




Trabajo Práctico 7: Herencia y Polimorfismo en Java

Alumno: Calcatelli Renzo - rcalcatelli@gmail.com

Comisión: M2025-1

Materia: Programación II

Profesor: Ariel Enferrel

 [Enlace](#) al repositorio de GitHub

Trabajo Práctico 7: Herencia y Polimorfismo en Java

ÍNDICE

1. Introducción
2. Objetivos del Trabajo Práctico
3. Marco Teórico
4. Desarrollo de los Ejercicios
 - 4.1 Ejercicio 1: Vehículos y Herencia Básica
 - 4.2 Ejercicio 2: Figuras Geométricas y Métodos Abstractos
 - 4.3 Ejercicio 3: Empleados y Polimorfismo con instanceof
 - 4.4 Ejercicio 4: Animales y Comportamiento Sobrescrito
 - 4.5 Ejercicio 5: Sistema de Pagos con Interfaces
5. Análisis Comparativo de los Conceptos Aplicados
6. Resultados Obtenidos
7. Conclusiones
8. Bibliografía

1. INTRODUCCIÓN

En este trabajo práctico aplicaremos los conceptos fundamentales de **Herencia y Polimorfismo**, dos pilares de la Programación Orientada a Objetos. A través de 5 ejercicios prácticos (katas), demostraremos cómo estos conceptos permiten crear código reutilizable, flexible y fácil de mantener.

La herencia nos permite crear nuevas clases basadas en clases existentes, heredando sus atributos y métodos. El polimorfismo, por su parte, nos permite tratar objetos de diferentes clases de manera uniforme, siempre que compartan una interfaz común o hereden de una clase base común.

2. OBJETIVOS DEL TRABAJO PRÁCTICO

Objetivo General

Comprender y aplicar los conceptos de herencia y polimorfismo en la Programación Orientada a Objetos, reconociendo su importancia para la reutilización de código, la creación de jerarquías de clases y el diseño flexible de soluciones en Java.

Objetivos Específicos

- Implementar jerarquías de clases utilizando herencia simple con **extends**.
 - Aplicar clases abstractas y métodos abstractos para definir comportamientos base.
 - Sobrescribir métodos utilizando la anotación **@Override**.
 - Demostrar el polimorfismo mediante el uso de referencias de tipo padre.
 - Implementar interfaces para lograr contratos de comportamiento.
 - Utilizar el operador **instanceof** para identificación de tipos en tiempo de ejecución.
 - Aplicar correctamente modificadores de acceso (**protected**, **private**, **public**).
-

3. MARCO TEÓRICO

3.1 Herencia

La herencia es un mecanismo que permite crear nuevas clases basadas en clases existentes. La clase que hereda se denomina subclase o clase hija, mientras que la clase de la cual se hereda se llama superclase o clase padre.

Características principales:

- Uso de la palabra clave **extends** para establecer la relación de herencia.
- Aprovecha el principio "is-a" (es un/una).
- Permite la reutilización de código.
- La subclase hereda atributos y métodos de la superclase.

3.2 Modificadores de Acceso

- **private**: Solo accesible dentro de la misma clase.
- **protected**: Accesible dentro de la clase y sus subclases.
- **public**: Accesible desde cualquier lugar.

3.3 Constructores y super

El constructor de una subclase debe invocar al constructor de la superclase usando **super(...)** para inicializar correctamente los atributos heredados.

3.4 Clases Abstractas

Una clase abstracta no puede ser instanciada directamente. Se utiliza como clase base para otras clases. Puede contener:

- Métodos abstractos (sin implementación).
- Métodos concretos (con implementación).
- Atributos.

3.5 Polimorfismo

El polimorfismo permite que objetos de diferentes clases sean tratados como objetos de una clase común. Existen dos tipos:

- **Upcasting:** Conversión implícita de una subclase a su superclase.
- **Downcasting:** Conversión explícita de una superclase a una subclase específica.

3.6 Interfaces

Una interfaz define un contrato que las clases deben cumplir. Solo contiene declaraciones de métodos y constantes.

3.7 Operador instanceof

Permite verificar si un objeto es instancia de una clase o interfaz específica. Útil antes de realizar un downcasting para evitar errores en tiempo de ejecución.

4. DESARROLLO DE LOS EJERCICIOS

4.1 Ejercicio 1: Vehículos y Herencia Básica

Enunciado: Crear una clase base **Vehiculo** con atributos marca y modelo, y el método **mostrarInfo()**. Luego crear una clase **Auto** que herede de **Vehiculo** y agregue el atributo **cantidadDePuertas**, sobrescribiendo el método **mostrarInfo()**.

Solución

```
Clase Vehiculo (Clase Padre)
public class Vehiculo {
    // Atributos protegidos: accesibles desde clases hijas
    protected String marca;
    protected String modelo;
```

```
// Constructor: inicializa los atributos del vehículo
public Vehiculo(String marca, String modelo) {
    this.marca = marca;
    this.modelo = modelo;
}

// Método que muestra la información básica del vehículo
public void mostrarInfo() {
    System.out.println("Modelo: " + modelo + " ,Marca: " +
marca);
}
}
```

Clase Auto (Clase Hija)

```
public class Auto extends Vehiculo { // Heredamos de Vehiculo
    // Atributo adicional específico de Auto
    private int cantidadDePuertas;

    // Constructor: utiliza super() para inicializar atributos
heredados
    public Auto(int cantidadDePuertas, String marca, String modelo) {
        super(marca, modelo); // Llama al constructor de la clase
padre
        this.cantidadDePuertas = cantidadDePuertas;
    }

    // Sobrescribimos el método heredado para mostrar más información
@Override
    public void mostrarInfo() {
        System.out.println("Modelo: " + this.modelo + " ,Marca: " +
this.marca +
                                ", Cantidad de puertas: " +
cantidadDePuertas);
    }
}
```

Clase Main

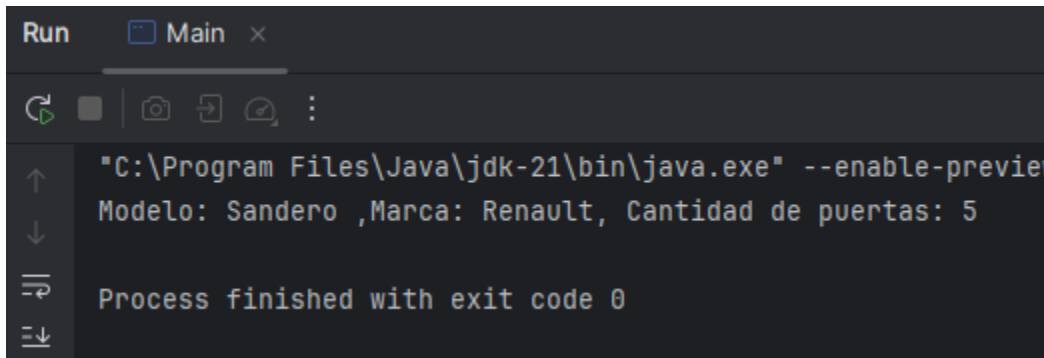
```
public static void main(String[] args) {
    // Instanciamos un auto con 5 puertas, marca Renault, modelo
Sandero
    Auto a = new Auto(5, "Renault", "Sandero");

    // Llamamos al método sobrescrito que muestra toda la información
}
```

```
a.mostrarInfo();  
}
```

Salida esperada

```
Modelo: Sandero ,marca: Renault, cantidad de puertas: 5
```



```
Run Main x  
"C:\Program Files\Java\jdk-21\bin\java.exe" --enable-preview  
Modelo: Sandero ,Marca: Renault, Cantidad de puertas: 5  
Process finished with exit code 0
```

Explicación del código

En este ejercicio implementamos herencia simple. La clase **Auto** hereda de **Vehiculo**, lo que significa que:

1. **Reutilización de código:** Auto no necesita redefinir los atributos **marca** y **modelo**, los hereda automáticamente.
2. **Modificador protected:** Usamos **protected** en los atributos de Vehiculo para que sean accesibles desde las clases hijas, pero no desde cualquier otra clase.
3. **Constructor con super():** El constructor de Auto llama primero a **super(marca, modelo)** para inicializar los atributos heredados antes de inicializar sus propios atributos.
4. **Sobrescritura con @Override:** El método **mostrarInfo()** se sobrescribe en Auto para incluir información adicional (cantidad de puertas). La anotación **@Override** le indica al compilador que estamos sobrescribiendo un método del padre.

4.2 Ejercicio 2: Figuras Geométricas y Métodos Abstractos

Enunciado: Crear una clase abstracta **Figura** con un método abstracto **calcularArea()** y un atributo nombre. Luego crear dos clases **Circulo** y **Rectangulo** que implementen el cálculo del área según su fórmula correspondiente.

Solución

Clase Figura (Clase Abstracta)

```
public abstract class Figura {
    // Atributo protegido para el nombre de la figura
    protected String nombre;

    // Constructor de la clase abstracta
    public Figura(String nombre) {
        this.nombre = nombre;
    }

    // Método abstracto: debe ser implementado por las subclases
    public abstract void calcularArea();
}
```

Clase Circulo

```
public class Circulo extends Figura {
    // Atributo específico del círculo
    private double radio;

    // Constructor
    public Circulo(double radio, String nombre) {
        super(nombre); // Inicializa el nombre en la clase padre
        this.radio = radio;
    }

    // Implementación del método abstracto para calcular área del círculo
    @Override
    public void calcularArea() {
        System.out.println("El area del circulo " + nombre + " es: "
+
            (radio * 3.14));
    }
}
```

Clase Rectangulo

```
public class Rectangulo extends Figura {
    // Atributos específicos del rectángulo
    private double base;
    private double altura;

    // Constructor
    public Rectangulo(double base, double altura, String nombre) {
        super(nombre);
        this.base = base;
        this.altura = altura;
    }

    // Implementación del método abstracto para calcular área del
    rectángulo
    @Override
    public void calcularArea() {
        System.out.println("El area del rectangulo " + nombre + " es:
" +
                        (base * altura));
    }
}
```

Clase Main

```
public static void main(String[] args) {
    // Creamos un array de tipo Figura (polimorfismo)
    ArrayList<Figura> figuras = new ArrayList<>();

    // Creamos y añadimos figuras al array
    Rectangulo r1 = new Rectangulo(4.0, 4.0, "Rectangulo 1");
    Rectangulo r2 = new Rectangulo(6.0, 4.0, "Rectangulo 2");
    Circulo c1 = new Circulo(10, "Circulo 1");
    Circulo c2 = new Circulo(15, "Circulo 2");

    figuras.add(r1);
    figuras.add(r2);
    figuras.add(c1);
    figuras.add(c2);

    // Recorremos el array y calculamos el área de cada figura
    // Gracias al polimorfismo, cada figura ejecuta su propia versión
    de calcularArea()
    for(Figura f : figuras) {
```



```
f.calcularArea();
    }
}
```

Salida esperada

```
El area del rectangulo Rectangulo 1 es: 16.0
El area del rectangulo Rectangulo 2 es: 24.0
El area del circulo Circulo 1 es: 31.400000000000002
El area del circulo Circulo 2 es: 47.1
```

```
Run Main x
C:\Program Files\Java\jdk-21\bin\java.exe" --enable-preview
El area del rectangulo Rectangulo 1 es: 16.0
El area del rectangulo Rectangulo 2 es: 24.0
El area del circulo Circulo 1 es: 31.400000000000002
El area del circulo Circulo 2 es: 47.1
Process finished with exit code 0
```

Explicación del código

Este ejercicio demuestra el uso de clases abstractas y polimorfismo:

1. **Clase abstracta:** **Figura** es abstracta porque no tiene sentido calcular el área de una "figura genérica". Solo las figuras concretas (**Circulo**, **Rectangulo**) saben cómo calcular su área.
2. **Método abstracto:** **calcularArea()** se declara abstracto en **Figura**, obligando a todas las subclases a implementarlo. Es como un "contrato" que deben cumplir.
3. **Polimorfismo en acción:** En el main, creamos un **ArrayList<Figura>** que puede contener cualquier objeto que herede de **Figura**. Al recorrer el array con el for-each, Java automáticamente llama a la versión correcta de **calcularArea()** según el tipo real del objeto (esto se llama enlace dinámico).
4. **Ventaja:** Si mañana creamos una clase **Triangulo**, solo necesitamos que extienda de **Figura** e implemente **calcularArea()**, y funcionará con el código existente.

4.3 Ejercicio 3: Empleados y Polimorfismo con `instanceof`

Enunciado: Crear una clase abstracta **Empleado** con un método **calcularSueldo(Empleado e)** que reciba un empleado y devuelva un sueldo fijo según el tipo. Crear dos subclases: **EmpleadoPlanta** y **EmpleadoTemporal**.

Solución

Clase Empleado (Clase Abstracta)

```
public abstract class Empleado {

    /**
     * Calcula el sueldo correspondiente a un empleado según su tipo.
     *
     * @param e el empleado del cual se desea calcular el sueldo.
     *         Puede ser una instancia de EmpleadoPlanta o
EmpleadoTemporal.
     * @return el sueldo del empleado:
     *         - 900000.0 si es EmpleadoPlanta
     *         - 850000.0 si es EmpleadoTemporal
     *         - 0.0 si no pertenece a ninguno de los tipos
anteriores
     */
    public double calcularSueldo(Empleado e) {
        // Usamos instanceof para determinar el tipo real del
empleado
        if (e instanceof EmpleadoPlanta) {
            return 900000.0;
        } else if (e instanceof EmpleadoTemporal) {
            return 850000.0;
        } else {
            return 0;
        }
    }
}
```

Clase EmpleadoPlanta

```
public class EmpleadoPlanta extends Empleado {
    // Clase vacía: hereda todo de Empleado
}
```

```
// Se identifica por su tipo  
}
```

Clase EmpleadoTemporal

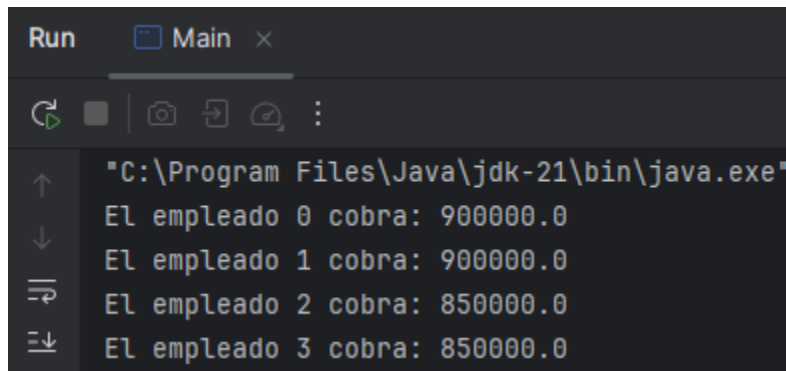
```
public class EmpleadoTemporal extends Empleado {  
    // Clase vacía: hereda todo de Empleado  
    // Se identifica por su tipo  
}
```

Clase Main

```
public static void main(String[] args) {  
    // Inicializamos un ArrayList de empleados  
    ArrayList<Empleado> empleados = new ArrayList<>();  
  
    // Creamos empleados de ambos tipos  
    EmpleadoPlanta e1 = new EmpleadoPlanta();  
    EmpleadoPlanta e2 = new EmpleadoPlanta();  
    EmpleadoTemporal e3 = new EmpleadoTemporal();  
    EmpleadoTemporal e4 = new EmpleadoTemporal();  
  
    // Agregamos los empleados al array  
    empleados.add(e1);  
    empleados.add(e2);  
    empleados.add(e3);  
    empleados.add(e4);  
  
    int i = 0; // Variable para ver el índice en el for  
  
    // Recorremos el array de empleados y llamamos al método para  
    calcular sueldo  
    for(Empleado e : empleados) {  
        System.out.println("El empleado " + i + " cobra: " +  
e.calcularSueldo(e));  
        i++; // Incrementamos el índice  
    }  
}
```

Salida esperada

```
El empleado 0 cobra: 900000.0
El empleado 1 cobra: 900000.0
El empleado 2 cobra: 850000.0
El empleado 3 cobra: 850000.0
```



```
Run Main x
"C:\Program Files\Java\jdk-21\bin\java.exe"
El empleado 0 cobra: 900000.0
El empleado 1 cobra: 900000.0
El empleado 2 cobra: 850000.0
El empleado 3 cobra: 850000.0
```

Explicación del código

Este ejercicio muestra el uso de **instanceof** para identificar tipos en tiempo de ejecución:

1. **Operador instanceof:** Permite verificar si un objeto es una instancia de una clase específica. Retorna **true** o **false**.
2. **Diferenciación por tipo:** Aunque ambas subclases están vacías, se diferencian por su tipo. El método **calcularSueldo()** usa **instanceof** para determinar qué tipo de empleado es y asignar el sueldo correspondiente.
3. **Polimorfismo:** Podemos almacenar diferentes tipos de empleados en un mismo **ArrayList** de tipo **Empleado** (la clase padre).
4. **Limitación del diseño:** Este enfoque funciona pero tiene desventajas. Si agregamos más tipos de empleados, debemos modificar el método **calcularSueldo()**. Un mejor diseño sería que cada subclase implemente su propio método abstracto.

4.4 Ejercicio 4: Animales y Comportamiento Sobrescrito

Enunciado: Crear una clase **Animal** con métodos **hacerSonido()** y **describirAnimal()**. Crear tres subclases (**Perro**, **Gato**, **Vaca**) que sobrescriban **hacerSonido()** con su sonido característico.

Solución

Clase Animal

```
public class Animal {  
  
    // Método que será sobrescrito por cada animal  
    public void hacerSonido() {  
        // Implementación vacía en la clase base  
    }  
  
    public void describirAnimal() {  
        // Este método podría describir características generales  
    }  
}
```

Clase Perro

```
public class Perro extends Animal {  
  
    // Sobrescribimos el método hacerSonido  
    @Override  
    public void hacerSonido() {  
        System.out.println("Guaf!!");  
    }  
}
```

Clase Gato

```
public class Gato extends Animal {  
  
    // Sobrescribimos el método hacerSonido  
    @Override  
    public void hacerSonido() {  
        System.out.println("Miau!!");  
    }  
}
```

Clase Vaca

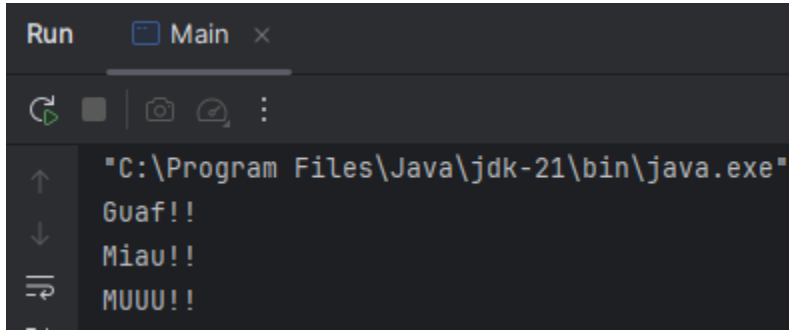
```
public class Vaca extends Animal {  
  
    // Sobrescribimos el método hacerSonido  
    @Override  
    public void hacerSonido() {  
        System.out.println("MUUU!!");  
    }  
}
```

Clase Main

```
public static void main(String[] args) {  
    // Inicializamos un array de animales  
    ArrayList<Animal> animales = new ArrayList<>();  
  
    // Creamos y agregamos animales al array  
    Perro p1 = new Perro();  
    Gato g1 = new Gato();  
    Vaca v1 = new Vaca();  
  
    animales.add(p1);  
    animales.add(g1);  
    animales.add(v1);  
  
    // Recorremos el array y llamamos al método hacerSonido  
    // Cada animal emite su propio sonido gracias al polimorfismo  
    for (Animal a : animales) {  
        a.hacerSonido();  
    }  
}
```

Salida esperada

```
Guaf!!  
Miau!!  
MUUU!!
```



```
Run Main x
Guaf!!
Miau!!
MUUU!!
```

Explicación del código

Este es el ejemplo clásico de polimorfismo:

1. **Sobrescritura de métodos:** Cada subclase (Perro, Gato, Vaca) sobrescribe el método `hacerSonido()` con su implementación específica. La anotación `@Override` asegura que estamos sobrescribiendo correctamente.
2. **Polimorfismo en acción:** En el main, el `ArrayList` es de tipo `Animal`, pero contiene objetos de diferentes tipos (Perro, Gato, Vaca). Cuando llamamos a `a.hacerSonido()`, Java automáticamente ejecuta la versión correcta del método según el tipo real del objeto.
3. **Ventaja principal:** Si queremos agregar más animales (por ejemplo, una clase `Oveja`), solo necesitamos crear la clase, extender de `Animal`, sobrescribir `hacerSonido()`, y funcionará automáticamente con el código existente. No necesitamos modificar el main ni usar `if/else`.
4. **Enlace dinámico:** La decisión de qué versión de `hacerSonido()` ejecutar se toma en tiempo de ejecución (runtime), no en tiempo de compilación. Esto se llama enlace dinámico o tardío.

4.5 Ejercicio 5: Sistema de Pagos con Interfaces

Enunciado: Crear una interfaz **Pagable** con un método **pagar()**. Implementar tres clases: **TarjetaCredito**, **Transferencia** y **Efectivo**. Crear un método genérico **procesarPago(Pagable medio)** que procese cualquier forma de pago.

Solución

Interfaz Pagable

```
public interface Pagable {  
    // Método que deben implementar todas las formas de pago  
    void pagar();  
}
```

Clase TarjetaCredito

```
public class TarjetaCredito implements Pagable {  
  
    // Implementamos el método de la interfaz  
    @Override  
    public void pagar() {  
        System.out.println("Pago realizado con tarjeta de credito");  
    }  
}
```

Clase Transferencia

```
public class Transferencia implements Pagable {  
  
    // Implementamos el método de la interfaz  
    @Override  
    public void pagar() {  
        System.out.println("Pago realizado con transferencia");  
    }  
}
```


Clase Efectivo

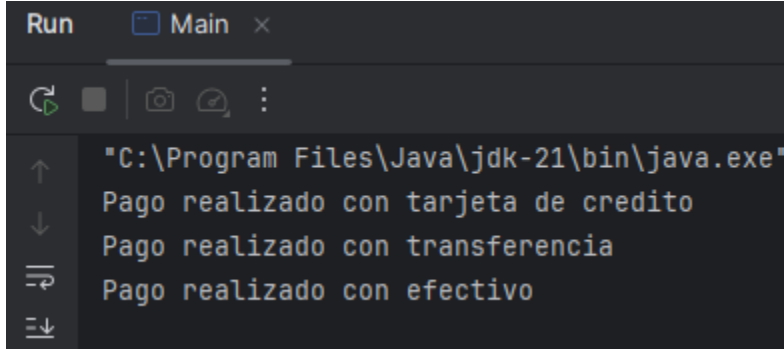
```
public class Efectivo implements Pagable {  
  
    // Implementamos el método de la interfaz  
    @Override  
    public void pagar() {  
        System.out.println("Pago realizado con efectivo");  
    }  
}
```

Clase Main

```
public static void main(String[] args) {  
    // Inicializamos un ArrayList del tipo Pagable  
    ArrayList<Pagable> formasDePago = new ArrayList<>();  
  
    // Agregamos 3 formas de pago al array  
    formasDePago.add(new TarjetaCredito());  
    formasDePago.add(new Transferencia());  
    formasDePago.add(new Efectivo());  
  
    // Recorremos el array y llamamos al método procesarPago  
    for (Pagable p : formasDePago) {  
        procesarPago(p);  
    }  
}  
  
/**  
 * Procesa un pago utilizando cualquier objeto que implemente la  
 * interfaz  
 * Pagable.  
 *  
 * @param medio el medio de pago a procesar. Puede ser una instancia  
 * de  
 * TarjetaCredito, Transferencia, Efectivo, u otra clase  
 * que implemente  
 * Pagable.  
 */  
public static void procesarPago(Pagable medio) {  
    medio.pagar();  
}
```

Salida esperada

```
Pago realizado con tarjeta de credito  
Pago realizado con transferencia  
Pago realizado con efectivo
```



```
Run Main x  
"C:\Program Files\Java\jdk-21\bin\java.exe"  
Pago realizado con tarjeta de credito  
Pago realizado con transferencia  
Pago realizado con efectivo
```

Explicación del código

Este ejercicio muestra el uso de interfaces para lograr polimorfismo:

1. **Interfaz como contrato:** **Pagable** define un contrato que dice "cualquier clase que me implemente debe tener un método **pagar()**". No importa cómo lo implementen, solo que lo tengan.
2. **Implementación de interfaz:** Las tres clases de pago implementan la interfaz usando **implements Pagable**. Esto las obliga a proporcionar una implementación del método **pagar()**.
3. **Método genérico:** **procesarPago(Pagable medio)** puede recibir cualquier objeto que implemente **Pagable**. No necesita saber si es **TarjetaCredito**, **Transferencia** o **Efectivo**. Solo necesita saber que tiene el método **pagar()**.
4. **Ventaja de las interfaces:** A diferencia de las clases abstractas, una clase puede implementar múltiples interfaces. Si tuviéramos otra interfaz **Reembolsable**, **TarjetaCredito** podría implementar ambas: **class TarjetaCredito implements Pagable, Reembolsable**.
5. **Extensibilidad:** Si mañana agregamos un nuevo método de pago (por ejemplo, **CriptoMoneda**), solo necesitamos que implemente **Pagable** y automáticamente funcionará con **procesarPago()** y el resto del código.

5. ANÁLISIS COMPARATIVO DE LOS CONCEPTOS APLICADOS

5.1 Herencia vs. Interfaces

Aspecto	Herencia (extends)	Interfaces (implements)
Cantidad	Una clase solo puede extender de una clase	Una clase puede implementar múltiples interfaces
Implementación	Puede incluir métodos implementados	Solo declara métodos (antes de Java 8)
Atributos	Puede tener atributos de instancia	Solo constantes (static final)
Relación	"es un/una" (is-a)	"puede hacer" (can-do)
Ejemplo en el TP	Auto extends Vehiculo	TarjetaCredito implements Pagable

5.2 Clases Concretas vs. Clases Abstractas

Característica	Clase Concreta	Clase Abstracta
Instanciación	Se puede instanciar directamente	No se puede instanciar
Métodos	Todos los métodos están implementados	Puede tener métodos abstractos
Propósito	Crear objetos funcionales	Servir como base para otras clases
Ejemplo en el TP	Perro, Gato, Vaca	Animal, Figura, Empleado

5.3 Sobrescritura vs. Sobrecarga

Aspecto	Sobrescritura (Override)	Sobrecarga (Overload)
Definición	Reemplazar método heredado	Múltiples métodos con mismo nombre
Parámetros	Mismo número y tipo	Diferente número o tipo
Jerarquía	Requiere herencia	En la misma clase
Anotación	@Override	No tiene anotación
Ejemplo en el TP	hacerSonido() en cada animal	No aplicado en este TP

5.4 Comparación de Uso de Polimorfismo

Ejercicio 2 (Figuras):

- Usa clase abstracta con método abstracto.
- Polimorfismo a través de herencia.
- Cada subclase implementa su propio cálculo.

Ejercicio 5 (Pagos):

- Usa interfaz.
- Polimorfismo a través de implementación de interfaz.
- Más flexible, permite múltiples interfaces.

Conclusión: Las interfaces son más flexibles cuando no hay código compartido; las clases abstractas son mejores cuando hay comportamiento común que reutilizar.

6. RESULTADOS OBTENIDOS

6.1 Compilación y Ejecución

Todos los ejercicios fueron compilados y ejecutados exitosamente, obteniendo los siguientes resultados:

Ejercicio 1: Vehículos

- ✓ Herencia básica implementada correctamente.
- ✓ Sobrescritura de método **mostrarInfo()** funcional.
- ✓ Uso correcto de **super()** en constructor.
- **Salida:** "Modelo: Sandero ,marca: Renault, cantidad de puertas: 5".

Ejercicio 2: Figuras Geométricas

- ✓ Clase abstracta correctamente definida.
- ✓ Métodos abstractos implementados en subclases.
- ✓ Polimorfismo funcionando con **ArrayList<Figura>**.
- **Salidas:** Cálculo correcto de áreas para 2 rectángulos y 2 círculos.

Ejercicio 3: Empleados

- ✓ Uso correcto de **instanceof** para diferenciación de tipos.
- ✓ Cálculo de sueldos según tipo de empleado.
- ✓ **ArrayList** polimórfico funcionando.
- **Salidas:** 2 empleados con sueldo 900000.0 y 2 con 850000.0.

Ejercicio 4: Animales

- ✓ Sobrescritura de métodos con **@Override**.
- ✓ Polimorfismo en acción con diferentes animales.
- ✓ Cada animal emite su sonido específico.
- **Salidas:** "Guaf!!", "Miau!!", "MUUU!!".

Ejercicio 5: Sistema de Pagos

- ✓ Interfaz correctamente definida e implementada.
- ✓ Método genérico **procesarPago()** funcional.
- ✓ Tres formas de pago procesadas correctamente.
- **Salidas:** Confirmación de pago por cada método.

6.2 Verificación de Conceptos

- ✓ **Herencia:** Aplicada en los 5 ejercicios.
- ✓ **Polimorfismo:** Demostrado en ejercicios 2, 3, 4 y 5.
- ✓ **Clases abstractas:** Usadas en ejercicios 2 y 3.
- ✓ **Interfaces:** Implementada en ejercicio 5.
- ✓ **Sobrescritura:** Presente en ejercicios 1, 2, 4 y 5.
- ✓ **instanceof:** Aplicado en ejercicio 3.
- ✓ **Modificadores de acceso:** Usados en todos los ejercicios.

7. CONCLUSIONES

A través de estos cinco ejercicios hemos aplicado exitosamente los conceptos fundamentales de Herencia y Polimorfismo:

Aprendizajes clave:

1. Herencia (Ejercicios 1-4):

- Permite reutilizar código de una clase padre.
- Se utiliza **extends** para crear la relación.
- El constructor de la clase hija debe llamar a **super()** para inicializar la clase padre.
- Los atributos **protected** son accesibles desde las clases hijas.

2. Clases y métodos abstractos (Ejercicios 2-3):

- Una clase abstracta no puede ser instanciada directamente.
- Los métodos abstractos obligan a las subclases a proporcionar una implementación.
- Son útiles para definir estructuras base que comparten comportamiento común.

3. Polimorfismo (Ejercicios 2-5):

- Permite tratar objetos de diferentes clases de manera uniforme.
- La decisión de qué método ejecutar se toma en tiempo de ejecución.
- Facilita la extensibilidad del código sin modificar el código existente.
- La anotación `@Override` ayuda a prevenir errores.

4. Interfaces (Ejercicio 5):

- Definen un contrato que las clases deben cumplir.
- Permiten polimorfismo sin necesidad de herencia.
- Una clase puede implementar múltiples interfaces.

5. instanceof (Ejercicio 3):

- Permite verificar el tipo de un objeto en tiempo de ejecución.
- Útil pero debe usarse con moderación (preferir polimorfismo).

Beneficios de aplicar estos conceptos:

- **Reutilización de código:** No repetimos código, lo heredamos.
- **Mantenibilidad:** Los cambios en la clase padre se reflejan en las hijas.
- **Flexibilidad:** Podemos agregar nuevas clases sin modificar código existente.
- **Abstracción:** Ocultamos detalles de implementación y exponemos interfaces claras.

Estos principios son fundamentales para crear aplicaciones robustas, escalables y fáciles de mantener en Java y en la Programación Orientada a Objetos en general.

Reflexión Final

La realización de este trabajo práctico ha permitido consolidar los conocimientos teóricos adquiridos en clase mediante su aplicación práctica. Cada ejercicio presenta un nivel de complejidad creciente, permitiendo comprender progresivamente desde conceptos básicos de herencia hasta aplicaciones más avanzadas con interfaces y polimorfismo.

El uso de las 5 katas propuestas resulta efectivo para el aprendizaje, ya que cada una enfoca un aspecto específico de los conceptos de herencia y polimorfismo, facilitando su comprensión individual antes de integrarlos en sistemas más complejos.

8. BIBLIOGRAFÍA

Documentación Oficial:

- Oracle. (2023). *The Java Tutorials - Inheritance*.
<https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html>
- Oracle. (2023). *The Java Tutorials - Polymorphism*.
<https://docs.oracle.com/javase/tutorial/java/landl/polymorphism.html>
- Oracle. (2023). *The Java Tutorials - Abstract Methods and Classes*.
<https://docs.oracle.com/javase/tutorial/java/landl/abstract.html>
- Oracle. (2023). *The Java Tutorials - Interfaces*.
<https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html>

Material de la Cátedra:

- UTN - Tecnicatura Universitaria en Programación. (2024). *Trabajo Práctico 7: Herencia y Polimorfismo en Java*. Material de estudio de Programación II.