Customizing JSON with Gson

Default settings of the Gson library are often enough for our basic needs, there are situations where more flexibility is required.

If the same developer/team is both serializing and deserializing the same Java objects, some of the customizations can be done when writing the class for the object.

You just make the object store the data in the same way you want it to be serialized.

We can use the **GsonBuilder** class to modify the default behavior of Gson objects.

If you need ultimate control over the fine details of serialization/deserialization, you can create your own classes that implement the JsonSerializer<T> and/or JsonDeserializer<T> interfaces.

GsonBuilder

The Gson library provides a convenient object for configuring the basic behavior of Gson's JsonObjects.

The GsonBuilder class is implemented using the builder design pattern, so you can chain a bunch of its methods together on an object instance to set the desired behavior.

```
GsonBuilder gsonBuilder = new GsonBuilder();
Gson gson = gsonBuilder
   .setPrettyPrinting()
   .serializeNulls()
   .excludeFieldsWithoutExposeAnnotation()
   .create();
```

setPrettyPrinting() formats the JSON so it is easier to read, serializeNulls() overrides the default behavior of non-serializing fields with null values, and excludeFieldsWithoutExposeAnnotation() only serializes fields explicitly annotated with @Expose. If we use @Expose only on some of our object's fields, Gson will serialize those and ignore the rest.

```
public class GuitarBrand {
   @Expose
   Date dateFounded;
   @Expose
   String name;
   String country;
```

```
    @Expose
    List<String> artistsUsedBy;

    // getters, setters, contructor
}
```

We can create an object representing the brand and then print the result of serializing it to see what we end up with. We'll pass the current date as the founding date and the artistsUsedBy will be set to null since no one has used our brand yet...

```
 GuitarBrand ultimateGuitars = new GuitarBrand(new Date(), "Ultimate
Guitars",
 "Canada", null);
 String jsonUltimateGuitars = gson.toJson(ultimateGuitars);
 System.out.println(jsonUltimateGuitars);
```

```
{
        "dateFounded": "Nov 9, 2021, 4:38:14 PM",
        "name": "Ultimate Guitars",
        "artistsUsedBy": null
}
```

As you can see, only the fields we annotated with @Expose are present, the artistsUsedBy field was serialized even though it is currently null, and the whole thing was printed with nice-looking formatting.

Custom serialization

If we don't want to use Gson's default settings, we can customize the way it converts our object into JSON.

We create a custom serializer class that implements the JsonSerializer<T> interface and overrides its serialize() method.

We configure it specifically for our class by passing our class in as a type argument.

```
public class GuitarBrandGsonSerializer implements
JsonSerializer<GuitarBrand> {

    @Override
    public JsonElement serialize(GuitarBrand guitar, Type type,
        JsonSerializationContext jsonSerializationContext) {

        JsonObject guitarJsonObj = new JsonObject();
```

```
        // Code to customize the JsonObject

        return guitarJsonObj;
    }
}
```

First, we can customize the name of the JSON properties so that they don't just follow the Java class's property names. To do this, we can use the addProperty() method. The first parameter is for the chosen name as a String , and the second is for the value of the object we are assigning to that name.

Since Gson maintains the insertion order, let's take this opportunity to put the brand name as the first property

```
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");

guitarJsonObj.addProperty("brand_name", guitar.getName());
guitarJsonObj.addProperty("country_founded_in", guitar.getCountry());
guitarJsonObj.addProperty("date_founded",
sdf.format(guitar.getDateFounded()));

JsonArray array = new JsonArray();
if (guitar.getArtistsUsedBy() != null) {
   guitar.getArtistsUsedBy().forEach(array::add);
}
guitarJsonObj.add("<strong>Artists Who Use</strong>", array);
```

If we want to add an array as a value, we can do it using add() method.

We need to pass the date in as a String, so we'll use a SimpleDateFormat object to format it for us.

We could also include HTML in the property name if it was necessary.

Custom deserialization

While the default deserialization is often good enough, there may be times when you want the data to be deserialized in a certain way.

The following example is a bit long-winded, but it demonstrates that inside the deserialize method you have the liberty to do any modifications you want to the JsonObject.

This can be done right before creating and returning the Java object.

However, if you define your custom serializer carefully, the default deserializer will often be enough.

```
public class GuitarBrandGsonDeserializer implements
JsonDeserializer<GuitarBrand> {

   private SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

   @Override
   public GuitarBrand deserialize(JsonElement json, Type type,
                           JsonDeserializationContext jsonDeserializationContext)
throws
                           JsonParseException {

      JsonObject jsonObject = json.getAsJsonObject();
      JsonElement jsonGuitarBrandName = jsonObject.get("name");
      JsonElement jsonDateFounded = jsonObject.get("dateFounded");
      JsonElement jsonCountry = jsonObject.get("country");
      JsonArray jsonArtistsUsedBy =
jsonObject.getAsJsonArray("artistsUsedBy");

      ArrayList<String> artistList = new ArrayList<>();
      if (jsonArtistsUsedBy != null) {
         for (int i = 0; i < jsonArtistsUsedBy.size(); i++) {
            artistList.add(jsonArtistsUsedBy.get(i).getAsString());
         }
      }

      GuitarBrand guitarBrand = new
GuitarBrand(sdf.parse(jsonDateFounded.getAsString()),
                           jsonGuitarBrandName.getAsString(),
                           jsonCountry.getAsString(), artistList);
      return guitarBrand;
   }
}
```

Custom deserialization is especially important when you don't have access to the class files for serialized objects, or when the JSON objects contain more information than you need.

Tying it all together

In order for a Gson object to use our customized class, we need to register it using a GsonBuilder object.

We do this using the registerTypeAdapter() method, to which we pass the class

we are serializing/deserializing and an instance of our custom class.

While we're at it, let's chain the disableHtmlEscaping() method too, since our custom serializer included some raw HTML.

```
Gson gson = new GsonBuilder()
    .registerTypeAdapter(GuitarBrand.class, new GuitarBrandGsonSerializer())
    .disableHtmlEscaping()
    .create();
```

The exact same process is used for registering a deserializer. Once this is done, the Gson object can be used in the same way as when we used the default Gson object, only now it will have the custom behavior applied.

Yes, typically when implementing JsonSerializer<T> for a certain type T, it's advisable to also implement JsonDeserializer<T>. This ensures that the serialization and deserialization processes are consistent and compatible with each other.

- **JsonElement**:
  - JsonElement is an abstract class that represents an element in a JSON structure.
  - It is a superclass for all types of JSON elements such as objects, arrays, primitive values (like strings, numbers, booleans), and null values.
  - You typically use JsonElement when you want to work with JSON data in a generic way, without knowing the specific type of the element beforehand.
- **JsonObject**:
  - JsonObject is a subclass of JsonElement and represents a JSON object, which is a collection of key-value pairs enclosed within curly braces {}.
  - It allows you to access and manipulate individual properties (key-value pairs) within the JSON object.
  - You use JsonObject when you know that the JSON structure is an object and you want to access its properties directly.