TEMA: CONCEPTO DE HERENCIA (UTILIZANDO JAVA)

Taller de Programación.

Módulo: Programación Orientada a Objetos

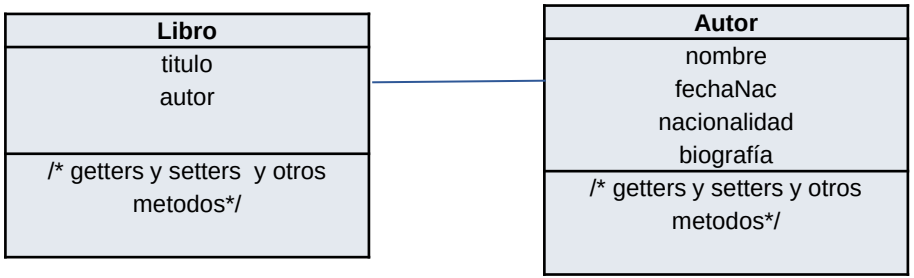
Introducción

Hasta ahora hemos trabajado en la creación de programas en los cuales **declaramos clases** con un conjunto de características (**atributos**) y comportamientos (**métodos**). A partir de esas clases hemos creado **objetos.**

Vimos que los objetos pueden interactuar entre ellos a través del **envío de mensajes** y que para realizar su tarea el objeto puede **delegar** trabajos en otro objeto que puede ser parte de él mismo o no.

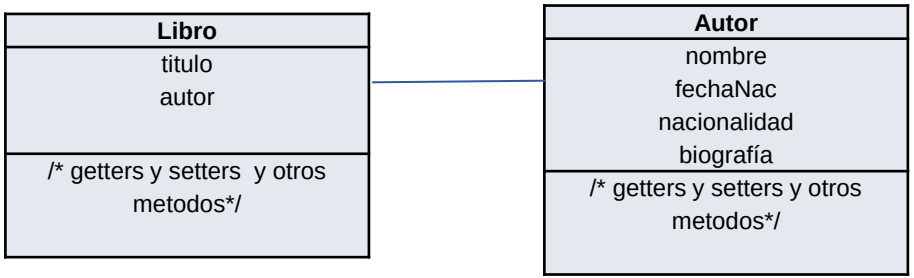
Introducción

Ahora retomemos el ejemplo del libro.



Introducción

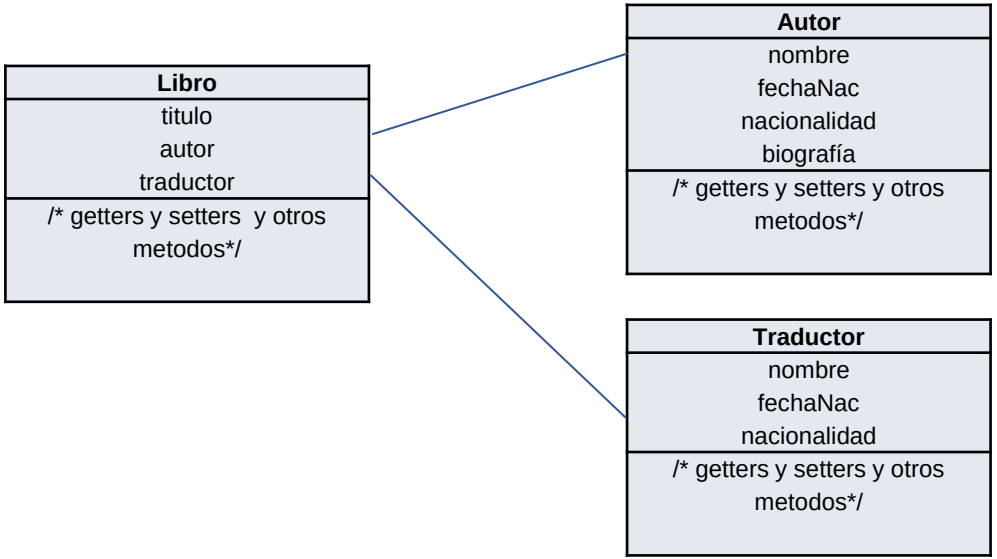
Ahora retomemos el ejemplo del libro.



¿Qué sucede si ahora en mi libro también quiero llevar información del traductor?

Introducción

¿Qué sucede si ahora quiero también en mi libro llevar información del traductor?



Autor y Traductor tienen similar representación y comportamiento

¿No podríamos reutilizar código?

Introducción

Si definimos las clases **Triángulo**, **Círculo** y **Cuadrado** por separado, vamos a replicar características y comportamiento común

Otro ejemplo de diferentes tipos de objetos con características y comportamiento común.

Triángulo 	**Círculo** 	**Cuadrado** 
• Lado1 / lado2 / lado3 • color de línea • color de relleno	• radio • color de línea • color de relleno	• 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

Herencia

Como solución a los problemas planteados vamos a recurrir a la **Herencia**

¿Qué es la Herencia?

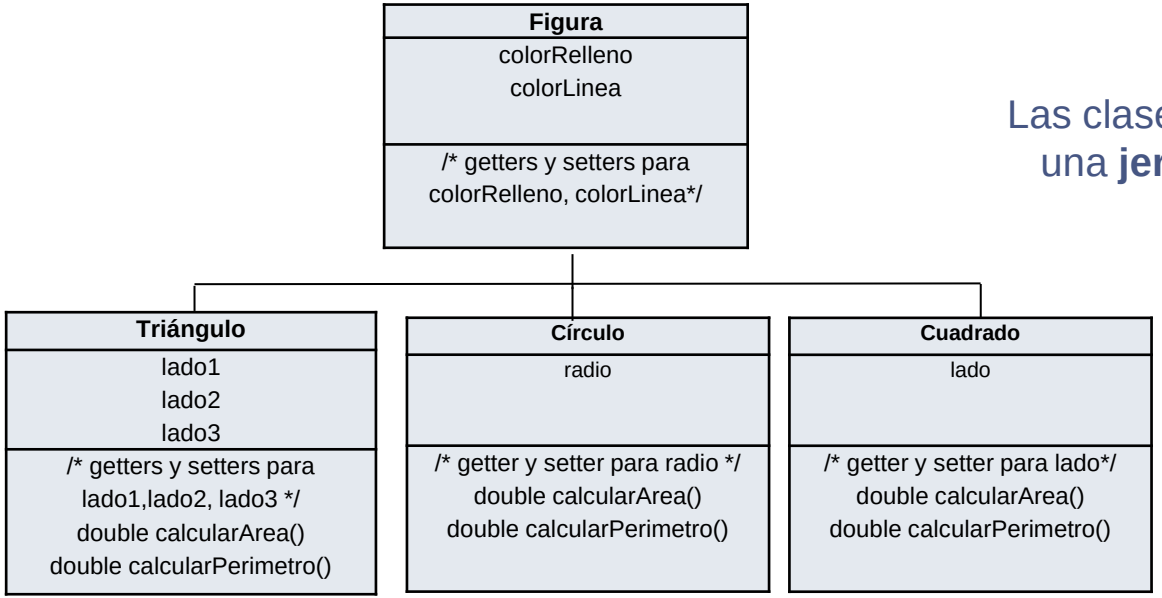
Es un mecanismo que permite que una clase ***herede*** características y comportamiento (atributos y métodos) de otra clase (clase padre o superclase). A su vez, la clase hija define sus propias características y comportamiento.

Ventaja: **reutilización de código**

En el caso anterior definiríamos **lo común en una clase Figura** (superclase) y las **clases Triángulo, Círculo y Cuadrado** heredarían de esta y serían más específicas**.**

Herencia. Ejemplo.

¿Cómo lo vemos en un Diagrama de clases?



Las clases forman una **jerarquía**.

Herencia. Ejemplo.

¿Cómo lo vemos en un Diagrama de clases?

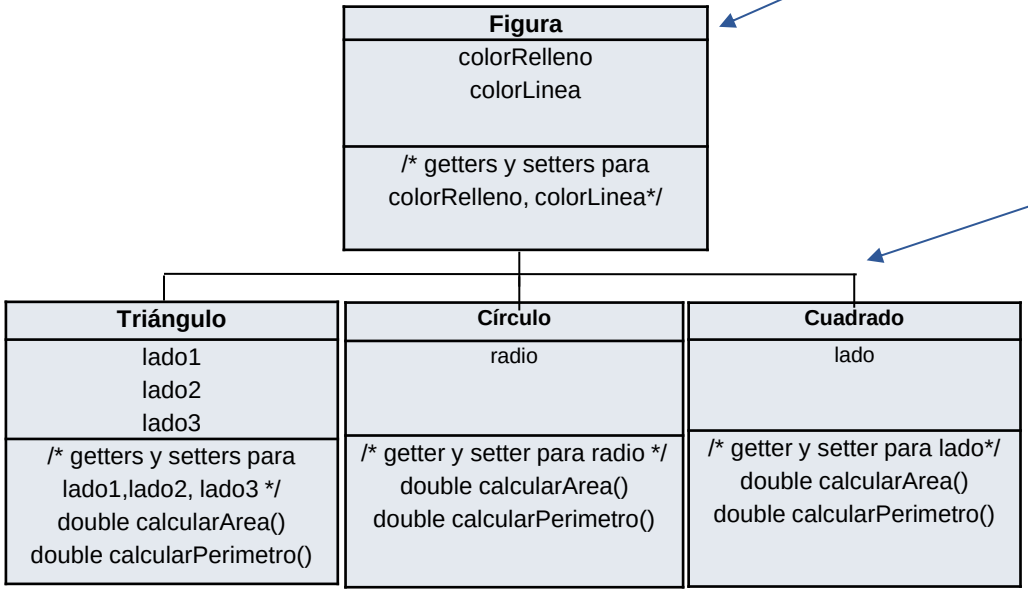


Figura es **superclase** de Triángulo, Círculo y Cuadrado

define atributos y comportamiento **común**

Triángulo, Círculo y Cuadrado son **subclases** de Figura.

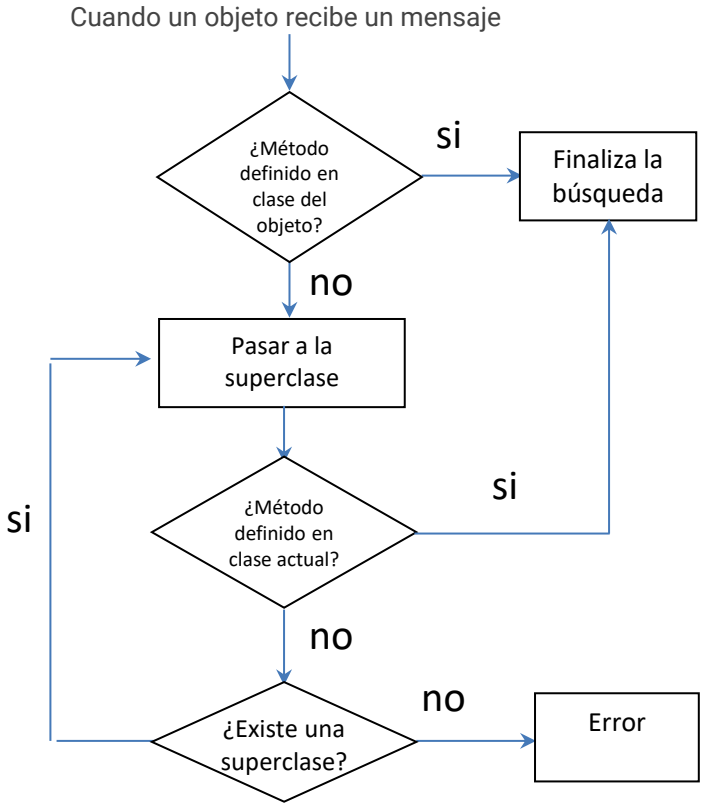
heredan atributos y métodos de Figura

definen atributos y métodos **propios**

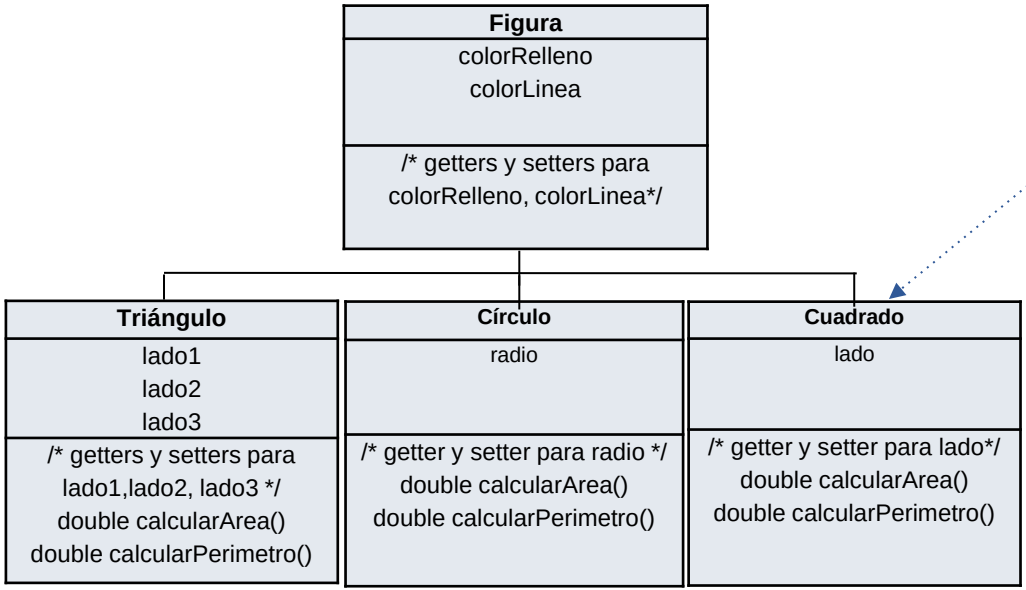
definen constructores.

deben *implementar calcularArea() y calcularPerimetro(). Cada uno deberá hacer cálculos diferentes de acuerdo a su clase →* ***POLIMORFISMO***

Búsqueda de método en la jerarquía de clases



Búsqueda de método en la jerarquía de clases



¿Qué mensajes le puedo enviar a un objeto **cuadrado**?

```
/*Ejemplo en el main*/
Cuadrado c = new Cuadrado(10,"rojo","negro");
System.out.println(c.calcularArea());
System.out.println(c.getColorRelleno());
System.out.println(c.getRadio()); ✗
```

Herencia en Java

¿Cómo defino una relación de herencia? Palabra clave *extends*.

```
public class ClaseA{
     /* Definir atributos propios */
     /* Definir constructores propios */
     /* Definir métodos propios */
}
```

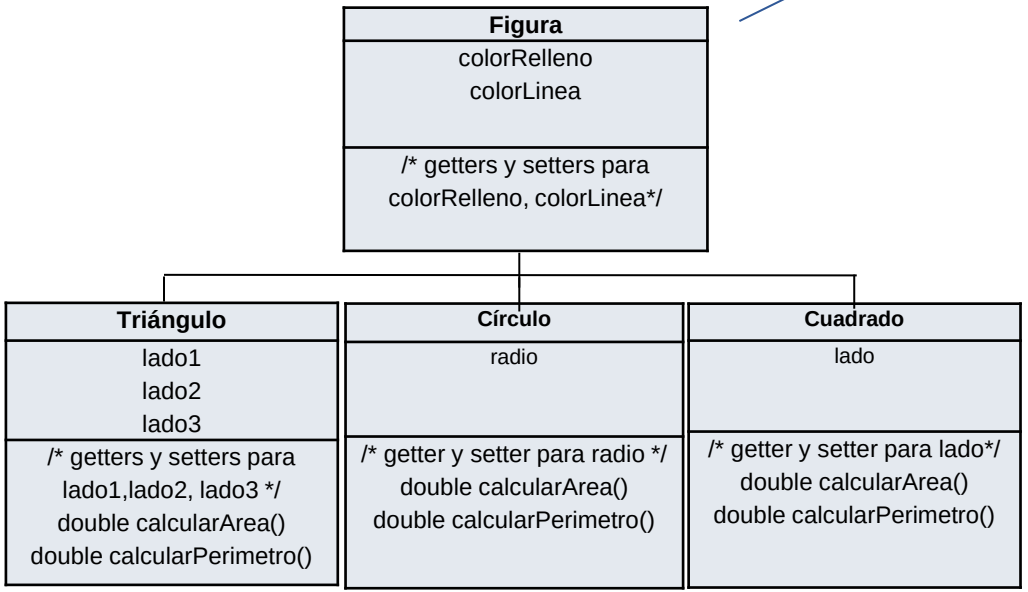
```
public class ClaseB extends ClaseA{
     /* Definir atributos propios */
     /* Definir constructores propios */
     /* Definir métodos propios */
}
```

La ClaseB (subclase de ClaseA) hereda los atributos y métodos de instancia declarados en la ClaseA

Aclaración: Los atributos declarados en una superclase como ***privados*** no son accesibles en sus subclases. Para *accederlos en* una subclase se deben usar los *getters y setters*.

Una subclase también puede declarar sus **propios** atributos, métodos y constructores

Ejemplo



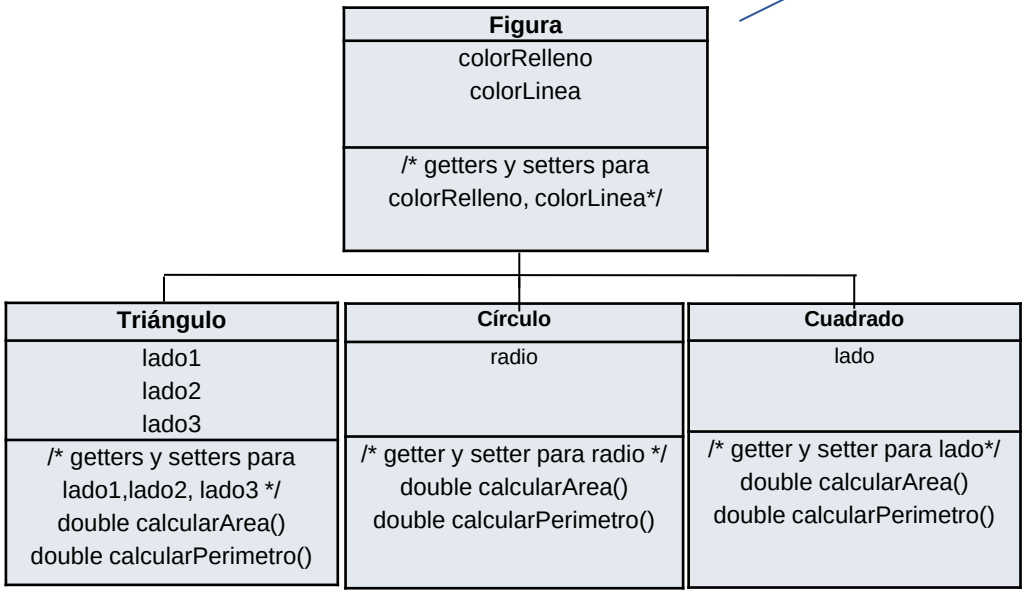

```
public class Figura{

    private String colorRelleno;
    private String colorLinea;

   /* Métodos getters y setters
   para colorRelleno y colorLinea*/
        …
}
```

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

    /* Métodos */
        …
}
```

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

    /* Métodos */
        …
}
```

...

Ejemplo



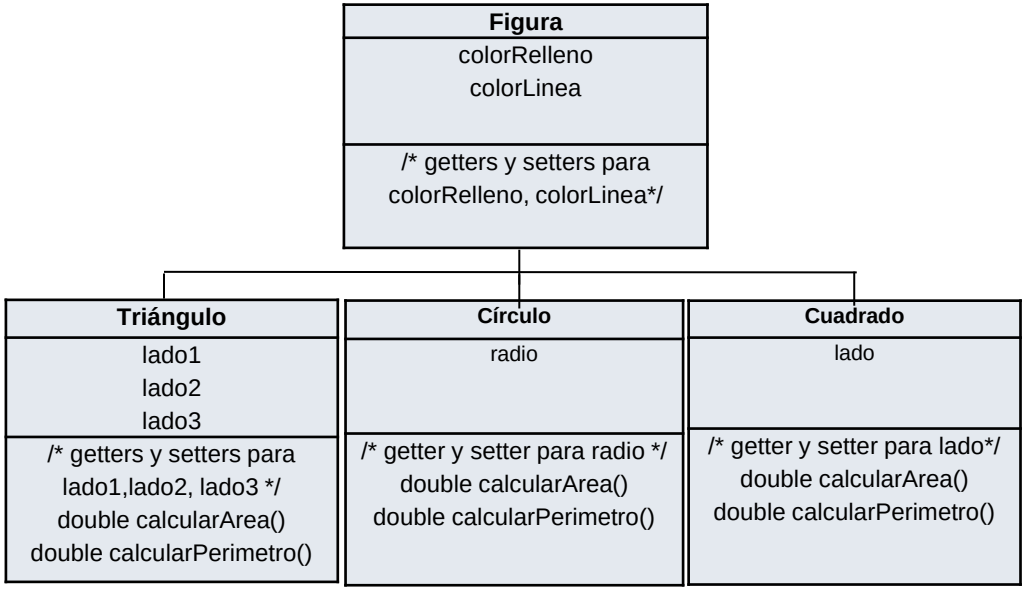

```
public class Figura{

    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 =….;
      }
}
```

colorRelleno es un atributo private de la superclase. Usar setter/getter para acceder al atributo

...

Clases y métodos abstractos



En algunas situaciones podemos encontrarnos con superclases de las cuales no nos resulta útil definir un nuevo objeto, sino que los nuevos objetos se van a instanciar desde sus subclases. Este sería el caso de la superclase Figura.

Lo que terminaremos definiendo serán Triángulos, Cuadrados, o Círculos.

La clase Figura sería una **clase abstracta**

Clases y métodos abstractos

¿Qué es un Clase abstracta?

Una clase abstracta es una clase que no puede ser instanciada (no se pueden crear objetos de esta clase). Define características y comportamiento común para un conjunto de clases (subclases). Puede definir ***métodos abstractos*** (sin implementación) que **deben** ser implementados por las subclases.

Los **métodos abstractos** son métodos para los cuales se define su encabezado pero no se los implementa. Las clases que hereden de una superclase abstracta deberán implementar sus métodos abstractos.

Ejemplo:

Para la clase *abstracta* Figura podemos declarar los métodos ***calcularArea*** *y* ***calcularPerimetro*** *como métodos abstractos.* Las clases Triángulo, Cuadrado y Círculo deberán implementar estos métodos.

Clases y métodos abstractos

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) */
}
```

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

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

Ejemplo:

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

MÉTODOS ABSTRACTOS

Subclase

La clase Cuadrado hereda de Figura sus atributos y métodos

y define un atributo propio

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

      /*Constructores*/
      public Cuadrado(double unLado,
                      String unColorR,
                      String unColorL){
             lado = unLado;
             colorRelleno = unColorR;
             colorLinea = unColorL;
       }

      /* Metodos *
      …

}
```

colorRelleno y colorLinea declarados “private” en Figura

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;

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

}
```

El objeto que está ejecutando (this) se envía un mensaje a sí mismo

setColorRelleno(unColorR)
ó **this.**setColorRelleno(unColorR)

Recordar ¿Cómo se busca el método a ejecutar en la jerarquía de clases?

Impleme

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;

      /*Constructores*/
      public Cuadrado(double unLado,
                      String unColorR,
                      String unColorL){
              lado = unLado;
              setColorRelleno(unColorR);
              setColorLinea(unColorL);
       }
   /* Metodos getLado y setLado  */
     …

   public double calcularPerime
         return lado*4;
   }

   public double calcularArea(
         return lado*lado;
   }
}
```

Impl

Otra opción: en vez de utilizar directamente la v.i. *lado* podemos hacer que el objeto se envíe un mensaje a si mismo para modificar/obtener dicho 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();
}
```

MÉTODOS ABSTRACTOS

Subclase

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

      /*Constructores*/
      public Cuadrado(double unLado,
                      String unColorR,
                      String unColorL){
              setLado(unLado);
              setColorRelleno(u
              setColorLinea(unC
       }
   /* Metodos getLado y setLado
     …

   public double calcularPerime
         return getLado()*4;
   }

   public double calcularArea()
         return getLado()*getLado();
   }
}
```

Implementa

Otra opción:
en vez de utilizar directamente la v.i. *lado* podemos hacer que el objeto se envíe un mensaje a si mismo para modificar/obtener dicho valor.
Buena práctica en POO

Ejemplo

Vamos a avanzar sobre el ejemplo de las figuras geométricas con 2 incisos.

a) Añadir la clase Círculo a la jerarquía de Figuras.

a) Añadir un método toString que retorne la representación en formato String de cada figura. Por ejemplo:

- Cuadrados: “CR: rojo CL: azul Lado: 3”
- Círculos: “CR: verde CL: negro Radio:4”

Subclases

La clase Cuadrado que habíamos creado

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

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

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




}
```

Código replicado

La nueva clase Círculo

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

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

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




}
```

Subclases

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

  /*Constructores*/
  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;
  }
}
```

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

      /*Constructores*/
      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;
       }
```

Código replicado

Código replicado

Veamos cómo mejorar esto ...

Ejemplo

Vamos a refactorizar el código, poniendo el código común en la superclase Figura y a “invocarlo” desde las subclases Círculo y Cuadrado.

Superclase

Agregamos a la superclase un **constructor** y un **toString**

```
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

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

```
public class Cuadrado
  private double lado

  /*Constructores*/
  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 es la referencia al objeto que está ejecutando **super.toString()** → El objeto se envía un mensaje a si mismo
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;

un
    }

    /* Metodos getLado y setLado  */

    /* Métodos calcularArea y calcularPerimetro */

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

Quiero añadir a la representación string el valor del área
¿en qué método toString lo hago?

Superclase

En toString() de Figura.
Evitamos repetir código en subclases.

¿Qué calcularArea() se ejecuta?

¿Cuándo se determina?

```
public abstract class Figura{
    private String colorRelleno, colorL
    public Figura(String unCR, String u
           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();
}
```

```
nds  Figura{

  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;
   }
}
```

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;
    }
```

Subclases

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

  /*Constructores*/
  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;
   }
```

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

  /*Constructores*/
  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;
   }
}
```

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

Resumen

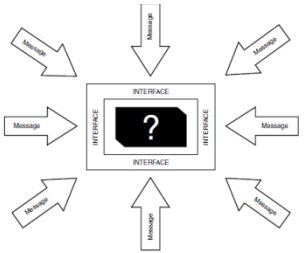
Hemos visto las bases de la POO.

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.



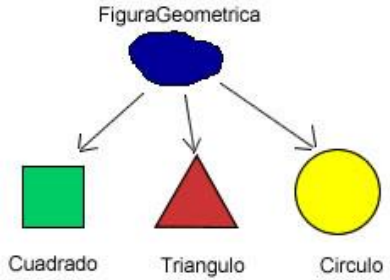
Resumen

Hemos visto las bases de la POO.

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.

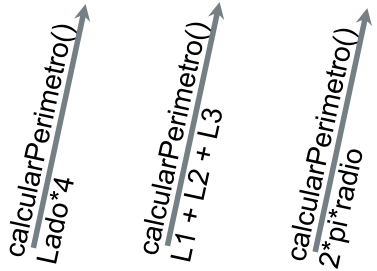


Resumen

Hemos visto las bases de la POO.

Polimorfismo

Objetos de clases distintas pueden responder a mensajes con nombre (selector) sintácticamente idénticos. Esto 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.

Resumen

Entre los beneficios de la POO, podemos mencionar producir SW 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 SW. 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.