**Nombre Alumno:** Kruchowski, Juan Ignacio   
**Nombre Profesores:** Cristina Greiner - Yanina Medina - Andrea Airaldi  
**Grupo Laboratorio:** 2   
**DNI:** 45.760.991   
**TP:** 5   
**Fecha de entrega: 20/10/2025**

### **Introducción a la Herencia y el Polimorfismo**

El presente trabajo práctico tuvo como objetivo principal la aplicación de conceptos avanzados de la Programación Orientada a Objetos, específicamente la **herencia** y el **polimorfismo**. Estos pilares permiten la creación de software flexible, reutilizable y de alta calidad, estableciendo relaciones jerárquicas entre clases y permitiendo que objetos de diferentes tipos respondan a un mismo mensaje de formas distintas.

A lo largo de los ejercicios, se exploró cómo la herencia facilita la reutilización de código al definir superclases que encapsulan atributos y comportamientos comunes, y cómo el polimorfismo, a través de la redefinición de métodos, permite especializar dicho comportamiento en las subclases.

### **Caso 1: Especialización de Clases - Rectangulo y Cuadrado**

Un claro ejemplo de especialización se trabajó con las clases Rectangulo y Cuadrado. Dado que un cuadrado es un caso particular de un rectángulo (donde ancho y alto son iguales), se aplicó herencia para que Cuadrado extendiera a Rectangulo.

Esto permitió a Cuadrado heredar automáticamente atributos como origen, ancho y alto, así como métodos para calcular superficie() y perimetro(). Sin embargo, se **redefinió** el método caracteristicas() en la clase Cuadrado para mostrar la información de una manera más específica a su naturaleza, haciendo referencia a "Lado" en lugar de "Alto" y "Ancho". Este es un ejemplo práctico de polimorfismo, donde un mismo método (caracteristicas()) se comporta de manera diferente según el objeto que lo invoca.

### **Caso 2: Generalización y Clases Abstractas - FiguraGeometrica**

En el desarrollo de la gestión de figuras para el jardín, se identificó la necesidad de tratar a todas las figuras (Cuadrado, Circulo, Triangulo, Elipse) de una manera uniforme. Para lograr esto, se aplicó el principio de generalización, creando una superclase abstracta FiguraGeometrica.

Esta clase abstracta definió una interfaz común para todas las figuras, incluyendo:

Un atributo origen de tipo Punto.

Métodos abstractos como superficie() y nombreFigura().

Al declarar estos métodos como abstract, se obliga a que cada subclase concreta proporcione su propia implementación, garantizando que cualquier objeto que sea una FiguraGeometrica sepa cómo calcular su superficie y reportar su nombre. Esto permitió gestionar todas las figuras en una única colección (ArrayList<FiguraGeometrica>) y operar sobre ellas de forma polimórfica, simplificando enormemente el código del Jardin.

### **Caso 3: Polimorfismo en el Cálculo de Costos - Alojamiento**

El ejercicio de la gerencia de alojamientos (Hotel y Cabaña) es un excelente ejemplo del poder del polimorfismo. Se definió una clase abstracta Alojamiento que declaraba un método costo().

Tanto la clase Hotel como la clase Cabaña heredaron de Alojamiento y **sobrescribieron** el método costo() para implementar su propia lógica de cálculo:

**Hotel:** Añade un adicional según si la habitación es "Single" o "Doble".

**Cabaña:** Añade un adicional por cada habitación que posee.

Esto permitió que la clase Gerencia pudiera simplemente invocar el método costo() sobre cualquier objeto de tipo Alojamiento en su colección, sin necesidad de saber si se trataba de un hotel o una cabaña. El sistema, gracias al enlace dinámico, se encarga de ejecutar la versión correcta del método en tiempo de ejecución, resultando en un código más limpio, mantenible y extensible.

### **Conclusión**

Este trabajo práctico ha sido fundamental para consolidar la comprensión de la herencia y el polimorfismo como herramientas clave para el diseño de software orientado a objetos. La capacidad de crear jerarquías de clases, reutilizar código a través de superclases (abstractas o no) y especializar el comportamiento en subclases mediante la redefinición de métodos, son técnicas que no solo mejoran la organización del código, sino que también sientan las bases para construir sistemas complejos de manera más eficiente y robusta.