

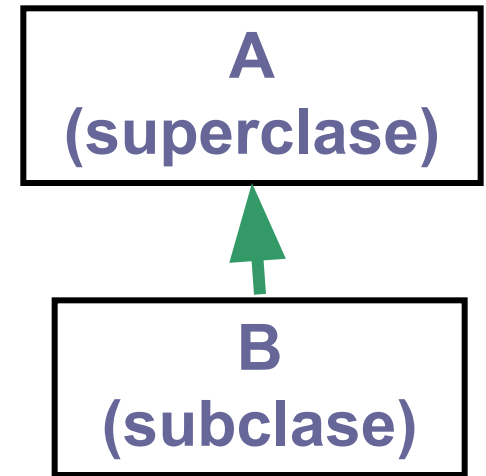
# Herencia y clases abstractas

- Herencia
  - La palabra clave **extends**
  - Sobreescritura
  - La palabra clave **super**
  - Upcasting-Downcasting
- La clase Object
  - Los métodos **equals (Object)** y **toString()**
- Clases abstractas

# Herencia

- La programación orientada a objetos (POO) 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 de que una clase hereda los atributos (variables) y el comportamiento (métodos) de otra clase.

- La clase que hereda se llama subclase o clase derivada. La clase B es subclase de la clase A.
- La clase A es la superclase o clase base de la clase B.

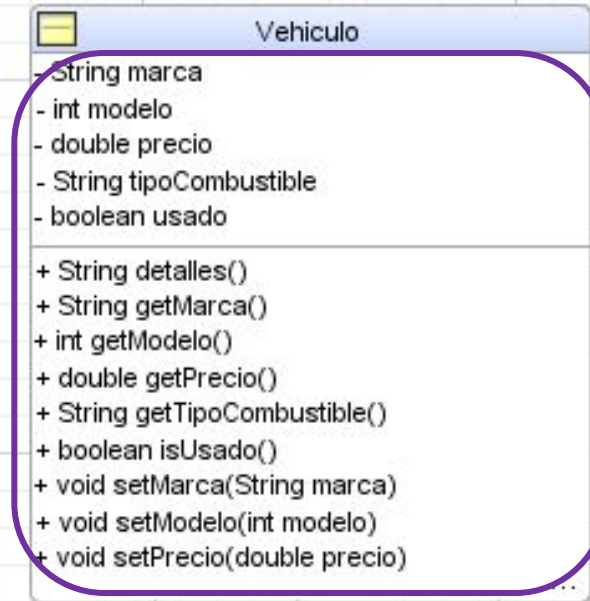


La **herencia** toma una clase existente y construye una versión especializada => **reusabilidad de código**

# Herencia

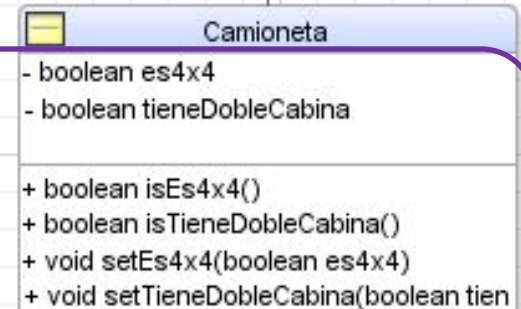
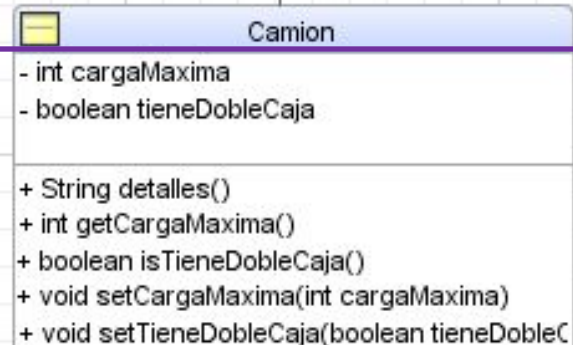
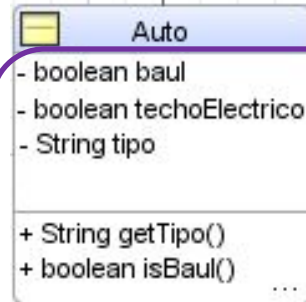
- Una subclase puede **agregar atributos y comportamiento** a su superclase.
- Una subclase puede **reemplazar o modificar el comportamiento** heredado. Esto es sobrescritura.

**Variables y métodos de instancia comunes a todos los vehículos**



**Java permite múltiples niveles de herencia, pero la herencia es simple**

**superclase**



**subclases**

**Variables y métodos de instancia ESPECÍFICOS de cada tipo de vehículo**

# Herencia

## La palabra clave `extends`

Automáticamente, la subclase obtiene las variables y métodos de la superclase

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

```
public class Camion extends Vehiculo {  
    private boolean tieneDobleCaja;  
    private int cargaMaxima;  
    public void setCargaMaxima(int cargaMaxima){  
        this.cargaMaxima = cargaMaxima;  
    }  
    public int getCargaMaxima() {  
        return cargaMaxima;  
    } . . .  
}
```

```
public class Camioneta extends Vehiculos {  
    private boolean tieneDobleCabina;  
    private boolean es4x4;  
    public void setTieneDobleCabina(boolean tieneDobleCabina){  
        this.tieneDobleCabina = tieneDobleCabina;  
    }  
  
    public boolean isTieneDobleCabina() {  
        return tieneDobleCabina;  
    }  
    . . .  
}
```

```
public class Auto extends Vehiculo {  
    private String tipo; //sedan, monovolumne,..  
    . . .  
}
```

# Herencia

## Invocación de métodos heredados

```
Vehiculo miAuto = new Vehiculo();
```

*¿Qué puedo hacer sobre el objeto miAuto?*

```
double p= miAuto.getPrecio();  
String c= miAuto.getTipoCombustible();  
if (miAuto.esUsado()) {...}
```

```
Camion miCamion = new Camion();
```

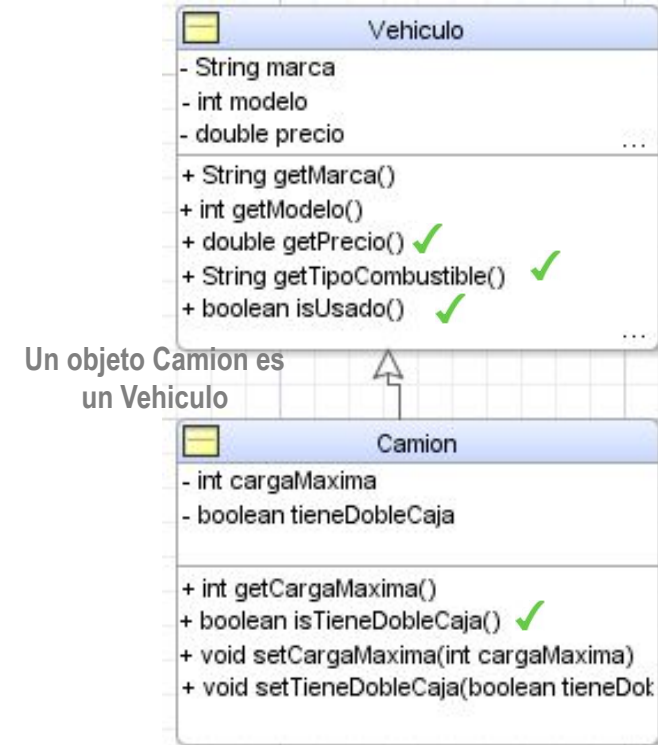
*¿Qué puedo hacer sobre el objeto miCamion ?*

```
double p= miCamion.getPrecio();
```

se pueden invocar todos los métodos heredados de Vehiculo.

```
if (miCamion.isTieneDobleCaja()) {...}
```

también se pueden invocar todos los métodos de Camión

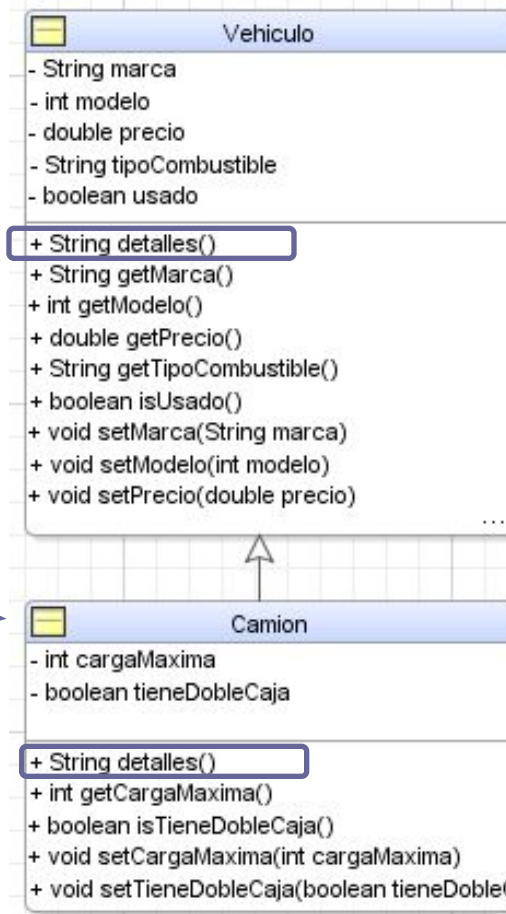


# Herencia

## Sobrescritura de métodos

Un método **sobrescribe** a otro método cuando se define en una subclase y coincide el **nombre**, **tipo de retorno** y **lista de argumentos** con un método ya definido en una superclase.

La clase **Camion** hereda todos los atributos de **Vehículo** y especifica dos adicionales, **cargaMaxima** y **tieneDobleCaja** y sobrescribe el método **detalles()**



Es posible crear una clase nueva basada en una existente, agregándole características adicionales y modificándole el comportamiento a la superclase.

# Herencia

## Sobrescritura de métodos

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

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

```
    // getters y setters: getMarca(), getPrecio()  
}
```

```
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();  
    }
```

```
    // getter y setter  
}
```

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());  
}
```

**Se ejecutan  
métodos  
diferentes !!**

**SALIDA**

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

# Herencia

## Sobrescritura de métodos – La palabra clave super

```
public class Vehiculo {  
    private String marca;  
    private double precio;  
    . . .  
  
    public String detalles() {  
        return "Vehiculo marca: " + getMarca() +  
            "\n" + "Precio: " + getPrecio();  
    }  
  
    . . .  
}
```

```
public class Camion extends Vehiculo {  
    private boolean tieneDobleCaja;  
    private int cargaMaxima;  
  
    public String detalles() {  
        return super.detalles() + "\n"  
            + "carga máxima:" + getCargaMaxima();  
    }  
  
    .  
}
```

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

**SI!!!!!!**

¿Cómo?

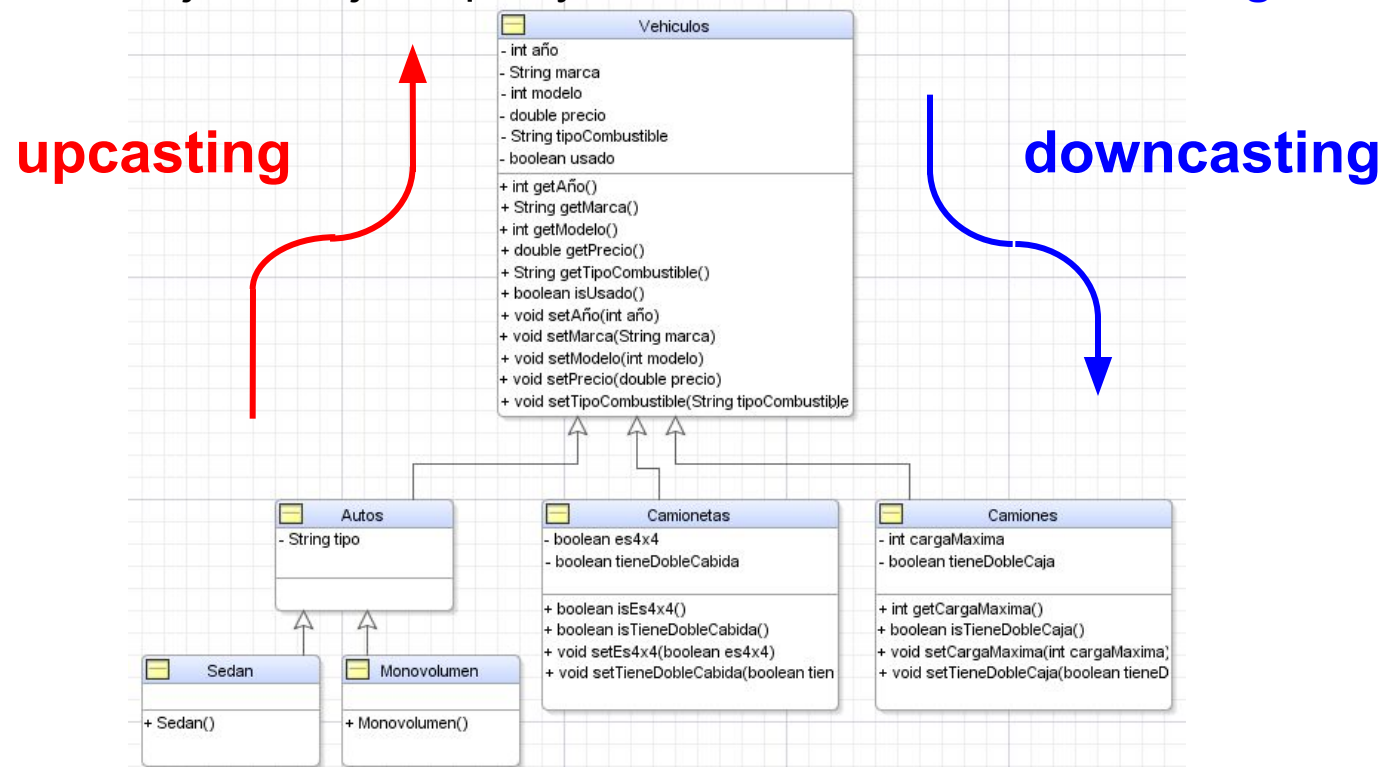
Usando la palabra clave **super**



# Herencia

## Upcasting - Downcasting

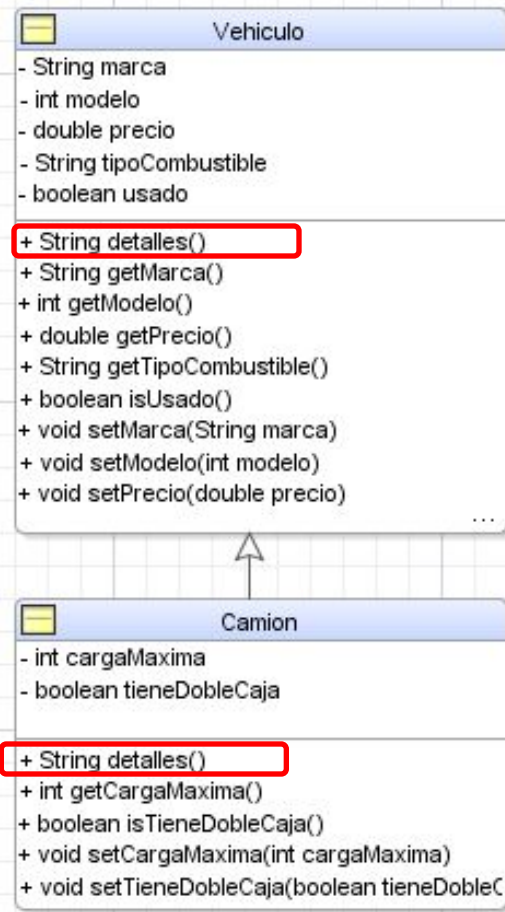
Tratar a una referencia de la clase derivada como una referencia de la clase base, se denomina upcasting. Con el **upcasting**, la conversión es hacia arriba en la jerarquía de herencia y se pierde el tipo específico del objeto. Para recuperar el tipo del objeto, se debe mover hacia abajo en la jerarquía y a esto se lo llama **downcasting**.



El **upcasting** es seguro, la clase base tiene una interface que es igual o es un subconjunto de la clase derivada. Pero, en el **downcasting** no ocurre lo mismo.

# Herencia

## Sobrescritura de métodos



upcasting

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

### ¿Qué método se ejecuta?

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

### ¿Qué imprime?

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

NO Compila

# Herencia

## La clase Object

- La clase `Object` es la raíz 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;  
    . . .  
  
    public String detalles() {  
        return "Vehiculo marca: "+getMarca()  
            + "\n"+"Precio: "+ getPrecio();  
    }  
    . . .  
}
```

```
public class Vehiculo extends Object{  
    private String marca;  
    private double precio;  
    . . .  
  
    public String detalles() {  
        return "Vehiculo marca: "+getMarca()  
            + "\n"+"Precio: "+ getPrecio();  
    }  
    . . .  
}
```

De esta manera, estamos habilitados para sobrescribir los métodos heredados de `Object`.

# Herencia

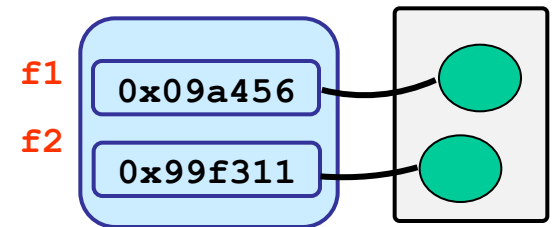
## La clase Object – Los métodos `equals(Object o)` y `toString()`

- El método `public boolean equals(Object obj) {}`, compara la igualdad de 2 objetos. La versión original del método `equals()`, devuelve `true` si las dos referencias son iguales, es decir si apuntan al mismo objeto en memoria. Es equivalente a usar el operador `==`.
- El método `public String toString() {}`, retorna la dirección del objeto como un `String`.

La clase **Fecha** definida debajo hereda los métodos `equals(Object o)` y `toString()` de `Object`:

```
public class Fecha {  
    private int dia= 1;  
    private int mes= 1;  
    private int año=2007;  
    // métodos de instancia  
}
```

```
Fecha f1 = new Fecha();  
Fecha f2 = new Fecha();  
f1.equals(f2) → false  
f1==f2 → false  
f1.toString() → Fecha@360be0
```



La intención del método `equals(Object o)` es comparar el contenido de dos objetos y la del `toString()` es producir una representación textual, concisa, legible y expresiva del contenido del objeto.

# Herencia

## La clase Object – Los métodos `equals(Object o)` y `toString()`

```
public class Fecha {
    private int dia = 1;
    private int mes = 1;
    private int año = 2006;

    public boolean equals(Object o){
        boolean result=false;
        if ((o!=null) && (o instanceof 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 "Fecha:"+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());
    }
}
```

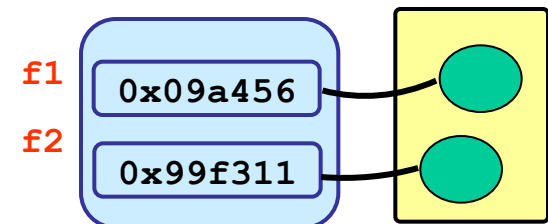
Es un operador que permite determinar la clase real del objeto

La salida es:  
**false**  
**true**  
**1-1-2006**

Sobrescribimos en la clase **Fecha**, los métodos `equals(Object o)` y `toString()` heredados de **Object**:

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 de objeto **Fecha** en una representación legible.

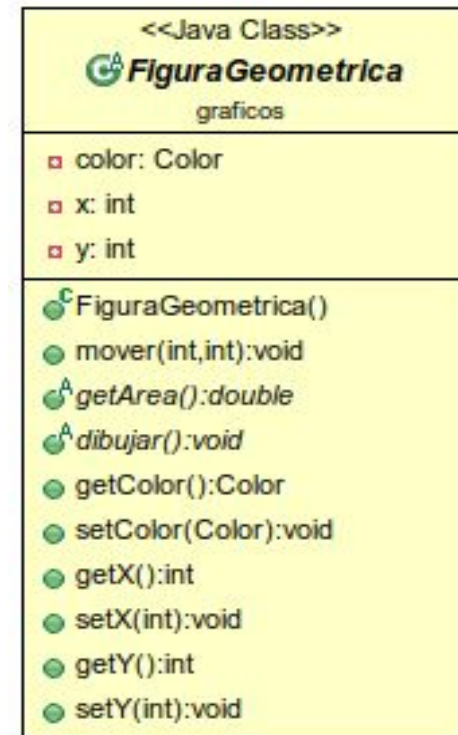


# Herencia y clases abstractas

En **Programación Orientada a Objetos** podríamos modelar un concepto abstracto mediante una clase abstracta cuya finalidad NO es crear instancias como en las clases que venimos definiendo hasta ahora.

Pensemos en una aplicación que dibuja figuras geométricas, podríamos dibujar por ejemplo: **círculos**, **rectángulos**, **triángulos**, **líneas rectas**, etc. Todas las figuras geométricas pueden cambiar de color, dibujarse en la pantalla, moverse, etc., pero cada una lo hace de una manera particular.

Por otro lado, una figura geométrica, es un concepto abstracto, no es posible dibujarla o redimensionarla, sólo sabemos que todas las figuras geométricas concretas, como los círculos, rectángulos, triángulos tienen esas capacidades.



# Herencia y clases abstractas

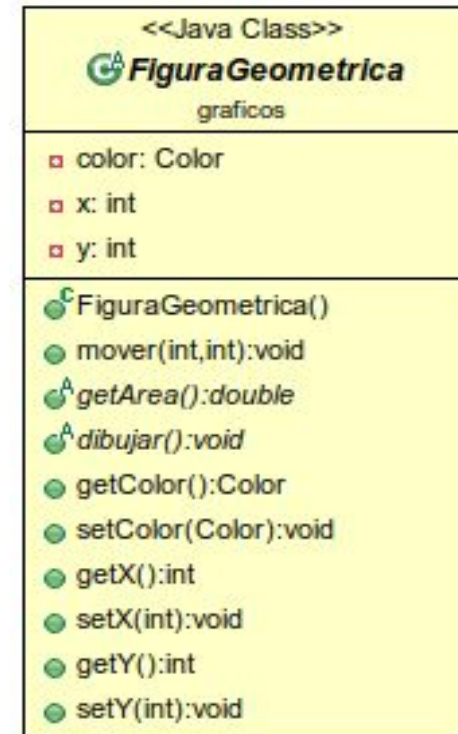
Si tratamos de codificar esta clase, podríamos hacerlo así:

```
package graficos;
import java.awt.Color;

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 double getArea() {
        ??
    }
    public void dibujar() {
        ??
    }
    public void setColor(Color color) {
        this.color = color;
        dibujar
    }
    . . .
}
```

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





# Herencia y clases abstractas

La definición de la clase abstracta es la siguiente:

```
package graficos;
import java.awt.Color;

public abstract class FiguraGeometrica {
    private Color color;
    private int x;
    private int y;

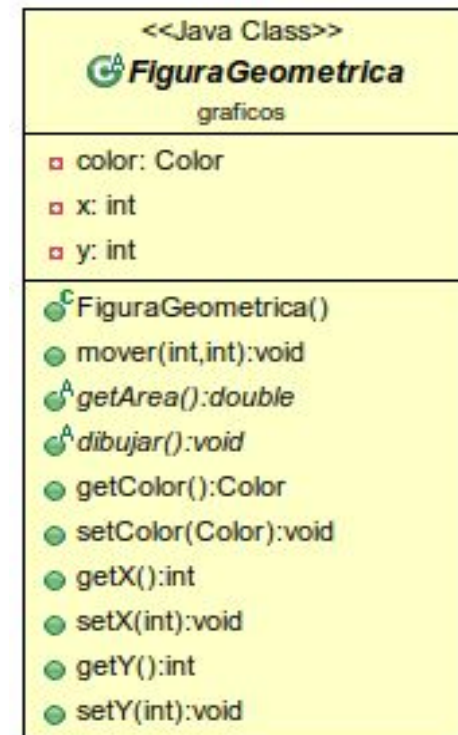
    public void mover(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract double getArea();
    public abstract void dibujar();

    public void setColor(Color color) {
        this.color = color;
        dibujar
    }
    . . .
}
```

De esta clase no se podrán crear instancias.

Se espera que las subclases implementen estos métodos.





# Herencia y clases abstractas

## Clase abstracta

El objetivo de definir una clase abstracta es lograr una **interface de comportamiento común** para los objetos de las subclases (de la clase abstracta) .

No se pueden crear instancias a partir de una clase abstracta. Se espera que una **clase abstracta** sea **extendida** por clases que implementen todos sus métodos abstractos.

## Método abstracto

Es un método que no tiene cuerpo! Se debe anteponer la palabra clave **abstract** al tipo de datos de retorno del método.

**public abstract double area();**

- Si dentro de una clase se declara un método abstracto, la clase debe declararse abstracta.
- Las clases abstractas pueden tener métodos concretos y métodos abstractos.
- Las clases concretas no pueden tener métodos abstractos, solo métodos concretos.

# Herencia y clases abstractas

La clase `FiguraGeometrica` es una clase abstracta. La primer clase concreta en la jerarquía debe tener implementados todos los métodos abstractos de la clase `FiguraGeometricas`.

```
package graficos;

public class Circulo extends FiguraGeometrica {

    @Override
    public double getArea() {
        . . .
    }

    @Override
    public void dibujar() {
        . . .
    }
}
```

