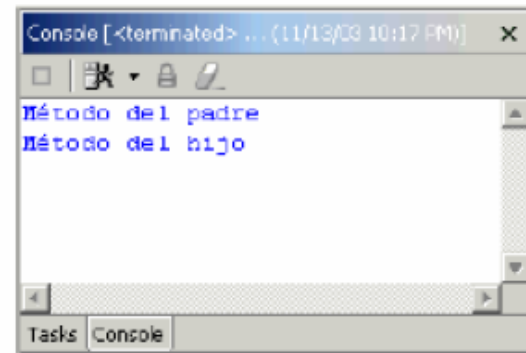
NOTA: Tiene que ser la primera línea del constructor. Sólo puede usarse una vez por constructor. Tendrá tantos parámetros como el constructor al que llama.



ACCESO A UN MÉTODO DE LA SUPERCLASE.- Ejemplo

```
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");
    }
}
```



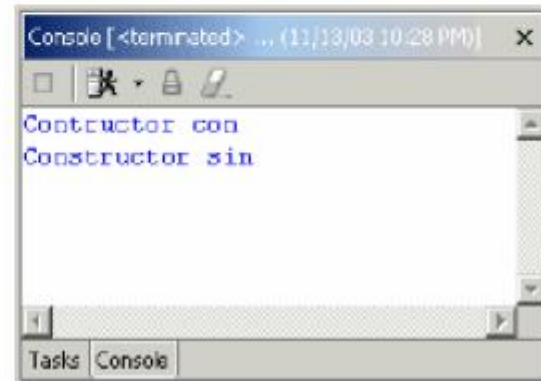
USO DE this

- * **this** es una referencia al objeto actual que se utiliza para acceder a propiedades y métodos del propio objeto.
- * Hay dos ocasiones en las que resulta útil, aunque parezca una redundancia:
 - Acceso a un constructor desde otro constructor.
 - Acceso a un atributo desde un método donde hay definidas unas variables locales con el mismo nombre que el atributo.

USO DE this -Ejemplo

```
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. Sólo puede usarse una vez por constructor.



CLASES ABSTRACTAS

- * Una clase abstracta **no** permite que se **instancien objetos**, sin embargo otras clases pueden heredar de ellas los métodos y propiedades que tenga definidas.
- * Se pueden utilizar para agrupar bajo un mismo tipo a clases similares y también para reutilizar código, ya que si varias clases utilizan un mismo método, éste podría ser definido en una clase abstracta de donde las otras lo heredarían.
- * Por ejemplo, la clase Figura podría ser abstracta, ya que nunca vamos a hacer una instancia suya, sino de sus hijos (círculos, triángulos, ...)

MÉTODOS ABSTRACTOS

- * Son métodos que no están implementados, es decir, no tienen instrucciones que ejecutar, solamente figura la cabecera con los parámetros que aceptan y el tipo de dato devuelto.
- * Una clase que tenga algún método abstracto tendrá que ser declarada abstracta.
- * Cuando una clase hereda de otra que tiene métodos abstractos, la clase hija deberá implementar obligatoriamente todos estos métodos.
- * Se utilizan precisamente para obligar a que las clases hijas tengan que implementarlos.

MÉTODOS ABSTRACTOS.- Ejemplo

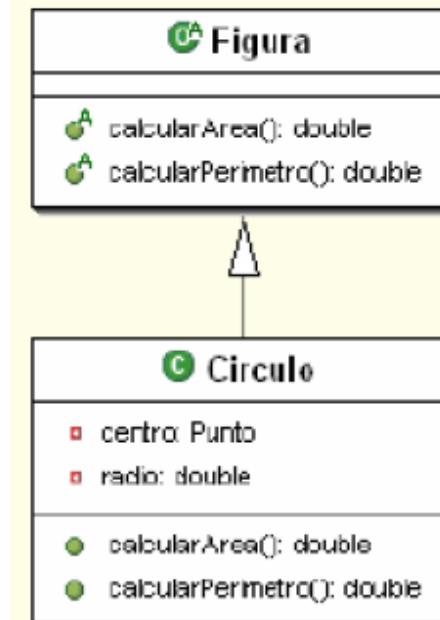
Modificador abstract

```
public abstract class Figura
{
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}

public class Circulo extends Figura
{
    private Punto centro = null;
    private double radio = 0.0;

    public double calcularArea()
    {
        return Math.PI*radio*radio;
    }

    public double calcularPerimetro()
    {
        return 2*Math.PI*radio;
    }
}
```



EL MODIFICADOR final

- * Se usa para **evitar** que un método se pueda **redefinir** en una subclase.

```
class Consultor extends Trabajador
{
    ...
    public final double calcularPaga ()
    {
        return horas*tarifa;
    }
    ...
}
```

Aunque creemos subclases de Consultor, no podremos modificar el cálculo de la paga

- * Evitar que se puedan **crear subclases** de una clase dada.

```
public final class Circulo extends Figura
...
```

No se pueden crear subclases de Circulo