

TEMA: CONCEPTO DE HERENCIA Y POLIMORFISMO

Taller de Programación.

Módulo: Programación Orientada a Objetos

Introducción

- Diferentes tipos de objetos con características y comportamiento común.

Triángulo



- Lado1 / lado2 / lado3
- color de línea
- color de relleno

Círculo



- radio
- color de línea
- color de relleno

Cuadrado



- lado
- color de línea
- color de relleno

- Devolver y modificar el valor de cada atributo

lado1 / lado2 / lado3

color de línea / color de relleno

- Calcular el área
- Calcular el perímetro

- Devolver y modificar el valor de cada atributo

radio

color de línea / color de relleno

- Calcular el área
- Calcular el perímetro

- Devolver y modificar el valor de cada atributo

lado

color de línea / color de relleno

- Calcular el área
- Calcular el perímetro

Inconvenientes hasta ahora. Herencia como solución.

- Esquema de trabajo hasta ahora
 - Definimos las clases Triángulo, Circulo, Cuadrado (sin relacionarlas).
 - Problema: Replicación de características y comportamiento común.
- Solución → Herencia
 - Se define **lo común en una clase Figura** y las clases **Triángulo, Círculo y Cuadrado lo heredan y definen lo específico.**
 - **Herencia:** mecanismo que permite que una clase **herede** características y comportamiento (atributos y métodos) de otra clase (clase padre o superclase). A su vez, la clase hija define características y comportamiento propio.
 - Ventaja: **reutilización de código**

Herencia. Ejemplo.

- Diagrama de clases.

Las clases forman una jerarquía.

“es un”

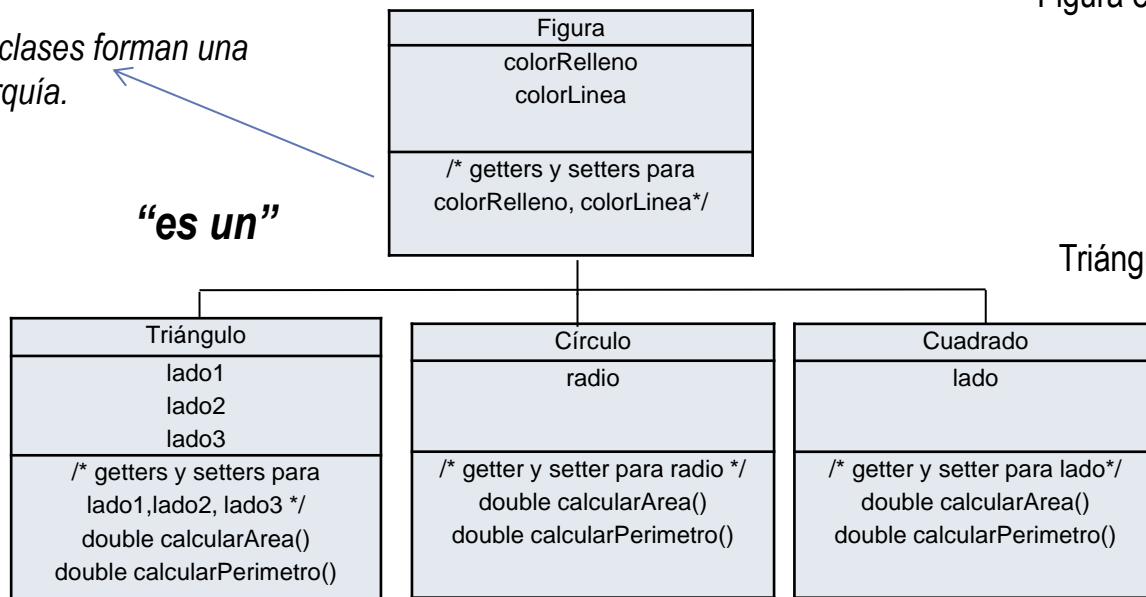


Figura es la **superclase** de Triángulo/Círculo/Cuadrado

define atributos y comportamiento **común**

Triángulo/Círculo/Cuadrado son **subclases** de Figura.

heredan atributos y métodos de Figura

definen atributos y métodos **propios**

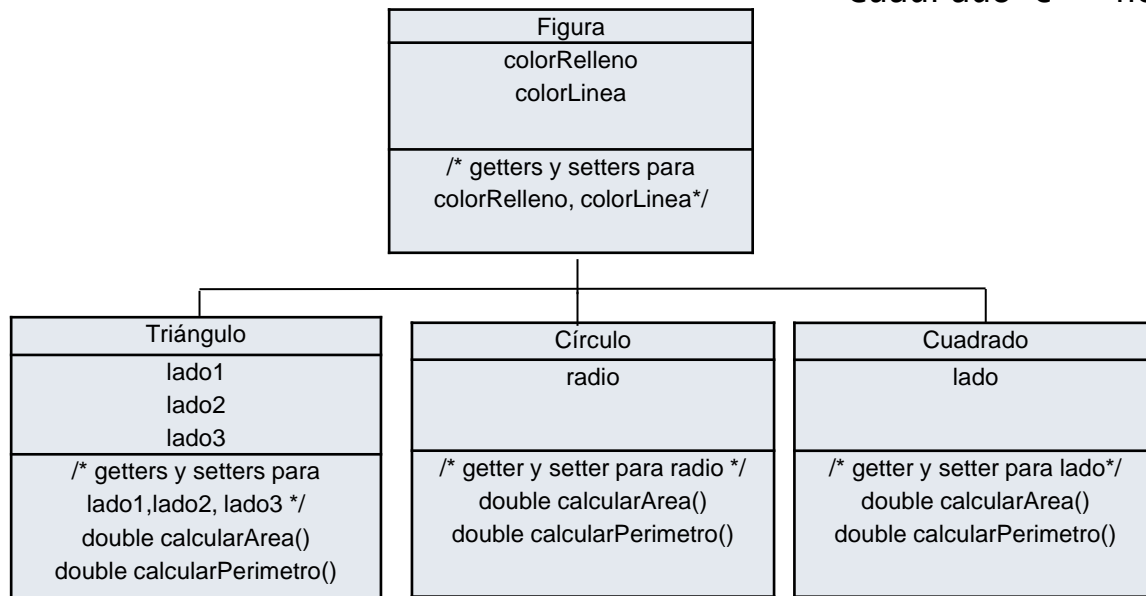
definen constructores.

deben implementar `calcularArea()` y `calcularPerimetro()` pero de manera diferente → **POLIMORFISMO**

Herencia. Ejemplo.

```
/* Ejemplo main */
```

```
Cuadrado c = new Cuadrado(10,"rojo","negro");
```



¿Qué variables de instancia almacena el objeto "c"?

¿Qué mensajes le puedo enviar al objeto "c"?

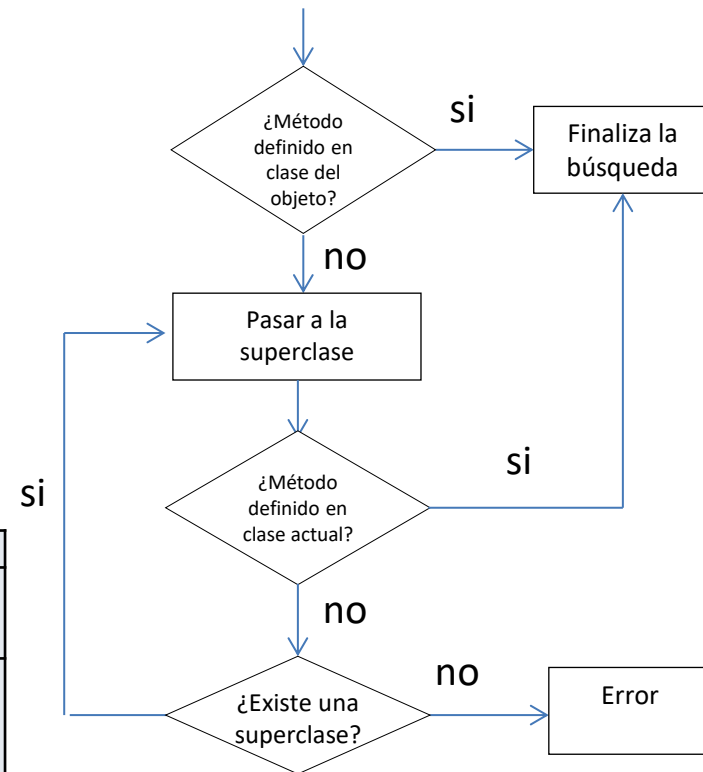
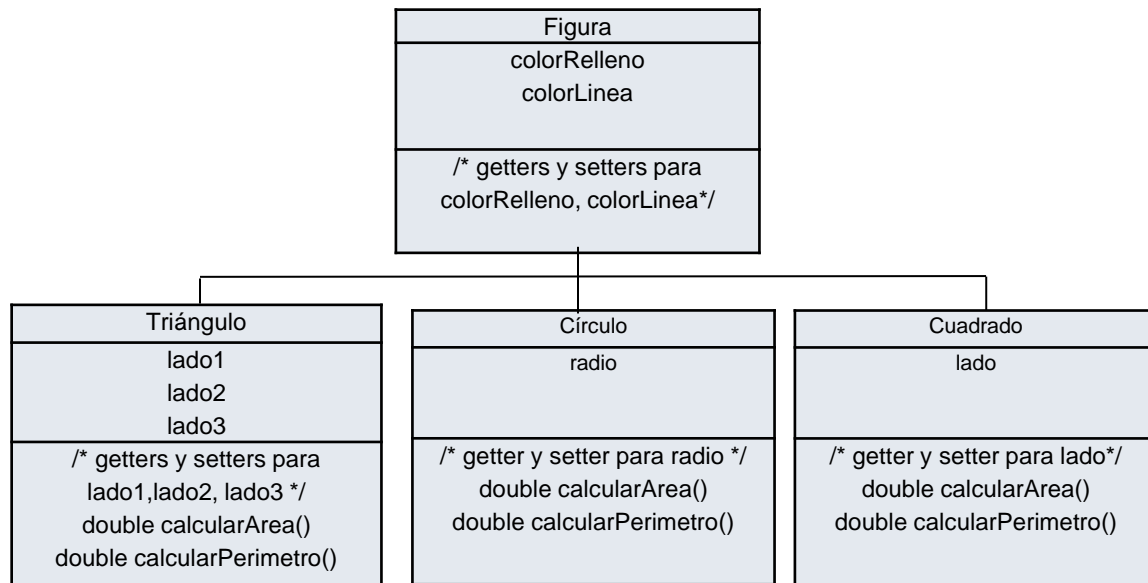
Búsqueda de método en la jerarquía de clases

```
/* Ejemplo main */
```

```
Cuadrado c = new Cuadrado(10,"rojo","negro");
```

```
System.out.println(c.calcularArea());
```

```
System.out.println(c.getColorRelleno());
```




Herencia en Java

- Definición de relación de herencia. Palabra clave ***extends***.

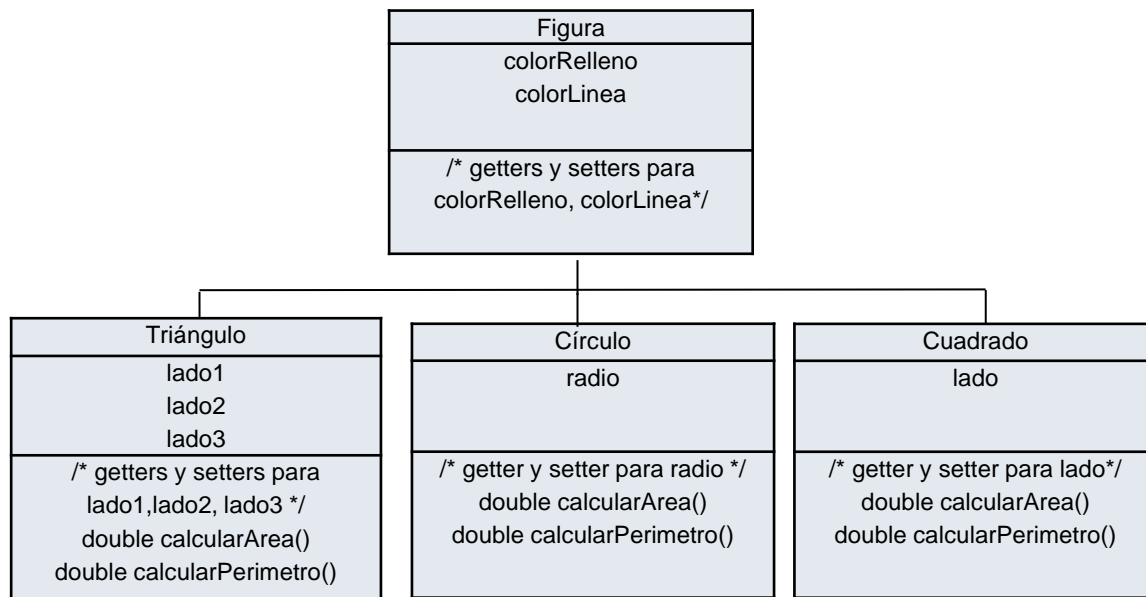
```
public class NombreSubclase extends NombreSuperclase {  
    /* Definir atributos propios */  
    /* Definir constructores propios */  
    /* Definir métodos propios */  
}
```

```
public class Figura{  
    private String colorRelleno;  
    private String colorLinea;  
  
    /* Métodos getters y setters  
    para colorRelleno y colorLinea*/  
    ...  
}
```

```
public class Cuadrado extends Figura{  
    private int lado;  
    /* Métodos */  
    ...  
    public void hacerAlgo(){  
        colorRelleno=...;  Usar setter/getter  
    }                               para acceder a  
    }                               atributo  
    }                               heredado
```

- La **subclase hereda** atributos y métodos de instancia declarados en la **superclase**.
 - Nota: los atributos declarados en una superclase como **privados** no son accesibles directamente en sus subclases. Para *accederlos* en una subclase se deben usar los *getters* y *setters* heredados.
- La **subclase puede declarar** atributos, métodos y constructores propios.

Clases y métodos abstractos. Introducción.



¿Tiene sentido instanciar un objeto a partir de la clase *Figura*?



¿Hay alguna manera de impedirlo?

Toda subclase de *Figura* debe implementar *calcularArea* y *calcularPerimetro*



¿Hay alguna manera de forzar a las subclases a definir estos métodos?

Clases y métodos abstractos. Definición.

- **Clase abstracta:** es una clase que no puede ser instanciada (no se pueden crear objetos a partir de ella). Se usa para definir características y comportamiento común para un conjunto de clases (subclases). Puede definir **métodos abstractos** (sin implementación) que deben ser implementados por las subclases.

Ejemplo La clase **Figura** es *abstracta*.

Figura puede declarar *métodos abstractos* `calcularArea` y `calcularPerimetro`.

- **Declaración de clase abstracta:** anteponer *abstract* a la palabra `class`.

```
public abstract class NombreClase {  
    /* Definir atributos */  
    /* Definir métodos no abstractos (con implementación) */  
    /* Definir métodos abstractos (sin implementación) */  
}
```

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

- **Declaración de método abstracto:** encabezado del método (sin código) anteponiendo *abstract* al tipo de retorno.

```
public abstract TipoRetorno nombreMetodo(lista parámetros formales);
```

Ejemplo

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...

    public abstract double calcularArea();
    public abstract double calcularPerimetro();

    MÉTODOS ABSTRACTOS
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructor*/
    public Cuadrado(double unLado,
                    String unColorR,
                    String unColorL){

        lado=unLado;
        colorRelleno=unColorR;
        colorLinea=unColorL;
    }
}
```

colorRelleno y
colorLinea declarados
"private" en Figura

Usar setter/getter
heredado

}

Ejemplo

Superclase

```
public abstract class Figura{  
    private String colorRelleno, colorLinea;  
  
    public String getColorRelleno(){  
        return colorRelleno;  
    }  
    public void setColorRelleno(String unColor){  
        colorRelleno = unColor;  
    }  
    ...  
}
```

```
public abstract double calcularArea();  
public abstract double calcularPerimetro();
```

MÉTODOS ABSTRACTOS

```
}
```

Subclase

```
public class Cuadrado extends Figura{  
    private double lado;  
  
    /*Constructor*/  
    public Cuadrado(double unLado,  
                    String unColorR,  
                    String unColorL){  
        lado=unLado;  
        setColorRelleno(unColorR);  
        setColorLinea(unColorL);  
    }  
}
```

nombreMétodo(parámetros) ó this.nombreMétodo(parámetros)

El objeto que está ejecutando (**this**) se autoenvía un mensaje

La búsqueda del método a ejecutar inicia en la clase del objeto

```
}
```

Ejemplo

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...

    public abstract double calcularArea();
    public abstract double calcularPerimetro();

    MÉTODOS ABSTRACTOS
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructor*/
    public Cuadrado(double unLado,
                    String unColorR,
                    String unColorL){
        lado=unLado;
        setColorRelleno(unColorR);
        setColorLinea(unColorL);
    }

    /* Metodos getLado y setLado */
    ...

    public double calcularPerimetro(){
        return lado*4;
    }
    public double calcularArea(){
        return lado*lado;
    }
}
```

IMPLEMENTA
calcularArea
calcularPerimetro

Ejemplo

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...

    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructor*/
    public Cuadrado(double unLado,
                    String unColorR,
                    String unColorL){
        lado=unLado;
        setColorRelleno(unColorR);
        setColorLinea(unColorL);
    }

    /* Metodos getLado y setLado */
    ...

    public double calcularPerimetro(){
        return lado*4;
    }
    public double calcularArea(){
        return lado*lado;
    }
}
```

Otra opción:
en vez de usar
directamente la v.i. "lado"
podemos hacer que el
objeto se autoenvíe un
mensaje para
modificar/obtener el
valor.
¿Cómo?

Ejemplo

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...

    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructor*/
    public Cuadrado(double unLado,
                    String unColorR,
                    String unColorL){
        setLado(unLado);
        setColorRelleno(unColorR);
        setColorLinea(unColorL);
    }

    /* Metodos getLado y setLado */
    ...

    public double calcularPerimetro(){
        return getLado()*4;
    }
    public double calcularArea(){
        return getLado()*getLado();
    }
}
```

Otra opción:
en vez de usar
directamente la v.i. "lado"
podemos hacer que el
objeto se autoenvíe un
mensaje para
modificar/obtener el
valor.

Buena práctica en POO

Ejemplo (Continuación)

- **Añadir** la clase `Círculo` a la jerarquía de Figuras.
- **Añadir** el método `toString` que retorne la representación String de cada figura, siguiendo el ejemplo:
 - Cuadrados: `"CR: rojo CL: azul Lado: 3"`
 - Círculos: `"CR: verde CL: negro Radio:4"`

Subclases de Figura

```
public class Cuadrado extends Figura{  
    private double lado;
```

```
    /*Constructor*/  
    public Cuadrado(double unLado,  
                    String unColorR,  
                    String unColorL){  
        setLado(unLado);  
        setColorRelleno(unColorR);  
        setColorLinea(unColorL);  
    }
```

```
    /* Metodos getLado y setLado */  
    /* Métodos calcularArea y calcularPerimetro */
```

```
    public String toString(){  
        String aux = "CR:" + getColorRelleno() +  
                    "CL:" + getColorLinea() +  
                    " Lado: " + getLado();  
        return aux;  
    }  
}
```

Código
replicado

**Solución:**

Factorizar código
común en la
superclase e
"invocarlo" desde las
subclases

```
public class Circulo extends Figura{  
    private double radio;
```

```
    /*Constructor*/  
    public Circulo(double unRadio,  
                   String unColorR,  
                   String unColorL){  
        setRadio(unRadio);  
        setColorRelleno(unColorR);  
        setColorLinea(unColorL);  
    }
```

```
    /* Metodos getRadio y setRadio */  
    /*Métodos calcularArea y calcularPerimetro*/
```

```
    public String toString(){  
        String aux = "CR:" + getColorRelleno() +  
                    "CL:" + getColorLinea() +  
                    "Radio:" + getRadio();  
        return aux;  
    }  
}
```


Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "CR:" + getColorRelleno() +
                     "CL:" + getColorLinea();

        return aux;
    }
    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructor*/
    public Cuadrado(double unLado,
                    String unColorR,
                    String unColorL){
        super(unColorR,unColorL);
        setLado(unLado);
    }

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
                     " Lado: " + getLado();

        return aux;
    }
}
```

super(...)
Invoco al constructor de la superclase. Al declarar un constructor en la superclase esta invocación *debe ir como primera línea*

super → referencia al objeto que está ejecutando
super.nombreMétodo(...) → El objeto se autoenvía un mensaje
 La búsqueda del método inicia en la clase superior a la actual

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "CR:" + getColorRelleno() +
                     "CL:" + getColorLinea();

        return aux;
    }
    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructor*/
    public Cuadrado(double unLado,
                    String unColorR,
                    String unColorL){
        super(unColorR,unColorL);
        setLado(unLado);
    }

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
                     " Lado: " + getLado();

        return aux;
    }
}
```

```
/* Ejemplo main */
```

```
Cuadrado c = new Cuadrado(10,"rojo","negro");
System.out.println(c.toString());
```



Ejecución

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "CR:" + getColorRelleno() +
                     "CL:" + getColorLinea();

        return aux;
    }
    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
                     " Lado: " + getLado();

        return aux;
    }
}
```

Añadir a la representación String el valor del área.
¿qué método toString modifico?

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "Area:" + this.calcularArea()+
                    "CR:" + getColorRelleno() +
                    "CL:" + getColorLinea();

        return aux;
    }
    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;
```

Modifico toString de Figura.
Así evitamos repetir código en subclases.
¿Qué *calcularArea* se ejecuta?
¿Cuándo se determina?

```
    }

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
                    " Lado: " + getLado();

        return aux;
    }
}
```

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }
    public String toString(){
        String aux = "Area:" + this.calcularArea()+
            "CR:" + getColorRelleno() +
            "CL:" + getColorLinea();
        return aux;
    }
    public String getColorRelleno(){
        return colorRelleno;
    }
    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }
    ...
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

¿Qué calcularArea se ejecuta?
¿Cuándo se determina?

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /*Constructor*/
    public Cuadrado(double unLado,
        String unColorR,
        String unColorL){
        super(unColorR,unColorL);
        setLado(unLado);
    }

    /* Metodos getLado y setLado */
    /* Métodos calcularArea y
        calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
            " Lado: " + getLado();
        return aux;
    }
}
```

Subclase

```
public class Circulo extends Figura{
    private double radio;

    /*Constructor*/
    public Circulo(double unRadio,
        String unColorR,
        String unColorL){
        super(unColorR,unColorL);
        setRadio(unRadio);
    }

    /* Metodos getRadio y setRadio */
    /*Métodos calcularArea y
        calcularPerimetro*/

    public String toString(){
        String aux = super.toString() +
            "Radio:" + getRadio();
        return aux;
    }
}
```

Ejecución

/* Ejemplo main */

```
Cuadrado c1 = new Cuadrado(10,"rojo","negro");
System.out.println(c1.toString());
Circulo c2 = new Circulo(5,"verde","azul");
System.out.println(c2.toString());
```

Superclase

```
public abstract class Figura{
    private String colorRelleno, colorLinea;

    public Figura(String unCR, String unCL){
        setColorRelleno(unCR);
        setColorLinea(unCL);
    }

    public String toString(){
        String aux = "Area:" + this.calcularArea()+
            "CR:" + getColorRelleno() +
            "CL:" + getColorLinea();

        return aux;
    }

    public String getColorRelleno(){
        return colorRelleno;
    }

    public void setColorRelleno(String unColor){
        colorRelleno = unColor;
    }

    ...
    public abstract double calcularArea();
    public abstract double calcularPerimetro();
}
```

Subclase

```
public class Cuadrado extends Figura{
    private double lado;

    /* Métodos calcularArea y
       calcularPerimetro */

    public String toString(){
        String aux = super.toString() +
            " Lado: " + getLado();
        return aux;
    }
}
```

Subclase

```
public class Circulo extends Figura{
    private double radio;

    /* Métodos calcularArea y
       calcularPerimetro*/

    public String toString(){
        String aux = super.toString() +
            "Radio:" + getRadio();
        return aux;
    }
}
```

Polimorfismo: objetos de clases distintas responden al mismo mensaje de distinta forma.

Binding dinámico: se determina en tiempo de ejecución el método a ejecutar para responder a un mensaje.

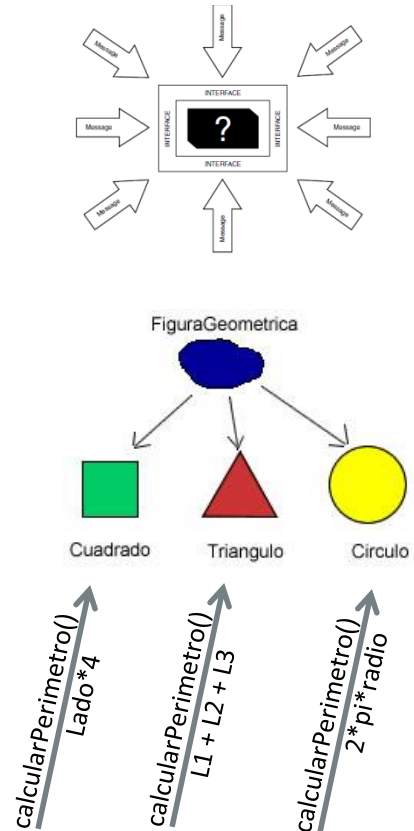
Ventaja: Código genérico, reusable.

/* Ejemplo main */

```
Cuadrado c1 = new Cuadrado(10,"rojo","negro");
System.out.println(c1.toString());
Circulo c2 = new Circulo(5,"verde","azul");
System.out.println(c2.toString());
```

Resumen de conceptos de POO

- **Encapsulamiento:** permite construir componentes autónomos de software, es decir independientes de los demás componentes. La independencia se logra ocultando detalles internos (implementación) de cada componente. Una vez encapsulado, el componente se puede ver como una caja negra de la cual sólo se conoce su interfaz.
- **Herencia:** permite definir una nueva clase en términos de una clase existente. La nueva clase hereda automáticamente todos los atributos y métodos de la clase existente, y a su vez puede definir atributos y métodos propios.
- **Polimorfismo:** objetos de clases distintas pueden responder a mensajes con selector (nombre) sintácticamente idéntico de distinta forma. Permite realizar código genérico, altamente reusable.
- **Binding dinámico:** mecanismo por el cual se determina en tiempo de ejecución el método (código) a ejecutar para responder a un mensaje.



Beneficios de la POO

- **Producir software que sea ...**
 - **Natural.** El programa queda expresado usando términos del problema a resolver, haciendo que sea más fácil de comprender.
 - **Fiable.** La POO facilita la etapa de prueba del software. Cada clase se puede probar y validar independientemente.
 - **Reusable.** Las clases implementadas pueden reusarse en distintos programas. Además gracias a la herencia podemos reutilizar el código de una clase para generar una nueva clase. El polimorfismo también ayuda a crear código más genérico.
 - **Fácil de mantener.** Para corregir un problema, nos limitamos a corregirlo en un único lugar.