JACKSON

data-processing tools for java

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Jackson is a very popular and efficient Java-based library to serialize or map Java objects to JSON and vice versa.

This tutorial uses a simple and intuitive way to explain the basic features of Jackson library API and how to use them in practice.

Audience

This tutorial will be useful for most Java developers, regardless of whether they are beginners or experts.

Prerequisites

Jackson is a Java-based library and it is imperative that you should have a thorough knowledge of Java programming language before proceeding with this tutorial.

Copyright & Disclaimer

Table of Contents


tutorialspoint
SIMPLYEASYLEARNING

1. OVERVIEW

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

Features of Jackson

- **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 quite fast, consumes less memory space, and is suitable for large object graphs or systems.

- **Clean JSON** – Jackson creates clean and compact JSON results which are easy to read.

- **No Dependency** – Jackson library does not require any other library apart from JDK.

- **Open Source** – Jackson library is open source and free to use.

Process JSON using Jackson

Jackson provides three different ways to process JSON:

- **Streaming API** – It reads and writes JSON content as discrete events. JsonParser reads the data, whereas JsonGenerator writes the data.
    - It is the most powerful approach among the three.
    - It has the lowest overhead and it provides the fastest way to perform read/write operations.
    - It is analogous to **Stax parser** for XML.

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

- **Data Binding** – It converts JSON to and from Plain Old Java Object (POJO) using property accessor or using annotations. ObjectMapper reads/writes JSON for both types of data bindings. Data binding is analogous to **JAXB parser** for XML. Data binding is of two types:
    - **Simple Data Binding** – It converts JSON to and from Java Maps, Lists, Strings, Numbers, Booleans, and null objects.
    - **Full Data Binding** – It converts JSON to and from any Java type.

2. ENVIRONMENT SETUP

This chapter describes how to set up the Jackson environment on your system.

Try-It Online Option

You really do not need to set up your own environment to start learning Jackson. We have set up an online Java Programming environment online, so that you can compile and execute all the available examples online. Feel free to modify any example and check the result with different options.

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

```
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 a **Try it** option to help you learn quickly through practice.

Local Environment Setup

If you still wish 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. Follow the given steps to set up the environment.

Java SE is freely available from the link Download Java. You can 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 have installed Java, you would need to set the environment variables to point to the 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'.
- Select 'Properties'.

- Click on the 'Environment variables' button under the 'Advanced' tab.
- Alter the 'Path' variable so that it also contains the path to the Java executable. For 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
- Add the following line at the end:

  'SET PATH=%PATH%;C:\Program Files\java\jdk\bin'

Setting up the Path for Linux, UNIX, Solaris, and 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.

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

Popular Java Editors

To write Java programs, you need a text editor. There are sophisticated IDEs available in the market to write Java programs, but for now, you can consider one of the following:

- **Notepad:** On Windows platform, you can use any simple text editor such as Notepad (recommended for this tutorial) or TextPad.
- **Netbeans:** It is an open-source Java IDE. It can be downloaded from http://www.netbeans.org/index.html.
- **Eclipse:** It is also a Java IDE developed by the Eclipse open-source community. It can be downloaded from http://www.eclipse.org/.

Download Jackson Archive

Download the latest version of Jackson jar file from jackson-all-1.9.0.jar.zip. In this tutorial, *jackson-1.9.0.jar* is downloaded and copied into C:\>jackson folder.

OS	Archive name
Windows	jackson-all-1.9.0.jar
Linux	jackson-all-1.9.0.jar
Mac	jackson-all-1.9.0.jar

Set up Jackson Environment

Set the **jackson_HOME** environment variable to point to the base directory location where Jackson jar is stored on your machine. Depending on the platform you are working on, the process varies as shown in the following table:

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.

OS	Output
Windows	Set the environment variable CLASSPATH to %CLASSPATH%;%jackson_HOME%\jackson-all-1.9.0.jar;.;
Linux	export CLASSPATH=$CLASSPATH:$jackson_HOME/jackson-all-1.9.0.jar:.
Mac	export CLASSPATH=$CLASSPATH:$jackson_HOME/jackson-all-1.9.0.jar:.

3. FIRST APPLICATION

Before going into the details of the Jackson library, let us see an application in action.

Jackson – Example

In the following example, we will create a Student class. Thereafter, we will create a JSON string with Student details and deserialize it to Student object and then serialize it back to a JSON string.

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

File: JacksonTester.java

```
import java.io.IOException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.map.SerializationConfig;

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);

         mapper.enable(SerializationConfig.Feature.INDENT_OUTPUT);
         jsonString = mapper.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
```

Execute 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 the JSON string or the source of the JSON string and the object type as parameters.

```
//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.writeValueAsString(student);
```

4. OBJECTMAPPER CLASS

ObjectMapper is the main actor class of Jackson library. ObjectMapper class provides functionalities to convert Java objects to matching JSON constructs and vice versa. It uses instances of JsonParser and JsonGenerator for implementing actual reading/writing of JSON.

ObjectMapper – Class Declaration

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

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

ObjectMapper – Nested Classes

Sr. No.	Class and 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.

ObjectMapper Class – Constructors

Sr. No.	Constructor and Description
1	**ObjectMapper()** It is the default constructor, which constructs the default JsonFactory as necessary. Use StdSerializerProvider as its SerializerProvider, and BeanSerializerFactory as its SerializerFactory.

2	**ObjectMapper(JsonFactory jf)** Construct mapper that uses specified JsonFactory for constructing necessary JsonParsers and/or JsonGenerators.
3	**ObjectMapper(JsonFactory jf, SerializerProvider sp, DeserializerProvider dp)**
4	**ObjectMapper(JsonFactory jf, SerializerProvider sp, DeserializerProvider dp, SerializationConfig sconfig, DeserializationConfig dconfig)**
5	**ObjectMapper(SerializerFactory sf)** Deprecated. Use other constructors instead; note that you can set the serializer factory with **setSerializerFactory(org.codehaus.jackson.map.SerializerFactory)**

ObjectMapper – Methods

Sr. No.	Method and Description
1	**protected void_configAndWriteValue (JsonGenerator jgen, Object value)** Method called to configure the generator as necessary and then call write functionality
2	**protected void_configAndWriteValue (JsonGenerator jgen, Object value, Class<?> viewClass)**
3	**protected Object _convert (Object fromValue, JavaType toValueType)**
4	**protected DeserializationContext_createDeserializationContext (JsonParser jp, DeserializationConfig cfg)**
5	**protected PrettyPrinter_defaultPrettyPrinter()** Helper method that should return default pretty-printer to use for generators constructed by this mapper, when instructed to use default pretty printer.

6	**protected JsonDeserializer<Object> _findRootDeserializer (DeserializationConfig cfg, JavaType valueType)** Method called to locate deserializer for the passed root-level value.
7	**protected JsonToken _initForReading (JsonParser jp)** Method called to ensure that given parser is ready for reading content for data binding.
8	**protected Object _readMapAndClose(JsonParser jp, JavaType valueType)**
9	**protected Object _readValue(DeserializationConfig cfg, JsonParser jp, JavaType valueType)** Actual implementation of value reading+binding operation.
10	**protected Object _unwrapAndDeserialize(JsonParser jp, JavaType rootType, DeserializationContext ctxt, JsonDeserializer<Object> deser)**
11	**boolean canDeserialize(JavaType type)** Method that can be called to check whether mapper thinks it could deserialize an Object of given type.
12	**boolean canSerialize(Class<?> type)** Method that can be called to check whether mapper thinks it could serialize an instance of given Class.
13	**ObjectMapper configure(DeserializationConfig.Feature f, boolean state)** Method for changing state of an on/off deserialization feature for this object mapper.
14	**ObjectMapper configure(JsonGenerator.Feature f, boolean state)**

	Method for changing state of an on/off JsonGenerator feature for JsonFactory instance this object mapper uses.
15	**ObjectMapper configure(JsonParser.Feature f, boolean state)** Method for changing state of an on/off JsonParser feature for JsonFactory instance this object mapper uses.
16	**ObjectMapper configure(SerializationConfig.Feature f, boolean state)** Method for changing state of an on/off serialization feature for this object mapper.
17	**JavaType constructType(Type t)** Convenience method for constructing JavaType out of given type (typically java.lang.Class), but without explicit context.
18	**<T> T convertValue(Object fromValue, Class<T> toValueType)** Convenience method for doing two-step conversion from given value, into instance of given value type.
19	**<T> T convertValue(Object fromValue, JavaType toValueType)**
20	**<T> T convertValue(Object fromValue, TypeReference toValueTypeRef)**
21	**DeserializationConfig copyDeserializationConfig()** Method that creates a copy of the shared default DeserializationConfig object that defines configuration settings for deserialization.
22	**SerializationConfig copySerializationConfig()** Method that creates a copy of the shared default SerializationConfig object that defines configuration settings for serialization.
23	**ArrayNode createArrayNode()**

	Note: return type is co-variant, as basic ObjectCodec abstraction cannot refer to concrete node types (as it is a part of core package, whereas impls are part of mapper package).
24	**ObjectNode createObjectNode()** Note: return type is co-variant, as basic ObjectCodec abstraction cannot refer to concrete node types (as it is part of core package, whereas impls are part of mapper package).
25	**ObjectWriter defaultPrettyPrintingWriter()** Deprecated. Since 1.9, use writerWithDefaultPrettyPrinter() instead.
26	**ObjectMapper disable(DeserializationConfig.Feature... f)** Method for enabling specified DeserializationConfig features.
27	**ObjectMapper disable(SerializationConfig.Feature... f)** Method for enabling specified DeserializationConfig features.
28	**ObjectMapper disableDefaultTyping()** Method for disabling automatic inclusion of type information; if so, only explicitly annotated types (ones with JsonTypeInfo) has additional embedded type information.
29	**ObjectMapper enable(DeserializationConfig.Feature... f)** Method for enabling specified DeserializationConfig features.
30	**ObjectMapper enable(SerializationConfig.Feature... f)** Method for enabling specified DeserializationConfig features.
31	**ObjectMapper enableDefaultTyping()** Convenience method that is equivalent to calling.

32	**ObjectMapper enableDefaultTyping(ObjectMapper.DefaultTyping dti)** Convenience method that is equivalent to calling.
33	**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 are annotated with JsonTypeInfo).
34	**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)
35	**ObjectWriter filteredWriter(FilterProvider filterProvider)** Deprecated. Since 1.9, use writer(FilterProvider) instead.
36	**JsonSchema generateJsonSchema(Class<?> t)** Generate Json-schema instance for specified class.
37	**JsonSchema generateJsonSchema(Class<?> t, SerializationConfig cfg)** Generate Json-schema instance for specified class, using specific serialization configuration
38	**DeserializationConfig getDeserializationConfig()** Method that returns the shared default DeserializationConfig object that defines configuration settings for deserialization.

39	**DeserializerProvider getDeserializerProvider()**
40	**JsonFactory getJsonFactory()** Method that can be used to get hold of JsonFactory that this mapper uses if it needs to construct JsonParsers and/or JsonGenerators.
41	**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.
42	**SerializationConfig getSerializationConfig()** Method that returns the shared default SerializationConfig object that defines configuration settings for serialization.
43	**SerializerProvider getSerializerProvider()**
44	**SubtypeResolver getSubtypeResolver()** Method for accessing subtype resolver in use.
45	**TypeFactory getTypeFactory()** Accessor for getting currently configured TypeFactory instance.
46	**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.
47	**boolean isEnabled(DeserializationConfig.Feature f)**
48	**boolean isEnabled(JsonGenerator.Feature f)**
49	**boolean isEnabled(JsonParser.Feature f)**
50	**boolean isEnabled(SerializationConfig.Feature f)**

51	**ObjectWriter prettyPrintingWriter(PrettyPrinter pp)**
52	**ObjectReader reader()** Factory method for constructing ObjectReader with default settings.
53	**ObjectReader reader(Class<?> type)** Factory method for constructing ObjectReader that reads or updates instances of specified type.
54	**ObjectReader reader(FormatSchema schema)** Factory method for constructing ObjectReader that will pass specific schema object to JsonParser used for reading content.
55	**ObjectReader reader(InjectableValues injectableValues)** Factory method for constructing ObjectReader that will use specified injectable values.
56	**ObjectReader reader(JavaType type)** Factory method for constructing ObjectReader that will read or update instances of specified type
57	**ObjectReader reader(JsonNodeFactory f)** Factory method for constructing ObjectReader that will use specified JsonNodeFactory for constructing JSON trees.
58	**ObjectReader reader(TypeReference<?> type)** Factory method for constructing ObjectReader that will read or update instances of specified type
59	**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.
60	**JsonNode readTree(byte[] content)** Method to deserialize JSON content as tree expressed using set of JsonNode instances.
61	**JsonNode readTree(File file)** Method to deserialize JSON content as tree expressed using set of JsonNode instances.
62	**JsonNode readTree(InputStream in)** Method to deserialize JSON content as tree expressed using set of JsonNode instances.
63	**JsonNode readTree(JsonParser jp)** Method to deserialize JSON content as tree expressed using set of JsonNode instances.
64	**JsonNode readTree(JsonParser jp, DeserializationConfig cfg)** Method to deserialize JSON content as tree expressed using set of JsonNode instances.
65	**JsonNode readTree(Reader r)** Method to deserialize JSON content as tree expressed using set of JsonNode instances.
66	**JsonNode readTree(String content)** Method to deserialize JSON content as tree expressed using set of JsonNode instances.

67	**JsonNode readTree(URL source)** Method to deserialize JSON content as tree expressed using set of JsonNode instances.
68	**<T> T readValue(byte[] src, Class<T> valueType)**
69	**<T> T readValue(byte[] src, int offset, int len, Class<T> valueType)**
70	**<T>** T readValue(byte[] src, int offset, int len, JavaType valueType)
71	**<T> T readValue(byte[] src, int offset, int len, TypeReference valueTypeRef)**
72	**<T> T readValue(byte[] src, JavaType valueType)**
73	**<T> T readValue(byte[] src, TypeReference valueTypeRef)**
74	**<T> T readValue(File src, Class<T> valueType)**
75	**<T> T readValue(File src, JavaType valueType)**
76	**<T> T readValue(File src, TypeReference valueTypeRef)**
77	**<T> T readValue(InputStream src, Class<T> valueType)**
78	**<T> T readValue(InputStream src, JavaType valueType)**
79	**<T> T readValue(InputStream src, TypeReference valueTypeRef)**
80	**<T> T readValue(JsonNode root, Class<T> valueType)** Convenience method for converting results from given JSON tree into given value type.
81	**<T> T readValue(JsonNode root, JavaType valueType)** Convenience method for converting results from given JSON tree into given value type.

82	**<T> T readValue(JsonNode root, TypeReference valueTypeRef)** Convenience method for converting results from given JSON tree into given value type.
83	**<T> T readValue(JsonParser jp, 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).
84	**<T> T readValue(JsonParser jp, Class<T> valueType, DeserializationConfig cfg)** 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).
85	**<T> T readValue(JsonParser jp, JavaType valueType)** Method to deserialize JSON content into a Java type, reference to which is passed as argument.
86	**<T> T readValue(JsonParser jp, JavaType valueType, DeserializationConfig cfg)** Method to deserialize JSON content into a Java type, reference to which is passed as argument.
87	**<T> T readValue(JsonParser jp, TypeReference<?> valueTypeRef)** Method to deserialize JSON content into a Java type, reference to which is passed as argument.
88	**<T> T readValue(JsonParser jp, TypeReference<?> valueTypeRef, DeserializationConfig cfg)** Method to deserialize JSON content into a Java type, reference to which is passed as argument.
89	**<T> T readValue(Reader src, Class<T> valueType)**

90	**<T> T readValue(Reader src, JavaType valueType)**
91	**<T> T readValue(Reader src, TypeReference valueTypeRef)**
92	**<T> T readValue(String content, Class<T> valueType)**
93	**<T> T readValue(String content, JavaType valueType)**
94	**<T> T readValue(String content, TypeReference valueTypeRef)**
95	**<T> T readValue(URL src, Class<T> valueType)**
96	**<T> T readValue(URL src, JavaType valueType)**
97	**<T> T readValue(URL src, TypeReference valueTypeRef)**
98	**<T> MappingIterator<T> readValues(JsonParser jp, Class<T> valueType)** Method for reading sequence of Objects from parser stream.
99	**<T> MappingIterator<T> readValues(JsonParser jp, JavaType valueType)** Method for reading sequence of Objects from parser stream.
100	**<T> MappingIterator<T> readValues(JsonParser jp, TypeReference<?> valueTypeRef)** Method for reading sequence of Objects from parser stream.
101	**void 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.
102	**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).
103	**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).
104	**ObjectReader schemaBasedReader(FormatSchema schema)**
105	**ObjectWriter schemaBasedWriter(FormatSchema schema)**
106	**ObjectMapper setAnnotationIntrospector(AnnotationIntrospector ai)** Method for changing AnnotationIntrospector used by this mapper instance for both serialization and deserialization
107	**void setDateFormat(DateFormat dateFormat)** Method for configuring the default DateFormat to use when serializing time values as Strings, and deserializing from JSON Strings.
108	**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.
109	**ObjectMapper setDeserializationConfig(DeserializationConfig cfg)** Method for replacing the shared default deserialization configuration object.
110	**ObjectMapper setDeserializerProvider(DeserializerProvider p)** Method for setting specific DeserializerProvider to use for handling the caching of JsonDeserializer instances.
111	

	void setFilters(FilterProvider filterProvider)
112	**void 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.
113	**ObjectMapper setInjectableValues(InjectableValues injectableValues)**
114	**ObjectMapper setNodeFactory(JsonNodeFactory f)** Method for specifying JsonNodeFactory to use for constructing root level tree nodes (via method createObjectNode()
115	**ObjectMapper setPropertyNamingStrategy(PropertyNamingStrategy s)** Method for setting custom property naming strategy to use.
116	**ObjectMapper setSerializationConfig(SerializationConfig cfg)** Method for replacing the shared default serialization configuration object.
117	**ObjectMapper setSerializationInclusion(JsonSerialize.Inclusion incl)** Method for setting defalt POJO property inclusion strategy for serialization.
118	**ObjectMapper setSerializerFactory(SerializerFactory f)** Method for setting specific SerializerFactory to use for constructing (bean) serializers.
119	**ObjectMapper setSerializerProvider(SerializerProvider p)** Method for setting specific SerializerProvider to use for handling caching of JsonSerializer instances.

120	**void setSubtypeResolver(SubtypeResolver r)** Method for setting custom subtype resolver to use.
121	**ObjectMapper setTypeFactory(TypeFactory f)** Method that can be used to override TypeFactory instance used by this mapper.
122	**ObjectMapper setVisibility(JsonMethod forMethod, JsonAutoDetect.Visibility visibility)** Convenience method that allows changing configuration for underlying VisibilityCheckers, to change details of what kinds of properties are auto-detected.
123	**void setVisibilityChecker(VisibilityChecker<?> vc)** Method for setting currently configured visibility checker; object used to determine whether given property element (method, field, constructor) can be auto-detected or not.
125	**JsonParser treeAsTokens(JsonNode n)** Method for constructing a JsonParser out of JSON tree representation.
126	**<T> T treeToValue(JsonNode n, Class<T> valueType)** Convenience conversion method that binds data provided the JSON tree contains a specific value (usually bean) type.
127	**ObjectWriter typedWriter(Class<?> rootType)** Deprecated. Since 1.9, use writerWithType(Class) instead.
128	**ObjectWriter typedWriter(JavaType rootType)** Deprecated. Since 1.9, use writerWithType(JavaType) instead.

129	**ObjectWriter typedWriter(TypeReference<?> rootType)** Deprecated. Since 1.9, use writerWithType(TypeReference) instead.
130	**ObjectReader updatingReader(Object valueToUpdate)** Deprecated. Since 1.9, use readerForUpdating(java.lang.Object) instead.
131	**<T extends JsonNode> T valueToTree(Object fromValue)** Reverse of treeToValue(org.codehaus.jackson.JsonNode, java.lang.Class); given a value (usually bean), constructs equivalent JSON Tree representation.
132	**Version version()** Method that will return version information stored and reads from jar that contains this class.
133	**ObjectWriter viewWriter(Class<?> serializationView)** Deprecated. Since 1.9, use writerWithView(Class) instead.
134	**ObjectMapper withModule(Module module)** **Fluent-style alternative to registerModule(org.codehaus.jackson.map.Module);**
135	**ObjectWriter writer()** Convenience method for constructing ObjectWriter with default settings.
136	**ObjectWriter writer(DateFormat df)** Factory method for constructing ObjectWriter that serializes objects using specified DateFormat; or, if null passed, using timestamp (64-bit number.
137	**ObjectWriter writer(FilterProvider filterProvider)**

	Factory method for constructing ObjectWriter that serializes objects using specified filter provider.
138	**ObjectWriter writer(FormatSchema schema)** Factory method for constructing ObjectWriter that passes specific schema object to JsonGenerator used for writing content.
139	**ObjectWriter writer(PrettyPrinter pp)** Factory method for constructing ObjectWriter that serializes objects using specified pretty printer for indentation (or if null, no pretty printer)
140	**ObjectWriter writerWithDefaultPrettyPrinter()** Factory method for constructing ObjectWriter that serializes objects using the default pretty printer for indentation
141	**ObjectWriter writerWithType(Class<?> rootType)** Factory method for constructing ObjectWriter that serializes objects using specified root type, instead of actual runtime type of value.
142	**ObjectWriter writerWithType(JavaType rootType)** Factory method for constructing ObjectWriter that serializes objects using specified root type, instead of actual runtime type of value.
143	**ObjectWriter writerWithType(TypeReference<?> rootType)** Factory method for constructing ObjectWriter that serializes objects using specified root type, instead of actual runtime type of value.
144	**ObjectWriter writerWithView(Class<?> serializationView)** Factory method for constructing ObjectWriter that serializes objects using specified JSON View (filter).
145	**void writeTree(JsonGenerator jgen, JsonNode rootNode)**

	Method to serialize given JSON Tree, using generator provided.
146	**void writeTree(JsonGenerator jgen, JsonNode rootNode, SerializationConfig cfg)** Method to serialize given Json Tree, using generator provided.
147	**void writeValue(File resultFile, Object value)** Method that can be used to serialize any Java value as JSON output, written to File provided.
148	**void writeValue(JsonGenerator jgen, Object value)** Method that can be used to serialize any Java value as JSON output, using provided JsonGenerator.
149	**void writeValue(JsonGenerator jgen, Object value, SerializationConfig config)** Method that can be used to serialize any Java value as JSON output, using provided JsonGenerator, configured as per passed configuration object.
150	**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).
151	**void writeValue(Writer w, Object value)** Method that can be used to serialize any Java value as JSON output, using Writer provided.
152	**byte[] writeValueAsBytes(Object value)** Method that used to serialize any Java value as a byte array.
153	**String writeValueAsString(Object value)** Method that can be used to serialize any Java value as a String.

ObjectMapper class inherits methods from java.lang.Object.

ObjectMapper Example

Create the following Java program using any editor of your choice and save it in the folder **C:/> Jackson_WORKSPACE**

File: JacksonTester.java

```
import java.io.IOException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.map.SerializationConfig;

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);
         mapper.enable(SerializationConfig.Feature.INDENT_OUTPUT);
         jsonString = mapper.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 execute the jacksonTester to see the result.

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

Verify the output:

```
Student [ name: Mahesh, age: 21 ]
```

```
{
  "name" : "Mahesh",
  "age" : 21
}
```

5. OBJECT SERIALIZATION

To understand object serialization in detail, let us serialize a Java object to a JSON file and then read that JSON file to get the object back.

Object Serialization – Example

In the following example, we will create a Student class. Thereafter we will create a student.json file which will have a JSON representation of Student object.

First of all, create a Java class file named JacksonTester in **C:\>Jackson_WORKSPACE**.

File: JacksonTester.java

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

import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.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, IOException{
      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 execute the jacksonTester to see the result.

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

Verify the Output:

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

6. DATA BINDING

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

- **Simple Data Binding** – It converts JSON to and from Java Maps, Lists, Strings, Numbers, Booleans, and null objects.
- **Full Data Binding** – It converts JSON to and from any Java type.

ObjectMapper reads/writes JSON for both types of data bindings. Data binding is analogous to JAXB parser for XML.

We will cover simple data binding in this chapter. Full data binding is discussed separately in the next chapter.

Simple Data Binding

Simple data binding refers to mapping of JSON to JAVA core data types. The following table illustrates the relationship between JSON types versus Java types.

S. 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

Simple Data Binding – Example

Let us take a simple example to understand simple data binding in detail. 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 org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.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]
```

7. 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);

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

Full Data Binding – Example

Let us take a simple example to understand full data binding in detail. In the following example, we will map a Java Object 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 org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.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, IOException{
      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 execute the jacksonTester to see the result.

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

Verify the Output:

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

8. GENERICS DATA BINDING

In simple data binding, we have used Map class which uses String as key and Object as a value object. Instead, we can have a concrete Java object and type cast it to use it 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.List;
import java.util.Map;

import org.codehaus.jackson.JsonGenerationException;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.type.TypeReference;

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

            Map userDataMap = new HashMap();
            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>() { };
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;
```

```
   }
}</map
```

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]
```

9. TREE MODEL

The Tree Model prepares an in-memory tree representation of a JSON document. It is the most flexible approach among the three processing modes that Jackson supports. It is quite similar to DOM parser in XML.

Create a 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\": [100,90,85]}";

//create tree from JSON
JsonNode rootNode = mapper.readTree(jsonString);
```

Traversing a Tree

Get each node using the relative path to the root node while traversing the tree and process the data. The following code snippet shows how to traverse a tree, provided you have information regarding the root node.

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

JsonNode marksNode = rootNode.path("marks");
Iterator iterator = marksNode.getElements();
```

Tree Model – Example

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

File: JacksonTester.java

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

import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;

public class JacksonTester {
   public static void main(String args[]){
      JacksonTester tester = new JacksonTester();
      try {
         ObjectMapper mapper = new ObjectMapper();
         String jsonString = "{\"name\":\"Mahesh Kumar\",
         \"age\":21,\"verified\":false,\"marks\": [100,90,85]}";
         JsonNode rootNode = mapper.readTree(jsonString);

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

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

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

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

```
         System.out.print("Marks: [ ");
         while (iterator.hasNext()) {
            JsonNode marks = iterator.next();
            System.out.print(marks.getIntValue() + " ");
         }
         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 execute 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 Conversion

In the following example, we will create a Tree using JsonNode and write it to a JOSN file and read it back.

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.Iterator;

import org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.node.ArrayNode;
import org.codehaus.jackson.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"));
```

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

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

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

         JsonNode marksNode1 = rootNode.path("marks");
         Iterator<JsonNode> iterator = marksNode1.getElements();
         System.out.print("Marks: [ ");
         while (iterator.hasNext()) {
            JsonNode marks = iterator.next();
            System.out.print(marks.getIntValue() + " ");
         }
         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 the following example, we will perform the following operations:

- Create a Tree using JsonNode
- Write it to a JSON file
- Read back the tree and then convert it to a Student object.

First of all, 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 org.codehaus.jackson.JsonNode;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;
import org.codehaus.jackson.node.ArrayNode;
import org.codehaus.jackson.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 execute 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 ]
```

10. STREAMING API

Streaming API reads and writes JSON content as discrete events. JsonParser reads the data, whereas JsonGenerator writes the data.

- It is the most powerful approach among the three processing modes that Jackson supports.
- It has the lowest overhead and it provides the fastest way to perform read/write operations.
- It is analogous to Stax parser for XML.

In this chapter, we will discuss how to read and write JSON data using Jackson streaming APIs. Streaming API works with the concept of token and every details of JSON is to be handled carefully. Following are the two classes which we will use in the examples given in this chapter:

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

JsonGenerator

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

Class Declaration

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

```
public abstract class JsonGenerator
   extends Object
      implements Closeable
```

Nested Class

S.N.	Class & Description
1	**static class JsonGenerator.Feature** Enumeration that defines all togglable features for generators.

Fields

protected PrettyPrinter_cfgPrettyPrinter - Object that handles pretty-printing (usually additional whitespace to make results more human-readable) during output.

Constructors

S.N.	Constructor & Description
1	<="" b="" style="box-sizing: border-box;"> Default constructor

Class Methods

S.N.	Method & Description
1	**boolean canUseSchema(FormatSchema schema)** Method that can be used to verify that a given schema can be used with this generator (using setSchema(org.codehaus.jackson.FormatSchema)).
2	**abstract void close()** Method called to close this generator, so that no more content can be written.
3	**JsonGenerator configure(JsonGenerator.Feature f, boolean state)** Method for enabling or disabling a specified feature: check JsonGenerator.Feature for a list of available features.
4	**abstract void copyCurrentEvent(JsonParser jp)** Method for copying the contents of the current event that the given parser instance points to.
5	**abstract void copyCurrentStructure(JsonParser jp)** Method for copying the contents of the current event and its following events that it encloses the given parser instance points to.

6	**abstract JsonGenerator disable(JsonGenerator.Feature f)** Method for disabling the specified features (check JsonGenerator.Feature for list of features)
7	**void disableFeature(JsonGenerator.Feature f)** Deprecated. Use disable(org.codehaus.jackson.JsonGenerator.Feature) instead
8	**abstract JsonGenerator enable(JsonGenerator.Feature f)** Method for enabling specified parser features: check JsonGenerator.Feature for list of available features.
9	**void enableFeature(JsonGenerator.Feature f)** Deprecated. Use enable(org.codehaus.jackson.JsonGenerator.Feature) instead
10	**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.
11	**CharacterEscapes getCharacterEscapes()** Method for accessing custom escapes factory uses for JsonGenerators it creates.
12	**abstract ObjectCodec getCodec()** Method for accessing the object used for writing Java object as Json content (using method writeObject(java.lang.Object)).
13	**int getHighestEscapedChar()** Accessor method for testing what is the highest unescaped character configured for this generator.
14	**abstract JsonStreamContext getOutputContext()**
15	**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.
16	**abstract boolean isClosed()** Method that can be called to determine whether this generator is closed or not.
17	**abstract boolean isEnabled(JsonGenerator.Feature f)** Method for checking whether a given feature is enabled.
18	**boolean isFeatureEnabled(JsonGenerator.Feature f)** Deprecated. Use isEnabled(org.codehaus.jackson.JsonGenerator.Feature) instead
19	**JsonGenerator setCharacterEscapes(CharacterEscapes esc)** Method for defining custom escapes factory uses for JsonGenerators it creates.
20	**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)).
21	**void setFeature(JsonGenerator.Feature f, boolean state)** Deprecated. Use configure(org.codehaus.jackson.JsonGenerator.Feature, boolean) instead
22	**JsonGenerator setHighestNonEscapedChar(int charCode)** Method that can be called to request that the generator escapes all the character codes above a 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).
23	**JsonGenerator setPrettyPrinter(PrettyPrinter pp)** Method for setting a custom pretty printer, which is usually used to add indentation for improved human readability.
24	**void setSchema(FormatSchema schema)** Method to call to make this generator use a specified schema.

25	**abstract JsonGenerator useDefaultPrettyPrinter()** Convenience method for enabling pretty-printing using the default pretty printer (DefaultPrettyPrinter).
26	**Version version()** Method called to detect the version of the component that implements this interface; returned version should never be null, but may return specific "not available" instance (see Version for details).
27	**void writeArrayFieldStart(String fieldName)** Convenience method for outputting a field entry ("member") (that will contain a JSON Array value), and the START_ARRAY marker.
28	**abstract void writeBinary(Base64Variant b64variant, byte[] data, int offset, int len)** Method that will output a given chunk of binary data as base64 encoded, as a complete String value (surrounded by double quotes).
29	**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).
30	**void writeBinary(byte[] data, int offset, int len)** Similar to writeBinary(Base64Variant,byte[],int,int), but defaults to using the Jackson default Base64 variant (which is Base64Variants.MIME_NO_LINEFEEDS).
31	**void writeBinaryField(String fieldName, byte[] data)** Convenience method for outputting a field entry ("member") that contains the specified data in base64-encoded form.
32	**abstract void writeBoolean(boolean state)** Method for outputting literal Json boolean value (one of Strings 'true' and 'false').

33	**void writeBooleanField(String fieldName, boolean value)** Convenience method for outputting a field entry ("member") that has a boolean value.
34	**abstract void writeEndArray()** Method for writing the closing marker of a JSON Array value (character ']'; plus possible whitespace decoration if pretty-printing is enabled).
35	**abstract void writeEndObject()** Method for writing the closing marker of a JSON Object value (character '}'; plus possible whitespace decoration if pretty-printing is enabled).
36	**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.
37	**void writeFieldName(SerializedString 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.
38	**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 whitespace if pretty-printing is enabled.
39	**abstract void writeNull()** Method for outputting literal Json null value.
40	**void writeNullField(String fieldName)** Convenience method for outputting a field entry ("member") that has JSON literal value null.
41	**abstract void writeNumber(BigDecimal dec)** Method for outputting indicate Json numeric value.

42	**abstract void writeNumber(BigInteger v)** Method for outputting a given value as a Json number.
43	**abstract void writeNumber(double d)** Method for outputting indicate Json numeric value.
44	**abstract void writeNumber(float f)** Method for outputting indicate Json numeric value.
45	**abstract void writeNumber(int v)** Method for outputting a given value as Json number.
46	**abstract void writeNumber(long v)** Method for outputting a given value as Json number.
47	**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.
48	**void writeNumberField(String fieldName, BigDecimal value)** Convenience method for outputting a field entry ("member") that has the specified numeric value.
49	**void writeNumberField(String fieldName, double value)** Convenience method for outputting a field entry ("member") that has the specified numeric value.
50	**void writeNumberField(String fieldName, float value)** Convenience method for outputting a field entry ("member") that has the specified numeric value.
51	**void writeNumberField(String fieldName, int value)** Convenience method for outputting a field entry ("member") that has the specified numeric value.

52	**void writeNumberField(String fieldName, long value)** Convenience method for outputting a field entry ("member") that has the specified numeric value.
53	**abstract void writeObject(Object pojo)** Method for writing a given Java object (POJO) as Json.
54	**void writeObjectField(String fieldName, Object pojo)** Convenience method for outputting a field entry ("member") that has contents of a specific Java object as its value.
55	**void writeObjectFieldStart(String fieldName)** Convenience method for outputting a field entry ("member") (that will contain a JSON Object value), and the START_OBJECT marker.
56	**abstract void writeRaw(char c)** Method that will force the 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).
57	**abstract void writeRaw(char[] text, int offset, int len)** Method that will force the 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).
58	**abstract void writeRaw(String text)** Method that will force the 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).
59	**abstract void writeRaw(String text, int offset, int len)** Method that will force the 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).
60	**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).
61	**abstract void writeRawValue(char[] text, int offset, int len)**
62	**abstract void writeRawValue(String text)** Method that will force the 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).
63	**abstract void writeRawValue(String text, int offset, int len)**
64	**abstract void writeStartArray()** Method for writing the starting marker of a JSON Array value (character '['; plus possible whitespace decoration if pretty-printing is enabled).
65	**abstract void writeStartObject()** Method for writing the starting marker of a JSON Object value (character '{'; plus possible whitespace decoration if pretty-printing is enabled).
66	**abstract void writeString(char[] text, int offset, int len)** Method for outputting a String value.
67	**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.
68	**abstract void writeString(String text)** Method for outputting a String value.
69	**void writeStringField(String fieldName, String value)** Convenience method for outputting a field entry ("member") that has a String value.

70	**abstract void writeTree(JsonNode rootNode)** Method for writing given JSON tree (expressed as a tree where given JsonNode is the root) using this generator.
71	**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 class:

- java.lang.Object

JsonParser

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 whitespace 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, the 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 JsonParseException _constructError(String msg)** Helper method for constructing JsonParseExceptions based on current state of the parser
2	**boolean canUseSchema(FormatSchema schema)** Method that can be used to verify that given schema can be used with this parser (using setSchema(org.codehaus.jackson.FormatSchema)).
3	**void clearCurrentToken()** Method called to "consume" the current token by effectively removing it so that hasCurrentToken() returns false, and getCurrentToken() null).

4	**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.
5	**JsonParser configure(JsonParser.Feature f, boolean state)** Method for enabling or disabling specified feature (check JsonParser.Feature for list of features)
6	**JsonParser disable(JsonParser.Feature f)** Method for disabling specified feature (check JsonParser.Feature for list of features)
7	**void disableFeature(JsonParser.Feature f)** Deprecated. Use disable(Feature) instead
8	**JsonParser enable(JsonParser.Feature f)** Method for enabling specified parser feature (check JsonParser.Feature for list of features)
9	**void enableFeature(JsonParser.Feature f)** Deprecated. Use enable(Feature) instead
10	**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.
11	**byte[] getBinaryValue()** Convenience alternative to getBinaryValue(Base64Variant) that defaults to using Base64Variants.getDefaultVariant() as the default encoding.
12	**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.

13	**boolean getBooleanValue()** Convenience accessor that can be called when the current token is JsonToken.VALUE_TRUE or JsonToken.VALUE_FALSE.
14	**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.
15	**abstract ObjectCodec getCodec()** Accessor for ObjectCodec associated with this parser, if any.
16	**abstract JsonLocation getCurrentLocation()** Method that returns location of the last processed character; usually for error reporting purposes.
17	**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.
18	**JsonToken getCurrentToken()** Accessor to find which token parser currently points to, if any; null will be returned if none.
19	**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.
20	**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.
21	**Object getEmbeddedObject()** Accessor that can be called if (and only if) the current token is JsonToken.VALUE_EMBEDDED_OBJECT.

22	**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.
23	**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.
24	**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.
25	**JsonToken getLastClearedToken()** Method that can be called to get the last token that was cleared using clearCurrentToken().
26	**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.
27	**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.
28	**abstract Number getNumberValue()** Generic number value accessor method that will work for all kinds of numeric values.
29	**abstract JsonStreamContext getParsingContext()** Method that can be used to access current parsing context reader is in.
30	**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.
31	**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.
32	**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.
33	**abstract int getTextLength()** Accessor used with getTextCharacters(), to know length of String stored in returned buffer.
34	**abstract int getTextOffset()** Accessor used with getTextCharacters(), to know offset of the first text content character within buffer.
35	**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.
36	**boolean getValueAsBoolean()** Method that will try to convert value of current token to a boolean.
37	**boolean getValueAsBoolean(boolean defaultValue)** Method that will try to convert value of current token to a boolean.
38	**double getValueAsDouble()** Method that will try to convert value of current token to a Java double.
39	**double getValueAsDouble(double defaultValue)** Method that will try to convert value of current token to a Java double.

40	**int getValueAsInt()** Method that will try to convert value of current token to a int.
41	**int getValueAsInt(int defaultValue)** Method that will try to convert value of current token to a int.
42	**long getValueAsLong()** Method that will try to convert value of current token to a long.
43	**long getValueAsLong(long defaultValue)** Method that will try to convert value of current token to a long.
44	**boolean hasCurrentToken()** Method for checking whether parser currently points to a token (and data for that token is available).
45	**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.
46	**abstract boolean isClosed()** Method that can be called to determine whether this parser is closed or not.
47	**boolean isEnabled(JsonParser.Feature f)** Method for checking whether specified JsonParser.Feature is enabled.
48	**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.
49	**boolean isFeatureEnabled(JsonParser.Feature f)** Deprecated. Use isEnabled(Feature) instead
50	**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.
51	**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.
52	**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:
53	**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:
54	**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.
55	**abstract JsonToken nextToken()** Main iteration method, which will advance stream enough to determine type of the next token, if any.
56	**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).
57	**<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).

58	**<T> T readValueAs(TypeReference<?> valueTypeRef)** Method to deserialize JSON content into a Java type, reference to which is passed as argument.
59	**JsonNode readValueAsTree()** Method to deserialize JSON content into equivalent "tree model", represented by root JsonNode of resulting model.
60	**<T> Iterator<T> readValuesAs(Class<T> valueType)** Method for reading sequence of Objects from parser stream, all with same specified value type.
61	**<T> Iterator<T> readValuesAs(TypeReference<?> valueTypeRef)** Method for reading sequence of Objects from parser stream, all with same specified value type.
62	**int releaseBuffered(OutputStream out)** Method that can be called to push back any content that has been read but not consumed by the parser.
63	**int releaseBuffered(Writer w)** Method that can be called to push back any content that has been read but not consumed by the parser.
64	**abstract void setCodec(ObjectCodec c)** Setter that allows defining ObjectCodec associated with this parser, if any.
65	**void setFeature(JsonParser.Feature f, boolean state)** Deprecated. Use configure(org.codehaus.jackson.JsonParser.Feature, boolean) instead
66	**void setSchema(FormatSchema schema)** Method to call to make this parser use specified schema.
67	**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.
68	**Version version()** Method called to detect version of the component that implements this interface; returned version should never be null, but may return specific "not available" instance (see Version for details).

Methods Inherited

This class inherits methods from the following class:

- java.lang.Object

Write to JSON using JsonGenerator

It is pretty simple to use JsonGenerator. First, create the JsonGenerator using JsonFactory.createJsonGenerator() method and use its write***() methods to write each JSON value.

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

Let us 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 org.codehaus.jackson.JsonEncoding;
import org.codehaus.jackson.JsonFactory;
```

```
import org.codehaus.jackson.JsonGenerator;
import org.codehaus.jackson.JsonParseException;
import org.codehaus.jackson.map.JsonMappingException;
import org.codehaus.jackson.map.ObjectMapper;

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

         JsonGenerator jsonGenerator = jasonFactory.createJsonGenerator(new
         File("student.json"), JsonEncoding.UTF8);
         // {
         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 execute 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 its nextToken() method to read each JSON string as token. Check each token and process accordingly.

```
JsonFactory jasonFactory = new JsonFactory();
JJsonParser 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 us 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 org.codehaus.jackson.JsonEncoding;
import org.codehaus.jackson.JsonFactory;
import org.codehaus.jackson.JsonGenerator;
import org.codehaus.jackson.JsonParseException;
```

```
import org.codehaus.jackson.JsonParser;
import org.codehaus.jackson.JsonToken;
import org.codehaus.jackson.map.JsonMappingException;

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

         JsonGenerator jsonGenerator = jasonFactory.createJsonGenerator(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 = 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());
            }
            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]
```