Gson User Guide

Overview

Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object.

Gson can work with arbitrary Java objects including pre-existing objects that you do not have source code of.

Goals for Gson

- Provide easy to use mechanisms like `toString()` and constructor (factory method) to convert Java to JSON and vice-versa
- Allow pre-existing unmodifiable objects to be converted to and from JSON
- Allow custom representations for objects
- Support arbitrarily complex objects
- Generate compact and readable JSON output

Gson Performance and Scalability

Here are some metrics that we obtained on a desktop (dual opteron, 8GB RAM, 64-bit Ubuntu) running lots of other things along-with the tests. You can rerun these tests by using the class `PerformanceTest`.

- Strings: Deserialized strings of over 25MB without any problems (see `disabled_testStringDeserializationPerformance` method in `PerformanceTest`)
- Large collections:
  - Serialized a collection of 1.4 million objects (see `disabled_testLargeCollectionSerialization` method in `PerformanceTest`)
  - Deserialized a collection of 87,000 objects (see `disabled_testLargeCollectionDeserialization` in `PerformanceTest`)
- Gson 1.4 raised the deserialization limit for byte arrays and collection to over 11MB from 80KB.

Note: Delete the `disabled_` prefix to run these tests. We use this prefix to prevent running these tests every time we run JUnit tests.

Gson Users

Gson was originally created for use inside Google where it is currently used in a number of projects. It is now used by a number of public projects and companies.

Using Gson

The primary class to use is `Gson` which you can just create by calling `new Gson()` . There is also a class `GsonBuilder` available that can be used to create a Gson instance with various settings like version control and so on.

The Gson instance does not maintain any state while invoking JSON operations. So, you are free to reuse the same object for multiple JSON serialization and deserialization operations.

Using Gson with Gradle/Android

```
dependencies {
    implementation 'com.google.code.gson:gson:2.10'
}
```

Using Gson with Maven

To use Gson with Maven2/3, you can use the Gson version available in Maven Central by adding the following dependency:

```
<dependencies>
    <!--  Gson: Java to JSON conversion -->
    <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.10</version>

      <scope>compile</scope>
    </dependency>
</dependencies>
```

That is it, now your Maven project is Gson enabled.

Primitives Examples

```
// Serialization
Gson gson = new Gson();
gson.toJson(1);            // ==> 1
gson.toJson("abcd");       // ==> "abcd"
gson.toJson(new Long(10)); // ==> 10
int[] values = { 1 };
gson.toJson(values);       // ==> [1]

// Deserialization
int one = gson.fromJson("1", int.class);
Integer one = gson.fromJson("1", Integer.class);
Long one = gson.fromJson("1", Long.class);
Boolean false = gson.fromJson("false", Boolean.class);
```

```
String str = gson.fromJson("\"abc\"", String.class);
String[] anotherStr = gson.fromJson("[\"abc\"]", String[].class);
```

Object Examples

```
class BagOfPrimitives {
  private int value1 = 1;
  private String value2 = "abc";
  private transient int value3 = 3;
  BagOfPrimitives() {
    // no-args constructor
  }
}

// Serialization
BagOfPrimitives obj = new BagOfPrimitives();
Gson gson = new Gson();
String json = gson.toJson(obj);

// ==> json is {"value1":1,"value2":"abc"}
```

Note that you can not serialize objects with circular references since that will result in infinite recursion.

```
// Deserialization
BagOfPrimitives obj2 = gson.fromJson(json, BagOfPrimitives.class);

// ==> obj2 is just like obj
```

Finer Points with Objects

- It is perfectly fine (and recommended) to use private fields.
- There is no need to use any annotations to indicate a field is to be included for serialization and deserialization. All fields in the current class (and from all super classes) are included by default.
- If a field is marked transient, (by default) it is ignored and not included in the JSON serialization or deserialization.
- This implementation handles nulls correctly.
  - While serializing, a null field is omitted from the output.
  - While deserializing, a missing entry in JSON results in setting the corresponding field in the object to its default value: null for object types, zero for numeric types, and false for booleans.
- If a field is *synthetic*, it is ignored and not included in JSON serialization or deserialization.
- Fields corresponding to the outer classes in inner classes are ignored and not included in serialization or deserialization.
- Anonymous and local classes are excluded. They will be serialized as JSON `null` and when deserialized their JSON value is ignored and `null` is returned. Convert the classes to

`static` nested classes to enable serialization and deserialization for them.

Nested Classes (including Inner Classes)

Gson can serialize static nested classes quite easily.

Gson can also deserialize static nested classes. However, Gson can **not** automatically deserialize the **pure inner classes since their no-args constructor also need a reference to the containing Object** which is not available at the time of deserialization. You can address this problem by either making the inner class static or by providing a custom InstanceCreator for it. Here is an example:

```
public class A {
  public String a;

  class B {

    public String b;

    public B() {
      // No args constructor for B
    }
  }
}
```

NOTE: The above class B can not (by default) be serialized with Gson.

Gson can not deserialize `{"b":"abc"}` into an instance of B since the class B is an inner class. If it was defined as static class B then Gson would have been able to deserialize the string. Another solution is to write a custom instance creator for B.

```
public class InstanceCreatorForB implements InstanceCreator<A.B> {
  private final A a;
  public InstanceCreatorForB(A a)  {
    this.a = a;
  }
  public A.B createInstance(Type type) {
    return a.new B();
  }
}
```

The above is possible, but not recommended.

Array Examples

```
Gson gson = new Gson();
int[] ints = {1, 2, 3, 4, 5};
String[] strings = {"abc", "def", "ghi"};
```

```
// Serialization
gson.toJson(ints);     // ==> [1,2,3,4,5]
gson.toJson(strings);  // ==> ["abc", "def", "ghi"]

// Deserialization
int[] ints2 = gson.fromJson("[1,2,3,4,5]", int[].class);
// ==> ints2 will be same as ints
```

We also support multi-dimensional arrays, with arbitrarily complex element types.

Collections Examples

```
Gson gson = new Gson();
Collection<Integer> ints = Arrays.asList(1,2,3,4,5);

// Serialization
String json = gson.toJson(ints);  // ==> json is [1,2,3,4,5]

// Deserialization
TypeToken<Collection<Integer>> collectionType = new TypeToken<Collection<Integer>>()
// Note: For older Gson versions it is necessary to use `collectionType.getType()` a
// this is however not type-safe and care must be taken to specify the correct type
Collection<Integer> ints2 = gson.fromJson(json, collectionType);
// ==> ints2 is same as ints
```

Fairly hideous: note how we define the type of collection. Unfortunately, there is no way to get around this in Java.

Collections Limitations

Gson can serialize collection of arbitrary objects but can not deserialize from it, because there is no way for the user to indicate the type of the resulting object. Instead, while deserializing, the Collection must be of a specific, generic type. This makes sense, and is rarely a problem when following good Java coding practices.

Maps Examples

Gson by default serializes any `java.util.Map` implementation as a JSON object. Because JSON objects only support strings as member names, Gson converts the Map keys to strings by calling `toString()` on them, and using `"null"` for `null` keys:

```
Gson gson = new Gson();
Map<String, String> stringMap = new LinkedHashMap<>();
stringMap.put("key", "value");
stringMap.put(null, "null-entry");

// Serialization
String json = gson.toJson(stringMap); // ==> json is {"key":"value","null":"null-ent
```

```
Map<Integer, Integer> intMap = new LinkedHashMap<>();
intMap.put(2, 4);
intMap.put(3, 6);

// Serialization
String json = gson.toJson(intMap); // ==> json is {"2":4,"3":6}
```

For deserialization Gson uses the `read` method of the `TypeAdapter` registered for the Map key type. Similar to the Collection example shown above, for deserialization a `TypeToken` has to be used to tell Gson what types the Map keys and values have:

```
Gson gson = new Gson();
TypeToken<Map<String, String>> mapType = new TypeToken<Map<String, String>>(){};
String json = "{\"key\": \"value\"}";

// Deserialization
// Note: For older Gson versions it is necessary to use `mapType.getType()` as argum
// this is however not type-safe and care must be taken to specify the correct type
Map<String, String> stringMap = gson.fromJson(json, mapType);
// ==> stringMap is {key=value}
```

Gson also supports using complex types as Map keys. This feature can be enabled with `GsonBuilder.enableComplexMapKeySerialization()` . If enabled, Gson uses the `write` method of the `TypeAdapter` registered for the Map key type to serialize the keys, instead of using `toString()` . When any of the keys is serialized by the adapter as JSON array or JSON object, Gson will serialize the complete Map as JSON array, consisting of key-value pairs (encoded as JSON array). Otherwise, if none of the keys is serialized as a JSON array or JSON object, Gson will use a JSON object to encode the Map:

```
class PersonName {
  String firstName;
  String lastName;

  PersonName(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  // ... equals and hashCode
}

Gson gson = new GsonBuilder().enableComplexMapKeySerialization().create();
Map<PersonName, Integer> complexMap = new LinkedHashMap<>();
complexMap.put(new PersonName("John", "Doe"), 30);
complexMap.put(new PersonName("Jane", "Doe"), 35);

// Serialization; complex map is serialized as a JSON array containing key-value pai
String json = gson.toJson(complexMap);
// ==> json is [[{"firstName":"John","lastName":"Doe"},30],[{"firstName":"Jane","las
```

```
Map<String, String> stringMap = new LinkedHashMap<>();
stringMap.put("key", "value");
// Serialization; non-complex map is serialized as a regular JSON object
String json = gson.toJson(stringMap); // json is {"key":"value"}
```

Important: Because Gson by default uses `toString()` to serialize Map keys, this can lead to malformed encoded keys or can cause mismatch between serialization and deserialization of the keys, for example when `toString()` is not properly implemented. A workaround for this can be to use `enableComplexMapKeySerialization()` to make sure the `TypeAdapter` registered for the Map key type is used for deserialization *and* serialization. As shown in the example above, when none of the keys are serialized by the adapter as JSON array or JSON object, the Map is serialized as a regular JSON object, as desired.

Note that when deserializing enums as Map keys, if Gson is unable to find an enum constant with a matching `name()` value respectively `@SerializedName` annotation, it falls back to looking up the enum constant by its `toString()` value. This is to work around the issue described above, but only applies to enum constants.

Serializing and Deserializing Generic Types

When you call `toJson(obj)` , Gson calls `obj.getClass()` to get information on the fields to serialize. Similarly, you can typically pass `MyClass.class` object in the `fromJson(json, MyClass.class)` method. This works fine if the object is a non-generic type. However, if the object is of a generic type, then the Generic type information is lost because of Java Type Erasure. Here is an example illustrating the point:

```
class Foo<T> {
  T value;
}
Gson gson = new Gson();
Foo<Bar> foo = new Foo<Bar>();
gson.toJson(foo); // May not serialize foo.value correctly

gson.fromJson(json, foo.getClass()); // Fails to deserialize foo.value as Bar
```

The above code fails to interpret value as type Bar because Gson invokes `foo.getClass()` to get its class information, but this method returns a raw class, `Foo.class` . This means that Gson has no way of knowing that this is an object of type `Foo<Bar>` , and not just plain `Foo` .

You can solve this problem by specifying the correct parameterized type for your generic type. You can do this by using the `TypeToken` class.

```
Type fooType = new TypeToken<Foo<Bar>>() {}.getType();
gson.toJson(foo, fooType);

gson.fromJson(json, fooType);
```

The idiom used to get `fooType` actually defines an anonymous local inner class containing a method `getType()` that returns the fully parameterized type.

Serializing and Deserializing Collection with Objects of Arbitrary Types

Sometimes you are dealing with JSON array that contains mixed types. For example: `['hello',5,{name:'GREETINGS',source:'guest'}]`

The equivalent `Collection` containing this is:

```
Collection collection = new ArrayList();
collection.add("hello");
collection.add(5);
collection.add(new Event("GREETINGS", "guest"));
```

where the `Event` class is defined as:

```
class Event {
  private String name;
  private String source;
  private Event(String name, String source) {
    this.name = name;
    this.source = source;
  }
}
```

You can serialize the collection with Gson without doing anything specific: `toJson(collection)` would write out the desired output.

However, deserialization with `fromJson(json, Collection.class)` will not work since Gson has no way of knowing how to map the input to the types. Gson requires that you provide a genericised version of collection type in `fromJson()` . So, you have three options:

1. Use Gson's parser API (low-level streaming parser or the DOM parser JsonParser) to parse the array elements and then use `Gson.fromJson()` on each of the array elements.This is the preferred approach. Here is an example that demonstrates how to do this.
2. Register a type adapter for `Collection.class` that looks at each of the array members and maps them to appropriate objects. The disadvantage of this approach is that it will screw up deserialization of other collection types in Gson.
3. Register a type adapter for `MyCollectionMemberType` and use `fromJson()` with `Collection<MyCollectionMemberType>` .

This approach is practical only if the array appears as a top-level element or if you can change the field type holding the collection to be of type `Collection<MyCollectionMemberType>` .

Built-in Serializers and Deserializers

Gson has built-in serializers and deserializers for commonly used classes whose default representation may be inappropriate, for instance

- `java.net.URL` to match it with strings like `"https://github.com/google/gson/"`
- `java.net.URI` to match it with strings like `"/google/gson/"`

For many more, see the internal class `TypeAdapters`.

You can also find source code for some commonly used classes such as JodaTime at this page.

Custom Serialization and Deserialization

Sometimes default representation is not what you want. This is often the case when dealing with library classes (DateTime, etc). Gson allows you to register your own custom serializers and deserializers. This is done by defining two parts:

- JSON Serializers: Need to define custom serialization for an object
- JSON Deserializers: Needed to define custom deserialization for a type
- Instance Creators: Not needed if no-args constructor is available or a deserializer is registered

```
GsonBuilder gson = new GsonBuilder();
gson.registerTypeAdapter(MyType2.class, new MyTypeAdapter());
gson.registerTypeAdapter(MyType.class, new MySerializer());
gson.registerTypeAdapter(MyType.class, new MyDeserializer());
gson.registerTypeAdapter(MyType.class, new MyInstanceCreator());
```

`registerTypeAdapter` call checks if the type adapter implements more than one of these interfaces and register it for all of them.

Writing a Serializer

Here is an example of how to write a custom serializer for JodaTime `DateTime` class.

```
private class DateTimeSerializer implements JsonSerializer<DateTime> {
  public JsonElement serialize(DateTime src, Type typeOfSrc, JsonSerializationContex
    return new JsonPrimitive(src.toString());
  }
}
```

Gson calls `serialize()` when it runs into a `DateTime` object during serialization.

Writing a Deserializer

Here is an example of how to write a custom deserializer for JodaTime DateTime class.

```
private class DateTimeDeserializer implements JsonDeserializer<DateTime> {
  public DateTime deserialize(JsonElement json, Type typeOfT, JsonDeserializationCon
      throws JsonParseException {
    return new DateTime(json.getAsJsonPrimitive().getAsString());
  }
}
```

Gson calls `deserialize` when it needs to deserialize a JSON string fragment into a DateTime object

Finer points with Serializers and Deserializers

Often you want to register a single handler for all generic types corresponding to a raw type

- For example, suppose you have an `Id` class for id representation/translation (i.e. an internal vs. external representation).
- `Id<T>` type that has same serialization for all generic types
  - Essentially write out the id value
- Deserialization is very similar but not exactly the same
  - Need to call `new Id(Class<T>, String)` which returns an instance of `Id<T>`

Gson supports registering a single handler for this. You can also register a specific handler for a specific generic type (say `Id<RequiresSpecialHandling>` needed special handling). The `Type` parameter for the `toJson()` and `fromJson()` contains the generic type information to help you write a single handler for all generic types corresponding to the same raw type.

Writing an Instance Creator

While deserializing an Object, Gson needs to create a default instance of the class. Well-behaved classes that are meant for serialization and deserialization should have a no-argument constructor.

- Doesn't matter whether public or private

Typically, Instance Creators are needed when you are dealing with a library class that does NOT define a no-argument constructor

Instance Creator Example

```
private class MoneyInstanceCreator implements InstanceCreator<Money> {
  public Money createInstance(Type type) {
    return new Money("1000000", CurrencyCode.USD);
  }
}
```

Type could be of a corresponding generic type

- Very useful to invoke constructors which need specific generic type information

- For example, if the `Id` class stores the class for which the Id is being created

InstanceCreator for a Parameterized Type

Sometimes the type that you are trying to instantiate is a parameterized type. Generally, this is not a problem since the actual instance is of raw type. Here is an example:

```
class MyList<T> extends ArrayList<T> {
}

class MyListInstanceCreator implements InstanceCreator<MyList<?>> {
    @SuppressWarnings("unchecked")
  public MyList<?> createInstance(Type type) {
    // No need to use a parameterized list since the actual instance will have the r
    return new MyList();
  }
}
```

However, sometimes you do need to create instance based on the actual parameterized type. In this case, you can use the type parameter being passed to the `createInstance` method. Here is an example:

```
public class Id<T> {
  private final Class<T> classOfId;
  private final long value;
  public Id(Class<T> classOfId, long value) {
    this.classOfId = classOfId;
    this.value = value;

  }
}

class IdInstanceCreator implements InstanceCreator<Id<?>> {
  public Id<?> createInstance(Type type) {
    Type[] typeParameters = ((ParameterizedType)type).getActualTypeArguments();
    Type idType = typeParameters[0]; // Id has only one parameterized type T
    return new Id((Class)idType, 0L);
  }
}
```

In the above example, an instance of the Id class can not be created without actually passing in the actual type for the parameterized type. We solve this problem by using the passed method parameter, `type`. The `type` object in this case is the Java parameterized type representation of `Id<Foo>` where the actual instance should be bound to `Id<Foo>`. Since `Id` class has just one parameterized type parameter, `T`, we use the zeroth element of the type array returned by `getActualTypeArgument()` which will hold `Foo.class` in this case.

Compact Vs. Pretty Printing for JSON Output Format

The default JSON output that is provided by Gson is a compact JSON format. This means that there will not be any whitespace in the output JSON structure. Therefore, there will be no whitespace between field names and its value, object fields, and objects within arrays in the JSON output. As well, "null" fields will be ignored in the output (NOTE: null values will still be included in collections/arrays of objects). See the Null Object Support section for information on configure Gson to output all null values.

If you would like to use the Pretty Print feature, you must configure your `Gson` instance using the `GsonBuilder`. The `JsonFormatter` is not exposed through our public API, so the client is unable to configure the default print settings/margins for the JSON output. For now, we only provide a default `JsonPrintFormatter` that has default line length of 80 character, 2 character indentation, and 4 character right margin.

The following is an example shows how to configure a `Gson` instance to use the default `JsonPrintFormatter` instead of the `JsonCompactFormatter`:

```
Gson gson = new GsonBuilder().setPrettyPrinting().create();
String jsonOutput = gson.toJson(someObject);
```

Null Object Support

The default behaviour that is implemented in Gson is that `null` object fields are ignored. This allows for a more compact output format; however, the client must define a default value for these fields as the JSON format is converted back into its Java form.

Here's how you would configure a `Gson` instance to output null:

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

NOTE: when serializing `null` s with Gson, it will add a `JsonNull` element to the `JsonElement` structure. Therefore, this object can be used in custom serialization/deserialization.

Here's an example:

```
public class Foo {
  private final String s;
  private final int i;

  public Foo() {
    this(null, 5);
  }

  public Foo(String s, int i) {
    this.s = s;
    this.i = i;
  }
}
```

```
Gson gson = new GsonBuilder().serializeNulls().create();
Foo foo = new Foo();
String json = gson.toJson(foo);
System.out.println(json);

json = gson.toJson(null);
System.out.println(json);
```

The output is:

```
{"s":null,"i":5}
null
```

Versioning Support

Multiple versions of the same object can be maintained by using @Since annotation. This annotation can be used on Classes, Fields and, in a future release, Methods. In order to leverage this feature, you must configure your `Gson` instance to ignore any field/object that is greater than some version number. If no version is set on the `Gson` instance then it will serialize and deserialize all fields and classes regardless of the version.

```
public class VersionedClass {
  @Since(1.1) private final String newerField;
  @Since(1.0) private final String newField;
  private final String field;

  public VersionedClass() {
    this.newerField = "newer";
    this.newField = "new";
    this.field = "old";
  }
}

VersionedClass versionedObject = new VersionedClass();
Gson gson = new GsonBuilder().setVersion(1.0).create();
String jsonOutput = gson.toJson(versionedObject);
System.out.println(jsonOutput);
System.out.println();

gson = new Gson();
jsonOutput = gson.toJson(versionedObject);
System.out.println(jsonOutput);
```

The output is:

```
{"newField":"new","field":"old"}

{"newerField":"newer","newField":"new","field":"old"}
```

Excluding Fields From Serialization and Deserialization

Gson supports numerous mechanisms for excluding top-level classes, fields and field types. Below are pluggable mechanisms that allow field and class exclusion. If none of the below mechanisms satisfy your needs then you can always use custom serializers and deserializers.

Java Modifier Exclusion

By default, if you mark a field as `transient` , it will be excluded. As well, if a field is marked as `static` then by default it will be excluded. If you want to include some transient fields then you can do the following:

```
import java.lang.reflect.Modifier;
Gson gson = new GsonBuilder()
    .excludeFieldsWithModifiers(Modifier.STATIC)
    .create();
```

NOTE: you can give any number of the `Modifier` constants to the `excludeFieldsWithModifiers` method. For example:

```
Gson gson = new GsonBuilder()
    .excludeFieldsWithModifiers(Modifier.STATIC, Modifier.TRANSIENT, Modifier.VOLATI
    .create();
```

Gson's `@Expose`

This feature provides a way where you can mark certain fields of your objects to be excluded for consideration for serialization and deserialization to JSON. To use this annotation, you must create Gson by using `new GsonBuilder().excludeFieldsWithoutExposeAnnotation().create()` . The Gson instance created will exclude all fields in a class that are not marked with `@Expose` annotation.

User Defined Exclusion Strategies

If the above mechanisms for excluding fields and class type do not work for you then you can always write your own exclusion strategy and plug it into Gson. See the `ExclusionStrategy` JavaDoc for more information.

The following example shows how to exclude fields marked with a specific `@Foo` annotation and excludes top-level types (or declared field type) of class `String` .

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
public @interface Foo {
  // Field tag only annotation
}
```

```
}

public class SampleObjectForTest {
  @Foo private final int annotatedField;
  private final String stringField;
  private final long longField;
  private final Class<?> clazzField;

  public SampleObjectForTest() {
    annotatedField = 5;
    stringField = "someDefaultValue";
    longField = 1234;
  }
}

public class MyExclusionStrategy implements ExclusionStrategy {
  private final Class<?> typeToSkip;

  private MyExclusionStrategy(Class<?> typeToSkip) {
    this.typeToSkip = typeToSkip;
  }

  public boolean shouldSkipClass(Class<?> clazz) {
    return (clazz == typeToSkip);
  }

  public boolean shouldSkipField(FieldAttributes f) {
    return f.getAnnotation(Foo.class) != null;
  }
}

public static void main(String[] args) {
  Gson gson = new GsonBuilder()
      .setExclusionStrategies(new MyExclusionStrategy(String.class))
      .serializeNulls()
      .create();
  SampleObjectForTest src = new SampleObjectForTest();
  String json = gson.toJson(src);
  System.out.println(json);
}
```

The output is:

```
{"longField":1234}
```

JSON Field Naming Support

Gson supports some pre-defined field naming policies to convert the standard Java field names (i.e., camel cased names starting with lower case --- `sampleFieldNameInJava`) to a JSON field name (i.e., `sample_field_name_in_java` or `SampleFieldNameInJava`). See the FieldNamingPolicy class for information on the pre-defined naming policies.

It also has an annotation based strategy to allows clients to define custom names on a per field basis. Note, that the annotation based strategy has field name validation which will raise "Runtime" exceptions if an invalid field name is provided as the annotation value.

The following is an example of how to use both Gson naming policy features:

```
private class SomeObject {
  @SerializedName("custom_naming") private final String someField;
  private final String someOtherField;

  public SomeObject(String a, String b) {
    this.someField = a;
    this.someOtherField = b;
  }
}

SomeObject someObject = new SomeObject("first", "second");
Gson gson = new GsonBuilder().setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CAS
String jsonRepresentation = gson.toJson(someObject);
System.out.println(jsonRepresentation);
```

The output is:

```
{"custom_naming":"first","SomeOtherField":"second"}
```

If you have a need for custom naming policy (see this discussion), you can use the @SerializedName annotation.

Sharing State Across Custom Serializers and Deserializers

Sometimes you need to share state across custom serializers/deserializers (see this discussion). You can use the following three strategies to accomplish this:

1. Store shared state in static fields
2. Declare the serializer/deserializer as inner classes of a parent type, and use the instance fields of parent type to store shared state
3. Use Java `ThreadLocal`

1 and 2 are not thread-safe options, but 3 is.

Streaming

In addition Gson's object model and data binding, you can use Gson to read from and write to a stream. You can also combine streaming and object model access to get the best of both approaches.

Issues in Designing Gson

See the Gson design document for a discussion of issues we faced while designing Gson. It also include a comparison of Gson with other Java libraries that can be used for JSON conversion.

Future Enhancements to Gson

For the latest list of proposed enhancements or if you'd like to suggest new ones, see the Issues section under the project website.