Informe: Aplicación de la Programación Orientada a Objetos en los Ejercicios 2 y 3

En los ejercicios 2 y 3 se desarrollaron dos programas diferentes aplicando los principios de la Programación Orientada a Objetos (POO). A través de ambos se pueden identificar conceptos como encapsulamiento, abstracción, modularidad y, en menor medida, composición. En este informe se explica cómo se aplicaron estos principios en cada caso, con ejemplos concretos del código, y también se aclara por qué no se usaron otros como la herencia o el polimorfismo.

Ejercicio 2: Calculadora orientada a objetos

En este ejercicio se creó una calculadora básica en TypeScript, con un menú que permite al usuario realizar operaciones como suma, resta, multiplicación y división. La lógica principal está dentro de una clase llamada Calculadora, que tiene un método para cada operación, y el programa principal (index.ts) se encarga de mostrar el menú e interactuar con el usuario.

El encapsulamiento se puede ver claramente en cómo cada operación está definida dentro de la clase. Por ejemplo, para realizar una suma, simplemente se llama al método correspondiente y se obtiene el resultado, sin necesidad de saber qué pasa internamente. Lo mismo ocurre con la división, que tiene una validación para evitar dividir por cero, la cual está dentro del propio método.

La abstracción también se aplica, ya que la clase Calculadora representa la idea general de una calculadora sin mostrar detalles innecesarios. El usuario o el programa principal solo ven los métodos disponibles (sumar, restar, multiplicar y dividir), mientras que la lógica interna y el manejo de errores quedan ocultos dentro de la clase.

Por otro lado, hay un uso claro de la modularidad, porque el código está dividido en archivos distintos: uno para la clase Calculadora y otro para la parte ejecutable. Esto hace que el programa sea más fácil de mantener y reutilizar.
Por ejemplo, la misma clase se podría usar en otro proyecto sin tener que copiar todo el código principal.

En este caso no se utilizó herencia, ya que no se crearon clases derivadas ni relaciones entre clases. Sin embargo, el diseño modular y orientado a objetos permitió mantener un código limpio, bien estructurado y preparado para futuras ampliaciones, como podría ser una calculadora científica.

Ejercicio 3: Gestor de tareas con varias clases

El ejercicio 3 fue más completo y aplicó varios conceptos de POO a un sistema de consola que permite agregar, editar, buscar, ver y eliminar tareas.
Para organizar el código, se crearon varias clases con responsabilidades bien definidas: Tarea, GestorTareas y Aplicacion.

La clase Tarea representa una tarea individual, con atributos como título, descripción, fecha de creación, estado y dificultad. Además, tiene métodos propios para mostrar y editar la información. Esto es un claro ejemplo de encapsulamiento, ya que toda la lógica relacionada con una tarea está dentro de esa clase, y el resto del programa solo la usa sin modificar su funcionamiento interno.

La abstracción se nota en cómo otras partes del sistema trabajan con las tareas. Por ejemplo, si se quiere editar una tarea, no se accede directamente a sus atributos, sino que se llama a un método que se encarga de hacerlo correctamente y con validaciones.

La clase GestorTareas se ocupa de manejar el conjunto de tareas: agregar, eliminar, editar y guardar la información en un archivo JSON.
Esto refuerza tanto la abstracción como la modularidad, ya que la aplicación no trabaja directamente con archivos, sino que lo hace a través de esta clase, que simplifica todo el proceso.

Por último, la clase Aplicacion es la que muestra el menú, pide los datos al usuario y coordina la interacción entre las demás clases.
No realiza cálculos ni gestiona archivos, sino que organiza cómo todo funciona en conjunto. Dentro de su constructor crea una instancia del GestorTareas, lo cual muestra un uso de la composición, ya que combina varias clases para lograr una funcionalidad más completa sin necesidad de herencia.

En este ejercicio tampoco fue necesario usar herencia ni polimorfismo, ya que todas las tareas tienen el mismo tipo y comportamiento. Sin embargo, si en el futuro se quisiera ampliar el sistema con tareas automáticas, recurrentes o con alarmas, sí se podrían aplicar estos principios para extender la funcionalidad.