¿QUÉ ES POLIMORFISMO?

El significado de la palabra: poli=”varias” o “muchas”, así que polimorfismo=”muchas formas”

En POO hay mucha literatura al respecto y hay quien distingue entre varios tipos de polimorfismo.

Para muchos, la sobrecarga de métodos es un tipo de polimorfismo ya que el mismo método puede adoptar muchas formas diferentes según el tipo de objeto o valor que reciba por parámetro.

Otro tipo de polimorfismo en Java es el que se basa en la “ligadura dinámica”. Normalmente cuando se dice polimorfismo sin mayor especificación siempre nos referimos a este polimorfismo relacionado con la ligadura dinámica. En este contexto una referencia de tipo superclase, puede adoptar muchas “poli” formas ya que la misma referencia puede ejecutar diferentes tipos de métodos de objetos subclase. ¡Mejor un ejemplo!

Un ejemplo del polimorfismo

Observa el código de abajo. Una variable referencia de tipo Instrumento genera un comportamiento polimórfico. Podemos procesar todo el array en un bucle sin preocuparnos del tipo de instrumentos, evitando una maraña de if, y escribiendo un código limpio y elegante. Imagínate que la orquesta tiene 30 tipos de instrumentos y queremos oir el sonido de los 30: ¡polimorfismo en acción!

```
abstract class Instrumento{
   abstract void tocarNota();
}

class Flauta extends Instrumento{
   void tocarNota(){
      System.out.println("así suena una flauta");
   }
}
class Piano extends Instrumento{
   void tocarNota(){
      System.out.println("así suena un piano");
   }
}
class App{
   public static void main(String[] args) {
      Instrumento[] orquesta={new Flauta(), new Piano(), new Flauta()};

      for(Instrumento i: orquesta){
         System.out.print("Atención señoras y señores ...");
         i.tocarNota();
      }

   }
}
```

polimorfismo: la misma expresión *i.tocarNota()* puede adquirir muchas “poli” formas ya que automáticamente va a actuar sobre un tipo de instrumento y para cada tipo de instrumento el método tocarNota() puede tener instrucciones totalmente diferentes.

Sería bonito que el ejemplo anterior además sonara... Lo dejamos para más adelante. Observa que el método tocarNota() de la clase Instrumento es abstracto ya que no tiene

sentido reproducir ningún sonido si no sé de qué instrumento se trata, pero me interesa mucho que aunque sea abstracto la clase Instrumento tenga el método tocarNota() para lucrarme del mecanismo de ligadura dinámica.

Sin polimorfismo hay copiar y pegar en App …

```
class App{
    public static void main(String[] args) {
        Flauta f1= new Flauta();
        Piano p= new Piano();
        Flauta f2= new Flauta();

        System.out.print("Atención señoras y señores ...");
        f1.tocarNota();
        System.out.print("Atención señoras y señores ...");
        p.tocarNota();
        System.out.print("Atención señoras y señores ...");
        f2.tocarNota();


    }
}
```

Ventajas de polimorfismo: El código que tiene que escribir el programador de App es fácil. No tiene que estar contemplando el tipo de instrumento y por tanto se evitan if, código duplicado, etc.

USO DE REFERENCIAS A INTERFACES.

Esta explicación "quedaba pendiente" en el boletín de interfaces para justificar su necesidad e importancia.

Si no se pueden instanciar objetos de tipo interface ¿Para qué quiero una referencia de un tipo *interface*?: para lucrarme de la capacidad de polimorfismo de forma similar a como utilizamos una referencia a superclase para referenciar a objetos de subclases.

Ejercicio U5_B6C_E1:

Volvemos al ejercicio de Series. Usamos las mismas clases y creamos en el siguiente main()

```
interface Serie{
    int obtenerSiguiente(); //devuelve el siguiente número de la serie,
    void restablecer(); //reinicia
    void establecerInicio(int x); //establece el valor inicial
}
class MasDos implements Serie{
    int inicio;
    int val;
    MasDos(){
        inicio=0;
        val=0;
    }
    public int obtenerSiguiente(){
        val +=2;
        return val;
    }
    public void restablecer(){
        inicio=0;
        val=0;
    }
    public void establecerInicio(int x){
        inicio=x;
        val=x;
    }
}
class MasTres implements Serie{
    int inicio;
    int val;
    MasTres(){
        inicio=0;
        val=0;
    }
    public int obtenerSiguiente(){
        val +=3;
        return val;
```

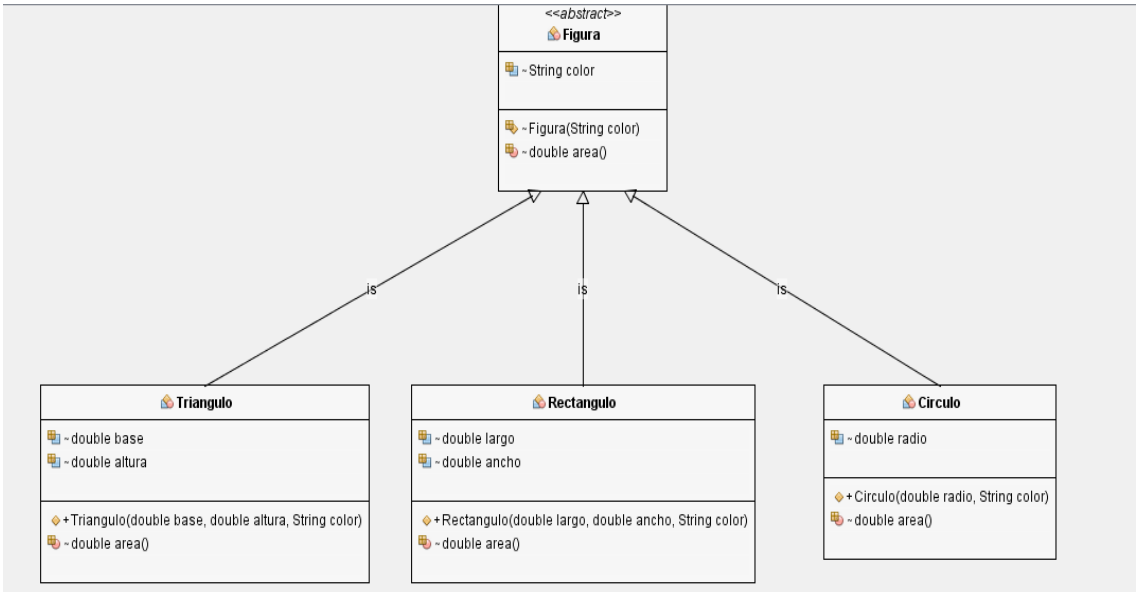
```
    }
    public void restablecer(){
        inicio=0;
        val=0;
    }
    public void establecerInicio(int x){
        inicio=x;
        val=x;
    }
}

class App{
    public static void main(String[] args) {
        MasDos serie1 = new MasDos();
        MasDos serie2= new MasDos();
        MasTres serie3 = new MasTres();
        MasTres serie4= new MasTres();
        serie2.establecerInicio(200);
        serie4.establecerInicio(300);

        System.out.print("\nSerie1: ");
        for(int i=0;i<5;i++)
            System.out.print(serie1.obtenerSiguiente()+" ");
        System.out.print("\nSerie2: ");
        for(int i=0;i<5;i++)
            System.out.print(serie2.obtenerSiguiente()+" ");
        System.out.print("\nSerie3: ");
        for(int i=0;i<5;i++)
            System.out.print(serie3.obtenerSiguiente()+" ");
        System.out.print("\nSerie4: ");
        for(int i=0;i<5;i++)
            System.out.print(serie4.obtenerSiguiente()+" ");
    }
}
```

Al escribir el código de App **me harté de copiar y pegar**, esto tiene que hacerme sospechar que algo va mal ya que tanto copiar y pegar va asociado a "código duplicado" de baja calidad. Imagina que tuviera que imprimir 100 series, y ahora imagina también que tuviera que hacer un cambio en el for. Tendré que hacer 100 cambios, lo que es una pérdida de tiempo y una fuente de errores.

SE PIDE: Generando exactamente la misma salida que con el código anterior, escribir un código mejorado que utilice un array de interfaces de tipo Serie para evitar tanto código duplicado.

Ejercicio U5_B6C_E2:

Escribe código para implantar la siguiente estructura. Luego en main de App crea 3 triángulos, 3 rectángulos y 3 círculos almacenando las figuras en un array. Recorre el array e imprime el área y color de todas las figuras, cambiando el color a "negro" en aquellas figuras cuya área se mayor que 4.0.



El operador *instance of*

En general "preferimos" trabajar con referencias de superclase para lucrarnos del mecanismo de polimorfismo, no obstante, puede darse el caso que necesite por alguna razón saber realmente a qué tipo de subclase está apuntando la referencia de la superclase. Para ello utilizamos el operador instanceof, por ejemplo si *f* es una referencia de tipo Figura

if(f instanceof Triangulo)

System.out.println("esta figura es un triángulo);

Ejercicio U5_B6C_E3: añade código a la solución del ejercicio anterior de forma que al final de la impresión de la lista me indique cuantas figuras eran triángulos.

El método getClass()

Todos los objetos heredan de Object el método getClass(). El método getClass() devuelve un objeto de tipo Class. Los objetos Class permite averiguar cosas sobre la estructura interna de un objeto como por ejemplo: saber el nombre de su clase, saber el nombre de sus campos, saber el nombre de sus métodos, ….
Observa el siguiente ejemplo, vamos a trabajar con un objeto tipo Class c vinculado a un objeto Persona p, de forma que a través del objeto c puedo averiguar cosas de la estructura de p. Ojo en este ejemplo con public, ya que los campos y métodos que queramos que devuelvan getFields() y getMethods() han de ser public:

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

class Persona{
   public String nombre;
   public int edad;

   public Persona(String nombre, int edad) {
      this.nombre = nombre;
      this.edad = edad;
   }

   public int rejuvenecer(){
      return edad-5;
   }
   public int envejecer(){
      return edad+5;
   }
}
public class App {
   public static void main(String[] args) throws ClassNotFoundException {
      Persona p = new Persona("juan",45);
      Class c=p.getClass();
      System.out.println("nombre de la clase del objeto referenciado por p: "+ c.getName());


      System.out.print("\ncampos(atributos) de la clase del objeto referenciado por p: ");
      for(Field f:c.getFields())
         System.out.print(f.getName()+" ");

      System.out.print("\nmétodos de la clase del objeto referenciado por p: ");
      for(Method m:c.getMethods())
         System.out.print(m.getName()+" ");

   }
}
```

Ejercicio U5_B6C_E4: resuelve el ejercicio U5_B6C_E3 ahora con getClass() en lugar de instance of