6️⃣

GSON—> TypeAdapter

0. Introducción: Typeadapter

Gson permite personalizar la serialización y deserialización mediante el método `registerTypeAdapter(Type tipo, Object tipoDeAdaptador)`. Con este método, se pueden registrar varios adaptadores:

- **Adaptadores de tipo:** `TypeAdapter` Personaliza la adaptación de tipos, implementando los métodos `write()` y `read()`.

- **Creadores de instancia:** `InstanceCreator<T>` Crea instancias de clases que no tienen constructor por defecto.
- **Serialización y deserialización personalizada:** `JsonSerializer<T>` y `JsonDeserializer<T>` Interfaces que permiten personalizar la serialización y deserialización de JSON.

> ! Nota: Si se registra un adaptador para un tipo específico, sobrescribirá cualquier registro anterior.

---

1. Soporte de versiones en Gson: Since y Until

Gson permite controlar qué atributos de una clase se serializan o deserializan según la versión utilizando las anotaciones `@Since` y `@Until`.

- `@Since(x.x)` : Indica que un campo debe incluirse en versiones iguales o superiores a la especificada.
- `@Until(x.x)` : Indica que un campo se incluye hasta la versión especificada.

Ejemplo:

```
public class Persoa {
    @Since(1.0) public String nome;
    @Since(1.0) public String apelidos;
    @Until(2.0) public String cidade;
    @Since(3.0) public String email;
}
```

Para crear un `GsonBuilder` que admita versiones:

```
GsonBuilder builder = new GsonBuilder();
builder.setVersion(2.0);
Gson gson = builder.create();
```

---

2. Creación de objetos personalizados en Gson: `InstanceCreator`

Gson usa el **constructor por defecto** de la clase para crear una nueva instancia. Sin embargo, si tu clase no tiene un constructor sin parámetros o si deseas inicializar

objetos con ciertos valores predeterminados, puedes utilizar `InstanceCreator`.

Ejemplo:

- **Clase Poeta**

```
public class Poeta {
    private String nombre;
    private int edad;

    // Constructor con parámetros, pero no hay constructor por defecto
    public Poeta(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

- **Implementación de** `InstanceCreator`

```
import com.google.gson.InstanceCreator;

public class PoetaInstanceCreator implements InstanceCreator<Poeta> {
    @Override
    public Poeta createInstance(Type tipo) {
        // Se puede inicializar el objeto de forma personalizada
        return new Poeta("Poeta por defecto", 25);  // Ejemplo con valores predetermin
ados
    }
}
```

- **Registro del** `InstanceCreator` **en** `GsonBuilder`

```
GsonBuilder gsonBuilder = new GsonBuilder()
                                        .registerTypeAdapter(Poeta.class, new PoetaInstance
                                        .create();
```

---

3. Serialización y Deserialización personalizadas

🗣️ Los serializadores personalizados pueden convertir valores Java a JSON personalizado, y los deserializadores personalizados pueden convertir JSON personalizado de nuevo a valores Java.

3.1. Serializador personalizado: `JsonSerializer`

Permite convertir valores Java a un formato JSON personalizado. Se debe implementar la interfaz `JsonSerializer<T>`.

▼ Subclases de j**sonElement**

💡 **JSONElement es una clase abstracta que representa un elemento JSON**. Subclases de JSONElement son:

- `JsonArray`: representa un array JSON, Podemos añadir elementos a un JsonArray con el método `add()` y obtener un elemento con el método `get(int i)`. También es posible obtener el array como un único elemento Java si contiene un único elemento: `getAsBoolean()`, `getAsCharacter()`, `getAsDouble()`, `getAsFloat()`, `getAsInt()`, `getAsString()`, etc.
- `JsonNull` / `JsonNull.INSTANCE`: representa un valor nulo en JSON.
- `JsonObject`: representa un objeto JSON. Podemos añadir elementos a un JsonObject con el método `add(String property, JsonElement value)` o `addProperty(String property, T value)` y obtener un elemento con el método `get(String nombreMiembro)` o como Array, Objeto y tipo primitivo con los métodos `getAsJsonArray()`, `getAsJsonObject()`, `getAsJsonPrimitive()`.
- `JsonPrimitive`, que son Boolean, Character, Number o String y permite crear un JSON primitivo: `new JsonPrimitive(1)`, `new JsonPrimitive("Wittgenstein")`, `new JsonPrimitive(true)`, `new JsonPrimitive('a')`.

Ejemplo:

```
public class BooleanSerializer implements JsonSerializer<Boolean> {
    public JsonElement serialize(Boolean valor, Type tipo, JsonSerializationContext cont
        return new JsonPrimitive(valor ? 1 : 0);  // true → 1, false → 0
    }
}
```

▼ **Explicación** `JsonSerializer<T>`

Esta interfaz se utiliza para convertir un objeto Java a su representación en formato JSON. Implementa el método:

Parámetros del método `serialize` :

- `T src` :

  Este es el objeto que estamos serializando (en tu caso, un objeto `Boolean`). `valor` es el valor que será convertido a JSON.

- `Type tipo` :

  Representa el tipo genérico del objeto que se está serializando. En la mayoría de los casos, será la clase del objeto (`BooleanSerializer.class`), pero este parámetro es útil cuando se trabaja con tipos genéricos como listas, mapas o clases parametrizadas.

- `JsonSerializationContext context` :

  👣 Contextos JSON

  Este contexto te permite delegar la serialización de campos o subcampos a la instancia de Gson. Si tienes objetos dentro del objeto que estás serializando, puedes usar este contexto para serializar esos subobjetos sin tener que implementarlo manualmente. Es decir, si quieres que otros campos del objeto se serialicen automáticamente, puedes llamar a `context.serialize(subcampo)`.

Retorno:

- `JsonElement` :Este es el resultado de la serialización: una representación en formato JSON del objeto que puede ser una instancia de `JsonPrimitive` (para valores simples como strings o números), `JsonObject` (para objetos), o `JsonArray` (para colecciones).

Registrar el serializador:

Registrar unserializador personalizado se hace empleando un objeto del tipo BooleanSerializer:

```
GsonBuilder builder = new GsonBuilder()
                                .registerTypeAdapter(Boolean.class, new BooleanSer
```

```
ializer()
                                        .create();
```

📌 Existen 4 subclases de JsonElement que pueden ser devueltas: `JsonArray`, `JsonNull.INSTANCE`, `JsonObject`, `JsonPrimitive`, que son Boolean, Character, Number o String.

Ten en cuenta que **el método serialize devuelve un objeto de tipo JsonElement**.

3.2. Deserializador personalizado: `JsonDeserializer`

Permite deserializar JSON personalizado. Se debe implementar la interfaz `JsonDeserializer<T>`.

Ejemplo:

```
public class BooleanDeserializer implements JsonDeserializer<Boolean> {
    public Boolean deserialize(JsonElement json, Type tipo, JsonDeserializationContext
        return json.getAsInt() == 1;  // 1 → true, 0 → false
    }
}
```

💡 ▼ **Explicación** `JsonDeserializer<T>`

Esta interfaz permite convertir una representación en formato JSON de un objeto de vuelta a una instancia de un objeto Java. Implementa el método:

Parámetros del método `deserialize`:

- `JsonElement json`:

  Este es el elemento JSON que queremos deserializar. Puede ser una instancia de `JsonPrimitive` (para valores simples), `JsonObject` (para objetos), o `JsonArray` (para colecciones).

- `Type tipo`:

  Es el tipo de destino que queremos obtener. Al igual que en el caso del serializador, generalmente será la clase del tipo de objeto que estamos deserializando (`Boolean.class`), pero también puede ser útil cuando se trabaja con tipos genéricos.

- `JsonDeserializationContext context`:

  Al igual que el `JsonSerializationContext`, este contexto te permite delegar la deserialización de campos o subcampos a Gson. Si el objeto JSON tiene campos que deben ser convertidos a otras clases, puedes usar `context.deserialize(jsonSubcampo, tipo)` para delegar la tarea de convertir ese subcampo a su correspondiente objeto Java.

Retorno:

- `T`:El método devuelve el objeto Java deserializado a partir del elemento JSON.

Registrar el deserializador:

```
GsonBuilder builder = new GsonBuilder();
                                .registerTypeAdapter(Boolean.class, new BooleanDeser
ializer())
                                .create();
```

---

4. Adaptadores de tipo: `TypeAdapter`

Un `TypeAdapter` permite definir cómo se leen y escriben tipos personalizados en JSON. Debe extender la clase abstracta `TypeAdapter<T>` y sobrescribir los métodos `read()` y `write()`.

Ejemplo:

Adaptador para una clase `Point`:

```
public class PointAdapter extends TypeAdapter<Point> {
    public Point read(JsonReader reader) throws IOException {
        String[] coords = reader.nextString().split(",");
        return new Point(Integer.parseInt(coords[0]), Integer.parseInt(coords[1]));
    }

    public void write(JsonWriter writer, Point punto) throws IOException {
        writer.value(punto.getX() + "," + punto.getY());
    }
}
```

Registrar el `TypeAdapter`:

```
GsonBuilder builder = new GsonBuilder();
builder.registerTypeAdapter(Point.class, new PointAdapter());
Gson gson = builder.create();
```

Ejemplo avanzado: Serialización de `LocalDateTime`

Modificar una clase `Examen` para usar `LocalDateTime` requiere un adaptador personalizado que maneje el formato de fecha.

4.1. Adaptador de `LocalDateTime`:

```
public class LocalDateTimeAdapter extends TypeAdapter<LocalDateTime> {
    private static final DateTimeFormatter formato = DateTimeFormatter.ofPattern("yyyy-

    public void write(JsonWriter writer, LocalDateTime fecha) throws IOException {
        writer.value(fecha.format(formato));
    }

    public LocalDateTime read(JsonReader reader) throws IOException {
        return LocalDateTime.parse(reader.nextString(), formato);
    }
}
```

Registrar el adaptador:

```
GsonBuilder builder = new GsonBuilder();
builder.registerTypeAdapter(LocalDateTime.class, new LocalDateTimeAdapter());
Gson gson = builder.create();
```

Resumen

- `registerTypeAdapter`: Registra adaptadores personalizados para tipos específicos.
- **Versionado con** `@Since` **y** `@Until`: Controla la versión de los atributos serializados.
- `InstanceCreator`: Crea objetos personalizados sin constructor por defecto.
- `JsonSerializer` **y** `JsonDeserializer`: Personalizan la serialización y deserialización de tipos Java.
- `TypeAdapter`: Ofrece un control más detallado para personalizar el proceso de lectura y escritura de JSON.

Con estas herramientas, Gson permite una alta flexibilidad para personalizar cómo se serializan y deserializan los datos JSON en tus aplicaciones Java.