EJERCICIO 5

En este cuaderno de ejercicios hemos explorado la persistencia de datos mediante el uso del formato JSON en el lenguaje de programación Java, utilizando la biblioteca Gson. Este enfoque se basa en una conversión explícita entre objetos Java y su representación en texto estructurado (JSON), el cual puede ser almacenado en archivos o transmitido a través de la red.

Pero no es la única forma, Java dispone desde sus primeras versiones de un mecanismo nativo de serialización que permite transformar un objeto (y su estado completo) en una secuencia de bytes. Esta secuencia puede ser almacenada directamente o enviada, y posteriormente reconstruida para recuperar el objeto original. Vamos a explorar ese enfoque tradicional, analizar su funcionamiento y compararlo con el uso de JSON mediante la librería Gson.

En primer lugar, una breve explicación sobre la serialización y la serialización nativa. La serialización es un proceso fundamental en la programación orientada a objetos y consiste en convertir un objeto en memoria , es decir, una instancia de una clase con sus atributos y estado actuales, en una forma que pueda ser almacenada o transmitida. Esta transformación convierte el objeto en una secuencia de bytes que puede guardarse en un archivo, enviarse a través de una red, almacenarse en una base de datos o simplemente conservarse para ser recuperada en un momento posterior. La operación inversa se llama deserialización, y su propósito es reconstruir el objeto original a partir de los datos serializados. Este mecanismo es muy útil porque permite mantener el estado de un programa entre ejecuciones o compartir objetos entre diferentes partes del sistema o incluso entre sistemas diferentes si se utiliza un formato de intercambio.

En Java, la serialización es una característica integrada desde las primeras versiones del lenguaje. Para que un objeto pueda ser serializado, su clase debe implementar la interfaz java.io.Serializable, que es una interfaz marcador, es decir, no contiene métodos que se deban implementar, sino que simplemente indica al sistema de entrada/salida de Java que ese objeto puede ser convertido a bytes. Una vez que la clase implementa esta interfaz, es posible usar las clases ObjectOutputStream y ObjectInputStream para escribir y leer esos objetos, respectivamente.

Este mecanismo, conocido como serialización nativa de Java, es el método estándar proporcionado por el propio lenguaje para transformar un objeto que implemente la interfaz Serializable en una secuencia binaria de datos. A diferencia de otros métodos de persistencia, como JSON, esta forma de serialización es completamente interna y específica de Java, lo que significa que solo puede ser comprendida y utilizada dentro del ecosistema del lenguaje. El proceso es automático: una vez que el objeto es marcado como serializable, todos sus atributos también son serializados, siempre y cuando ellos mismos sean serializables. En caso de que un atributo no lo sea —por ejemplo, una conexión de red activa o una interfaz gráfica— debe ser declarado como transient para que sea ignorado durante la serialización. Esta técnica es útil para

guardar el estado de una aplicación entre sesiones, enviar objetos entre distintos procesos Java o realizar copias rápidas de objetos en memoria. Su principal ventaja es la simplicidad de uso y el bajo esfuerzo de implementación, pero también presenta desventajas importantes, como la falta de interoperabilidad con otros lenguajes y la fragilidad ante cambios en la estructura de las clases.

Uno de los aspectos más críticos al utilizar la serialización, especialmente la nativa de Java, es la seguridad. Aunque puede parecer una operación inocua, la deserialización de objetos desde fuentes no confiables representa un riesgo real y frecuente en muchas aplicaciones. Esto se debe a que, al deserializar, el sistema reconstruye objetos directamente desde los datos binarios sin validación estricta sobre su contenido, lo cual abre la posibilidad de que se inyecte código malicioso a través de un archivo de entrada especialmente manipulado. De hecho, existen ataques conocidos que aprovechan la deserialización para ejecutar código arbitrario o para provocar fallos deliberados en los sistemas.

La seguridad es particularmente problemática cuando se reciben datos serializados a través de la red. Por ejemplo, si un servidor acepta objetos serializados desde un cliente sin verificar su procedencia, es posible que se deserialice un objeto que intente ejecutar código en el servidor o acceder a recursos restringidos. Esta vulnerabilidad ha sido aprovechada en múltiples escenarios y ha llevado a que muchos expertos en seguridad recomienden evitar completamente la serialización nativa en aplicaciones distribuidas o accesibles públicamente.

Por este motivo, se han propuesto alternativas más seguras, como formatos de intercambio de datos más estructurados y predecibles, por ejemplo, JSON, XML o incluso protocolos binarios como Protobuf o Avro, que requieren que el desarrollador defina de manera explícita qué datos se envían y cómo deben ser interpretados. Estas alternativas no solo ofrecen mayor seguridad, sino también mejor interoperabilidad entre sistemas de distintos lenguajes o plataformas.

En respuesta a estas vulnerabilidades, a partir de Java 9 se introdujeron mecanismos de filtrado de clases deserializables que permiten controlar qué tipos de objetos pueden reconstruirse desde un flujo de datos serializados. Aun así, la recomendación general sigue siendo evitar la serialización nativa en entornos donde la seguridad sea una prioridad.

A continuación, se presentan las principales diferencias entre ambos enfoques:

Característica	Serialización nativa (Serializable)	JSON con Gson
Formato	Binario	Texto (legible por humanos)
Portabilidad	Solo en Java	Multilenguaje
Legibilidad	No legible sin herramientas	Legible y editable
Compatibilidad con versiones	Frágil si cambia la clase	Más flexible con cambios
Tamaño de archivo	**Más compacto**	**Más grande debido al texto**
Requisitos	Implementar Serializable, mantener serialVersionUID	Requiere biblioteca externa (Gson)
Control de serialización	A través de métodos especiales (writeObject, readObject)	Totalmente manual, basado en atributos públicos o getters/setters
Transparencia para el programador	Transparente (automático)	Explícita (declarativa)
Velocidad de procesamiento	Rápida	Algo más lenta
Soporte para estructuras complejas	Completo (con restricciones)	Bueno pero necesita configuración extra para tipos genéricos o anidados