



**UNIVERSIDAD DE LAS FUERZAS ARMADAS (ESPE)**

**SEGUNDO SEMESTRE**

**CARRERAS TECNICAS**

**DEPARTAMENTO DE CIENCIAS EXACTAS**

**SEGUNDO PARCIAL**

**“POLIMORFISMO Y CLASES ABSTRACTAS”**

**AUTORES:**

- Guerrero Steven.
- Sislema Gabriel.
- Shaffik Samir.

**NRC:**

1322

**DOCENTE:**

**SANGOLQUI – ECUADOR**

## **Introducción**

En el ámbito del desarrollo de software orientado a objetos, los conceptos de polimorfismo y clases abstractas son clave para una adecuada organización y reutilización del código. El polimorfismo permite que una misma interfaz sea empleada para distintos tipos de datos, lo que aporta flexibilidad y escalabilidad al sistema. Por otro lado, las clases abstractas actúan como modelos base que definen estructuras que deben ser implementadas por sus clases derivadas, favoreciendo así la coherencia y un diseño estructurado.

En este documento, profundizaremos en estos conceptos, analizando su aplicación en el diseño de sistemas y su repercusión en la eficiencia del desarrollo de software. Además, presentaremos un resumen de sus características, ejemplos prácticos y la relevancia que tienen en la programación contemporánea.

## **1. OBJETIVOS**

### **1.1 OBJETIVO GENERAL**

Analizar y comprender los conceptos de polimorfismo y clases abstractas en la programación orientada a objetos, destacando su importancia en el diseño de sistemas de software eficientes y flexibles mediante la elaboración de un resumen con ejemplos prácticos.

### **1.2 OBJETIVOS ESPECÍFICO**

Explicar el concepto de polimorfismo y clases abstractas, identificando sus características y diferencias en la programación orientada a objetos.

Aplicar los conocimientos adquiridos mediante la elaboración de un resumen y ejemplos prácticos, demostrando su utilidad en el diseño de sistemas de software eficientes.

## 2. MARCO TEÓRICO

### Polimorfismo

El polimorfismo en la programación orientada a objetos se refiere a la capacidad de una función, método o interfaz para adoptar distintas formas. Esto permite que una misma operación se comporte de manera diferente según el objeto que la invoque. Esta característica aumenta la flexibilidad y la reutilización del código, lo que a su vez facilita el mantenimiento y la escalabilidad del software.

```
Declaro la función:
function estacionar( Vehiculo ) {}

Invoco la función: (soporto polimorfismo)
estacionar( Coche ) ;
estacionar( Moto ) ;
estacionar( Bici ) ;

No puedo invocar la función: (errores sintácticos, porque no son declaraciones de funciones de vehículos)
estacionar( Moto ) ;
estacionar( 123 ) ;

En el futuro sí podría: (si como las clases "Van" o "Habré especial" y herencia de Vehículo)
estacionar( Van ) ;
estacionar( Habré especial ) ;
```

*Ilustración 1*

### Clases abstractas

Una clase abstracta en programación orientada a objetos es un tipo de clase que no se puede instanciar directamente, funcionando más bien como una plantilla para otras clases. Incluye métodos que pueden tener o no una implementación concreta, lo que obliga a las clases derivadas a definir su propia versión de ciertos métodos.

Estas clases abstractas son útiles para establecer comportamientos comunes entre un conjunto de clases relacionadas, fomentando así la reutilización del código y promoviendo un diseño estructurado en el desarrollo de software.

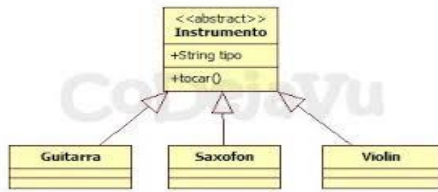
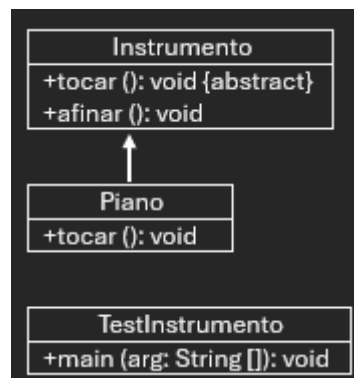
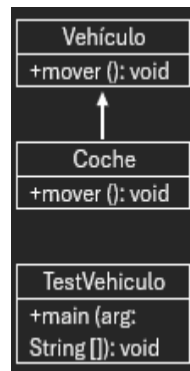
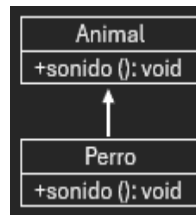
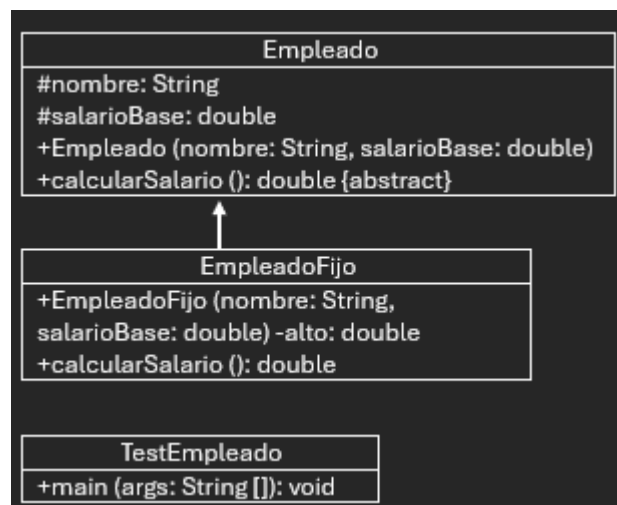
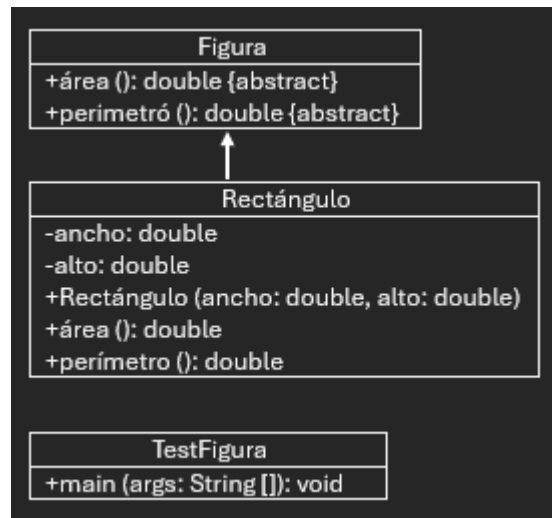


Ilustración 2

## UML





## 1. Jerarquía de Clases de "Animal"

- **Clase Animal:**
  - **Método +sonido(): void** (público).
  - Define una función general sonido(), que será sobrescrita por las subclases.
- **Clase Perro (hereda de Animal):**
  - **También tiene el método +sonido(): void.**
  - Sobrescribe el método de Animal para proporcionar una implementación específica (por ejemplo, "ladrar").

## 2. Jerarquía de Clases de "Vehículo"

- **Clase Vehículo:**
  - Método +mover(): void (público).
  - Define un comportamiento genérico para los vehículos.
- **Clase Coche (hereda de Vehículo):**
  - También tiene el método +mover(): void.
  - Sobrescribe el método de Vehículo, proporcionando una implementación más específica.
- **Clase TestVehiculo:**
  - Contiene un método main(String[] args): void, que probablemente se usa para probar la jerarquía de Vehículo y Coche.

## 3. Jerarquía de Clases de "Instrumento"

- **Clase Instrumento (abstracta):**
  - Método +tocar(): void {abstract} (público y abstracto).
  - Método +afinar(): void (público, pero con implementación concreta).
  - tocar() es un método abstracto que debe ser implementado por las subclases.
- **Clase Piano (hereda de Instrumento):**
  - Implementa el método +tocar(): void, proporcionando su propia versión del comportamiento.
- **Clase TestInstrumento:**
  - Contiene el método main(String[] args): void, que probablemente se usa para probar la jerarquía de Instrumento y Piano.

#### 4. Jerarquía de Clases de "Figura"

- **Clase Figura (abstracta):**
  - Métodos abstractos:
    - **+área():** double {abstract}
    - **+perímetro():** double {abstract}
  - Define una estructura genérica para figuras geométricas, sin implementación específica.
- **Clase Rectángulo (hereda de Figura):**
  - Atributos privados:
    - **-ancho:** double
    - **-alto:** double
  - Constructor que recibe ancho y alto.
  - Implementa los métodos **+área():** double y **+perímetro():** double, ya que son abstractos en Figura.
- **Clase TestFigura:**
  - Contiene el método **main(String[] args):** void, usado para probar la jerarquía de figuras.

#### 5. Jerarquía de Clases de "Empleado"

- **Clase Empleado (abstracta):**
  - **Atributos privados:**
    - **#nombre:** String (protegido).
    - **#salarioBase:** double (protegido).
  - Constructor que recibe nombre y salarioBase.
  - **Método abstracto +calcularSalario():** double {abstract}, que debe ser implementado por las subclases.



- **Clase EmpleadoFijo (hereda de Empleado):**
  - Constructor que recibe nombre, salarioBase y un atributo alto (posiblemente un bono o incremento).
  - Implementa el método +calcularSalario(): double, proporcionando su propia lógica.
- **Clase TestEmpleado:**
  - Contiene el método main(String[] args): void, usado para probar la jerarquía de empleados.

## CÓDIGO POLIMORFISMO – EJEMPLOS

### 1) EJEMPLOS DE SOBRECARGA DE MÉTODOS

```
1 public class Calculadora {
2     // Método para sumar dos números
3     public int sumar(int a, int b) {
4         return a + b;
5     }
6
7     // Método sobrecargado para sumar tres números
8     public int sumar(int a, int b, int c) {
9         return a + b + c;
10    }
11 }
12 }
```

Ilustración 3

#### Variable a

- Actúa como el primer operando en la operación de suma.
- Se recibe como parámetro cuando se llama al método sumar.
- Participa en todas las sumas realizadas en ambos métodos.

#### Variable b

- Es el segundo operando en la operación de suma.
- Se usa en ambos métodos (sumar con dos parámetros y sumar con tres parámetros).

#### Variable c

- Solo se usa en el segundo método sumar(int a, int b, int c).
- Actúa como un tercer operando opcional, permitiendo realizar la suma de tres números en lugar de solo dos.

```
1 public class TestCalculadora {
2     public static void main(String[] args) {
3         Calculadora calc = new Calculadora();
4         System.out.println("Suma de 2 números: " + calc.sumar(5, 3));
5         System.out.println("Suma de 3 números: " + calc.sumar(5, 3, 2));
6     }
7 }
8 }
```

Ilustración 4

**Calculadora calc = new Calculadora();**

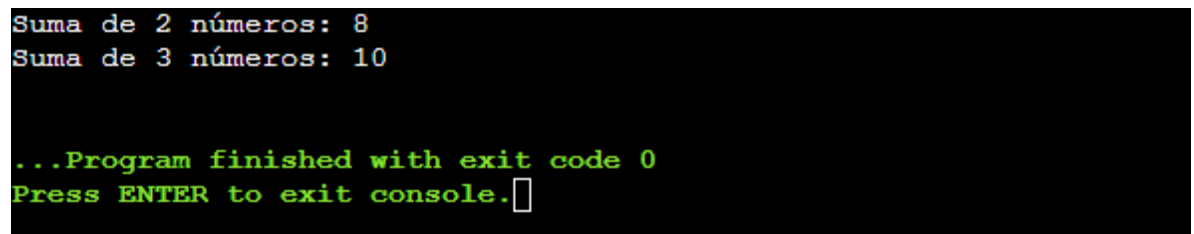
- **calc:** Es una variable de tipo Calculadora que representa un objeto de la clase Calculadora.
- **new Calculadora();** Crea una nueva instancia de la clase Calculadora, permitiendo el uso de sus métodos.

**System.out.println("Suma de 2 números: " + calc.sumar(5, 3));**

- **calc.sumar(5, 3):** Llama al método sumar de la clase Calculadora, pasando los valores 5 y 3 como parámetros.
- **"Suma de 2 números: ":** Es un texto que se concatena con el resultado de `calc.sumar(5,3)` para imprimirlo en pantalla.

**System.out.println("Suma de 3 números: " + calc.sumar(5, 3, 2));**

- **calc.sumar(5, 3, 2):** Llama a un método sobrecargado sumar, que suma tres números (5, 3 y 2).
- **"Suma de 3 números: ":** Texto que acompaña el resultado de la operación.



```
Suma de 2 números: 8
Suma de 3 números: 10

...Program finished with exit code 0
Press ENTER to exit console.█
```

Ilustración 5

**Explicación de la salida:**

**1. Suma de 2 números: 8**

- Corresponde a la ejecución de `calc.sumar(5, 3)`, lo que indica que la clase Calculadora tiene un método que suma dos números y devuelve  $5 + 3 = 8$ .

## 2. Suma de 3 números: 10

- Proviene de `calc.sumar(5, 3, 2)`, lo que sugiere que la clase `Calculadora` tiene un método sobrecargado que permite sumar tres números, devolviendo  $5 + 3 + 2 = 10$ .

## 3. Program finished with exit code 0

- Indica que el programa se ejecutó correctamente sin errores.

## 4. Press ENTER to exit console.

- Es un mensaje del entorno de desarrollo que espera una entrada del usuario para cerrar la consola.

### Ejemplo 2:

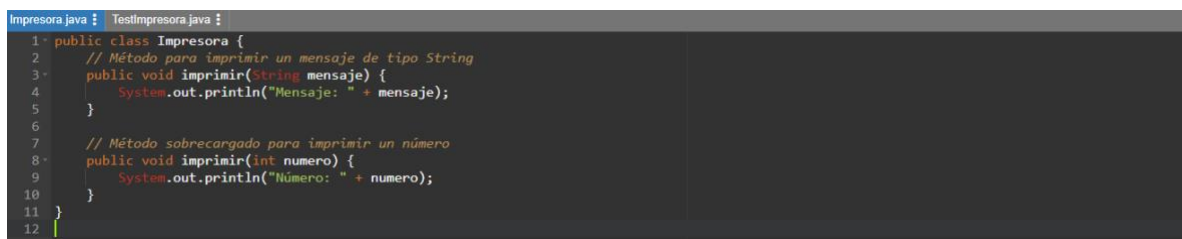


Ilustración 6

### Clase `Impresora`

- Se define una clase pública llamada `Impresora` que tiene dos métodos para imprimir distintos tipos de datos en la consola.

#### Método `imprimir(String mensaje)`

- **Parámetro:** Recibe un `String` llamado `mensaje`.
- **Acción:** Imprime en la consola "Mensaje: " seguido del contenido del parámetro `mensaje`.

#### Método `imprimir(int numero)` (Sobrecarga del anterior)

- **Parámetro:** Recibe un `int` llamado `numero`.
- **Acción:** Imprime en la consola "Número: " seguido del valor del parámetro `numero`.

```

Impresora.java : TestImpresora.java :
1 public class TestImpresora {
2     public static void main(String[] args) {
3         Impresora imp = new Impresora();
4         imp.imprimir("Hola, mundo");
5         imp.imprimir(2025);
6     }
7 }
8

```

### args (String[] args):

- Es un parámetro del método main. Se usa para recibir argumentos desde la línea de comandos cuando se ejecuta el programa.

### imp (Impresora imp = new Impresora());:

- Es una variable de tipo Impresora, que representa un objeto de la clase Impresora.
- Se crea una instancia de Impresora utilizando new Impresora(), lo que permite acceder a los métodos de esa clase.

### Creación de un objeto Impresora:

- Impresora imp = new Impresora(); crea una nueva instancia de la clase Impresora.

### Llamadas al método imprimir:

- **imp.imprimir("Hola, mundo");:** Llama al método imprimir con un String como argumento.
- **imp.imprimir(2025);:** Llama al método imprimir con un int como argumento.

### Confirmación de impresión:

```

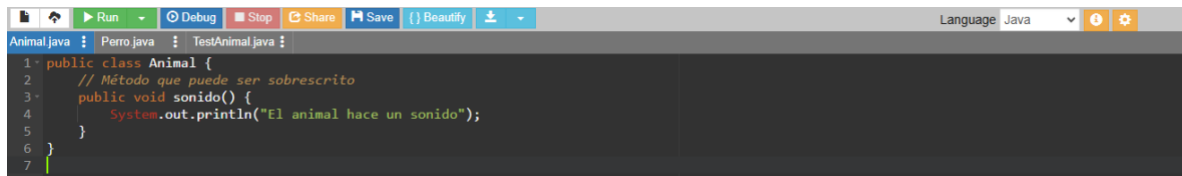
Mensaje: Hola, mundo
Número: 2025

...Program finished with exit code 0
Press ENTER to exit console.

```

Ilustración 7

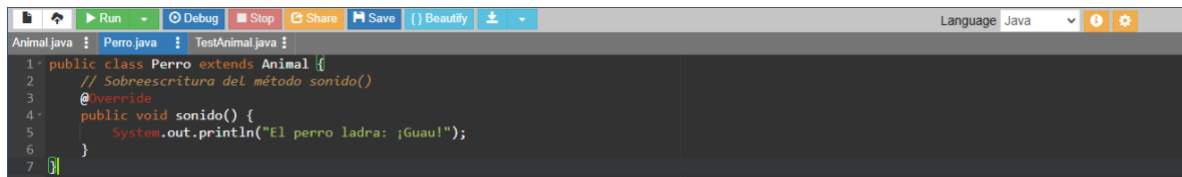
## 2) SOBRE ESCRITURA DE MÉTODOS



```
1 public class Animal {
2     // Método que puede ser sobrescrito
3     public void sonido() {
4         System.out.println("El animal hace un sonido");
5     }
6 }
7
```

Ilustración 8

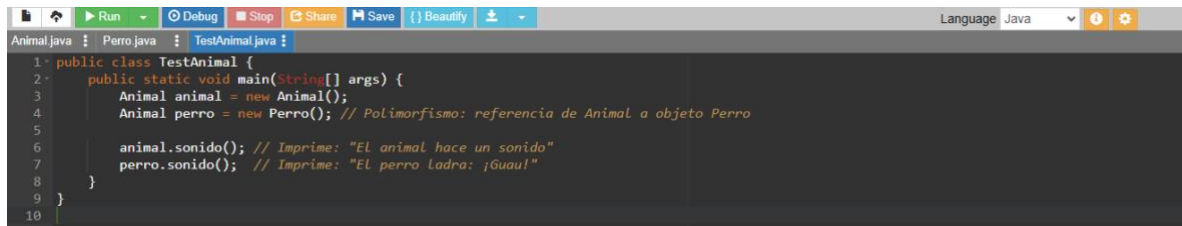
Se define un método sonido(), que imprime "El animal hace un sonido" y puede ser sobrescrito en clases hijas.



```
1 public class Perro extends Animal {
2     // Sobrescritura del método sonido()
3     @Override
4     public void sonido() {
5         System.out.println("El perro ladra: ¡Guau!");
6     }
7 }
```

Ilustración 9

La clase Perro hereda de Animal y sobrescribe el método sonido() para imprimir "El perro ladra: ¡Guau!".



```
1 public class TestAnimal {
2     public static void main(String[] args) {
3         Animal animal = new Animal();
4         Animal perro = new Perro(); // Polimorfismo: referencia de Animal a objeto Perro
5
6         animal.sonido(); // Imprime: "El animal hace un sonido"
7         perro.sonido();  // Imprime: "El perro ladra: ¡Guau!"
8     }
9 }
10
```

Ilustración 10

### Variables:

#### Animal

- **Tipo:** Animal
- **Valor:** new Animal(); Se crea un objeto de la clase Animal.
- **Acción:** Al llamar animal.sonido(), ejecuta el método de Animal, imprimiendo "El animal hace un sonido".

#### Perro

- **Tipo:** Animal (pero instancia de Perro)

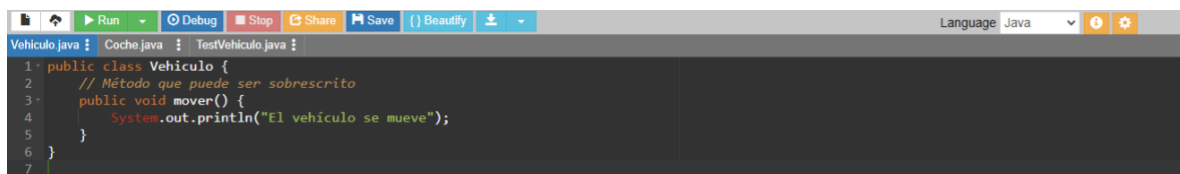
- **Valor:** `new Perro();` Se crea un objeto de `Perro`, pero con una referencia de `Animal` (polimorfismo).
- **Acción:** Al llamar `perro.sonido()`, ejecuta el método sobrescrito en `Perro`, imprimiendo "El perro ladra: ¡Guau!".

#### Confirmación de impresión:

```
El animal hace un sonido
El perro ladra: ¡Guau!

...Program finished with exit code 0
Press ENTER to exit console.
```

Ilustración 11

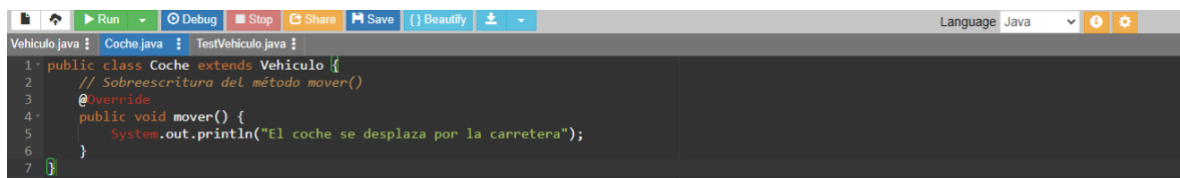


```
1 public class Vehiculo {
2     // Método que puede ser sobrescrito
3     public void mover() {
4         System.out.println("El vehículo se mueve");
5     }
6 }
7
```

Ilustración 12

**Clase Vehículo:** Es una clase base que define un método llamado `mover()`.

**Método mover():** Muestra el mensaje "El vehículo se mueve", lo cual indica un comportamiento genérico para cualquier vehículo.



```
1 public class Coche extends Vehiculo {
2     // Sobreescritura del método mover()
3     @Override
4     public void mover() {
5         System.out.println("El coche se desplaza por la carretera");
6     }
7 }
8
```

Ilustración 13

**Clase Coche:** Hereda de `Vehiculo`, lo que significa que `Coche` es un tipo de `Vehiculo`.

**Sobreescritura del método mover():** En lugar de usar el comportamiento definido en `Vehiculo`, redefine el método para que muestre "El coche se desplaza por la carretera", lo cual es un ejemplo de polimorfismo.

```

1: public class TestVehiculo {
2:     public static void main(String[] args) {
3:         Vehiculo vehiculo = new Vehiculo();
4:         Vehiculo coche = new Coche(); // Polimorfismo: referencia de Vehiculo a objeto Coche
5:
6:         vehiculo.mover(); // Imprime: "El vehículo se mueve"
7:         coche.mover();    // Imprime: "El coche se desplaza por la carretera"
8:     }
9: }

```

Ilustración 14

**Vehiculo vehiculo = new Vehiculo();**

- Crea un objeto de tipo Vehiculo y lo almacena en la variable vehiculo.
- Llama al método mover(), que imprime: "El vehículo se mueve".

**Vehiculo coche = new Coche(); (Polimorfismo)**

- Declara una variable de tipo Vehiculo, pero le asigna un objeto Coche.
- Gracias al polimorfismo, aunque la variable es de tipo Vehiculo, el método mover() llamado será el de Coche, ya que está sobrescrito en la subclase.
- Por eso, cuando se llama coche.mover(), imprime: "El coche se desplaza por la carretera"

**Confirmación de imprecion:**

```

El vehículo se mueve
El coche se desplaza por la carretera

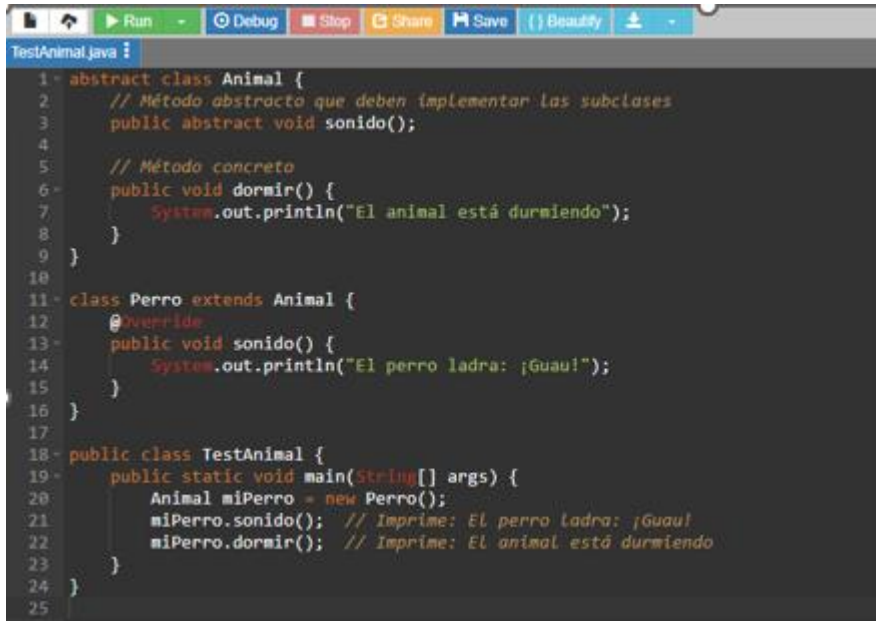
...Program finished with exit code 0
Press ENTER to exit console.

```

Ilustración 15



## CÓDIGO CLASES ABSTRACTAS – EJEMPLOS



```
1- abstract class Animal {
2-     // Método abstracto que deben implementar las subclases
3-     public abstract void sonido();
4-
5-     // Método concreto
6-     public void dormir() {
7-         System.out.println("El animal está durmiendo");
8-     }
9- }
10
11- class Perro extends Animal {
12-     @Override
13-     public void sonido() {
14-         System.out.println("El perro ladra: ¡Guau!");
15-     }
16- }
17
18- public class TestAnimal {
19-     public static void main(String[] args) {
20-         Animal miPerro = new Perro();
21-         miPerro.sonido(); // Imprime: El perro ladra: ¡Guau!
22-         miPerro.dormir(); // Imprime: El animal está durmiendo
23-     }
24- }
25
```

Ilustración 16

**Animal miPerro = new Perro();**

- **Tipo:** Animal (superclase).
- **Objeto instanciado:** new Perro() (subclase).
- **Explicación:** Se usa **polimorfismo**, ya que miPerro es una variable de tipo Animal, pero almacena un objeto de tipo Perro.

**Comportamiento:**

- **miPerro.sonido();** : Llama al método sobrescrito en Perro, mostrando "El perro ladra: ¡Guau!".
- **miPerro.dormir();** : Usa el método heredado de Animal, mostrando "El animal está durmiendo".

## Confirmación de impresión:

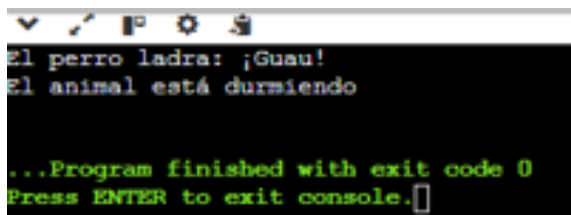


Ilustración 17

## Ejemplo 2:

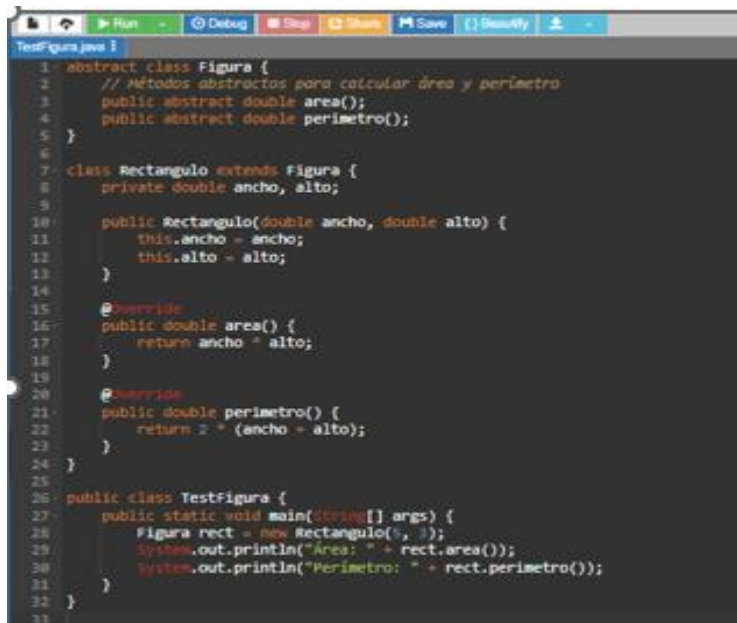


Ilustración 18

## Figura rect = new Rectangulo(5, 3);

- **Tipo:** Figura (superclase abstracta).
- **Objeto instanciado:** new Rectangulo(5, 3); (subclase concreta).
- **Explicación:** Se usa polimorfismo, ya que rect es una variable de tipo Figura, pero almacena un objeto de tipo Rectangulo.

## Comportamiento:

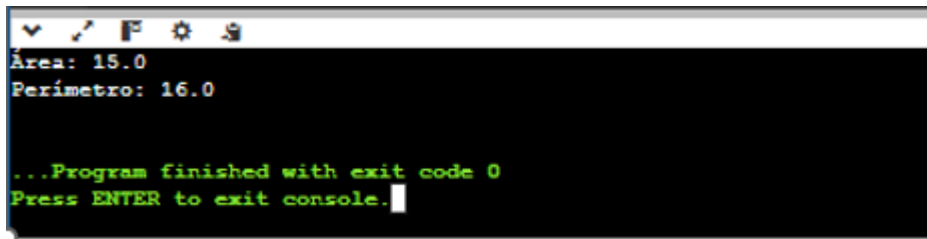
- **rect.area();** : Llama al método sobrescrito en Rectangulo, devolviendo  $5 \times 3 = 15$ .
- **rect.perimetro();** : Llama al método sobrescrito en Rectangulo, devolviendo  $2 \times (5 + 3) = 16$ .

**private double ancho, alto;**

- **Ubicación:** Dentro de la clase Rectangulo.
- **Explicación:** Son variables de instancia que almacenan el ancho y alto del rectángulo.

Se inicializan en el constructor con los valores pasados como parámetros.

**Confirmación de imprecion:**

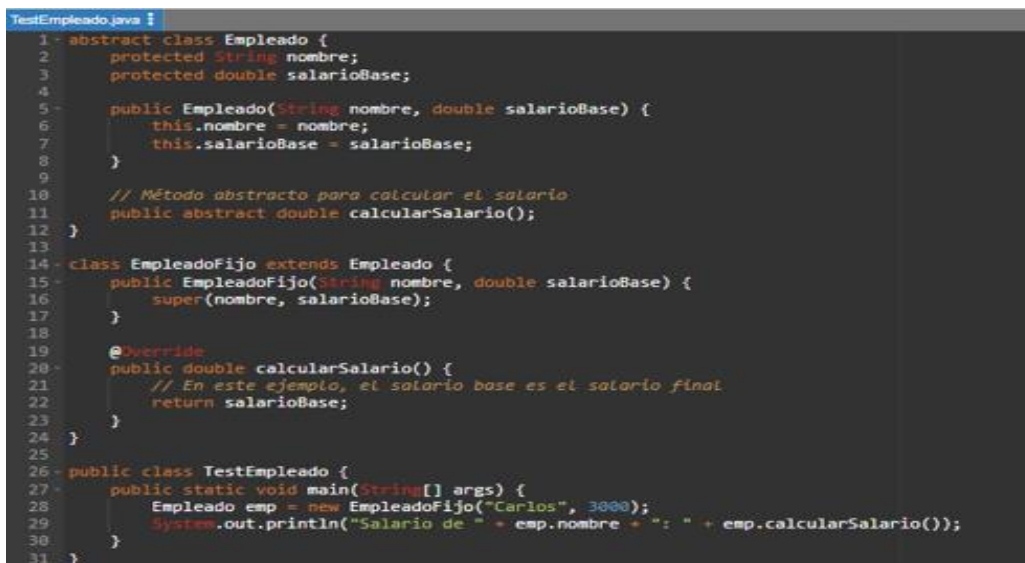


```
Área: 15.0
Perímetro: 16.0

...Program finished with exit code 0
Press ENTER to exit console.
```

Ilustración 19

**Ejemplo 3:**



```
TestEmpleado.java
1- abstract class Empleado {
2-     protected String nombre;
3-     protected double salarioBase;
4-
5-     public Empleado(String nombre, double salarioBase) {
6-         this.nombre = nombre;
7-         this.salarioBase = salarioBase;
8-     }
9-
10-    // Método abstracto para calcular el salario
11-    public abstract double calcularSalario();
12- }
13-
14- class EmpleadoFijo extends Empleado {
15-     public EmpleadoFijo(String nombre, double salarioBase) {
16-         super(nombre, salarioBase);
17-     }
18-
19-     @Override
20-     public double calcularSalario() {
21-         // En este ejemplo, el salario base es el salario final
22-         return salarioBase;
23-     }
24- }
25-
26- public class TestEmpleado {
27-     public static void main(String[] args) {
28-         Empleado emp = new EmpleadoFijo("Carlos", 3000);
29-         System.out.println("Salario de " + emp.nombre + ": " + emp.calcularSalario());
30-     }
31- }
```

Ilustración 20

**nombre (línea 3)**

- **Tipo:** String
- **Modificador:** protected
- **Propósito:** Almacena el nombre del empleado.
- Se asigna en el constructor y se puede acceder en clases hijas.

## **salarioBase**

**Tipo:** double

- Modificador: protected
- Propósito: Representa el salario base del empleado.
- Se asigna en el constructor y se puede acceder en clases hijas.

### **Constructor Empleado(String nombre, double salarioBase)**

Inicializa los atributos nombre y salarioBase cuando se crea un objeto de tipo Empleado o sus subclases.

- **this.nombre** = nombre; Asigna el valor del parámetro nombre al atributo de la clase.
- **this.salarioBase** = salarioBase; Asigna el valor del parámetro salarioBase al atributo de la clase.

### **Método abstracto calcularSalario()**

- No tiene implementación en esta clase, por lo que cualquier subclase que herede de Empleado debe implementarlo.

### **Clase EmpleadoFijo (Hereda de Empleado)**

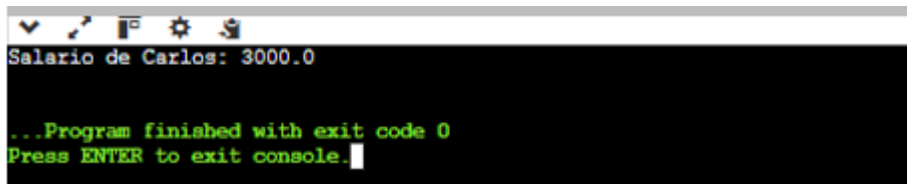
#### **Constructor EmpleadoFijo(String nombre, double salarioBase)**

Llama al constructor de Empleado con super(nombre, salarioBase);, lo que inicializa los atributos nombre y salarioBase.

#### **Método calcularSalario()**

- Implementa el método abstracto de Empleado.
- Retorna salarioBase, ya que en este ejemplo el salario fijo es igual al salario base.

## Confirmación de impresión:



```
Salario de Carlos: 3000.0

...Program finished with exit code 0
Press ENTER to exit console.
```

Ilustración 21

## Ejemplo 4:



```
1 abstract class Instrumento {
2     // Método abstracto que deben implementar las subclases
3     public abstract void tocar();
4
5     // Método concreto para afinar el instrumento
6     public void afinar() {
7         System.out.println("Afinando el instrumento.");
8     }
9 }
10
11 class Piano extends Instrumento {
12     @Override
13     public void tocar() {
14         System.out.println("Tocando el piano.");
15     }
16 }
17
18 public class TestInstrumento {
19     public static void main(String[] args) {
20         Instrumento miPiano = new Piano();
21         miPiano.afinar(); // Imprime: Afinando el instrumento.
22         miPiano.tocar();  // Imprime: Tocando el piano.
23     }
24 }
25
```

Ilustración 22

## Clase Instrumento (Clase Abstracta)

- **Método afinar():** Es un método concreto, lo que significa que tiene una implementación. Muestra el mensaje "Afinando el instrumento." cuando se llama.

## Clase Piano (Hereda de Instrumento)

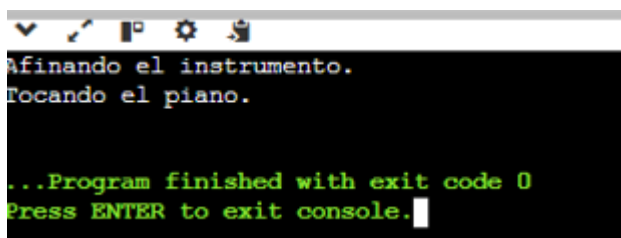
- **Herencia:** Piano extiende Instrumento, lo que significa que hereda sus métodos.
- **Implementación del método tocar():** Como Instrumento tiene un método abstracto tocar(), Piano está obligado a implementarlo. En este caso, imprime "Tocando el piano."

## Clase TestInstrumento (Clase con main)

- **Método main:** Es el punto de entrada del programa.

- **Instancia de Piano:** Se crea un objeto de la clase Piano, pero se almacena en una variable de tipo Instrumento (polimorfismo).
- **Llamada a afinar():** Se ejecuta el método heredado de Instrumento, mostrando "Afinando el instrumento."
- **Llamada a tocar():** Se ejecuta el método sobrescrito en Piano, mostrando "Tocando el piano."

#### Confirmación de imprecion:

A screenshot of a terminal window with a dark background. The output text is displayed in a monospaced font. The first two lines are in red: "Afinando el instrumento." and "Tocando el piano.". The next two lines are in green: "...Program finished with exit code 0" and "Press ENTER to exit console.". A white cursor is visible at the end of the last line.

```
Afinando el instrumento.  
Tocando el piano.  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Ilustración 23

### **3. CONCLUSIONES**

El polimorfismo y las clases abstractas son conceptos fundamentales en la programación orientada a objetos, ya que permiten un diseño de software más flexible, escalable y reutilizable. Gracias a estas técnicas, se puede mejorar la organización del código, reducir la redundancia y facilitar el mantenimiento de los sistemas de software.

La implementación de polimorfismo y clases abstractas en la programación orientada a objetos permite establecer una base sólida para el desarrollo de software. Estas herramientas proporcionan una mejor organización del código, fomentan la reutilización y permiten la creación de sistemas más eficientes y estructurados, facilitando su comprensión y mantenimiento a largo plazo.

### **4. RECOMENDACIONES**

Recomendamos utilizar polimorfismo cuando sea necesario para evitar código repetitivo y mejorar la escalabilidad de los sistemas. Es fundamental analizar cuándo es más conveniente la sobrecarga de métodos y cuándo la sobrescritura para optimizar el diseño del software.

Es importante asegurarse de que las clases abstractas contengan solo los métodos esenciales que deben ser implementados por las subclasses. Se debe evitar agregar métodos concretos innecesarios en estas clases para no afectar la flexibilidad del diseño.

## 5. BIBLIOGRAFÍA

Deitel, P., & Deitel, H. (2016). *Cómo programar en Java* (10ª ed.). Pearson Educación.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2002). *Patrones de diseño: Elementos de software orientado a objetos reutilizable*. Addison-Wesley.

Sommerville, I. (2011). *Ingeniería del software* (9ª ed.). Pearson Educación.

Horstmann, C. S. (2013). *Java: Fundamentos y técnicas de programación orientada a objetos* (2ª ed.). Pearson Educación.

Pressman, R. S., & Maxim, B. R. (2010). *Ingeniería del software: Un enfoque práctico* (7ª ed.). McGraw-Hill.

Stroustrup, B. (2014). *El lenguaje de programación C++* (4ª ed.). Addison-Wesley.

Fowler, M. (2005). *UML Gota a Gota: Guía breve del lenguaje de modelado estándar* (3ª ed.). Addison-Wesley.