Guía práctica de estudio 08: Polimorfismo

Elaborado por:
M.C. M. Angélica Nakayama C.
Ing. Jorge A. Solano Gálvez

Autorizado por:
M.C. Alejandro Velázquez Mena

Guía práctica de estudio 08: Polimorfismo

Objetivo:

Implementar el concepto de polimorfismo en un lenguaje de programación orientado a objetos.

Actividades:

- A partir de una jerarquía de clases, implementar referencias que se comporten como diferentes objetos.

Introducción

El término **polimorfismo** es constantemente referido como uno de los pilares de la programación orientada a objetos (junto con la Abstracción, el Encapsulamiento y la Herencia).

El término **polimorfismo** es una palabra de origen griego que significa **muchas formas.** En la programación orientada a objetos se refiere a la **propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos**.

El **polimorfismo** consiste en conseguir que un objeto de una clase se comporte como un objeto de cualquiera de sus subclases. Se puede aplicar tanto a métodos como a tipos de datos. Los métodos pueden evaluar y ser aplicados a diferentes tipos de datos de manera indistinta. Los tipos polimórficos son tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

El polimorfismo se puede clasificar en dos grandes grupos:

- Polimorfismo dinámico (o paramétrico): es aquel en el que **no se especifica el tipo de datos sobre el que se trabaja** y, por ende, se puede recibir utilizar todo tipo de datos compatible. Este tipo de polimorfismo también se conoce como programación genérica.
- Polimorfismo estático (o ad hoc): es aquel en el que **los tipos de datos que se pueden utilizar deben ser especificados** de manera explícita antes de ser utilizados.

NOTA: En esta guía se tomará como caso de estudio el lenguaje de programación JAVA, sin embargo, queda a criterio del profesor el uso de éste u otro lenguaje orientado a objetos.

Polimorfismo

Polimorfismo se refiere a la habilidad de **tener diferentes formas**. Como se vio en la práctica de herencia, el término **IS-A** se refiere a la pertenencia de un objeto con un tipo, es decir, si se crea una instancia de tipo A, se dice que el objeto creado es un A.

En Java, cualquier objeto que pueda comportarse como más de un **IS-A** (es un) puede ser considerado **polimórfico**. Por lo tanto, todos los objetos en Java pueden ser considerados polimórficos porque todos se pueden comportar como objetos de su propio tipo y como objetos de la clase *Object*.

Por otro lado, debido a que la única manera de acceder a un objeto es a través de su referencia, existen algunos puntos clave que se deben recordar sobre las mismas:

- Una referencia puede ser solo de un tipo y, una vez declarado, el tipo no puede ser cambiado.
- Una referencia es una variable, por lo tanto, ésta puede ser reasignada a otros objetos (a menos que la referencia sea declarada como *final*).
- El tipo de una referencia determina los métodos que pueden ser invocados del objeto al que referencia, es decir, solo se pueden ejecutar los métodos definidos en el tipo de la referencia.
- A una referencia se le puede asignar cualquier objeto que sea del mismo tipo con el que fue declarada la referencia o de algún subtipo (**Polimorfismo**).

Dada la siguiente jerarquía de clases:

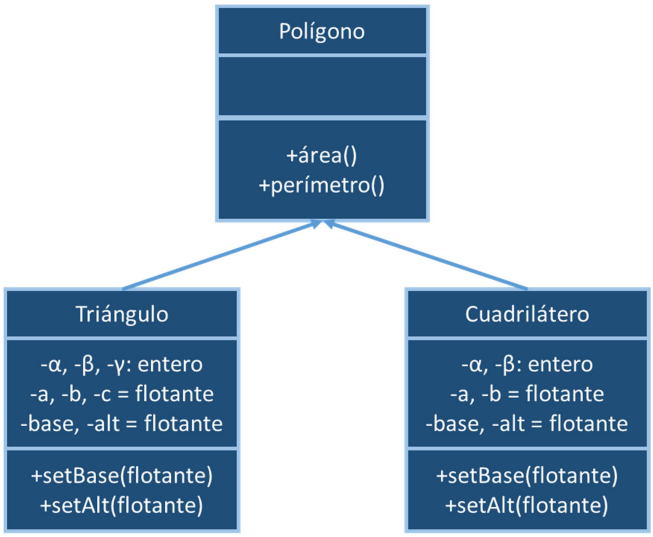

Figura 1. Jerarquía de clases

Ejemplo:

```
public class Poligono {

    public double area(){
        return 0d;
    }

    public double perimetro(){
        return 0d;
    }

    public String toString(){
        return "Polígono";
    }

}
```

```
public class Cuadrilatero extends Poligono {

    private int alfa, beta;
    private float a, b;
    private float base, altura;

    public void setBase(){}

    public void setAltura(){}

    public String toString(){
        return "Cuadrilátero";
    }
}
```

```
public class Triangulo extends Poligono {

    private int alfa, beta, gama;
    private float a, b, c;
    private float base, altura;

    public void setBase(){}

    public void setAltura(){}

    public String toString(){
        return "Triángulo";
    }
}
```

```
public class PruebaFigurasGeometricas{

    public static void main (String [] args){
                Poligono poligono = new Poligono();
                // Polígono puede comportarse como Objeto
                Object objeto = poligono;
                System.out.println(objeto);

                // Una referencia puede ser reasignada a otros objetos
                poligono = new Triangulo();
                System.out.println(poligono);
                poligono = new Cuadrilatero();
                System.out.println(poligono);

                // Solo se pueden ejecutar los métodos que están definidos
                // en la referencia, sin embargo, se ejecutarán como están
                // implementados en la instancia.
                // El método toString se puede ejecutar porque está definido
                // en Polígono, sin embargo, se va a ejecutar como está
                // implementado en la instancia (Cuadrilátero en este caso).
                System.out.println(poligono);
                // El método setBase no está definido en Polígono, por lo tanto
                // la siguiente instrucción marcaría un error:
                // poligono.setBase(5.5);
        }

}
```

Así mismo, un método puede recibir cualquier tipo de dato como parámetro. Cuando el parámetro definido es una referencia a una clase, el método es capaz de recibir un objeto de ese tipo o de **cualquier subtipo** de esa clase. A esto se le conoce también como **polimorfismo**, pero en métodos. Como ya se mencionó en la práctica anterior, la palabra reservada *instanceof* permite identificar el tipo de objeto enviado, es decir, no revisa la referencia si no el objeto en sí.

Código (Métodos polimórficos)

```
public class MetodoPolimorfico {
        public static void main (String [] arg){
                Poligono poligono = new Poligono();
                getPoligono(poligono);
                poligono = new Triangulo();
                getPoligono(poligono);
                poligono = new Cuadrilatero();
                getPoligono(poligono);
        }

        public static void getPoligono(Poligono p){
                if (p instanceof Triangulo){
                        System.out.println("El objeto es un triángulo.");
                } else {
                        if (p instanceof Cuadrilatero){
                                System.out.println("El objeto es un cuadrilátero.");
                        } else {
                                System.out.println("El objeto es un polígono.");
                        }
                }
        }

}
```

Clases abstractas

Una clase **abstracta** es una clase de la que no se pueden crear objetos, debido a que define la existencia de métodos, pero no su implementación.

Las clases **abstractas** sirven como modelo para la creación de clases derivadas. Algunas características de éstas son:

- Pueden contener métodos abstractos y métodos concretos.
- Pueden contener atributos.
- Pueden heredar de otras clases.

Para declarar una clase **abstracta** en Java solo es necesario anteponer la palabra reservada **abstract** antes de palabra reservada *class*, es decir:

```
public abstract class Poligono {
        // Métodos abstractos o concretos
}
```

Una clase **abstracta** puede tener métodos declarados **abstractos**, en cuyo caso no se da definición del método (no se implementa). **Si una clase tiene algún método abstracto es obligatorio que la clase sea abstracta.** La sintaxis para declarar un método abstracto es:

```
public abstract double perimetro( );
```

La clase que hereda de una clase **abstracta** debe implementar los métodos abstractos definidos en la clase base:

```
public class Triangulo extends Poligono {
        public double perimetro() {
                // bloque de código para obtener el perímetro
        }
}
```

Es posible crear referencias de una clase abstracta:

```
Poligono figura;
```

Sin embargo, una clase **abstracta** **no se puede instanciar**, es decir, no se pueden crear objetos de una clase abstracta, la siguiente línea de código generaría un error en tiempo de compilación:

```
Poligono figura = new Poligono();
```

El que una clase **abstracta** no se pueda instanciar es coherente dado que este tipo de clases no tiene completa su implementación y encaja bien con la idea de que un ente abstracto no puede materializarse. Sin embargo, una referencia abstracta sí puede contener un objeto concreto, es decir:

Poligono figura = new Triangulo();

Ejemplo:

```java
public abstract class Poligono {

    public abstract double area();

    public abstract double perimetro();

    public String toString(){
        return "Polígono";
    }

}
```

```java
public class Triangulo extends Poligono {

    private int alfa, beta, gama;

    private float a, b, c;

    private float base, altura;

    // La clase Triangulo está obligada a sobrescribir los métodos
    // abstractos que definió la clase abstracta de la que hereda.

    public double area(){
        return (base*altura)/2;
    }

    public double perimetro(){
        return a*b*c;
    }

    public String toString(){
        return "Triángulo";
    }

}
```

```java
public class Cuadrilatero extends Poligono {

    private int alfa, beta;

    private float a, b;

    private float base, altura;

    // La clase Cuadrilatero  está obligada a sobrescribir los métodos
    // abstractos que definió la clase abstracta de la que hereda.

    public double area(){
        return base*altura;
    }

    public double perimetro(){
        return 2*a*b;
    }

    public String toString(){
        return "Cuadrilátero";
    }

}
```

```java
public class PruebaFigurasGeometricas{

    public static void main (String [] args){
        // No se pueden crear objetos de clases abstractas
        //Poligono poligono = new Poligono();
        Poligono poligono;

        // Una referencia de una clase abstracta sí
        // puede almacenar un objeto de una clase concreta
        poligono = new Triangulo();
        System.out.println(poligono);
        poligono = new Cuadrilatero();
        System.out.println(poligono);
    }

}
```

Interfaces

El concepto de **Interfaces** lleva un paso más adelante la idea de las clases abstractas. Una **interfaz** es una clase abstracta pura, es decir, una clase donde **todos los métodos son abstractos** (no se implementa ninguno). Una interfaz es un contrato sobre *qué* puede hacer la clase, sin decir *cómo* lo va a hacer. Debido a que es una clase 100% abstracta, **no es posible crear instancias de una interfaz**.

Este tipo de diseños permiten establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno), pero no bloques de código.

Para crear una interfaz en Java, se utiliza la palabra reservada **interface** en lugar de *class*. La interfaz puede definirse pública o sin modificador de acceso y tiene el mismo significado que para las clases, su sintaxis es la siguiente:

```
public interface NombreInterfaz {
        tipoRetorno nombreMetodo([Parametros]);
}
```

Todos los **métodos** que declara una interfaz son siempre **públicos y abstractos**. Así mismo, una interfaz puede contener **atributos**, pero estos son siempre **públicos, estáticos y finales.**

Las **interfaces** pueden ser **implementadas** por cualquier clase. **La clase que implementa una interfaz está obligada a definir (implementar) los métodos que la interfaz declaró** y, en ese sentido, adquieren una conducta o modo de funcionamiento particular (contrato).

La sintaxis para implementar una interfaz es la siguiente:

```
public class MiClase implements NombreInterfaz
```

El compilador se encargará de verificar que la clase efectivamente declare e implemente todos los métodos de la interfaz.

Ejemplo:

```java
public interface InstrumentoMusical {

    // Por defecto, todos los métodos definidos dentro de
    // una interfaz son públicos (public) y abstractos (abstract)

    void tocar();

    void afinar();

    String tipoInstrumento();
}


public class InstrumentoViento extends Object implements InstrumentoMusical {

    // Por defecto, todos los métodos de la interfaz son públicos,
    // por lo tanto, así se deben implementar

    public void tocar() {
        System.out.println("Tocando instrumento de viento.");
    }

    public void afinar() {
        System.out.println("Afinando instrumento de viento.");
    }

    public String tipoInstrumento() {
        return "Instrumento de viento";
    }

}


public class Flauta extends InstrumentoViento {

    // La clase Flauta puede sobrescribir algún método

    public String tipoInstrumento() {
        return "Flauta";
    }

}


public class PruebaInstrumento {

    public static void main (String [] args){

        //Se puede crear una referencia de una interfaz.
        InstrumentoMusical instrumento;

        // Pero no es posible crear una instancia de una interfaz.
        // instrumento = new InstrumentoMusical();

        // Una referencia a interfaz puede ser asignada cualquier
        // objeto que la implemente.

        instrumento = new Flauta();
        instrumento.tocar();
        System.out.println(instrumento.tipoInstrumento());
    }

}
```

Implementación múltiple

Una clase puede **implementar** cualquier cantidad de **interfaces**, la única restricción es que, dentro del cuerpo de la clase, se deben **implementar todos los métodos** de las interfaces que se implementen. La sintaxis es la siguiente la siguiente:

```
public class NombreClase implements Interfaz1, Interfaz2, …, InterfazN {
        // Implementar métodos de la Interfaz1
        // Implementar métodos de la interfaz 2
        // …
        // Implementar métodos de la interfaz N
}
```

El orden de la implementación de las interfaces y el orden en el que se implementan los métodos de las interfaces dentro de la clase no es importante.

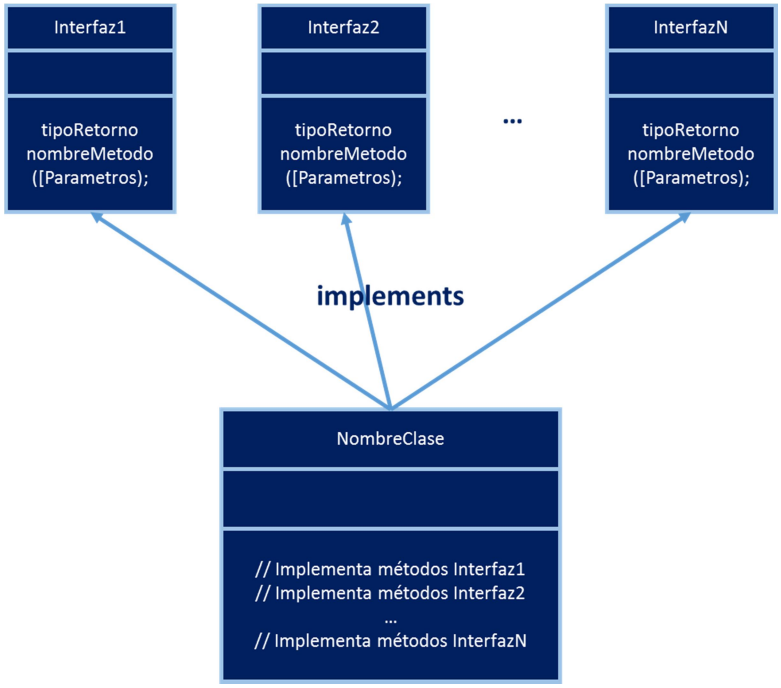

Figura 2. Implementación múltiple de interfaces en una clase.

Herencia múltiple entre interfaces

Las interfaces pueden heredar de otras interfaces y, a diferencia de las clases, **una interfaz puede heredar de una o más interfaces** (herencia múltiple), utilizando la siguiente sintaxis:

```
public interface HeredaInterfaz extends Interfaz1, Interfaz2, …, InterfazN {
        tipoRetorno nombreMetodo([Parametros]);
}
```

En este caso, se aplican las mismas reglas que en la herencia, es decir, la interfaz que hereda de otras interfaces posee todos los métodos definidos en ellas. Por otro lado, la clase que implemente esta interfaz (la interfaz que hereda de otras interfaces) debe definir los métodos de todas las interfaces.

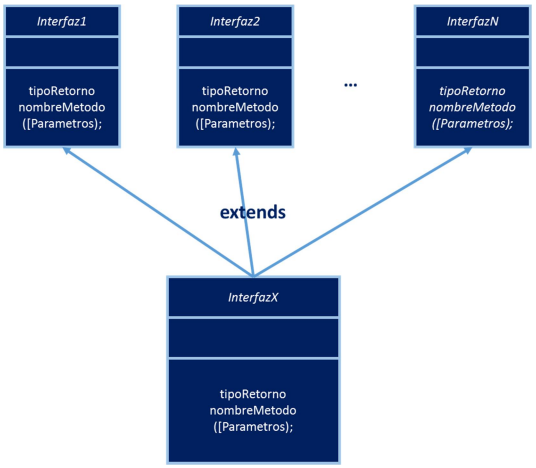

Figura 3. Herencia múltiple entre interfaces.

Atributos en las interfaces

Dado que, por definición, todos los datos miembros (atributos) que se definen en una interfaz son públicos, estáticos y finales y dado que las interfaces no pueden instanciarse, resultan una buena herramienta para implantar grupos de constantes que pueden ser llamados sin crear objetos.

Ejemplo:

```
public interface Meses {

    int UNO = 1, DOS = 2, TRES = 3, CUATRO = 4, CINCO = 5, SEIS = 6;

    int SIETE = 7, OCHO = 8, NUEVE = 9, DIEZ = 10, ONCE = 11, DOCE = 12;

    String [] NOMBRES_MESES = {"", "Enero", "Febrero", "Marzo",
            "Abril", "Mayo", "Junio", "Agosto", "Septiembre", "Octubre",
            "Noviembre", "Diciembre"};
}
```

```
public class PruebaMeses {

    public static void main (String [] args){

        // Se puede acceder a las variables de la interfaz sin crear instancias
        System.out.println("El mes " + Meses.DOS + " corresponde a:");
        System.out.println(Meses.NOMBRES_MESES[Meses.DOS]);
    }

}
```

Bibliografía

Barnes David, Kölling Michael
Programación Orientada a Objetos con Java.
Tercera Edición.
Madrid
Pearson Educación, 2007

Deitel Paul, Deitel Harvey.
Como programar en Java
Septima Edición.
México
Pearson Educación, 2008

Martín, Antonio
Programador Certificado Java 2.
Segunda Edición.
México
Alfaomega Grupo Editor, 2008

Dean John, Dean Raymond.
Introducción a la programación con Java
Primera Edición.
México
Mc Graw Hill, 2009