	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	143/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Guía práctica de estudio 11: Polimorfismo (2da parte)

---





---

***Elaborado por:***

M.C. M. Angélica Nakayama C.  
Ing. Jorge A. Solano Gálvez

***Autorizado por:***

M.C. Alejandro Velázquez Mena

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	144/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Guía práctica de estudio 11: Polimorfismo (2da parte)

### Objetivo:

Implementar el concepto de polimorfismo desde un nivel muy general y hasta un nivel muy específico en un lenguaje de programación orientado a objetos.

### Actividades:


- Crear una jerarquía de clases desde un nivel muy abstracto hasta un nivel muy específico.
- Crear una referencia de la clase base más general.
- Crear instancias de diferentes clases derivadas.

### Introducción

Una jerarquía de clases determina la relación que existe entre clases, determinando diversos niveles dentro de la estructura. La relación que existe entre las clases se puede modelar utilizando una estructura de datos no lineal tipo grafo, separados por niveles, donde los nodos en el nivel más alto representan los elementos más generales o abstractos y los nodos en el nivel más bajo representan los elementos más específicos.

Las clases más generales (las del nivel más alto) tienen por objeto determinar un comportamiento estándar para toda la jerarquía de clases. Por otro lado, las clases más específicas (las del nivel más bajo) determinan el último eslabón de la jerarquía, lo que implica que la jerarquía ya no puede hacerse más especializada y, por ende, ha llegado a su fin.

Cada lenguaje de programación modela de manera diferente el elemento más específico y el elemento más general, pero estos conceptos son soportados por la mayoría de los lenguajes.

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	145/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

**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.

## Clases abstractas

Las clases abstractas sirven como modelo para la creación de otras clases (clases derivadas), es decir, definen las clases más generales de la jerarquía de clases.

Una clase abstracta permite definir la existencia de métodos, pero no su implementación, es decir, permite identificar métodos comunes para la toda jerarquía de clase sin especificar cómo se deben realizar las acciones de dichos métodos.

Las características principales de las clases abstractas son:


- Pueden definir métodos abstractos y métodos concretos (con o sin implementación).
- Pueden definir atributos.
- Pueden heredar de otras clases.

Una clase abstracta se declara simplemente con el modificador `abstract`, adicionalmente, se puede usar un modificador de acceso. La sintaxis de una clase abstracta es la siguiente:

```
[modificadores] abstract class NombreClase {
    [modificadores] abstract float metodo();
}
```

Un método abstracto se declara de utilizando el modificador `abstract`, pero no se define una implementación, es decir, sólo se define la firma de la función que deben respetar todas las clases que sobrescriban dicho método.

La clase que herede de una clase abstracta debe declarar e implementar el comportamiento de los métodos abstractos (a menos que ésta, la que hereda, sea abstracta). De no declararse (implícitamente), el compilador generará un error indicando que no se han implementado

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	146/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

todos los métodos abstractos y que, o bien se implementen, o bien se declare la clase como abstracta.

Dada la siguiente jerarquía de clases:

Polígono

|


Triángulo

La implementación de la clase Polígono define la base de todas las clases que deriven de ella, porque todas heredan de ella. Por lo tanto, es un buen lugar para definir la existencia de métodos que se deseen tengan todas las clases derivadas, en este caso, área y perímetro.

```
public abstract class Poligono {
    public abstract double area();

    public abstract double perimetro();

    @Override
    public String toString(){
        return "Polígono";
    }
}
```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	147/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

public class Triangulo extends Poligono {
    protected float base, altura, ladoA, ladoB, ladoC;

    public Triangulo() {}

    public Triangulo(float base, float altura,
        float ladoA, float ladoB, float ladoC) {
        this.base = base;
        this.altura = altura;
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;
    }

    public Boolean tieneLadosParalelos(){
        return false;
    }

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

    @Override
    public double perimetro() {
        return ladoA + ladoB + ladoC;
    }

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


```

Cuando se crean objetos, la parte izquierda de la ecuación se conoce como referencia. 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();*

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	148/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

El que una clase **abstracta** no se pueda instanciar es coherente con la definición, 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:

```
public abstract class Poligono {
    public abstract double area();

    public abstract double perimetro();

    @Override
    public String toString(){
        return "Polígono";
    }
}
```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	149/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

public class Triangulo extends Poligono {
    protected float base, altura, ladoA, ladoB, ladoC;

    public Triangulo() {}

    public Triangulo(float base, float altura,
        float ladoA, float ladoB, float ladoC) {
        this.base = base;
        this.altura = altura;
        this.ladoA = ladoA;
        this.ladoB = ladoB;
        this.ladoC = ladoC;
    }

    public Boolean tieneLadosParalelos(){
        return false;
    }

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

    @Override
    public double perimetro() {
        return ladoA + ladoB + ladoC;
    }

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

```

```

public class Cuadrilatero extends Poligono {
    protected float ladoA, ladoB;
    protected float base, altura;

    public Cuadrilatero() {}

    public Cuadrilatero(float base, float altura,
        float ladoA, float ladoB) {
        this.base = base;
        this.altura = altura;
        this.ladoA = ladoA;
        this.ladoB = ladoB;
    }

    public Boolean tieneLadosParalelos(){
        return true;
    }

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

    @Override
    public double perimetro() {
        return ladoA + ladoB;
    }

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

```

```


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

        // Sí se pueden crear referencias de clases abstractas
        Poligono poligono;

        // Las referencias abstractas pueden ser asignadas
        // con objetos concretos mientras estén en la misma
        // jerarquía de clases
        poligono = new Triangulo();
        System.out.println(poligono.area());
        poligono = new Cuadrilatero(2, 4, 2, 4);
        System.out.println(poligono.area());
    }
}

```



	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	150/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Clase final

Una clase que representa el final de la jerarquía de clases, es decir, es un elemento suficientemente específico que no necesita modificarse. Cuando esto ocurre el funcionamiento de la clase se puede proteger evitando ser modificada por otras clases.


En java para evitar que una clase pueda ser modificada se agrega la palabra reservada final a la declaración de la misma, consiguiendo que ésta no pueda ser heredada y, por ende, modificada.

```
public final class TrianguloIsosceles extends Triangulo {
    @Override
    public double perimetro(){
        return 3 * ladoA;
    }
}
```

cannot inherit from final TrianguloIsosceles  
----  
(Alt-Enter shows hints)

```
class HeredaTrianguloIsosceles extends TrianguloIsosceles {  
}
```



	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	151/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## Interfaces

De igual manera, debido a que una interfaz es una clase abstracta pura (todos los métodos son abstractos).

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

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

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

Las interfaces son implementadas por las clases. Del mismo modo que con las clases abstractas, la clase que implemente una o varias interfaces, está obligada a darle un comportamiento a los métodos que defina la(s) interfaz(ces). La sintaxis para que una clase pueda implementar una interfaz es la siguiente:


```
class NombreClase implements NombreInterfaz {
    // definición de los métodos que define la interfaz
}
```

Al igual que con las clases abstractas, es posible crear referencias de una interfaz:

```
InstrumentoMusical instrumento;
```

Sin embargo, **no se puede instanciar**, es decir:

```
InstrumentoMusical instrumento = new InstrumentoMusical();
```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	152/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

Lo que sí se puede hacer, como en las clases abstractas, es crear una referencia abstracta que contenga un objeto concreto, es decir:

*InstrumentoMusical instrumento = new Flauta();*

Ejemplo:

```

public interface InstrumentoMusical {
    void tocar();

    void afinar();


    String tipoInstrumento();
}

public class InstrumentoViento extends Object implements InstrumentoMusical {
    // Por defecto, todos los métodos declarados en una interfaz son
    // públicos, por tanto, se deben implementar con el mismo nivel de acceso
    @Override
    public void tocar() {
        System.out.println("Tocando un instrumento de viento");
    }

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

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

```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	153/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

```

public class Flauta extends InstrumentoViento {
    // La clase Flauta puede modificar el comportamiento
    // de alguno de los métodos heredados

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

```


```

public class PruebaInstrumento {
    public static void main (String [] args) {
        // Se puede crear una referencia del tipo interfaz
        InstrumentoMusical instrumento;

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

        // Una referencia del tipo interfaz, puede ser asignada
        // a cualquier instancia que la implemente
        instrumento = new Flauta();
        instrumento.tocar();
        System.out.println(instrumento.tipoInstrumento());
    }
}

```

	<b>Manual de prácticas del Laboratorio de Modelos de programación orientada a objetos</b>	Código:	MADO-21
		Versión:	02
		Página	154/166
		Sección ISO	8.3
		Fecha de emisión	14 de junio de 2018
Facultad de Ingeniería		Área/Departamento: Laboratorio de computación salas A y B	
La impresión de este documento es una copia no controlada			

## 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*