TECNICATURAUNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

PROGRAMACIÓN II

Trabajo Práctico 6: Colecciones y Sistema de Stock

Autor: IACONO, Federico

OBJETIVO GENERAL

Desarrollar estructuras de datos dinámicas en Java mediante el uso de colecciones (**ArrayList**) y enumeraciones (**enum**), implementando un sistema de stock con funcionalidades progresivas que refuerzan conceptos clave de la programación orientada a objetos..

MARCO TEÓRICO

Concepto	Aplicación en el proyecto
ArrayList	Estructura principal para almacenar productos en el inventario.
Enumeraciones (enum)	Representan las categorías de productos con valores predefinidos.
Relaciones 1 a N	Relación entre Inventario (1) y múltiples Productos (N).
Métodos en enum	Inclusión de descripciones dentro del enum para mejorar legibilidad.
Ciclo for-each	Recorre colecciones de productos para listado, búsqueda o filtrado.
Búsqueda y filtrado	Por ID y por categoría, aplicando condiciones.
Ordenamientos y reportes	Permiten organizar la información y mostrar estadísticas útiles.
Encapsulamiento	Restringir el acceso directo a los atributos de una clase

Consideraciones Previas y Diseño de la Solución

Para la resolución del Trabajo Práctico 6, se ha realizado un análisis de los tres ejercicios propuestos, enfocándose en la Programación Orientada a Objetos (POO) y el manejo de colecciones en Java.

El diseño de la solución se basa en las siguientes consideraciones:

1. Estructura General y Encapsulamiento

- **Estructura de Paquetes y Clases:** Cada ejercicio (Stock, Biblioteca, Universidad) se organizará en su propio paquete (o conjunto de archivos). Cada clase (`Producto`, `Inventario`, `Libro`, `Autor`, `Biblioteca`, `Profesor`, `Curso`, `Universidad`) se implementará en su propio archivo `.java`.
- **Clases de Prueba:** Se creará una clase `Main` (ej. `MainStock`, `MainBiblioteca`, `MainUniversidad`) para cada ejercicio, la cual será responsable de instanciar los objetos y ejecutar la lista de "Tareas a realizar" especificadas en el documento.
- **Encapsulamiento:** Todos los atributos de las clases de modelo (ej. `Producto`, `Libro`, `Profesor`) se declararán como `private`. El acceso y la modificación de estos atributos se gestionarán a través de métodos públicos (getters, setters) o métodos de lógica de negocio (ej. `agregarProducto`), respetando el principio de encapsulamiento.

2. Diseño Específico del Caso 1: Sistema de Stock

- **Relación:** Se implementa una relación 1 a N unidireccional. La clase `Inventario` (el "1") gestionará una colección de `Producto` (el "N").
- **Colección:** Se utilizará un `ArrayList<Producto>` dentro de la clase `Inventario` para almacenar los productos.
- **Lógica Centralizada:** La clase `Inventario` será la clase "gestora" (manager). Contendrá toda la lógica de negocio: `agregarProducto`, `listarProductos`, `buscarProductoPorId`, `filtrarPorCategoria`, etc. . La clase `Producto` será una clase de datos (POJO) con su método `mostrarInfo()`.
- **Enumeración:** Se creará el `enum CategoriaProducto` tal como se especifica, incluyendo el constructor privado y el método `getDescripcion()` para asociar un texto a cada valor.

3. Diseño Específico del Caso 2: Biblioteca y Libros

- **Relación (Composición):** El ejercicio describe una relación de **composición 1 a N**. La `Biblioteca` (el "1") contiene múltiples `Libros` (el "N"). La existencia de los `Libros` depende de la `Biblioteca`.
- **Lógica de Creación:** A diferencia del primer caso, el método `agregarLibro` de la clase `Biblioteca` no recibirá un objeto `Libro`, sino los atributos necesarios para *crearlo* (`isbn`, `titulo`, `anio`, `autor`). Esto refuerza la idea de composición, donde la `Biblioteca` es responsable de la creación (y ciclo de vida) de sus `Libros`.

- **Relación (Agregación):** La relación entre `Libro` y `Autor` es de agregación. El `Autor` es un objeto independiente que es *referenciado* por el `Libro`. Los autores se crearán por separado y se pasarán como parámetro al crear un libro.

4. Diseño Específico del Caso 3: Universidad y Cursos

Este es el caso más complejo, ya que requiere mantener la integridad de una **asociación bidireccional 1 a N** .

- **Relación:**

1. Un `Profesor` tiene una `List<Curso>` (los cursos que dicta).
2. Un `Curso` tiene una referencia a un único `Profesor` (el profesor responsable).

- **El Desafío (Invariante de Asociación):** El punto crítico es mantener la consistencia de los datos en *ambos lados* de la relación. Si un curso se asigna a un profesor, la lista del profesor debe actualizarse, y la referencia del curso también.
- **Diseño de Sincronización:** Para garantizar esto, la lógica de sincronización se encapsulará en las clases del modelo:

1. La responsabilidad principal recaerá en el método `setProfesor(Profesor p)` de la clase `Curso`.
2. Cuando se llame a `setProfesor(nuevoProfesor)`:

   - El curso primero debe desvincularse del `profesorAnterior` (si existe), accediendo a la lista de cursos de ese profesor y eliminándose a sí mismo.
   - Luego, el curso asignará la referencia al `nuevoProfesor`.
   - Finalmente, el curso se agregará a sí mismo a la `List<Curso>` del `nuevoProfesor`.

3. Los métodos `agregarCurso` y `eliminarCurso` de la clase `Profesor` simplemente llamarán a `curso.setProfesor(this)` o `curso.setProfesor(null)` para delegar la sincronización.

- **Rol de la Universidad:** La clase `Universidad` actuará como gestora de las listas maestras de cursos y profesores. Su método `asignarProfesorACurso` simplemente buscará los objetos y llamará a `curso.setProfesor(profesor)`, delegando la lógica de sincronización.

Link a repositorio: GitHub