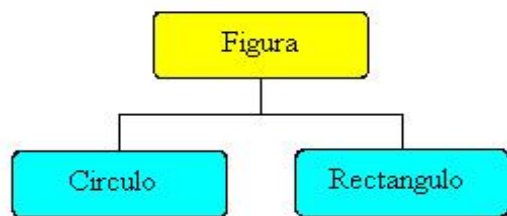


# La jerarquía de clases

Consideremos las figuras planas cerradas como el rectángulo, y el círculo. Tales figuras comparten características comunes como es la posición de la figura, de su centro, y el área de la figura, aunque el procedimiento para calcular dicha área sea completamente distinto. Podemos por tanto, diseñar una jerarquía de clases, tal que la clase base denominada *Figura*, tenga las características comunes y cada clase derivada las específicas. La relación jerárquica se muestra en la figura



La clase *Figura* es la que contiene las características comunes a dichas figuras concretas por tanto, no tiene forma ni tiene área. Esto lo expresamos declarando *Figura* como una clase abstracta, declarando la función miembro *area* **abstract**.

Las clases abstractas solamente se pueden usar como clases base para otras clases. No se pueden crear objetos pertenecientes a una clase abstracta. Sin embargo, se pueden declarar variables de dichas clases.

En el juego del ajedrez podemos definir una clase base denominada *Pieza*, con las características comunes a todas las piezas, como es su posición en el tablero, y derivar de ella las características específicas de cada pieza particular. Así pues, la clase *Pieza* será una clase abstracta con una función **abstract** denominada *mover*, y cada tipo de pieza definirá dicha función de acuerdo a las reglas de su movimiento sobre el tablero.

- La clase *Figura*

La definición de la clase abstracta *Figura*, contiene la posición *x* e *y* de la figura particular, de su centro, y la función *area*, que se va a definir en las clases derivadas para calcular el área de cada figura en particular.

```
public abstract class Figura {
    protected int x;
    protected int y;
    public Figura(int x, int y) {
        this.x=x;
        this.y=y;
    }
    public abstract double area();
}
```

- La clase *Rectángulo*

Las clases derivadas heredan los miembros dato  $x$  e  $y$  de la clase base, y definen la función *area*, declarada **abstract** en la clase base *Figura*, ya que cada figura particular tiene una fórmula distinta para calcular su área. Por ejemplo, la clase derivada *Rectangulo*, tiene como datos, aparte de su posición  $(x, y)$  en el plano, sus dimensiones, es decir, su anchura *ancho* y altura *alto*.

```
class Rectangulo extends Figura{
    protected double ancho, alto;
    public Rectangulo(int x, int y, double ancho, double alto){
        super(x,y);
        this.ancho=ancho;
        this.alto=alto;
    }
    public double area(){
        return ancho*alto;
    }
}
```

La primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base, para ello se emplea la palabra reservada **super**. El constructor de la clase derivada llama al constructor de la clase base y le pasa las coordenadas del punto  $x$  e  $y$ . Después inicializa sus miembros dato *ancho* y *alto*.

En la definición de la función *area*, se calcula el área del rectángulo como producto de la anchura por la altura, y se devuelve el resultado

- La clase *Circulo*

```
class Circulo extends Figura{
    protected double radio;
    public Circulo(int x, int y, double radio){
        super(x,y);
        this.radio=radio;
    }
    public double area(){
        return Math.PI*radio*radio;
    }
}
```

Como vemos, la primera sentencia en el constructor de la clase derivada es una llamada al constructor de la clase base empleando la palabra reservada **super**. Posteriormente, se inicializa el miembro dato *radio*, de la clase derivada *Circulo*.

En la definición de la función *area*, se calcula el área del círculo mediante la conocida fórmula  $\pi * r^2$ , o bien  $\pi * r * r$ . La constante [\*Math.PI\*](#) es una aproximación decimal del número irracional  $\pi$ .

## Uso de la jerarquía de clases

Creamos un objeto *c* de la clase *Circulo* situado en el punto (0, 0) y de 5.5 unidades de radio. Calculamos y mostramos el valor de su área.

```
Circulo c=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+c.area());
```

Creamos un objeto *r* de la clase *Rectangulo* situado en el punto (0, 0) y de dimensiones 5.5 de anchura y 2 unidades de largo. Calculamos y mostramos el valor de su área.

```
Rectangulo r=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+r.area());
```

Veamos ahora, una forma alternativa, guardamos el valor devuelto por **new** al crear objetos de las clases derivadas en una variable *f* del tipo *Figura* (clase base).

```
Figura f=new Circulo(0, 0, 5.5);
System.out.println("Area del círculo "+f.area());
f=new Rectangulo(0, 0, 5.5, 2.0);
System.out.println("Area del rectángulo "+f.area());
```

## Enlace dinámico

En el lenguaje C, los identificadores de la función están asociados siempre a direcciones físicas antes de la ejecución del programa, esto se conoce como enlace temprano o estático. Ahora bien, el lenguaje C++ y Java permiten decidir a que función llamar en tiempo de ejecución, esto se conoce como enlace tardío o dinámico. Vamos a ver un ejemplo de ello.

Podemos crear un array de la clase base *Figura* y guardar en sus elementos los valores devueltos por **new** al crear objetos de las clases derivadas.

```
Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 7.0);
fig[1]=new Circulo(0,0, 5.0);
fig[2]=new Circulo(0, 0, 7.0);
fig[3]=new Rectangulo(0,0, 4.0, 6.0);
```

Tenemos una declaración de la función *area* y varias definiciones distintas de dicha función. La sentencia

```
fig[i].area();
```

¿a qué función *area* llamará?. La respuesta será, según sea el índice *i*. Si *i* es cero, el primer elemento del array guarda una referencia a un objeto de la clase *Rectangulo*, luego llamará a la función miembro *area* de *Rectangulo*. Si *i* es uno, el segundo elemento del array guarda una referencia un objeto de la clase *Circulo*, luego llamará también a la función *area* de *Circulo*, y así sucesivamente. Pero podemos introducir el valor del índice *i*, a través del teclado, o seleccionando un control en un applet, en el

momento en el que se ejecuta el programa. Luego, la decisión sobre qué función *area* se va a llamar se retrasa hasta el tiempo de ejecución.

```
public class FiguraApp {
    public static void main(String[] args) {
//enlace temprano
        Circulo c=new Circulo(0, 0, 5.5);
        System.out.println("Area del círculo "+c.area());
        Rectangulo r=new Rectangulo(0, 0, 5.5, 2.0);
        System.out.println("Area del rectángulo "+r.area());
//enlace tardío
        Figura f=new Circulo(0, 0, 5.5);
        System.out.println("Area del círculo "+f.area());
        f=new Rectangulo(0, 0, 5.5, 2.0);
        System.out.println("Area del rectángulo "+f.area());
//array de objetos
        Figura fig[]=new Figura[4];
        fig[0]=new Rectangulo(0,0, 5.0, 7.0);
        fig[1]=new Circulo(0,0, 5.0);
        fig[2]=new Circulo(0, 0, 7.0);
        fig[3]=new Rectangulo(0,0, 4.0, 6.0);
        int i=2;
        System.out.println("Area "+fig[i].area());
    }
}
```

## El polimorfismo en acción (opcional)

Supongamos que deseamos saber la figura que tiene mayor área independientemente de su forma. Primero, programamos una función que halle el mayor de varios números reales positivos.

```
double valorMayor(double[] x){
    double mayor=0.0;
    for (int i=0; i<x.length; i++)
        if(x[i]>mayor){
            mayor=x[i];
        }
    return mayor;
}
```

Ahora, la llamada a la función *valorMayor*

```
double numeros[]={3.2, 3.0, 5.4, 1.2};
System.out.println("El valor mayor es "+valorMayor(numeros));
```

La función *figuraMayor* que compara el área de figuras planas es semejante a la función *valorMayor* anteriormente definida, se le pasa el array de objetos de la clase base *Figura*. La función devuelve una referencia al objeto cuya área es la mayor.

```
static Figura figuraMayor(Figura[] figuras){
    Figura mFigura=null;
    double areaMayor=0.0;
```

```

        for(int i=0; i<figuras.length; i++){
            if(figuras[i].area()>areaMayor){
                areaMayor=figuras[i].area();
                mFigura=figuras[i];
            }
        }
        return mFigura;
    }
}

```

La clave de la definición de la función está en las líneas

```

        if(figuras[i].area()>areaMayor){
            areaMayor=figuras[i].area();
            mFigura=figuras[i];
        }
    }
}

```

En la primera línea, se llama a la versión correcta de la función *area* dependiendo de la referencia al tipo de objeto que guarda el elemento *figuras[i]* del array. En *areaMayor* se guarda el valor mayor de las áreas calculadas, y en *mFigura*, la figura cuya área es la mayor.

La principal ventaja de la definición de esta función estriba en que la función *figuraMayor* está definida en términos de variable *figuras* de la clase base *Figura*, por tanto, trabaja no solamente para una colección de círculos y rectángulos, sino también para cualquier otra figura derivada de la clase base *Figura*. Así si se deriva *Triangulo* de *Figura*, y se añade a la jerarquía de clases, la función *figuraMayor* podrá manejar objetos de dicha clase, sin modificar para nada el código de la misma.

Veamos ahora la llamada a la función *figuraMayor*

```

Figura[] fig=new Figura[4];
fig[0]=new Rectangulo(0,0, 5.0, 7.0);
fig[1]=new Circulo(0,0, 5.0);
fig[2]=new Circulo(0, 0, 7.0);
fig[3]=new Rectangulo(0,0, 4.0, 6.0);
Figura fMayor=figuraMayor(fig);
System.out.println("El área mayor es "+fMayor.area());

```

Pasamos el array *fig* a la función *figuraMayor*, el valor que retorna lo guardamos en *fMayor*. Para conocer el valor del área, desde *fMayor* se llamará a la función miembro *area*. Se llamará a la versión correcta dependiendo de la referencia al tipo de objeto que guarde por *fMayor*. Si *fMayor* guarda una referencia a un objeto de la clase *Circulo*, llamará a la función *area* definida en dicha clase. Si *fMayor* guarda una referencia a un objeto de la clase *Rectangulo*, llamará a la función *area* definida en dicha clase, y así sucesivamente.

La combinación de herencia y enlace dinámico se denomina polimorfismo. El polimorfismo es, por tanto, la técnica que permite pasar un objeto de una clase derivada a funciones que conocen el objeto solamente por su clase base.

## El operador *instanceof*(opcional)

El operador **instanceof** tiene dos operandos: un objeto en el lado izquierdo y una clase en el lado derecho. Esta expresión devuelve **true** o **false** dependiendo de que el objeto situado a la izquierda sea o no una instancia de la clase situada a la derecha o de alguna de sus clases derivadas.

Por ejemplo.

```
Rectangulo rect=new Rectangulo(0, 0, 5.0, 2.0);
rect instanceof String           //false
rect instanceof Rectangulo       //true
```

El objeto *rect* de la clase *Rectangulo* no es un objeto de la clase *String*. El objeto *rect* si es un objeto de la clase *Rectangulo*.

Veamos la relación entre *rect* y las clases de la jerarquía

```
rect instanceof Figura           //true
rect instanceof Cuadrado         //false
```

*rect* es un objeto de la clase base *Figura* pero no es un objeto de la clase derivada *Cuadrado*