Jackson - Quick Guide

Previous Page Next Page

Overview

Jackson is a simple java based library to serialize java objects to JSON and vice versa.

Features

- **Easy to use.** - jackson API provides a high level facade to simplify commonly used use cases.
- **No need to create mapping.** - jackson API provides default mapping for most of the objects to be serialized.
- **Performance.** - jackson is quiet fast and is of low memory footprint and is suitable for large object graphs or systems.
- **Clean JSON.** - jackson creates a clean and compact JSON results which is easy to read.
- **No Dependency.** - jackson library does not require any other library apart from jdk.
- **Open Source** - jackson library is open source and is free to use.

Three ways of processing JSON

Jackson provides three alternative ways to process JSON

- **Streaming API** - reads and writes JSON content as discrete events. JsonParser reads the data whereas JsonGenerator writes the data. It is most powerful approach among the three and is of lowest overhead and fastest in read/write opreations. It is Analogus to Stax parser for XML.

Tree Model - prepares a in-memory tree representation of the JSON document. ObjectMapper build tree of JsonNode nodes. It is most flexible approach. It is analogus to DOM parser for XML.

Data Binding - converts JSON to and from POJO (Plain Old Java Object) using property accessor or using annotations. It is of two type.

Simple Data Binding - Converts JSON to and from Java Maps, Lists, Strings, Numbers, Booleans and null objects.

Full Data Binding - Converts JSON to and from any JAVA type.

ObjectMapper reads/writes JSON for both types of data bindings. Data Binding is most convenient way and is analogus to JAXB parer for XML.

Environment Setup

Try it Option Online

You really do not need to set up your own environment to start learning Guava, a JAVA based library. Reason is very simple, we already have setup Java Programming environment online, so that you can compile and execute all the available examples online at the same time when you are doing your theory work. This gives you confidence in what you are reading and to check the result with different options. Feel free to modify any example and execute it online.

Try following example using **Try it** option available at the top right corner of the below sample code box:

```
public class MyFirstJavaProgram {

    public static void main(String []args) {
       System.out.println("Hello World");
    }
}
```

For most of the examples given in this tutorial, you will find **Try it** option, so just make use of it and enjoy your learning.

Local Environment Setup

If you are still willing to set up your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Please follow the following steps to set up the environment.

Java SE is freely available from the link Download Java . So you download a version based on your operating system.

Follow the instructions to download java and run the **.exe** to install Java on your machine. Once you installed Java on your machine, you would need to set environment variables to point to correct installation directories:

Setting up the path for windows 2000/XP:

Assuming you have installed Java in *c:\Program Files\java\jdk* directory:

Right-click on 'My Computer' and select 'Properties'.

Click on the 'Environment variables' button under the 'Advanced' tab.

Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

Setting up the path for windows 95/98/ME:

Assuming you have installed Java in *c:\Program Files\java\jdk* directory:

Edit the 'C:\autoexec.bat' file and add the following line at the end:
'SET PATH=%PATH%;C:\Program Files\java\jdk\bin'

Setting up the path for Linux, UNIX, Solaris, FreeBSD:

Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation if you have trouble doing this.

Example, if you use *bash* as your shell, then you would add the following line to the end of your '.bashrc: export PATH=/path/to/java:$PATH'

Popular Java Editors:

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following:

Notepad: On Windows machine you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.

Netbeans:is a Java IDE that is open-source and free which can be downloaded from http://www.netbeans.org/index.html .

Eclipse: is also a Java IDE developed by the eclipse open-source community and can be downloaded from http://www.eclipse.org/ .

Download jackson archive

Download the latest version of Jackson jar file from Maven Repository - Jackson . In this tutorial, jackson-core-2.8.9.jar,jackson-annotations-2.8.9.jar and jackson-databind-2.8.9.jar are downloaded and copied into C:\> jackson folder.

OS	Archive name
Windows	jackson-xxx-2.8.9.jar
Linux	jackson-xxx-2.8.9.jar
Mac	jackson-xxx-2.8.9.jar

Set jackson environment

Set the **jackson_HOME** environment variable to point to the base directory location where Guava jar is stored on your machine. Assuming, we've extracted jackson-core-2.8.9.jar,jackson-annotations-2.8.9.jar and jackson-databind-2.8.9.jar in jackson folder on various Operating Systems as follows.

OS	Output
Windows	Set the environment variable jackson_HOME to C:\jackson
Linux	export jackson_HOME=/usr/local/jackson
Mac	export jackson_HOME=/Library/jackson

Set CLASSPATH variable

Set the **CLASSPATH** environment variable to point to the jackson jar location. Assuming, we've stored jackson-core-2.8.9.jar,jackson-annotations-2.8.9.jar and jackson-databind-2.8.9.jar in jackson folder on various Operating Systems as follows.

OS	Output
Windows	Set the environment variable CLASSPATH to %CLASSPATH%;%jackson_HOME%\jackson-core-2.8.9.jar;%jackson_HOME%\jackson-databind-2.8.9.jar;%jackson_HOME%\jackson-annotations-2.8.9.jar;.;
Linux	export CLASSPATH=$CLASSPATH:$jackson_HOME/jackson-core-2.8.9.jar:$jackson_HOME/jackson-databind-2.8.9.jar:$jackson_HOME/jackson-annotations-2.8.9.jar:.

Mac	export CLASSPATH=$CLASSPATH:$jackson_HOME/jackson-core-2.8.9.jar:$jackson_HOME/jackson-databind-2.8.9.jar:$jackson_HOME/jackson-annotations-2.8.9.jar:.

First Application

Before going into the details of the jackson library, let's see an application in action. In this example, we've created Student class. We'll create a JSON string with student details and deserialize it to student object and then serialize it to an JSON String.

Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){

      ObjectMapper mapper = new ObjectMapper();
      String jsonString = "{\"name\":\"Mahesh\", \"age\":21}";

      //map json to student
      try{
         Student student = mapper.readValue(jsonString, Student.class);

         System.out.println(student);

         jsonString = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(student);

         System.out.println(jsonString);
      }
      catch (JsonParseException e) { e.printStackTrace();}
      catch (JsonMappingException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace(); }
   }
}

class Student {
   private String name;
   private int age;
   public Student(){}
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Student [ name: Mahesh, age: 21 ]
{
  "name" : "Mahesh",
  "age" : 21
}
```

Steps to remember

Following are the important steps to be considered here.

Step 1: Create ObjectMapper object.

Create ObjectMapper object. It is a reusable object.

```
ObjectMapper mapper = new ObjectMapper();
```

Step 2: DeSerialize JSON to Object.

Use readValue() method to get the Object from the JSON. Pass json string/ source of json string and object type as parameter.

```
//Object to JSON Conversion
Student student = mapper.readValue(jsonString, Student.class);
```

Step 3: Serialize Object to JSON.

Use writeValueAsString() method to get the JSON string representation of an object.

```
//Object to JSON Conversion
jsonString = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(student);
```

ObjectMapper Class

ObjectMapper is the main actor class of Jackson library. ObjectMapper class ObjectMapper provides functionality for reading and writing JSON, either to and from basic POJOs (Plain Old Java Objects), or to and from a general-purpose JSON Tree Model (JsonNode), as well as related functionality for performing conversions. It is also highly customizable to work both with different styles of JSON content, and to support more advanced Object concepts such as polymorphism and Object identity. ObjectMapper also acts as a factory for more advanced ObjectReader and ObjectWriter classes.

Class Declaration

Following is the declaration for **com.fasterxml.jackson.databind.ObjectMapper** class −

```
public class ObjectMapper
   extends ObjectCodec
      implements Versioned, Serializable
```

Nested Classes

S.No.	Class & Description
1	**static class ObjectMapper.DefaultTypeResolverBuilder** Customized TypeResolverBuilder that provides type resolver builders used with so-called "default typing" (see enableDefaultTyping() for details).
2	**static class ObjectMapper.DefaultTyping** Enumeration used with enableDefaultTyping() to specify what kind of types (classes) default typing should be used for.

Fields

protected DeserializationConfig _deserializationConfig - Configuration object that defines basic global settings for the serialization process.

protected DefaultDeserializationContext _deserializationContext - Blueprint context object; stored here to allow custom sub-classes.

protected InjectableValues _injectableValues - Provider for values to inject in deserialized POJOs.

protected JsonFactory _jsonFactory - Factory used to create JsonParser and JsonGenerator instances as necessary.

protected SimpleMixInResolver _mixIns - Mapping that defines how to apply mix-in annotations: key is the type to received additional annotations, and value is the type that has annotations to "mix in".

protected ConfigOverrides _propertyOverrides - Currently active per-type configuration overrides, accessed by declared type of property.

protected Set<Object> _registeredModuleTypes - Set of module types (as per Module.getTypeId() that have been registered; kept track of iff MapperFeature.IGNORE_DUPLICATE_MODULE_REGISTRATIONS is enabled, so that duplicate registration calls can be ignored (to avoid adding same handlers multiple times, mostly).

protected ConcurrentHashMap<JavaType,JsonDeserializer<Object>> _rootDeserializers - We will use a separate main-level Map for keeping track of root-level deserializers.

protected SerializationConfig _serializationConfig - Configuration object that defines basic global settings for the serialization process.

protected SerializerFactory _serializerFactory - Serializer factory used for constructing serializers.

protected DefaultSerializerProvider _serializerProvider - Object that manages access to serializers used for serialization, including caching.

protected SubtypeResolver _subtypeResolver - Thing used for registering sub-types, resolving them to super/sub-types as needed.

protected TypeFactory _typeFactory - Specific factory used for creating JavaType instances; needed to allow modules to add more custom type handling (mostly to support types of non-Java JVM languages).

protected static AnnotationIntrospector DEFAULT_ANNOTATION_INTROSPECTOR

protected static BaseSettings DEFAULT_BASE - Base settings contain defaults used for all ObjectMapper instances.

protected static VisibilityChecker<?> STD_VISIBILITY_CHECKER

Constructors

S.No.	Constructor & Description
1	**ObjectMapper()** Default constructor, which will construct the default JsonFactory as necessary, use SerializerProvider as its SerializerProvider, and BeanSerializerFactory as its SerializerFactory.
2	**ObjectMapper(JsonFactory jf)**

	Constructs instance that uses specified JsonFactory for constructing necessary JsonParsers and/or JsonGenerators.
3	**ObjectMapper(JsonFactory jf, SerializerProvider sp, DeserializerProvider dp)** Constructs instance that uses specified JsonFactory for constructing necessary JsonParsers and/or JsonGenerators, and uses given providers for accessing serializers and deserializers.
4	**protected ObjectMapper(ObjectMapper src)** Copy-constructor, mostly used to support copy().

Methods

Overridable helper method used for constructing SerializerProvider to use for serialization. void addMixInAnnotations(Class<?> target, Class<?> mixinSource) - Deprecated. Since 2.5: replaced by a fluent form of the method; addMixIn(Class, Class). protected DefaultDeserializationContext createDeserializationContext(JsonParser p, DeserializationConfig cfg) - Internal helper method called to create an instance of DeserializationContext for deserializing a single root value. JsonSchema generateJsonSchema(Class<?> t) - Deprecated. Since 2.6 use external JSON Schema generator (https://github.com/FasterXML/jackson-module-jsonSchema) (which under the hood calls acceptJsonFormatVisitor(JavaType, JsonFormatVisitorWrapper)) void registerSubtypes(Class<?>... classes) - Method for registering specified class as a subtype, so that typename-based resolution can link supertypes to subtypes (as an alternative to using annotations). void setFilters(FilterProvider filterProvider) - Deprecated. Since 2.6, use setFilterProvider(com.fasterxml.jackson.databind.ser.FilterProvider) instead (allows chaining) Factory method for constructing ObjectReader that will use specified character escaping details for output.

1	**protected void _checkInvalidCopy(Class<?> exp)**
2	**protected void _configAndWriteValue(JsonGenerator g, Object value)** - Method called to configure the generator as necessary and then call write functionality
3	**protected Object _convert(Object fromValue, JavaType toValueType)** - Actual conversion implementation: instead of using existing read and write methods, much of code is inlined.
4	**protected JsonDeserializer<Object> _findRootDeserializer(DeserializationContext ctxt, JavaType valueType)** - Method called to locate deserializer for the passed root-level value.
5	**protected JsonToken _initForReading(JsonParser p)** - Method called to ensure that given parser is ready for reading content for data binding.
6	**protected ObjectReader _newReader(DeserializationConfig config)** - Factory method sub-classes must override, to produce ObjectReader instances of proper sub-type

7	**protected ObjectReader _newReader(DeserializationConfig config, JavaType valueType, Object valueToUpdate, FormatSchema schema, InjectableValues injectableValues)** - Factory method sub-classes must override, to produce ObjectReader instances of proper sub-type
8	**protected ObjectWriter _newWriter(SerializationConfig config)** - Factory method sub-classes must override, to produce ObjectWriter instances of proper sub-type
9	**protected ObjectWriter _newWriter(SerializationConfig config, FormatSchema schema)** - Factory method sub-classes must override, to produce ObjectWriter instances of proper sub-type
10	**protected ObjectWriter _newWriter(SerializationConfig config, JavaType rootType, PrettyPrinter pp)** - Factory method sub-classes must override, to produce ObjectWriter instances of proper sub-type.
11	**protected Object _readMapAndClose(JsonParser p0, JavaType valueType)**
12	**protected Object _readValue(DeserializationConfig cfg, JsonParser p, JavaType valueType)** - Actual implementation of value reading+binding operation.
13	**protected DefaultSerializerProvider _serializerProvider(SerializationConfig config)**
14	**protected Object _unwrapAndDeserialize(JsonParser p, DeserializationContext ctxt, DeserializationConfig config, JavaType rootType, JsonDeserializer<Object> deser)**
15	**protected void _verifySchemaType(FormatSchema schema)**
16	**void acceptJsonFormatVisitor(Class<?> type, JsonFormatVisitorWrapper visitor)** - Method for visiting type hierarchy for given type, using specified visitor.
17	**void acceptJsonFormatVisitor(JavaType type, JsonFormatVisitorWrapper visitor)** - Method for visiting type hierarchy for given type, using specified visitor.
18	**ObjectMapper addHandler(DeserializationProblemHandler h)** - Method for adding specified DeserializationProblemHandler to be used for handling specific problems during deserialization.
19	**ObjectMapper addMixIn(Class<?> target, Class<?> mixinSource)** - Method to use for adding mix-in annotations to use for augmenting specified class or interface.
20	**boolean canDeserialize(JavaType type)** - Method that can be called to check whether mapper thinks it could deserialize an Object of given type.
21	**boolean canDeserialize(JavaType type, AtomicReference<Throwable> cause)** - Method similar to canDeserialize(JavaType) but that can return actual Throwable that was thrown when trying to construct serializer: this may be useful in figuring out what the actual problem is.
22	**boolean canSerialize(Class<?> type)** - Method that can be called to check whether

	mapper thinks it could serialize an instance of given Class.
23	**boolean canSerialize(Class<?> type, AtomicReference<Throwable> cause)** - Method similar to canSerialize(Class) but that can return actual Throwable that was thrown when trying to construct serializer: this may be useful in figuring out what the actual problem is.
24	**ObjectMapper clearProblemHandlers()** - Method for removing all registered DeserializationProblemHandlers instances from this mapper.
25	**MutableConfigOverride configOverride(Classlt;?> type)** - Accessor for getting a mutable configuration override object for given type, needed to add or change per-type overrides applied to properties of given type.
26	**ObjectMapper configure(DeserializationFeature f, boolean state)** - Method for changing state of an on/off deserialization feature for this object mapper.
27	**ObjectMapper configure(JsonGenerator.Feature f, boolean state)** - Method for changing state of an on/off JsonGenerator feature for generator instances this object mapper creates.
28	**ObjectMapper configure(JsonParser.Feature f, boolean state)** - Method for changing state of specified JsonParser.Features for parser instances this object mapper creates.
29	**ObjectMapper configure(MapperFeature f, boolean state)** - Method for changing state of an on/off mapper feature for this mapper instance.
30	**ObjectMapper configure(SerializationFeature f, boolean state)** - Method for changing state of an on/off serialization feature for this object mapper.
31	**JavaType constructType(Type t)** - Convenience method for constructing JavaType out of given type (typically java.lang.Class), but without explicit context.
32	**<T> T convertValue(Object fromValue, Class<T> toValueType)** - Convenience method for doing two-step conversion from given value, into instance of given value type, if (but only if!) conversion is needed.
33	**<T> T convertValue(Object fromValue, JavaType toValueType)** - See convertValue(Object, Class)
34	**<T> T convertValue(Object fromValue, TypeReference<?> toValueTypeRef)** - See convertValue(Object, Class)
35	**ObjectMapper copy()** - Method for creating a new ObjectMapper instance that has same initial configuration as this instance.
36	**ArrayNode createArrayNode()** - Note: return type is co-variant, as basic ObjectCodec abstraction can not refer to concrete node types (as it's part of core package, whereas impls are part of mapper package)
37	**ObjectNode createObjectNode()** - Note: return type is co-variant, as basic ObjectCodec abstraction can not refer to concrete node types (as it's part of core package, whereas impls

	are part of mapper package)
38	**protected ClassIntrospector defaultClassIntrospector()** - Overridable helper method used to construct default ClassIntrospector to use.
39	**ObjectMapper disable(DeserializationFeature feature)** - Method for enabling specified DeserializationConfig features.
40	**ObjectMapper disable(DeserializationFeature first, DeserializationFeature... f)** - Method for enabling specified DeserializationConfig features.
41	**ObjectMapper disable(JsonGenerator.Feature... features)** - Method for disabling specified JsonGenerator.Features for parser instances this object mapper creates.
42	**ObjectMapper disable(JsonParser.Feature... features)** - Method for disabling specified JsonParser.Features for parser instances this object mapper creates.
43	**ObjectMapper disable(MapperFeature... f)** - Method for enabling specified DeserializationConfig features.
44	**ObjectMapper disable(SerializationFeature f)** - Method for enabling specified DeserializationConfig features.
45	**ObjectMapper disable(SerializationFeature first, SerializationFeature... f)** - Method for enabling specified DeserializationConfig features.
46	**ObjectMapper disableDefaultTyping()** - Method for disabling automatic inclusion of type information; if so, only explicitly annotated types (ones with JsonTypeInfo) will have additional embedded type information.
47	**ObjectMapper enable(DeserializationFeature feature)** - Method for enabling specified DeserializationConfig features.
48	**ObjectMapper enable(DeserializationFeature first, DeserializationFeature... f)** - Method for enabling specified DeserializationConfig features.
49	**ObjectMapper enable(JsonGenerator.Feature... features)** - Method for enabling specified JsonGenerator.Features for parser instances this object mapper creates.
50	**ObjectMapper enable(JsonParser.Feature... features)** - Method for enabling specified JsonParser.Features for parser instances this object mapper creates.
51	**ObjectMapper enable(MapperFeature... f)** - Method for enabling specified MapperConfig features.
52	**ObjectMapper enable(SerializationFeature f)** - Method for enabling specified DeserializationConfig feature.
53	**ObjectMapper enable(SerializationFeature first, SerializationFeature... f)** - Method for enabling specified DeserializationConfig features.
54	**ObjectMapper enableDefaultTyping()** - Convenience method that is equivalent to calling

55	**ObjectMapper enableDefaultTyping(ObjectMapper.DefaultTyping dti)** - Convenience method that is equivalent to calling
56	**ObjectMapper enableDefaultTyping(ObjectMapper.DefaultTyping applicability, JsonTypeInfo.As includeAs)** - Method for enabling automatic inclusion of type information, needed for proper deserialization of polymorphic types (unless types have been annotated with JsonTypeInfo).
57	**ObjectMapper enableDefaultTypingAsProperty(ObjectMapper.DefaultTyping applicability, String propertyName)** - Method for enabling automatic inclusion of type information -- needed for proper deserialization of polymorphic types (unless types have been annotated with JsonTypeInfo) -- using "As.PROPERTY" inclusion mechanism and specified property name to use for inclusion (default being "@class" since default type information always uses class name as type identifier)
58	**ObjectMapper findAndRegisterModules()** - Convenience method that is functionally equivalent to: mapper.registerModules(mapper.findModules());
59	**Class<?> findMixInClassFor(Class<?> cls)**
60	**static List<Module> findModules()** - Method for locating available methods, using JDK ServiceLoader facility, along with module-provided SPI.
61	**static List<Module> findModules(ClassLoader classLoader)** - Method for locating available methods, using JDK ServiceLoader facility, along with module-provided SPI.
62	**DateFormat getDateFormat()**
63	**DeserializationConfig getDeserializationConfig()** - Method that returns the shared default DeserializationConfig object that defines configuration settings for deserialization.
64	**DeserializationContext getDeserializationContext()** - Method for getting current DeserializationContext.
65	**JsonFactory getFactory()** - Method that can be used to get hold of JsonFactory that this mapper uses if it needs to construct JsonParsers and/or JsonGenerators.
66	**InjectableValues getInjectableValues()**
67	**JsonFactory getJsonFactory()** - Deprecated. Since 2.1: Use getFactory() instead
68	**JsonNodeFactory getNodeFactory()** - Method that can be used to get hold of JsonNodeFactory that this mapper will use when directly constructing root JsonNode instances for Trees.
69	**PropertyNamingStrategy getPropertyNamingStrategy()**
70	**SerializationConfig getSerializationConfig()** - Method that returns the shared default SerializationConfig object that defines configuration settings for serialization.
71	**SerializerFactory getSerializerFactory()** - Method for getting current SerializerFactory.
72	**SerializerProvider getSerializerProvider()** - Accessor for the "blueprint" (or, factory)

	instance, from which instances are created by calling DefaultSerializerProvider.createInstance(com.fasterxml.jackson.databind.SerializationConfig, com.fasterxml.jackson.databind.ser.SerializerFactory).
73	**SerializerProvider getSerializerProviderInstance()** - Accessor for constructing and returning a SerializerProvider instance that may be used for accessing serializers.
74	**SubtypeResolver getSubtypeResolver()** - Method for accessing subtype resolver in use.
75	**TypeFactory getTypeFactory()** - Accessor for getting currently configured TypeFactory instance.
76	**VisibilityChecker<?> getVisibilityChecker()** - Method for accessing currently configured visibility checker; object used for determining whether given property element (method, field, constructor) can be auto-detected or not.
77	**boolean isEnabled(DeserializationFeature f)** - Method for checking whether given deserialization-specific feature is enabled.
78	**boolean isEnabled(JsonFactory.Feature f)** - Convenience method, equivalent to:
79	**boolean isEnabled(JsonGenerator.Feature f)**
80	**boolean isEnabled(JsonParser.Feature f)**
81	**boolean isEnabled(MapperFeature f)** - Method for checking whether given MapperFeature is enabled.
82	**boolean isEnabled(SerializationFeature f)** - Method for checking whether given serialization-specific feature is enabled.
83	**int mixInCount()**
84	**ObjectReader reader()** - Factory method for constructing ObjectReader with default settings.
85	**ObjectReader reader(Base64Variant defaultBase64)** - Factory method for constructing ObjectReader that will use specified Base64 encoding variant for Base64-encoded binary data.
86	**ObjectReader reader(Class<?> type)** - Deprecated. Since 2.5, use readerFor(Class) instead
87	**ObjectReader reader(ContextAttributes attrs)** - Factory method for constructing ObjectReader that will use specified default attributes.
88	**ObjectReader reader(DeserializationFeature feature)** - Factory method for constructing ObjectReader with specified feature enabled (compared to settings that this mapper instance has).
89	**ObjectReader reader(DeserializationFeature first, DeserializationFeature... other)** - Factory method for constructing ObjectReader with specified features enabled (compared to settings that this mapper instance has).

90	**ObjectReader reader(FormatSchema schema)** - Factory method for constructing ObjectReader that will pass specific schema object to JsonParser used for reading content.
91	**ObjectReader reader(InjectableValues injectableValues)** - Factory method for constructing ObjectReader that will use specified injectable values.
92	**ObjectReader reader(JavaType type)** - Deprecated. Since 2.5, use readerFor(JavaType) instead
93	**ObjectReader reader(JsonNodeFactory f)** - Factory method for constructing ObjectReader that will use specified JsonNodeFactory for constructing JSON trees.
94	**ObjectReader reader(TypeReference<?> type)** - Deprecated. Since 2.5, use readerFor(TypeReference) instead
95	**ObjectReader readerFor(Class<?> type)** - Factory method for constructing ObjectReader that will read or update instances of specified type
96	**ObjectReader readerFor(JavaType type)** - Factory method for constructing ObjectReader that will read or update instances of specified type
97	**ObjectReader readerFor(TypeReference<?> type)** - Factory method for constructing ObjectReader that will read or update instances of specified type
98	**ObjectReader readerForUpdating(Object valueToUpdate)** - Factory method for constructing ObjectReader that will update given Object (usually Bean, but can be a Collection or Map as well, but NOT an array) with JSON data.
99	**ObjectReader readerWithView(Class<?> view)** - Factory method for constructing ObjectReader that will deserialize objects using specified JSON View (filter).
100	**JsonNode readTree(byte[] content)** - Method to deserialize JSON content as tree expressed using set of JsonNode instances.
101	**JsonNode readTree(File file)** - Method to deserialize JSON content as tree expressed using set of JsonNode instances.
102	**JsonNode readTree(InputStream in)** - Method to deserialize JSON content as tree expressed using set of JsonNode instances.
103	**<T extends TreeNode> T readTree(JsonParser p)** - Method to deserialize JSON content as tree expressed using set of JsonNode instances.
104	**JsonNode readTree(Reader r)** - Method to deserialize JSON content as tree expressed using set of JsonNode instances.
105	**JsonNode readTree(String content)** - Method to deserialize JSON content as tree expressed using set of JsonNode instances.
106	**JsonNode readTree(URL source)** - Method to deserialize JSON content as tree expressed using set of JsonNode instances.

107	**<T> T readValue(byte[] src, Class<T> valueType)**
108	**<T> T readValue(byte[] src, int offset, int len, Class<T> valueType)**
109	**<T> T readValue(byte[] src, int offset, int len, JavaType valueType)**
110	**<T> T readValue(byte[] src, int offset, int len, TypeReference valueTypeRef)**
111	**<T> T readValue(byte[] src, JavaType valueType)**
112	**<T> T readValue(byte[] src, TypeReference valueTypeRef)**
113	**<T> T readValue(DataInput src, Class<T> valueType)**
114	**<T> T readValue(DataInput src, JavaType valueType)**
115	**<T> T readValue(File src, Class<T> valueType)** - Method to deserialize JSON content from given file into given Java type.
116	**<T> T readValue(File src, JavaType valueType)** - Method to deserialize JSON content from given file into given Java type.
117	**<T> T readValue(File src, TypeReference valueTypeRef)** - Method to deserialize JSON content from given file into given Java type.
118	**<T> T readValue(InputStream src, Class<T> valueType)**
119	**<T> T readValue(InputStream src, JavaType valueType)**
120	**<T> T readValue(InputStream src, TypeReference valueTypeRef)**
121	**<T> T readValue(JsonParser p, Class<T> valueType) - Method to deserialize JSON content into a non-container type (it can be an array type, however): typically a bean, array or a wrapper type (like Boolean).**
122	**<T> T readValue(JsonParser p, JavaType valueType)** - Type-safe overloaded method, basically alias for readValue(JsonParser, Class).
123	**<T> T readValue(JsonParser p, ResolvedType valueType)** - Method to deserialize JSON content into a Java type, reference to which is passed as argument.
124	**<T> T readValue(JsonParser p, TypeReference<?> valueTypeRef)** - Method to deserialize JSON content into a Java type, reference to which is passed as argument.
125	**<T> T readValue(Reader src, Class<T> valueType)** -
1	**<T> T readValue(Reader src, JavaType valueType)**
126	**<T> T readValue(Reader src, TypeReference valueTypeRef)**
127	**<T> T readValue(String content, Class<T> valueType)** - Method to deserialize JSON content from given JSON content String.
128	**<T> T readValue(String content, JavaType valueType)** - Method to deserialize JSON content from given JSON content String.

129	**<T> T readValue(String content, TypeReference valueTypeRef)** - Method to deserialize JSON content from given JSON content String.
130	**<T> T readValue(URL src, Class<T> valueType)** - Method to deserialize JSON content from given resource into given Java type.
131	**<T> T readValue(URL src, JavaType valueType)**
132	**<T> T readValue(URL src, TypeReference valueTypeRef)** - Method to deserialize JSON content from given resource into given Java type.
133	**<T> MappingIterator<T> readValues(JsonParser p, Class<T> valueType)** - Convenience method, equivalent in function to:
134	**<T> MappingIterator<T> readValues(JsonParser p, JavaType valueType)** - Convenience method, equivalent in function to:
135	**<T> MappingIterator<T> readValues(JsonParser p, ResolvedType valueType)** - Convenience method, equivalent in function to:
136	**<T> MappingIterator<T> readValues(JsonParser p, TypeReference<?>valueTypeRef)** - Method for reading sequence of Objects from parser stream.
137	**ObjectMapper registerModule(Module module)** - Method for registering a module that can extend functionality provided by this mapper; for example, by adding providers for custom serializers and deserializers.
138	**ObjectMapper registerModules(Iterable<Module> modules)** - Convenience method for registering specified modules in order; functionally equivalent to:
139	**ObjectMapper registerModules(Module... modules)** - Convenience method for registering specified modules in order; functionally equivalent to:
140	**void registerSubtypes(NamedType... types)** - Method for registering specified class as a subtype, so that typename-based resolution can link supertypes to subtypes (as an alternative to using annotations).
141	**ObjectMapper setAnnotationIntrospector(AnnotationIntrospector ai)** - Method for setting AnnotationIntrospector used by this mapper instance for both serialization and deserialization.
142	**ObjectMapper setAnnotationIntrospectors(AnnotationIntrospector serializerAI, AnnotationIntrospector deserializerAI)** - Method for changing AnnotationIntrospector instances used by this mapper instance for serialization and deserialization, specifying them separately so that different introspection can be used for different aspects.
143	**ObjectMapper setBase64Variant(Base64Variant v)** - Method that will configure default Base64Variant that byte[] serializers and deserializers will use.
144	**ObjectMapper setConfig(DeserializationConfig config)** - Method that allows overriding of the underlying DeserializationConfig object.

145	**ObjectMapper setConfig(SerializationConfig config)** - Method that allows overriding of the underlying SerializationConfig object, which contains serialization-specific configuration settings.
146	**ObjectMapper setDateFormat(DateFormat dateFormat)** - Method for configuring the default DateFormat to use when serializing time values as Strings, and deserializing from JSON Strings.
147	**ObjectMapper setDefaultPrettyPrinter(PrettyPrinter pp)** - Method for specifying PrettyPrinter to use when "default pretty-printing" is enabled (by enabling SerializationFeature.INDENT_OUTPUT)
148	**ObjectMapper setDefaultTyping(TypeResolverBuilder<?> typer)** - Method for enabling automatic inclusion of type information, using specified handler object for determining which types this affects, as well as details of how information is embedded.
149	**ObjectMapper setFilterProvider(FilterProvider filterProvider)** - Method for configuring this mapper to use specified FilterProvider for mapping Filter Ids to actual filter instances.
150	**Object setHandlerInstantiator(HandlerInstantiator hi)** - Method for configuring HandlerInstantiator to use for creating instances of handlers (such as serializers, deserializers, type and type id resolvers), given a class.
151	**ObjectMapper setInjectableValues(InjectableValues injectableValues)** - Method for configuring InjectableValues which used to find values to inject.
152	**ObjectMapper setLocale(Locale l)** - Method for overriding default locale to use for formatting.
153	**void setMixInAnnotations(Map<Class<?>,Class<?>> sourceMixins)** - Deprecated. Since 2.5: replaced by a fluent form of the method; setMixIns(java.util.Map<java.lang.Class<?>, java.lang.Class<?>>).
154	**ObjectMapper setMixInResolver(ClassIntrospector.MixInResolver resolver)** - Method that can be called to specify given resolver for locating mix-in classes to use, overriding directly added mappings.
155	**ObjectMapper setMixIns(Map<Class<?>,Class<?>> sourceMixins)** - Method to use for defining mix-in annotations to use for augmenting annotations that processable (serializable / deserializable) classes have.
156	**ObjectMapper setNodeFactory(JsonNodeFactory f)** - Method for specifying JsonNodeFactory to use for constructing root level tree nodes (via method createObjectNode()
157	**ObjectMapper setPropertyInclusion(JsonInclude.Value incl)** - Method for setting default POJO property inclusion strategy for serialization.
158	**ObjectMapper setPropertyNamingStrategy(PropertyNamingStrategy s)** - Method for setting custom property naming strategy to use.

159	**ObjectMapper setSerializationInclusion(JsonInclude.Include incl)** - Convenience method, equivalent to calling:
160	**ObjectMapper setSerializerFactory(SerializerFactory f)** - Method for setting specific SerializerFactory to use for constructing (bean) serializers.
161	**ObjectMapper setSerializerProvider(DefaultSerializerProvider p)** - Method for setting "blueprint" SerializerProvider instance to use as the base for actual provider instances to use for handling caching of JsonSerializer instances.
162	**ObjectMapper setSubtypeResolver(SubtypeResolver str)** - Method for setting custom subtype resolver to use.
163	**ObjectMapper setTimeZone(TimeZone tz)** - Method for overriding default TimeZone to use for formatting.
164	**ObjectMapper setTypeFactory(TypeFactory f)** - Method that can be used to override TypeFactory instance used by this mapper.
165	**ObjectMapper setVisibility(PropertyAccessor forMethod, JsonAutoDetect.Visibility visibility)** - Convenience method that allows changing configuration for underlying VisibilityCheckers, to change details of what kinds of properties are auto-detected.
166	**ObjectMapper setVisibility(VisibilityChecker<?> vc)** - Method for setting currently configured VisibilityChecker, object used for determining whether given property element (method, field, constructor) can be auto-detected or not.
167	**void setVisibilityChecker(VisibilityChecker<?> vc)** - Deprecated. Since 2.6 use setVisibility(VisibilityChecker) instead.
168	**JsonParser treeAsTokens(TreeNode n)** - Method for constructing a JsonParser out of JSON tree representation.
169	**<T> T treeToValue(TreeNode n, Class<T> valueType)** - Convenience conversion method that will bind data given JSON tree contains into specific value (usually bean) type.
170	**<T extends JsonNode> T valueToTree(Object fromValue)** - Reverse of treeToValue(com.fasterxml.jackson.core.TreeNode, java.lang.Class<T>); given a value (usually bean), will construct equivalent JSON Tree representation.
171	**Version version()** - Method that will return version information stored in and read from jar that contains this class.
172	**ObjectWriter writer()** - Convenience method for constructing ObjectWriter with default settings.
173	**ObjectWriter writer(Base64Variant defaultBase64)** - Factory method for constructing ObjectWriter that will use specified Base64 encoding variant for Base64-encoded binary data.
174	**ObjectWriter writer(CharacterEscapes escapes)** -

175	**ObjectWriter writer(ContextAttributes attrs)** - Factory method for constructing ObjectWriter that will use specified default attributes.
176	**ObjectWriter writer(DateFormat df)** - Factory method for constructing ObjectWriter that will serialize objects using specified DateFormat; or, if null passed, using timestamp (64-bit number.
177	**ObjectWriter writer(FilterProvider filterProvider)** - Factory method for constructing ObjectWriter that will serialize objects using specified filter provider.
178	**ObjectWriter writer(FormatSchema schema)** - Factory method for constructing ObjectWriter that will pass specific schema object to JsonGenerator used for writing content.
179	**ObjectWriter writer(PrettyPrinter pp)** - Factory method for constructing ObjectWriter that will serialize objects using specified pretty printer for indentation (or if null, no pretty printer)
180	**ObjectWriter writer(SerializationFeature feature)** - Factory method for constructing ObjectWriter with specified feature enabled (compared to settings that this mapper instance has).
181	**ObjectWriter writer(SerializationFeature first, SerializationFeature... other)** - Factory method for constructing ObjectWriter with specified features enabled (compared to settings that this mapper instance has).
182	**ObjectWriter writerFor(Class<?> rootType)** - Factory method for constructing ObjectWriter that will serialize objects using specified root type, instead of actual runtime type of value.
183	**ObjectWriter writerFor(JavaType rootType)** - Factory method for constructing ObjectWriter that will serialize objects using specified root type, instead of actual runtime type of value.
184	**ObjectWriter writerFor(TypeReference<?> rootType)** - Factory method for constructing ObjectWriter that will serialize objects using specified root type, instead of actual runtime type of value.
185	**ObjectWriter writerWithDefaultPrettyPrinter()** - Factory method for constructing ObjectWriter that will serialize objects using the default pretty printer for indentation.
186	**ObjectWriter writerWithType(Class<?> rootType)** - Deprecated. Since 2.5, use writerFor(Class) instead.
187	**ObjectWriter writerWithType(JavaType rootType)** - Deprecated. Since 2.5, use writerFor(JavaType) instead.
188	**ObjectWriter writerWithType(TypeReference<?> rootType)** - Deprecated. Since 2.5, use writerFor(TypeReference) instead.
189	**ObjectWriter writerWithView(Class<?> serializationView)** - Factory method for constructing ObjectWriter that will serialize objects using specified JSON View (filter).

190	**void writeTree(JsonGenerator jgen, JsonNode rootNode)** - Method to serialize given JSON Tree, using generator provided.
191	**void writeTree(JsonGenerator jgen, TreeNode rootNode)**
192	**void writeValue(DataOutput out, Object value)**
193	**void writeValue(File resultFile, Object value)** - Method that can be used to serialize any Java value as JSON output, written to File provided.
194	**void writeValue(JsonGenerator g, Object value)** - Method that can be used to serialize any Java value as JSON output, using provided JsonGenerator.
195	**void writeValue(OutputStream out, Object value)** - Method that can be used to serialize any Java value as JSON output, using output stream provided (using encoding JsonEncoding.UTF8).
196	**void writeValue(Writer w, Object value)** - Method that can be used to serialize any Java value as JSON output, using Writer provided.
197	**byte[] writeValueAsBytes(Object value)** - Method that can be used to serialize any Java value as a byte array.
198	**String writeValueAsString(Object value)** - Method that can be used to serialize any Java value as a String.

Methods inherited

This class inherits methods from the following classes:

java.lang.Object

ObjectMapper Example

Create the following java program using any editor of your choice in say **C:/> Jackson_WORKSPACE**

File: JacksonTester.java

```
import java.io.IOException;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){

      ObjectMapper mapper = new ObjectMapper();
      String jsonString = "{\"name\":\"Mahesh\", \"age\":21}";

      //map json to student
      try{
         Student student = mapper.readValue(jsonString, Student.class);

         System.out.println(student);

         jsonString = mapper.writerWithDefaultPrettyPrinter().writeValueAsString(student);

         System.out.println(jsonString);
      }
      catch (JsonParseException e) { e.printStackTrace();}
      catch (JsonMappingException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace(); }
   }
}

class Student {
   private String name;
   private int age;
   public Student(){}
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Student [ name: Mahesh, age: 21 ]
{
  "name" : "Mahesh",
  "age" : 21
}
```

Object Serialization

let's serialize a java object to a json file and then read that json file to get the object back. In this example, we've created Student class. We'll create a student.json file which will have a json representation of Student object.

Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerationException;
import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         Student student = new Student();
         student.setAge(10);
         student.setName("Mahesh");
         tester.writeJSON(student);

         Student student1 = tester.readJSON();
         System.out.println(student1);

      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   private void writeJSON(Student student) throws JsonGenerationException, JsonMappingException,
      ObjectMapper mapper = new ObjectMapper();
      mapper.writeValue(new File("student.json"), student);
   }

   private Student readJSON() throws JsonParseException, JsonMappingException, IOException{
      ObjectMapper mapper = new ObjectMapper();
      Student student = mapper.readValue(new File("student.json"), Student.class);
      return student;
   }
}

class Student {
   private String name;
   private int age;
   public Student(){}
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
```

```
    public int getAge() {
       return age;
    }
    public void setAge(int age) {
       this.age = age;
    }
    public String toString(){
       return "Student [ name: "+name+", age: "+ age+ " ]";
    }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Student [ name: Mahesh, age: 10 ]
```

Data Binding

Data Binding API is used to convert JSON to and from POJO (Plain Old Java Object) using property accessor or using annotations. It is of two type.

> **Simple Data Binding** - Converts JSON to and from Java Maps, Lists, Strings, Numbers, Booleans and null objects.
>
> **Full Data Binding** - Converts JSON to and from any JAVA type.

ObjectMapper reads/writes JSON for both types of data bindings. Data Binding is most convenient way and is analogus to JAXB parer for XML.

Simple Data Binding

Simple data binding refers to mapping of JSON to JAVA Core data types. Following table illustrates the relationship between JSON types vs Java Types.

Sr. No.	JSON Type	Java Type
1	object	LinkedHashMap<String,Object>
2	array	ArrayList<Object>
3	string	String

4	complete number	Integer, Long or BigInteger
5	fractional number	Double / BigDecimal
6	true | false	Boolean
7	null	null

Let's see simple data binding in action. Here we'll map JAVA basic types directly to JSON and vice versa.

Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
         try {
            ObjectMapper mapper = new ObjectMapper();

            Map<String,Object> studentDataMap = new HashMap<String,Object>();
            int[] marks = {1,2,3};

            Student student = new Student();
            student.setAge(10);
            student.setName("Mahesh");
            // JAVA Object
            studentDataMap.put("student", student);
            // JAVA String
            studentDataMap.put("name", "Mahesh Kumar");
            // JAVA Boolean
            studentDataMap.put("verified", Boolean.FALSE);
            // Array
            studentDataMap.put("marks", marks);

            mapper.writeValue(new File("student.json"), studentDataMap);
            //result student.json
                       //{
            //   "student":{"name":"Mahesh","age":10},
            //   "marks":[1,2,3],
            //   "verified":false,
            //   "name":"Mahesh Kumar"
            //}
            studentDataMap = mapper.readValue(new File("student.json"), Map.class);

            System.out.println(studentDataMap.get("student"));
            System.out.println(studentDataMap.get("name"));
            System.out.println(studentDataMap.get("verified"));
            System.out.println(studentDataMap.get("marks"));
      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
            e.printStackTrace();
      }
   }
}

class Student {
   private String name;
   private int age;
   public Student(){}
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
{name=Mahesh, age=10}

Mahesh Kumar

false

[1, 2, 3]
```

Full Data Binding

Full data binding refers to mapping of JSON to any JAVA Object.

```
//Create an ObjectMapper instance
ObjectMapper mapper = new ObjectMapper();
//map JSON content to Student object
Student student = mapper.readValue(new File("student.json"), Student.class);
```

```java
//map Student object to JSON content
mapper.writeValue(new File("student.json"), student);
```

Let's see simple data binding in action. Here we'll map JAVA Object directly to JSON and vice versa.

Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```java
import java.io.File;
import java.io.IOException;

import com.fasterxml.jackson.core.JsonGenerationException;
import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         Student student = new Student();
         student.setAge(10);
         student.setName("Mahesh");
         tester.writeJSON(student);

         Student student1 = tester.readJSON();
         System.out.println(student1);

      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   private void writeJSON(Student student) throws JsonGenerationException, JsonMappingException,
      ObjectMapper mapper = new ObjectMapper();
      mapper.writeValue(new File("student.json"), student);
   }

   private Student readJSON() throws JsonParseException, JsonMappingException, IOException{
      ObjectMapper mapper = new ObjectMapper();
      Student student = mapper.readValue(new File("student.json"), Student.class);
      return student;
   }
}

class Student {
   private String name;
   private int age;
   public Student(){}
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Student [ name: Mahesh, age: 10 ]
```

Data Binding with Generics

In simple data binding, we've used Map class which use String as key and Object as a value object. Instead we can have concrete java object and type cast it to be used in JSON binding.

Consider the following example with a class UserData, a class to hold user specific data.

Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.Map;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
         try {
```

```java
            ObjectMapper mapper = new ObjectMapper();

            Map<String, UserData> userDataMap = new HashMap<String, UserData>();
            UserData studentData = new UserData();
            int[] marks = {1,2,3};

            Student student = new Student();
            student.setAge(10);
            student.setName("Mahesh");
            // JAVA Object
            studentData.setStudent(student);
            // JAVA String
            studentData.setName("Mahesh Kumar");
            // JAVA Boolean
            studentData.setVerified(Boolean.FALSE);
            // Array
            studentData.setMarks(marks);
            TypeReference ref = new TypeReference<Map<String,UserData>>() { };
            userDataMap.put("studentData1", studentData);
            mapper.writeValue(new File("student.json"), userDataMap);
            //{
            //   "studentData1":
            //   {
            //          "student":
            //          {
            //                  "name":"Mahesh",
            //                  "age":10
            //      },
            //      "name":"Mahesh Kumar",
            //      "verified":false,
            //      "marks":[1,2,3]
            //   }
            //}
            userDataMap = mapper.readValue(new File("student.json"), ref);

            System.out.println(userDataMap.get("studentData1").getStudent());
            System.out.println(userDataMap.get("studentData1").getName());
            System.out.println(userDataMap.get("studentData1").getVerified());
            System.out.println(Arrays.toString(userDataMap.get("studentData1").getMarks()));
        } catch (JsonParseException e) {
           e.printStackTrace();
        } catch (JsonMappingException e) {
           e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class Student {
   private String name;
   private int age;
   public Student(){}
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public String toString(){
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }
}

class UserData {
   private Student student;
   private String name;
   private Boolean verified;
   private int[] marks;

   public UserData(){}

   public Student getStudent() {
      return student;
   }
   public void setStudent(Student student) {
      this.student = student;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public Boolean getVerified() {
      return verified;
   }
   public void setVerified(Boolean verified) {
      this.verified = verified;
   }
   public int[] getMarks() {
      return marks;
   }
   public void setMarks(int[] marks) {
      this.marks = marks;
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Student [ name: Mahesh, age: 10 ]
Mahesh Kumar
```

```
false
[1, 2, 3]
```

Tree Model

Tree Model prepares a in-memory tree representation of the JSON document. ObjectMapper build tree of JsonNode nodes. It is most flexible approach. It is analogus to DOM parser for XML.

Create Tree from JSON

ObjectMapper provides a pointer to root node of the tree after reading the JSON. Root Node can be used to traverse the complete tree. Consider the following code snippet to get the root node of a provided JSON String.

```
//Create an ObjectMapper instance
ObjectMapper mapper = new ObjectMapper();
String jsonString = "{\"name\":\"Mahesh Kumar\", \"age\":21,\"verified\":false,\"marks\": [100,90
//create tree from JSON
JsonNode rootNode = mapper.readTree(jsonString);
```

Traversing Tree Model

Get each node using relative path to the root node while traversing tree and process the data. Consider the following code snippet traversing the tree provided the root node.

```
JsonNode nameNode = rootNode.path("name");
System.out.println("Name: "+ nameNode.textValue());

JsonNode marksNode = rootNode.path("marks");
Iterator<JsonNode> iterator = marksNode.elements();
```

Example

Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.IOException;
import java.util.Iterator;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      try {
         ObjectMapper mapper = new ObjectMapper();
         String jsonString = "{\"name\":\"Mahesh Kumar\",  \"age\":21,\"verified\":false,\"marks\
         JsonNode rootNode = mapper.readTree(jsonString);

         JsonNode nameNode = rootNode.path("name");
         System.out.println("Name: "+ nameNode.textValue());

         JsonNode ageNode = rootNode.path("age");
         System.out.println("Age: " + ageNode.intValue());

         JsonNode verifiedNode = rootNode.path("verified");
         System.out.println("Verified: " + (verifiedNode.booleanValue() ? "Yes":"No"));

         JsonNode marksNode = rootNode.path("marks");
         Iterator<JsonNode> iterator = marksNode.elements();
         System.out.print("Marks: [ ");

         while (iterator.hasNext()) {
            JsonNode marks = iterator.next();
            System.out.print(marks.intValue() + " ");
         }

         System.out.println("]");
      }
      catch (JsonParseException e) { e.printStackTrace(); }
      catch (JsonMappingException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace(); }
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Name: Mahesh Kumar
Age: 21
Verified: No
Marks: [ 100 90 85 ]
```

Tree to JSON

In this example, we've created a Tree using JsonNode and write it to a json file and read back.

Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.IOException;
import java.util.Iterator;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){

      try {
         ObjectMapper mapper = new ObjectMapper();
         String jsonString = "{\"name\":\"Mahesh Kumar\",  \"age\":21,\"verified\":false,\"marks\
         JsonNode rootNode = mapper.readTree(jsonString);

         JsonNode nameNode = rootNode.path("name");
         System.out.println("Name: "+ nameNode.textValue());

         JsonNode ageNode = rootNode.path("age");
         System.out.println("Age: " + ageNode.intValue());

         JsonNode verifiedNode = rootNode.path("verified");
         System.out.println("Verified: " + (verifiedNode.booleanValue() ? "Yes":"No"));

         JsonNode marksNode = rootNode.path("marks");
         Iterator<JsonNode> iterator = marksNode.elements();
         System.out.print("Marks: [ ");

         while (iterator.hasNext()) {
            JsonNode marks = iterator.next();
            System.out.print(marks.intValue() + " ");
         }

         System.out.println("]");
      }
      catch (JsonParseException e) { e.printStackTrace(); }
      catch (JsonMappingException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace(); }
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Name: Mahesh Kumar
Age: 21
Verified: No
Marks: [ 100 90 85 ]
```

Tree to Java Objects

In this example, we've created a Tree using JsonNode and write it to a json file and read back tree and then convert it as a Student object.

Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```java
import java.io.File;
import java.io.IOException;
import java.util.Iterator;

import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.node.ArrayNode;
import com.fasterxml.jackson.databind.node.ObjectNode;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         ObjectMapper mapper = new ObjectMapper();

         JsonNode rootNode = mapper.createObjectNode();
         JsonNode marksNode = mapper.createArrayNode();
         ((ArrayNode)marksNode).add(100);
         ((ArrayNode)marksNode).add(90);
         ((ArrayNode)marksNode).add(85);
         ((ObjectNode) rootNode).put("name", "Mahesh Kumar");
         ((ObjectNode) rootNode).put("age", 21);
         ((ObjectNode) rootNode).put("verified", false);
         ((ObjectNode) rootNode).put("marks",marksNode);

         mapper.writeValue(new File("student.json"), rootNode);

         rootNode = mapper.readTree(new File("student.json"));

         Student student = mapper.treeToValue(rootNode, Student.class);

         System.out.println("Name: "+ student.getName());
         System.out.println("Age: " + student.getAge());
         System.out.println("Verified: " + (student.isVerified() ? "Yes":"No"));
         System.out.println("Marks: "+Arrays.toString(student.getMarks()));
      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
class Student {
   String name;
   int age;
   boolean verified;
   int[] marks;
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getAge() {
      return age;
   }
   public void setAge(int age) {
      this.age = age;
   }
   public boolean isVerified() {
      return verified;
   }
   public void setVerified(boolean verified) {
      this.verified = verified;
   }
   public int[] getMarks() {
      return marks;
   }
   public void setMarks(int[] marks) {
      this.marks = marks;
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Name: Mahesh Kumar

Age: 21

Verified: No

Marks: [ 100 90 85 ]
```

Streaming API

Streaming API reads and writes JSON content as discrete events. JsonParser reads the data whereas JsonGenerator writes the data. It is most powerful approach among the three and is of lowest overhead and fastest in read/write opreations. It is Analogus to Stax parser for XML.

In this Article, we'll showcase using Jackson streaming APIs to read and write JSON data. Streaming API works with concept of token and every details of Json is to be handle carefuly. Following are two class which we'll use in the examples:

JsonGenerator	- Write to JSON String.
JsonParser	- Parse JSON String.

Writing JSON using JsonGenerator

Using JsonGenerator is pretty simple. First create the JsonGenerator using JsonFactory.createJsonGenerator() method and use it's write***() methods to write each json value.

```
JsonFactory jsonFactory = new JsonFactory();
JsonGenerator jsonGenerator = jsonFactory.createGenerator(new File(
   "student.json"), JsonEncoding.UTF8);
// {
jsonGenerator.writeStartObject();
// "name" : "Mahesh Kumar"
jsonGenerator.writeStringField("name", "Mahesh Kumar");
```

Let's see JsonGenerator in action. Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;
import java.util.Map;

import com.fasterxml.jackson.core.JsonEncoding;
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){

      try {
         JsonFactory jsonFactory = new JsonFactory();
         JsonGenerator jsonGenerator = jsonFactory.createGenerator(new File("student.json"), Json

         jsonGenerator.writeStartObject();

         // "name" : "Mahesh Kumar"
         jsonGenerator.writeStringField("name", "Mahesh Kumar");

         // "age" : 21
         jsonGenerator.writeNumberField("age", 21);

         // "verified" : false
         jsonGenerator.writeBooleanField("verified", false);

         // "marks" : [100, 90, 85]
         jsonGenerator.writeFieldName("marks");

         // [
         jsonGenerator.writeStartArray();
         // 100, 90, 85
         jsonGenerator.writeNumber(100);
         jsonGenerator.writeNumber(90);
         jsonGenerator.writeNumber(85);
         // ]

         jsonGenerator.writeEndArray();

         jsonGenerator.writeEndObject();
         jsonGenerator.close();

         //result student.json
         //{
         //   "name":"Mahesh Kumar",
         //   "age":21,
         //   "verified":false,
         //   "marks":[100,90,85]
         //}

         ObjectMapper mapper = new ObjectMapper();
         Map<String,Object> dataMap = mapper.readValue(new File("student.json"), Map.class);

         System.out.println(dataMap.get("name"));
         System.out.println(dataMap.get("age"));
         System.out.println(dataMap.get("verified"));
         System.out.println(dataMap.get("marks"));
      }
      catch (JsonParseException e) { e.printStackTrace(); }
      catch (JsonMappingException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace(); }
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Mahesh Kumar

21
```

```
false

[100, 90, 85]
```

Reading JSON using JsonParser

Using JsonParser is again pretty simple. First create the JsonParser using JsonFactory.createJsonParser() method and use it's nextToken() methods to read each json string as token. Check each token and process accordingly

```
JsonFactory jasonFactory = new JsonFactory();
JsonParser jsonParser = jasonFactory.createJsonParser(new File("student.json"));
while (jsonParser.nextToken() != JsonToken.END_OBJECT) {
   //get the current token
   String fieldname = jsonParser.getCurrentName();
   if ("name".equals(fieldname)) {
      //move to next token
      jsonParser.nextToken();
      System.out.println(jsonParser.getText());
   }
}
```

Let's see JsonParser in action. Create a java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

```
import java.io.File;
import java.io.IOException;
import java.util.Map;

import com.fasterxml.jackson.core.JsonEncoding;
import com.fasterxml.jackson.core.JsonFactory;
import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.core.JsonParseException;
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         JsonFactory jsonFactory = new JsonFactory();

         JsonGenerator jsonGenerator = jsonFactory.createGenerator(new File(
            "student.json"), JsonEncoding.UTF8);
         jsonGenerator.writeStartObject();
         jsonGenerator.writeStringField("name", "Mahesh Kumar");
         jsonGenerator.writeNumberField("age", 21);
         jsonGenerator.writeBooleanField("verified", false);
         jsonGenerator.writeFieldName("marks");
         jsonGenerator.writeStartArray(); // [
         jsonGenerator.writeNumber(100);
         jsonGenerator.writeNumber(90);
         jsonGenerator.writeNumber(85);
         jsonGenerator.writeEndArray();
         jsonGenerator.writeEndObject();
         jsonGenerator.close();

         //result student.json
         //{
         //   "name":"Mahesh Kumar",
         //   "age":21,
         //   "verified":false,
         //   "marks":[100,90,85]
         //}

         JsonParser jsonParser = jsonFactory.createParser(new File("student.json"));
         while (jsonParser.nextToken() != JsonToken.END_OBJECT) {
            //get the current token
            String fieldname = jsonParser.getCurrentName();
            if ("name".equals(fieldname)) {
               //move to next token
               jsonParser.nextToken();
               System.out.println(jsonParser.getText());
            }
            if("age".equals(fieldname)){
               //move to next token
               jsonParser.nextToken();
               System.out.println(jsonParser.getNumberValue());
            }
            if("verified".equals(fieldname)){
               //move to next token
               jsonParser.nextToken();
               System.out.println(jsonParser.getBooleanValue());
            }
            if("marks".equals(fieldname)){
               //move to [
               jsonParser.nextToken();
               // loop till token equal to "]"
               while (jsonParser.nextToken() != JsonToken.END_ARRAY) {
                  System.out.println(jsonParser.getNumberValue());
               }
            }
         }
      } catch (JsonParseException e) {
         e.printStackTrace();
      } catch (JsonMappingException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Verify the result

Compile the classes using **javac** compiler as follows:

```
C:\Jackson_WORKSPACE>javac JacksonTester.java
```

Now run the jacksonTester to see the result:

```
C:\Jackson_WORKSPACE>java JacksonTester
```

Verify the Output

```
Mahesh Kumar
21
false
[100, 90, 85]
```

JsonGenerator Class

Introduction

JsonParser is the base class to define public API for reading Json content. Instances are created using factory methods of a JsonFactory instance.

Class declaration

Following is the declaration for **org.codehaus.jackson.JsonParser** class:

```
public abstract class JsonParser
   extends Object
      implements Closeable, Versioned
```

Nested Classes

S.N.	Class & Description
1	**static class JsonParser.Feature** Enumeration that defines all togglable features for parsers.
2	**static class JsonParser.NumberType** Enumeration of possible "native" (optimal) types that can be used for numbers.

Fields

protected PrettyPrinter _cfgPrettyPrinter - Object that handles pretty-printing (usually additional white space to make results more human-readable) during output.

protected JsonToken _currToken - Last token retrieved via nextToken(), if any.

protected int _features - Bit flag composed of bits that indicate which JsonParser.Features are enabled.

protected JsonToken _lastClearedToken - Last cleared token, if any: that is, value that was in effect when clearCurrentToken() was called.

Constructors

S.N.	Constructor & Description
1	**protected JsonParser()** Default constructor
2	**protected JsonParser(int features)**

Class Methods

S.N.	Method & Description
1	**protected void _reportError(String msg)** - Helper method used for constructing and throwing JsonGenerationException with given base message.
2	**protected void _reportUnsupportedOperation()**
3	**protected void _throwInternal()**
4	**protected void _writeSimpleObject(Object value)**- Helper method to try to call appropriate write method for given untyped Object.
5	**boolean canOmitFields()** - Introspection method to call to check whether it is ok to omit writing of Object fields or not.
6	**boolean canUseSchema(FormatSchema schema)** - Method that can be used to verify that given schema can be used with this generator (using setSchema(com.fasterxml.jackson.core.FormatSchema)).
7	**boolean canWriteBinaryNatively()** - Introspection method that may be called to see if the underlying data format supports "native" binary data; that is, an efficient output of binary content without encoding.
8	**boolean canWriteObjectId()** - Introspection method that may be called to see if the underlying data format supports some kind of Object Ids natively (many do not; for example, JSON doesn't).
9	**boolean canWriteTypeId()** - Introspection method that may be called to see if the underlying data format supports some kind of Type Ids natively (many do not; for example, JSON doesn't).
10	**abstract void close()** - Method called to close this generator, so that no more content can be written.
11	**JsonGenerator configure(JsonGenerator.Feature f, boolean state)** - Method for enabling or disabling specified feature: check JsonGenerator.Feature for list of available features.
12	**void copyCurrentEvent(JsonParser jp)** - Method for copying contents of the current event that the given parser instance points to.
13	**void copyCurrentStructure(JsonParser jp)** - Method for copying contents of the

	current event and following events that it encloses the given parser instance points to.
14	**abstract JsonGenerator disable(JsonGenerator.Feature f) - Method for disabling specified features (check JsonGenerator.Feature for list of features)**
15	**abstract JsonGenerator enable(JsonGenerator.Feature f)** - Method for enabling specified parser features: check JsonGenerator.Feature for list of available features.
16	**abstract void flush()** - Method called to flush any buffered content to the underlying target (output stream, writer), and to flush the target itself as well.
17	**CharacterEscapes getCharacterEscapes()** - Method for accessing custom escapes factory uses for JsonGenerators it creates.
18	**abstract ObjectCodec getCodec()** - Method for accessing the object used for writing Java object as Json content (using method writeObject(java.lang.Object)).
19	**abstract int getFeatureMask()** - Bulk access method for getting state of all standard JsonGenerator.Features.
20	**int getHighestEscapedChar()** - Accessor method for testing what is the highest unescaped character configured for this generator.
21	**abstract JsonStreamContext getOutputContext()**
22	**Object getOutputTarget()**- Method that can be used to get access to object that is used as target for generated output; this is usually either OutputStream or Writer, depending on what generator was constructed with.
23	**PrettyPrinter getPrettyPrinter()**- Accessor for checking whether this generator has a configured PrettyPrinter; returns it if so, null if none configured.
24	**FormatSchema getSchema()** - Method for accessing Schema that this parser uses, if any.
25	**abstract boolean isClosed()** - Method that can be called to determine whether this generator is closed or not.
26	**abstract boolean isEnabled(JsonGenerator.Feature f)** - Method for checking whether given feature is enabled.
27	**JsonGenerator setCharacterEscapes(CharacterEscapes esc)**-Method for defining custom escapes factory uses for JsonGenerators it creates.
28	**abstract JsonGenerator setCodec(ObjectCodec oc)**- Method that can be called to set or reset the object to use for writing Java objects as JsonContent (using method writeObject(java.lang.Object)).
29	**abstract JsonGenerator setFeatureMask(int mask)**-Bulk set method for (re)settting states of all standard JsonGenerator.Features
30	**JsonGenerator setHighestNonEscapedChar(int charCode) - Method that can be called to request that generator escapes all character codes above specified code**

	point (if positive value); or, to not escape any characters except for ones that must be escaped for the data format (if -1).
31	**JsonGenerator setPrettyPrinter(PrettyPrinter pp)** - Method for setting a custom pretty printer, which is usually used to add indentation for improved human readability.
32	**JsonGenerator setRootValueSeparator(SerializableString sep)** - Method that allows overriding String used for separating root-level JSON values (default is single space character)
33	**void setSchema(FormatSchema schema)** - Method to call to make this generator use specified schema.
33	**abstract JsonGenerator useDefaultPrettyPrinter()**- Convenience method for enabling pretty-printing using the default pretty printer (DefaultPrettyPrinter).
34	**abstract Version version()**- Accessor for finding out version of the bundle that provided this generator instance.
35	**void writeArrayFieldStart(String fieldName)**- Convenience method for outputting a field entry ("member") (that will contain a JSON Array value), and the START_ARRAY marker.
36	**abstract void writeBinary(Base64Variant b64variant, byte[] data, int offset, int len)**- Method that will output given chunk of binary data as base64 encoded, as a complete String value (surrounded by double quotes).
37	**abstract int writeBinary(Base64Variant b64variant, InputStream data, int dataLength)** - Method similar to writeBinary(Base64Variant,byte[],int,int), but where input is provided through a stream, allowing for incremental writes without holding the whole input in memory.
38	**void writeBinary(byte[] data)**- Similar to writeBinary(Base64Variant,byte[],int,int), but assumes default to using the Jackson default Base64 variant (which is Base64Variants.MIME_NO_LINEFEEDS).
39	**void writeBinary(byte[] data, int offset, int len)** - Similar to writeBinary(Base64Variant,byte[],int,int), but default to using the Jackson default Base64 variant (which is Base64Variants.MIME_NO_LINEFEEDS).
40	**int writeBinary(InputStream data, int dataLength)** - Similar to writeBinary(Base64Variant,InputStream,int), but assumes default to using the Jackson default Base64 variant (which is Base64Variants.MIME_NO_LINEFEEDS).
41	**void writeBinaryField(String fieldName, byte[] data)** - Convenience method for outputting a field entry ("member") that contains specified data in base64-encoded form.
42	**abstract void writeBoolean(boolean state)** - Method for outputting literal Json boolean value (one of Strings 'true' and 'false').
43	**void writeBooleanField(String fieldName, boolean value)** - Convenience method for

	outputting a field entry ("member") that has a boolean value.
44	**abstract void writeEndArray()** - Method for writing closing marker of a JSON Array value (character ']'; plus possible white space decoration if pretty-printing is enabled).
45	**abstract void writeEndObject()** - Method for writing closing marker of a JSON Object value (character '}'; plus possible white space decoration if pretty-printing is enabled).
46	**abstract void writeFieldName(SerializableString name)** - Method similar to writeFieldName(String), main difference being that it may perform better as some of processing (such as quoting of certain characters, or encoding into external encoding if supported by generator) can be done just once and reused for later calls.
47	**abstract void writeFieldName(String name)** - Method for writing a field name (JSON String surrounded by double quotes: syntactically identical to a JSON String value), possibly decorated by white space if pretty-printing is enabled.
48	**abstract void writeNull()** - Method for outputting literal Json null value.
49	**void writeNullField(String fieldName)** - Convenience method for outputting a field entry ("member") that has JSON literal value null.
50	**abstract void writeNumber(BigDecimal dec)** - Method for outputting indicate Json numeric value.
51	**abstract void writeNumber(BigInteger v)** - Method for outputting given value as Json number.
52	**abstract void writeNumber(double d)** - Method for outputting indicate Json numeric value.
53	**abstract void writeNumber(float f)** - Method for outputting indicate Json numeric value.
54	**abstract void writeNumber(int v)** - Method for outputting given value as Json number.
55	**abstract void writeNumber(long v)** - Method for outputting given value as Json number.
56	**void writeNumber(short v)** - Method for outputting given value as Json number.
57	**abstract void writeNumber(String encodedValue)** - Write method that can be used for custom numeric types that can not be (easily?) converted to "standard" Java number types.
58	**void writeNumberField(String fieldName, BigDecimal value)** - Convenience method for outputting a field entry ("member") that has the specified numeric value.
59	**void writeNumberField(String fieldName, double value)** - Convenience method for outputting a field entry ("member") that has the specified numeric value.
60	**void writeNumberField(String fieldName, float value)** - Convenience method for outputting a field entry ("member") that has the specified numeric value.

61	**void writeNumberField(String fieldName, int value)** - Convenience method for outputting a field entry ("member") that has the specified numeric value.
62	**void writeNumberField(String fieldName, long value)** - Convenience method for outputting a field entry ("member") that has the specified numeric value.
63	**abstract void writeObject(Object pojo)** - Method for writing given Java object (POJO) as Json.
64	**void writeObjectField(String fieldName, Object pojo)** - Convenience method for outputting a field entry ("member") that has contents of specific Java object as its value.
65	**void writeObjectFieldStart(String fieldName)** - Convenience method for outputting a field entry ("member") (that will contain a JSON Object value), and the START_OBJECT marker.
66	**void writeObjectId(Object id)** - Method that can be called to output so-called native Object Id.
67	**void writeObjectRef(Object id)** - Method that can be called to output references to native Object Ids.
68	**void writeOmittedField(String fieldName) Method called to indicate that a property in this position was skipped.**
69	**abstract void writeRaw(char c)** - Method that will force generator to copy input text verbatim with no modifications (including that no escaping is done and no separators are added even if context [array, object] would otherwise require such).
70	**abstract void writeRaw(char[] text, int offset, int len)** - Method that will force generator to copy input text verbatim with no modifications (including that no escaping is done and no separators are added even if context [array, object] would otherwise require such).
71	**void writeRaw(SerializableString raw)** - Method that will force generator to copy input text verbatim with no modifications (including that no escaping is done and no separators are added even if context [array, object] would otherwise require such).
72	**abstract void writeRaw(String text)** - Method that will force generator to copy input text verbatim with no modifications (including that no escaping is done and no separators are added even if context [array, object] would otherwise require such).
73	**abstract void writeRaw(String text, int offset, int len)** - Method that will force generator to copy input text verbatim with no modifications (including that no escaping is done and no separators are added even if context [array, object] would otherwise require such).
74	**abstract void writeRawUTF8String(byte[] text, int offset, int length)** - Method similar to writeString(String) but that takes as its input a UTF-8 encoded String that is to

	be output as-is, without additional escaping (type of which depends on data format; backslashes for JSON).
75	**abstract void writeRawValue(char[] text, int offset, int len)**
76	**abstract void writeRawValue(String text)**- Method that will force generator to copy input text verbatim without any modifications, but assuming it must constitute a single legal JSON value (number, string, boolean, null, Array or List).
77	**abstract void writeRawValue(String text, int offset, int len)**
78	**abstract void writeStartArray()**- Method for writing starting marker of a JSON Array value (character '['; plus possible white space decoration if pretty-printing is enabled).
79	**abstract void writeStartObject()** - Method for writing starting marker of a JSON Object value (character '{'; plus possible white space decoration if pretty-printing is enabled).
80	**abstract void writeString(char[] text, int offset, int len)** - Method for outputting a String value.
81	**abstract void writeString(SerializableString text)** - Method similar to writeString(String), but that takes SerializableString which can make this potentially more efficient to call as generator may be able to reuse quoted and/or encoded representation.
82	**abstract void writeString(String text)** - Method for outputting a String value.
83	**void writeStringField(String fieldName, String value)** - Convenience method for outputting a field entry ("member") that has a String value.
84	**abstract void writeTree(TreeNode rootNode)** - Method for writing given JSON tree (expressed as a tree where given JsonNode is the root) using this generator.
85	**void writeTypeId(Object id)** - Method that can be called to output so-called native Type Id.
86	**abstract void writeUTF8String(byte[] text, int offset, int length)** - Method similar to writeString(String) but that takes as its input a UTF-8 encoded String which has not been escaped using whatever escaping scheme data format requires (for JSON that is backslash-escaping for control characters and double-quotes; for other formats something else).

Methods inherited

This class inherits methods from the following classes:

java.lang.Object

JsonParser Class

Introduction

JsonParser is the base class to define public API for reading Json content. Instances are created using factory methods of a JsonFactory instance.

Class declaration

Following is the declaration for **com.fasterxml.jackson.core.JsonParser** class:

```
public abstract class JsonParser
   extends Object
      implements Closeable, Versioned
```

Nested Classes

S.N.	Class & Description
1	**static class JsonParser.Feature** Enumeration that defines all togglable features for parsers.
2	**static class JsonParser.NumberType** Enumeration of possible "native" (optimal) types that can be used for numbers.

Fields

protected int _features - Bit flag composed of bits that indicate which JsonParser.Features are enabled.

Constructors

S.N.	Constructor & Description
1	protected JsonParser() Default constructor
2	protected JsonParser(int features)

Class Methods

S.N.	Method & Description
1	**protected JsonParseException _constructError(String msg)** - Helper method for constructing JsonParseExceptions based on current state of the parser.
2	**protected void _reportUnsupportedOperation()** - Helper method to call for operations that are not supported by parser implementation.
3	**boolean canReadObjectId()** - Introspection method that may be called to see if the underlying data format supports some kind of Object Ids natively (many do not; for

	example, JSON doesn't).
4	**boolean canReadTypeId()** - Introspection method that may be called to see if the underlying data format supports some kind of Type Ids natively (many do not; for example, JSON doesn't).
5	**boolean canUseSchema(FormatSchema schema)** - Method that can be used to verify that given schema can be used with this parser (using setSchema(com.fasterxml.jackson.core.FormatSchema)).
6	**abstract void clearCurrentToken()** - Method called to "consume" the current token by effectively removing it so that hasCurrentToken() returns false, and getCurrentToken() null).
7	**abstract void close()** - Closes the parser so that no further iteration or data access can be made; will also close the underlying input source if parser either owns the input source, or feature JsonParser.Feature.AUTO_CLOSE_SOURCE is enabled.
8	**JsonParser configure(JsonParser.Feature f, boolean state)** - Method for enabling or disabling specified feature (check JsonParser.Feature for list of features)
9	**JsonParser disable(JsonParser.Feature f)** - Method for disabling specified feature (check JsonParser.Feature for list of features)
10	**JsonParser enable(JsonParser.Feature f)** - Method for enabling specified parser feature (check JsonParser.Feature for list of features)
11	**abstract BigInteger getBigIntegerValue()** - Numeric accessor that can be called when the current token is of type JsonToken.VALUE_NUMBER_INT and it can not be used as a Java long primitive type due to its magnitude.
12	**byte[] getBinaryValue()** - Convenience alternative to getBinaryValue(Base64Variant) that defaults to using Base64Variants.getDefaultVariant() as the default encoding.
13	**abstract byte[] getBinaryValue(Base64Variant b64variant)** - Method that can be used to read (and consume -- results may not be accessible using other methods after the call) base64-encoded binary data included in the current textual JSON value.
14	**boolean getBooleanValue()** - Convenience accessor that can be called when the current token is JsonToken.VALUE_TRUE or JsonToken.VALUE_FALSE.
15	**byte getByteValue()** - Numeric accessor that can be called when the current token is of type JsonToken.VALUE_NUMBER_INT and it can be expressed as a value of Java byte primitive type.
16	**abstract ObjectCodec getCodec()** - Accessor for ObjectCodec associated with this parser, if any.
17	**abstract JsonLocation getCurrentLocation()** - Method that returns location of the last processed character; usually for error reporting purposes.
18	**abstract String getCurrentName()** - Method that can be called to get the name

	associated with the current token: for JsonToken.FIELD_NAMEs it will be the same as what getText() returns; for field values it will be preceding field name; and for others (array values, root-level values) null.
19	**abstract JsonToken getCurrentToken()** - Accessor to find which token parser currently points to, if any; null will be returned if none.
20	**abstract int getCurrentTokenId()** - Method similar to getCurrentToken() but that returns an int instead of JsonToken (enum value).
21	**abstract BigDecimal getDecimalValue()** - Numeric accessor that can be called when the current token is of type JsonToken.VALUE_NUMBER_FLOAT or JsonToken.VALUE_NUMBER_INT.
22	**abstract double getDoubleValue()** - Numeric accessor that can be called when the current token is of type JsonToken.VALUE_NUMBER_FLOAT and it can be expressed as a Java double primitive type.
23	**abstract Object getEmbeddedObject()** - Accessor that can be called if (and only if) the current token is JsonToken.VALUE_EMBEDDED_OBJECT.
24	**int getFeatureMask()** - Bulk access method for getting state of all standard JsonParser.Features.
25	**abstract float getFloatValue()** - Numeric accessor that can be called when the current token is of type JsonToken.VALUE_NUMBER_FLOAT and it can be expressed as a Java float primitive type.
26	**Object getInputSource()** - Method that can be used to get access to object that is used to access input being parsed; this is usually either InputStream or Reader, depending on what parser was constructed with.
27	**abstract int getIntValue()** - Numeric accessor that can be called when the current token is of type JsonToken.VALUE_NUMBER_INT and it can be expressed as a value of Java int primitive type.
28	**abstract JsonToken getLastClearedToken()** - Method that can be called to get the last token that was cleared using clearCurrentToken().
29	**abstract long getLongValue()** - Numeric accessor that can be called when the current token is of type JsonToken.VALUE_NUMBER_INT and it can be expressed as a Java long primitive type.
30	**abstract JsonParser.NumberType getNumberType()** - If current token is of type JsonToken.VALUE_NUMBER_INT or JsonToken.VALUE_NUMBER_FLOAT, returns one of JsonParser.NumberType constants; otherwise returns null.
31	**abstract Number getNumberValue()** - Generic number value accessor method that will work for all kinds of numeric values.
32	**Object getObjectId()** - Method that can be called to check whether current token (one

	that was just read) has an associated Object id, and if so, return it.
33	**abstract JsonStreamContext getParsingContext()** - Method that can be used to access current parsing context reader is in.
34	**FormatSchema getSchema()** - Method for accessing Schema that this parser uses, if any.
35	**short getShortValue()** - Numeric accessor that can be called when the current token is of type JsonToken.VALUE_NUMBER_INT and it can be expressed as a value of Java short primitive type.
36	**abstract String getText()** - Method for accessing textual representation of the current token; if no current token (before first call to nextToken(), or after encountering end-of-input), returns null.
37	**abstract char[] getTextCharacters()** - Method similar to getText(), but that will return underlying (unmodifiable) character array that contains textual value, instead of constructing a String object to contain this information.
38	**abstract int getTextLength()** - Accessor used with getTextCharacters(), to know length of String stored in returned buffer.
39	**abstract int getTextOffset()** - Accessor used with getTextCharacters(), to know offset of the first text content character within buffer.
40	**abstract JsonLocation getTokenLocation()** - Method that return the starting location of the current token; that is, position of the first character from input that starts the current token.
41	**Object getTypeId()** - Method that can be called to check whether current token (one that was just read) has an associated type id, and if so, return it.
42	**boolean getValueAsBoolean()** - Method that will try to convert value of current token to a boolean.
43	**boolean getValueAsBoolean(boolean defaultValue)** - Method that will try to convert value of current token to a boolean.
44	**double getValueAsDouble()** - Method that will try to convert value of current token to a Java double.
45	**double getValueAsDouble(double defaultValue)** - Method that will try to convert value of current token to a Java double.
46	**int getValueAsInt()** - Method that will try to convert value of current token to a int.
47	**int getValueAsInt(int defaultValue)** - Method that will try to convert value of current token to a int.
48	**long getValueAsLong()** - Method that will try to convert value of current token to a long.

49	**long getValueAsLong(long defaultValue)** - Method that will try to convert value of current token to a long.
50	**String getValueAsString()** - Method that will try to convert value of current token to a String.
51	**abstract String getValueAsString(String defaultValue)** - Method that will try to convert value of current token to a String.
52	**abstract boolean hasCurrentToken()** - Method for checking whether parser currently points to a token (and data for that token is available).
53	**abstract boolean hasTextCharacters()** - Method that can be used to determine whether calling of getTextCharacters() would be the most efficient way to access textual content for the event parser currently points to.
54	**abstract boolean isClosed()** - Method that can be called to determine whether this parser is closed or not.
55	**boolean isEnabled(JsonParser.Feature f)** - Method for checking whether specified JsonParser.Feature is enabled.
56	**boolean isExpectedStartArrayToken()** - Specialized accessor that can be used to verify that the current token indicates start array (usually meaning that current token is JsonToken.START_ARRAY) when start array is expected.
57	**Boolean nextBooleanValue()** - Method that fetches next token (as if calling nextToken()) and if it is JsonToken.VALUE_TRUE or JsonToken.VALUE_FALSE returns matching Boolean value; otherwise return null.
58	**boolean nextFieldName(SerializableString str)** - Method that fetches next token (as if calling nextToken()) and verifies whether it is JsonToken.FIELD_NAME with specified name and returns result of that comparison.
59	**int nextIntValue(int defaultValue)** - Method that fetches next token (as if calling nextToken()) and if it is JsonToken.VALUE_NUMBER_INT returns 32-bit int value; otherwise returns specified default value It is functionally equivalent to:
60	**long nextLongValue(long defaultValue)** - Method that fetches next token (as if calling nextToken()) and if it is JsonToken.VALUE_NUMBER_INT returns 64-bit long value; otherwise returns specified default value It is functionally equivalent to:
61	**String nextTextValue()** - Method that fetches next token (as if calling nextToken()) and if it is JsonToken.VALUE_STRING returns contained String value; otherwise returns null.
62	**abstract JsonToken nextToken()** - Main iteration method, which will advance stream enough to determine type of the next token, if any.
63	**abstract JsonToken nextValue()** - Iteration method that will advance stream enough to determine type of the next token that is a value type (including JSON Array and Object start/end markers).

64	**abstract void overrideCurrentName(String name)** - Method that can be used to change what is considered to be the current (field) name.
65	**int readBinaryValue(Base64Variant b64variant, OutputStream out)** - Similar to readBinaryValue(OutputStream) but allows explicitly specifying base64 variant to use.
66	**int readBinaryValue(OutputStream out)** - Method that can be used as an alternative to getBigIntegerValue(), especially when value can be large.
67	**<T> T readValueAs(Class<T> valueType)** - Method to deserialize JSON content into a non-container type (it can be an array type, however): typically a bean, array or a wrapper type (like Boolean).
68	**<T> T readValueAs(TypeReference<?> valueTypeRef)**- Method to deserialize JSON content into a Java type, reference to which is passed as argument.
69	**<T extends TreeNode> T readValueAsTree()** - Method to deserialize JSON content into equivalent "tree model", represented by root TreeNode of resulting model.
70	**<T> Iterator<T> readValuesAs(Class<T> valueType)** - Method for reading sequence of Objects from parser stream, all with same specified value type.
71	**<T> Iterator<T> readValuesAs(TypeReference<?> valueTypeRef)**- Method for reading sequence of Objects from parser stream, all with same specified value type.
72	**int releaseBuffered(OutputStream out)** - Method that can be called to push back any content that has been read but not consumed by the parser.
73	**int releaseBuffered(Writer w) - Method that can be called to push back any content that has been read but not consumed by the parser.**
74	**boolean requiresCustomCodec()** - Method that can be called to determine if a custom ObjectCodec is needed for binding data parsed using JsonParser constructed by this factory (which typically also implies the same for serialization with JsonGenerator).
75	**abstract void setCodec(ObjectCodec c)** - Setter that allows defining ObjectCodec associated with this parser, if any.
76	**JsonParser setFeatureMask(int mask)** - Bulk set method for (re)settting states of all standard JsonParser.Features
77	**void setSchema(FormatSchema schema)** - Method to call to make this parser use specified schema.
78	**abstract JsonParser skipChildren() - Method that will skip all child tokens of an array or object token that the parser currently points to, iff stream points to JsonToken.START_OBJECT or JsonToken.START_ARRAY.**
79	**abstract Version version()** - Accessor for getting version of the core package, given a parser instance.

Methods inherited

This class inherits methods from the following classes:

java.lang.Object

Previous Page Next Page