🧩 Experiencia de Aprendizaje 2

🎯 Objetivo de este documento

Comprender y aplicar los mecanismos de **reutilización de código** y **gestión dinámica de objetos** mediante:

- **Herencia y jerarquías de clases**.
- **Sobrecarga y sobrescritura de métodos.**
- **Uso de colecciones (`ArrayList`).**
- **Polimorfismo e interfaces**.

🧱 Conceptos Clave

Tema	Descripción
Herencia (`extends`)	Permite crear nuevas clases que heredan atributos y métodos de una clase base, promoviendo la reutilización de código y la especialización.
Superclase y Subclase	La superclase define el comportamiento general; las subclases lo extienden o modifican con sus propios atributos/métodos.
Constructores y `super()`	Las subclases pueden invocar constructores de la superclase para inicializar atributos heredados.
Sobrecarga (`overload`)	Varios métodos con el mismo nombre pero diferentes parámetros.
Sobrescritura (`override`)	Una subclase redefine un método de su superclase para cambiar su comportamiento.
Palabras clave	`extends`, `super`, `@Override`, `this`.
Colecciones (`ArrayList`)	Estructuras dinámicas que almacenan objetos (por tipo genérico `ArrayList<T>`), permitiendo CRUD y recorrido mediante bucles.
Polimorfismo	Permite tratar objetos de distintas subclases como instancias de una superclase común, ejecutando su versión específica del método sobrescrito.
Interfaces (`implements`)	Contratos que obligan a implementar ciertos métodos, usados para comportamientos comunes entre clases no relacionadas.

🔗 Repositorio GitHub: https://github.com/profe-robert/java-ant-poo

El proyecto Java Ant POO es una aplicación educativa desarrollada en Java con Apache NetBeans (Ant) para ilustrar los principios fundamentales de la Programación Orientada a Objetos (POO). Incluye ejemplos prácticos de herencia, polimorfismo, sobrecarga, sobrescritura, interfaces y colecciones (`ArrayList`), organizados en una arquitectura en capas con paquetes de modelo, repositorio, servicio y aplicación. Está pensado como material de apoyo para la Experiencia de Aprendizaje 2, permitiendo a los estudiantes explorar el diseño modular, el uso de patrones DAO y Service Layer, y la implementación de CRUDs con orientación al dominio.

A continuación una guía de los conceptos clave aplicados al proyecto compartido en GitHub:

🧱 Herencia, `super()` y `@Override`

- **Clases:**
  `modelo/Producto` ← **superclase**
  `modelo/Electronico` y `modelo/Ropa` ← **subclases**

- **Qué mirar:**
  - Constructores de `Electronico`/`Ropa` llaman a `super(...)`.
  - `toString()` **sobrescrito** (`@Override`) en subclases.

- **Para probar:** listar productos desde el menú (consola) y ver la salida especializada.

🔁 Sobrecarga (overload)

- **Dónde:** en `Producto` puedes tener **métodos con el mismo nombre** y distinta firma, p. ej.:
  `calcularDescuento(double porcentaje)` y `calcularDescuento(double porcentaje, boolean redondear)`.

- **Para probar:** invoca ambos desde `servicio/ProductoServiceImpl` o un pequeño snippet en `app/Consola`.

📚 Colecciones (`ArrayList<T>`) y CRUD

- **Clases:**
  `repositorio/RepositorioProductos` o `InMemoryProductoRepository` (según la variante que adoptaste)
  `servicio/ServicioProductos`

- **Qué mirar:**
  - Atributo `List<Producto> productos = new ArrayList<>();`
  - **CRUD**: `agregar`, `buscarPorId`, `eliminar`, `listar`, `total`.

- **Extras recomendados:** usar `Optional<Producto>` en `buscarPorId` y validar duplicados.

🧠 Polimorfismo (en tiempo de ejecución)

- **Clases:**
  `modelo/Empleado` (abstracta/base)
  `modelo/EmpleadoAsalariado`, `modelo/EmpleadoPorHora` (subtipos)

- **Colección polimórfica:**
  `servicio/ServicioEmpleados` → `List<Empleado> empleados`

- **Qué mirar:**
  - Llamadas como `e.calcularSalario()` ejecutan la **versión específica** del subtipo.

- **Para probar:** opción de menú que **lista salarios** y **totales**.

📄 Interfaces (contratos comunes)

- **Clase:** `modelo/Bonificable`

- **Implementaciones:** `EmpleadoAsalariado`, `EmpleadoPorHora`

- **Uso polimórfico por interfaz:**
  En `ServicioEmpleados.bonusTotal()`, se itera sobre `List<Empleado>` y se usa `instanceof Bonificable` (o *pattern matching*) para sumar `calcularBonus()` **sin importar el subtipo**.

✅ Validaciones y calidad de dominio

- **Dónde agregar/ver:**
    - Validaciones en constructores/setters de `Producto`/`Empleado` (p. ej., precio ≥ 0).
    - `equals`/`hashCode` por `id` en `Producto`/`Empleado` (mejor comportamiento en colecciones).
- **Impacto:** evita duplicados y garantiza invariantes.

🌟 Mensaje del profesor

Felicitaciones por llegar hasta esta etapa del curso.
Cada concepto, ejercicio y proyecto que han desarrollado en esta experiencia representa un paso más hacia su crecimiento como **futuros profesionales del área tecnológica**.

Los animo a seguir explorando, probando ideas nuevas y enfrentando los desafíos que vienen con la misma curiosidad y constancia que demostraron hasta ahora.
Recuerden que **programar no solo es escribir código**, sino aprender a **pensar en soluciones, trabajar en equipo y construir conocimiento**.

Les deseo el mayor de los éxitos en las próximas evaluaciones y, sobre todo, en su camino profesional.
¡Sigan creciendo, creando y creyendo en su potencial! 🚀

– Roberto