

Contents

- 1 **1** Authors: Inderjeet Singh, Joel Leitch
 - 1 **1.1** Overview
 - 1 **1.2** Goals for Gson
 - 1 **1.3** Gson Performance and Scalability
 - 1 **1.4** Gson Users
 - 1 **1.5** Using Gson
 - 1 **1.5.1** Primitives Examples
 - 1 **1.5.2** Object Examples
 - 1 **1.5.2.1** Finer Points with Objects
 - 1 **1.5.3** Nested Classes (including Inner Classes)
 - 1 **1.5.4** Array Examples
 - 1 **1.5.5** Collections Examples
 - 1 **1.5.5.1** Collections Limitations
 - 1 **1.5.6** Serializing and Deserializing Generic Types
 - 1 **1.5.7** Built-in Serializers and Deserializers
 - 1 **1.5.8** Custom Serialization and Deserialization
 - 1 **1.5.8.1** Writing a Serializer
 - 1 **1.5.8.2** Writing a Deserializer
 - 1 **1.5.9** Writing an Instance Creator
 - 1 **1.5.9.1** InstanceCreator for a Parameterized Type
 - 1 **1.5.10** Compact Vs. Pretty Printing for JSON Output Format
 - 1 **1.5.11** Null Object Support
 - 1 **1.5.12** Versioning Support
 - 1 **1.5.13** Excluding Fields From Serialization and Deserialization
 - 1 **1.5.13.1** Java Modifier Exclusion
 - 1 **1.5.13.2** Gson's @Expose
 - 1 **1.5.13.3** User Defined Exclusion Strategies
 - 1 **1.5.14** JSON Field Naming Support
 - 1 **1.5.15** Sharing State Across Custom Serializers and Deserializers
 - 1 **1.6** Issues in Designing Gson
 - 1 **1.7** Future Enhancements to Gson

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 is an open-source project hosted at <http://code.google.com/p/google-gson>.

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 object
- Generate compact and readability 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. See details [here](#).

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.

Primitives Examples

(Serialization)

```
Gson gson = new Gson();  
gson.toJson(1);           ==> prints 1  
gson.toJson("abcd");      ==> prints "abcd"  
gson.toJson(new Long(10)); ==> prints 10  
int[] values = { 1 };  
gson.toJson(values);      ==> prints [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();  
Gsongson = newGson();  
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 serialization, a null field is skipped from the output
 - While deserialization, a missing entry in JSON results in setting the corresponding field in the object to null
- 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, anonymous classes, and local classes are ignored and not included in serialization or deserialization

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);    ==> prints [1,2,3,4,5]
```

```
gson.toJson(strings); ==> prints ["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 = Lists.immutableList(1,2,3,4,5);
```

(Serialization)

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

(Deserialization)

```
Type collectionType = new TypeToken<Collection<Integer>>().getType();  
Collection<Integer> ints2 = gson.fromJson(json, collectionType);
```

ints2 is same as ints

Fairly hideous: note how we define the type of collection

Unfortunately, no way to get around this in Java

Collections Limitations

- 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
- While deserializing, Collection must be of a specific generic type

All of this makes sense, and is rarely a problem when following good Java coding practices

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:

```
List<String> myStrings = new List<String>();  
gson.toJson(myStrings); // Will cause a runtime exception  
  
gson.fromJson(json, myStrings.getClass());
```

The above call results in a runtime exception because `Gson` invokes `myStrings.getClass()` to get its class information, but this method returns a raw class, `List.class`. This means that `Gson` has no way of knowing that this is a list of Strings, and not plain objects.

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 listType = new TypeToken<List<String>>() {}.getType();
gson.toJson(myStrings, listType);

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

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

Built-in Serializers and Deserializers

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

Here is a list of such classes:

- 2 `java.net.URL` to match it with strings like `"http://code.google.com/p/google-gson/"`.
- 3 `java.net.URI` to match it with strings like `"/p/google-gson/"`.

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 Serialiers:** 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

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

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,
        JsonSerializationContext context) {
        new JsonPrimitive(src.toString());
    }
}
```

```
}
```

Gson calls `toJson()` 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,
        JsonDeserializationContext context)
        throws JsonParseException {
        return new DateTime(json.getAsJsonPrimitive().getString());
    }
}
```

Gson calls `fromJson()` 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 that 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 raw type anyway.
        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 Id.get((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 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.

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

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

NOTE: when serializing `nulls` 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);

===== OUTPUT =====
{"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(someObject);
System.out.println(jsonOutput);
System.out.println();

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

===== OUTPUT =====
{"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 mechanism that allow field and class exclusion. If none of the below mechanism 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()
    .excludeFieldsWithModifier(Modifier.STATIC)
    .create();

```

NOTE: you can use any number of the `Modifier` constants to "excludeFieldsWithModifier" method. For example:

```

Gson gson = new GsonBuilder()
    .excludeFieldsWithModifier(Modifier.STATIC, Modifier.TRANSIENT,
Modifier.VOLATILE)
    .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 {
    @Fooprivate 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);
}

===== OUTPUT =====
{"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_CASE).creat
e();

```

```
String jsonRepresentation = gson.toJson(someObject);
System.out.println(jsonRepresentation);
```

===== OUTPUT =====

```
{"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:

- 4 Store shared state in static fields
- 5 Declare the serializer/deserializer as inner classes of a parent type, and use the instance fields of parent type to store shared state
- 6 Use Java ThreadLocal

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

Issues in Designing Gson

See the Gson design document for a discussion of issues we faced while designing Gson. It also includes 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.