TEMA: CONCEPTO DE HERENCIA (UTILIZANDO JAVA)

Taller de Programación.

Módulo: Programación Orientada a Objetos

Introducción

- 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

Inconvenientes hasta ahora. Herencia como solución.

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

Herencia. Ejemplo.

- Diagrama de clases.

Las clases forman una jerarquía.

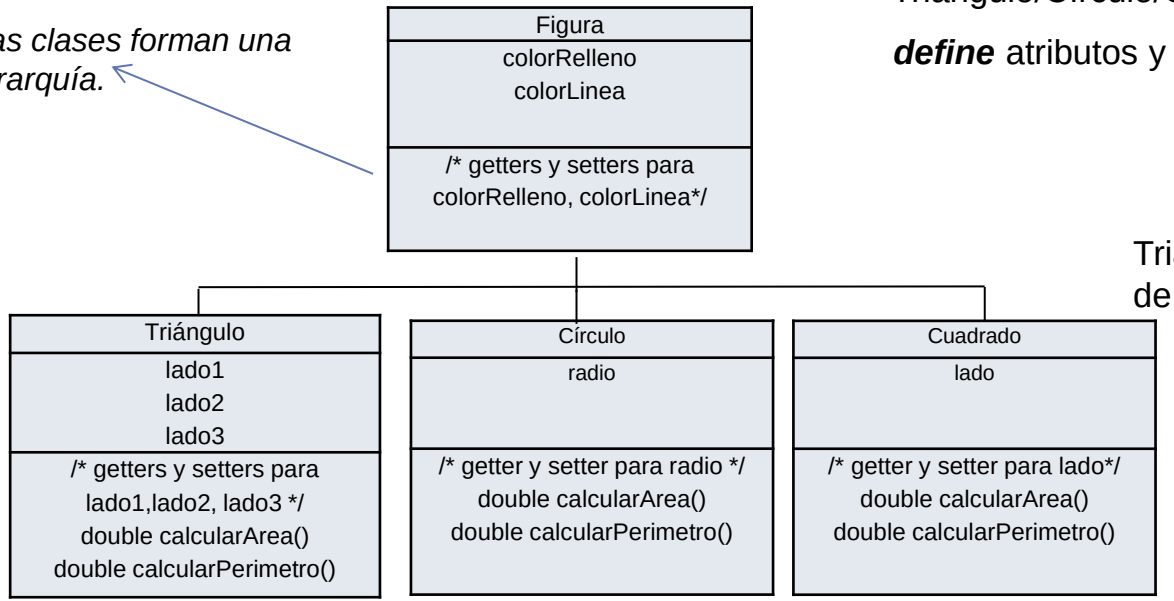


Figura es la **superclase** de Triángulo/Círculo/Cuadrado

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.

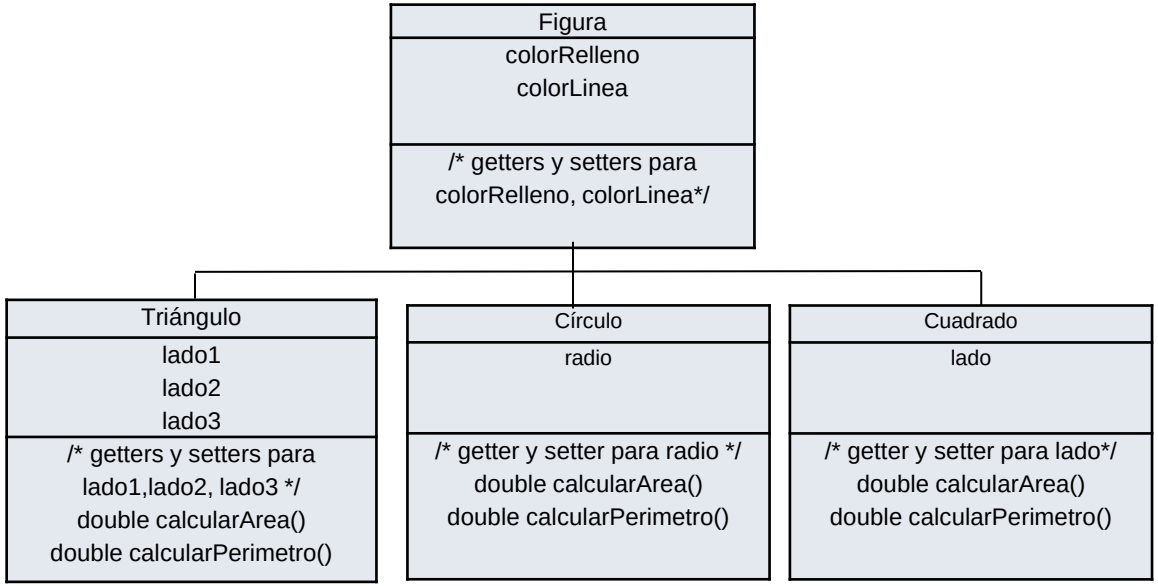
deben *implementar calcularArea() y calcularPerimetro() pero de manera diferente …* ***POLIMORFISMO***

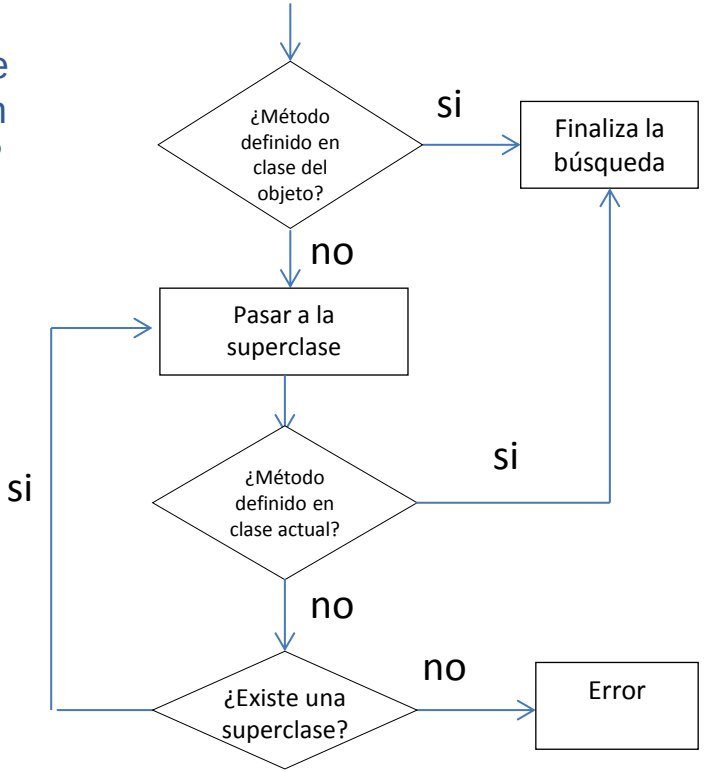
Búsqueda de método en la jerarquía de clases

- Ejemplo … en el main

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

¿Qué mensajes le puedo enviar a un objeto cuadrado?





Herencia en Java

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

```
public class Cuadrado extends  Figura{
        ….
}
```

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

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

- La **subclase** *hereda*
  - Atributos declarados en la **superclase**. Como son *privados* son accesibles sólo en métodos de la clase que los declaró. *En la subclase accederlos a través de getters y setters heredados. Ej: getColorRelleno() ó setColorRelleno(#)*
  - Métodos de instancia declarados en la **superclase**.
- La **subclase** *puede declarar*
  - Atributos / métodos / constructores propios.

Clases y métodos abstractos

- **Clase abstracta:** es una clase que no puede ser instanciada (no se pueden crear objetos). **Uso:** define características y comportamiento común para un conjunto de clases (subclases). Puede definir ***métodos abstractos*** (sin implementación) que *deben* ser implementados por las subclases.
  - Ejemplos:
    - La clase **Figura** es *abstracta*.
    - **Figura** puede declarar *métodos abstractos* ***calcularArea /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:
    - Sólo se pone el 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;

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

colorRelleno y colorLinea declarados "private" en Figura

setColorRelleno(unColorR)
setColorLinea(unColorL);

El objeto se envía un mensaje a si mismo

setColorRelleno(unColorR) **equivale a** **this.**setColorRelleno(unColorR)

this es el objeto que está ejecutando

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

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

Implementa

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

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.
¿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(unColorR);
            setColorLinea(unColorL);
      }

      /* Metodos getLado y setLado */
      ….

      public double calcularPerimetro(){
            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

- Añadir la clase Círculo a la jerarquía de Figuras.

- 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

```
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

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

```
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();
}
```

```
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;
    }
}
```

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

super es el 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.

```
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();
}
```

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

    /*Constructores*/
    p
    }

    /* 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. **Pero …**
¿en qué método toString?

```
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;
    }

    /* Métodos getters y setters */
    ….

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

```
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;
    }
}
```

En toString() de Figura.
Evitamos repetir código en subclases.
¿Qué calcularArea() se ejecuta?
¿Cuándo se determina?

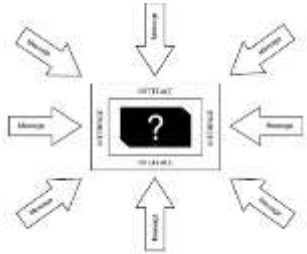
```
public abstract class Figura{
    private String colorRelleno, colorLinea;

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

    /* Métodos getters y setters */
    ….


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

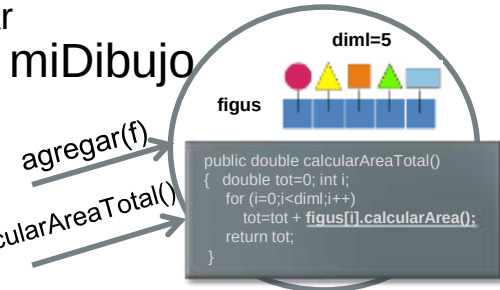
Polimorfismo: objetos de clases distintas responden al mismo mensaje de distinta forma.
Binding dinámico: se determina en tiempo de ejecución el método a ejecutar para responder a un mensaje.
Ventaja: Código genérico, reusable.

```
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;
    }
}
```

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.
  - 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.





Resumen

- Algunos beneficios de la POO: 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.