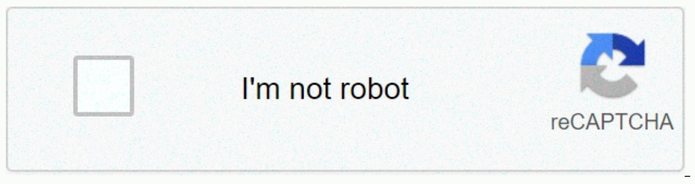

Continue

Jackson json number format

@Target(value={ANNOTATION_TYPE,FIELD,METHOD,PARAMETER,TYPE}) @Retention(value=RUNTIME) public @interface JsonFormat General-purpose annotation used for configuring details of how values of properties are to be serialized. Unlike most other Jackson annotation, annotation has no specific universal explanation: instead, the effect depends on the type of data of the annotation asset (or more specifically, the deserializer and serializer being used). Common uses include choosing between alternative reps -- for example, whether dates are to be serialized as numbers (Java timesn) or strings (such as ISO-8601 compatible time values) -- as well as configuring details precisely with pattern() assets. As of Jackson 2.6, special treats are known to include: Date: Shape can be JsonFormat.Shape.STRING or JsonFormat.Shape.NUMBER; the template may contain a SimpleDateFormat compatible template definition. Can be used on classes (types) as well, for modified default behavior, which can be overwritten by each asset annotates Enums: Shapes JsonFormat.Shape.STRING and JsonFormat.Shape.NUMBER can be used to change between numbers (indicators) and text (name or toString()); but it is also possible to use JsonFormat.Shape.OBJECT to serialize (but not deserialize) Enums as JSON objects (as if they were POJOs). NOTE: serialization such as JSON objects only works with class annotants; will not work by annoted by property. Collections can be posted as (and deserialized from) JSON objects, if JsonFormat.Shape.OBJECT is used. NOTE: Can only be used as class annots; will not work by annoted by property. Subclasses can be serialized as full objects if JsonFormat.Shape.OBJECT is used. Otherwise, the default behavior of serializing a quantity value will be preferred. NOTE: Can only be used as class annots; will not work by annoted by property. Since: 2.0 Public Abstract Series Specific Datatype Templates additional configurations can be used to further refine the formatting aspects. This can, for example, define the low-level string format used for serialization dates; However, the correct use is determined by specific JsonSerializer Defaults: public abstract JsonFormat.Shape configurations to use for serialization: the definition of mapping depends on the datatype, but usually has straight forward partners in the data format (JSON). Note that there is usually only one child set of shapes available; and if an 'invalid' value is selected, the default is usually used. Default: com.fasterxml.jackson.annotation.JsonFormat.Shape.ANY public abstract indigenous string to use for serialization (if necessary). The special value of DEFAULT_LOCALE can be used to mean just using the default, where the default is indicated by the serialization context, which in turn defaults to the default system (Locale.getDefault()) unless a clearly set to another locale. Default: ##default publicly abstract Time zone series to use for serialization (if necessary). The special value DEFAULT_TIMEZONE can be used to just use the default, where the default is specified by the serialization context, which in turn defaults to the default system (TimeZone.getDefault()) unless explicitly set to another locale. Default: ##default A public summary of JsonFormat.Feature[] with Set of JsonFormat.Features to explicitly trigger the processing of annotation properties. This would take preceding over possible global configurations. Since: 2.6 Default: {} public summary JsonFormat.Feature[] without the JsonFormat.Features Set to explicitly disable the processing of annotation properties. This would take preceding over possible global configurations. As of: 2.6 Defaults: {} [Last updated: August 11, 2020] @JsonFormat can also be used to synly layers of java.util.Collection and java.lang.Number as POJOs instead of the corresponding numberless and directionless elements and values. It only works if the annotate is used at the @JsonFormat #shape=JsonFormat.Shape.OBJECT. Java Objects For simplicity, we're wrapping ArrayList in an AbstractList subsort: @JsonFormat(shape = JsonFormat.Shape.OBJECT) public class ArrayListEx<T> extends AbstractList<T> { private List>T> wrapperList = new ArrayList <> (); ..... } The following is a class of examples of Numbers: @JsonFormat(shape = JsonFormat.Shape.OBJECT) public class BigIntegerEx extension BigInteger { ..... } Use the above classes in our main Java audience: public class staff { private string name; private String room; private ArrayListEx phone number; payroll<String> BigIntegerEx private; ..... } Public class ExampleMain { public static void main(String[] args) throws IOException { Employee employee = new Employee(); employee.setName(Amy); employee.setDept ArrayListEx<String> = ArrayListEx <> new (); list.add(111-111-111); list.add(222-222-222); employee.setPhoneNumbers(list); employee.setSalary(new BigIntegerEx(4000)); System.out.println(-- pre-insized --); System.out.println (employee); System.out.println(-- after sedation --); ObjectMapper om = New ObjectMapper(); JsonString chain = om.writeValueAsString (employee); System.out.println (jsonString); } }-- before posting --Employee{name='Amy', dept='Admin', phoneNumbers=[111-111-111, 222-222-222], salary=4000}--after serialization --{name:Amy,dept:Admin,phoneNumbers:{wrapperList:[111-111-111,222-222-222]},salary:{lowestSetBit:5}} Remove @JsonFormat from both our child classes ArrayListEx and BigIntegerEx, in which case the input will be Employee{name='Amy', dept='Admin', phoneNumbers=[111-111-111, 222-222-222], salary=4000} -- after serialization - {name:Amy,dept:Admin,phoneNumbers:[111-111-111,222-222-222],salary:4000} By default The number posted as an innumerable value (primitive). Also in the case of Collection layering, it is also possible to @JsonFormat #shape=JsonFormat.Shape.OBJECTis used. Dependency and used:jackson-databind 2.9.6: General</String> </String> </T> </T> functionality for Jackson: works on the core streaming API. JDK 10Maven 3.5.4 Currently, I'm using Jackson to send JSON results from my spring-based web app. The problem I'm having is trying to get all the money fields to the top with 2 decimal digits. I have not been able to solve this problem using setScale (2), as numbers like 25.50 are cut short to 25.5 etc. Has anyone else handled this problem? I was thinking about making a money class with a custom Jackson serializer... Can you make a custom serializer for a field variable? You may be able to ... But even if still, how can I get my client serializer to add numbers as a number with 2 decimal digits? The Jackson ObjectMapper class (com.fasterxml.jackson.databind.ObjectMapper) is the easiest way to analyze JSON with Jackson. Jackson ObjectMapper can analyze the JSON syntax from a string, line, or file, and create a Java object or object graph that represents JSON in syntax analysis. Analyzing the JSON syntax into Java objects is also called to desalination java objects from JSON. Jackson ObjectMapper can also create JSON from Java objects. Creating JSON from Java objects is also called serialize Java objects into JSON. Jackson Object mapping people can analyze JSON into objects of layers developed by you, or into objects of the integrated JSON tree model explained later in this tutorial. By the way, the reason it's called ObjectMapper is because it mapped JSON as Java Objects (deserialization), or Java Objects to JSON (serialization). Jackson Databind The ObjectMapper is part of the Jackson Databind project, so your app will need that project on its classification line to work. See jackson installation instructions for more information. Jackson ObjectMapper Example This is a java jackson quick ObjectMapper example: ObjectMapper objectMapper = new objectMapper(); CarJson chain = { \brand\ : \Mercedes\, \door\ : 5 }; try { Car car = objectMapper.readValue(carJson, Car.class); System.out.println (car brand = + car.getBrand()); System.out.println (car door = + car.getDoors()); } catch (IOException e) { e.printStackTrace(); } The car class is made by me. As you can see, car.class analysis is the second parameters for readValue(). The first parameters of readValue() are the source JSON (string, thread, or file). Here's what the Car class look like: public class car { private String brand = null; private int door = 0; public String getBrand() { return this.brand; } public void setBrand(String brand){ this.brand = brand;} public int getDoors() { return this.doors; } void public setDoors (int doors) { this.doors = doors; } } How Jackson ObjectMapper matches JSON Fields to Java Fields To read Java objects from JSON with Jackson properly, it is important to know how Jackson maps the fields of JSON objects to the fields of a Java object, so I'll explain how Jackson does it. By default Jackson mapped the fields of a JSON object to the fields in a Java object by combining the names of the field for getter and setter methods in Java objects. Jackson removed the get and set portions of the getter's name and setter methods, and converted the first character of the other name to lowercase. For example, the JSON field has a brand name that matches Java getter and setter methods called getBrand() and setBrand(). The JSON field named EngineNumber matches the getter and setter named getEngineNumber() and setEngineNumber(). If you need to combine JSON object fields with Java object fields in a different way, you need to use custom serializer and deserializer or use some Jackson annots. Jackson Jackson notes contain a set of Java annotatings that you can use to modify the way Jackson reads and writes JSON to and from Java objects. Jackson's caption is explained in my Jackson caption guide. Reading objects from JSON String Reading a Java object from a JSON string is pretty easy. You've actually seen an example of how. The JSON series is transmitted as the first parameters for objectmapper readValue() method. Here's a simple example: ObjectMapper objectMapper = new objectMapper(); CarJson chain = { \brand\ : \Mercedes\, \door\ : 5 }; Car = objectMapper.readValue (carJson, Car.class); Read objects from JSON Reader You can also read an object from JSON downloaded through a Reader version. Here's an example of how to do it: ObjectMapper objectMapper = new objectMapper(); CarJson chain = { \brand\ : \Mercedes\, \door\ : 4 }; Reader reader = new stringreader (carJson); Car = objectMapper.readValue (read, Car.class); Reading objects from JSON File Reading JSON from a file can of course be done through a FileReader (instead of a StringReader – see previous section), but also with a File object. Here's an example of reading JSON from a file: ObjectMapper objectMapper = new objectMapper(); File = New file(data/car.json); Car = objectMapper.readValue(file, Car.class); Read objects from JSON via URL You can read an object from JSON through a URL (java.net.URL) like this: ObjectMapper objectMapper = new objectMapper(); URL = new URL(file:data/car.json); Car = objectMapper.readValue(url, Car.class); This example uses a file URL, but you can also use http URLs (similar to . Read objects from JSON InputStream It is also possible to read an object from JSON through an InputStream with ObjectMapper Jackson. Here's an example of reading JSON from an InputStream: ObjectMapper objectMapper = new objectMapper(); InputStream input = new FileInputStream(data/car.json); Car = objectMapper.readValue (input, Car.class); Reading objects from JSON Byte Array Jackson also supports reading objects from a JSON byte array. Here's an example of reading an object from a JSON byte array: ObjectMapper objectMapper = new objectMapper(); CarJson chain = { \brand\ : \Mercedes\, \door\ : 5 }; bytes[] = carJson.getByte(UTF-8); carJson.getByte(UTF-8); car = objectMapper.readValue(bytes, Car.class); Read Object Array From JSON Array String The Jackson ObjectMapper can also read an array of objects from a JSON array sequence. Here's an example of reading an array of objects from a JSON array: JsonArray series = [{\brand\:\ford\}, {\brand\:\Fiat\}]; ObjectMapper objectMapper = new objectMapper(); Car [] cars2 = objectMapper.readValue(jsonArray, Car[].class); Pay attention to how the Car Plate class is adopted as the second parameters to readValue() method to show theMapper object that you want to read an array of vehicle cases. Read arrays of objects that also work with JSON sources other than a string. For example, a file, URL, InputStream, Reader etc. Read the object list from JSON Array String The Jackson ObjectMapper can also read a Java list of objects from a JSON array string. Here's an example of reading the List of objects from a JSON array series: JsonArray series = [{\brand\:\ford\}, {\brand\:\Fiat\}; ObjectMapper objectMapper = new objectMapper(); List<Car> books cars1 = objectMapper.readValue(jsonArray, TypeReference >> <Car>>(){}); TypeReference parameters message passed to readValue(). This parameters for Jackson to read the list of car objects. Reading maps from JSON String The Jackson ObjectMapper can also read a Java map from a JSON string. This can be useful if you don't know in advance the exact JSON structure that you'll be analyzing. Usually you will read a JSON object into a Java Map. Each field in the JSON object becomes a pair of keys and values in Java Map. Here's an example of reading a Java Map from a JSON string with Jackson ObjectMapper: String jsonObject = {\brand\:\ford\, \doors\:5}; ObjectMapper objectMapper = new objectMapper(); <String, object=> jsonMap = objectMapper.readValue(jsonObject, TypeReference <> <String,Object>>(){}); Ignore The Do Not Specify JSON Field Sometimes you have more fields in JSON than you do in the Java object you want to read from JSON. By default, Jackson throws an exception in that case, sayinging that it does not know the XYZ field because it is not found in Java objects. However, it will sometimes be allowed to have more fields in JSON than in the corresponding Java object. For example, if you're parsing JSON from a REST service that contains more data than you need. In that case, Jackson allows you to skip these additional fields with the Jackson configuration. Here's how to configure the Jackson ObjectMapper to ignore the field of in-name looks: objectMapper.configure (DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false); Failure on the Null JSON value for the original Type Can configure Jackson ObjectMapper fails if a JSON string containing a field has its value set to null, for a field that in the corresponding Java object is a primitive type (int, long, float, double, etc.). To explain what I mean in more detail, look at this class of vehicles: public class vehicles { private string brands =</String,Object> </String,> </Car> </Car> private int door = 0; public String getBrand() { return this.brand; } public void setBrand(String brand){ this.brand = brand;} public int getDoors(){ return this.doors; } public void setDoors (int doors) { this.doors = doors; } } Notice how the door field is an int which is a primitive type in Java (not an object). Now imagine you have a JSON sequence corresponding to a car object that looks like this: { brand:Toyota, doors:null } Notice how the door field contains null values. A primitive type in Java cannot have null values. Therefore Jackson ObjectMapper by default ignores a null value for a primitive field. However, you can configure Jackson ObjectMapper to fail instead. Here's how you configure jackson objectmapper failures for JSON null values for primitive fields in Java classes: ObjectMapper objectMapper = new objectMapper(); objectMapper.configure(DeserializationFeature.FAIL_ON_NULL_FOR_PRIMITIVES, true); With FAIL_ON_NULL_FOR_PRIMITIVES configuration value set to true, you'll get an exception when you try to analyze a JSON null field into a primitive Java field. Here is an example java jackson objectmapper will fail because a JSON field contains a null value for a primitive Java field: ObjectMapper objectMapper = new objectMapper(); objectMapper.configure(DeserializationFeature.FAIL_ON_NULL_FOR_PRIMITIVES, true); String carJson = { \brand\:\Toyota\, \doors\:null }; Car = objectMapper.readValue (carJson, Car.class); Notice how the JSON chain has the door field set to null. Exceptions thrown from this code should look like this: Exceptions in the main thread com.fasterxml.jackson.databind.exc.MismatchedInputException: Cannot map 'null' to type int (set DeserializationConfig.DeserializationFeature.FAIL_ON_NULL_FOR_PRIMITIVES to 'false' to allow) at [Source: (String) { brand:Toyota, doors:null }; line: 1, column: 29] Car[doors]) Custom Deserializer Sometimes you may want to read a JSON string into a Java object in a different way than jackson objectmapper does this by default. You can add a custom deserializer to objectmapper that can perform deserialization as you want it done. Here's how you sign up and use a custom deserializer with ObjectMapper Jackson: String json = { \brand\ : \Ford\, \door\ : 6 }; SimpleModule Module = New SimpleModule (CarDeserializer, New Version (3, 1, 8, null, null, null)); module.addDeserializer (Car.class, New CarDeserializer (Vehicle.class)); ObjectMapper mapping = new objectMapper(); mapper.registerModule (module); Car = mapper.readValue (json, Car.class); And here's how the CarDeserializer class looks: import com.fasterxml.jackson.core.JsonParser; imports com.fasterxml.jackson.core.JsonToken; enter com.fasterxml.jackson.databind.DeserializationContext; import com.fasterxml.jackson.databind.deser.std.StdDeserializer; java.io.IOException; public class cardeserializer StdDeserializer<Car> { public CarDeserializer(Class <?? > vc) { super(vc); } @Override public Car deserialize. DeserializationContext deserializer) throws IOException { Car = new Car(); while(!parser.isClosed()){ JsonToken jsonToken = parser.nextToken(); if(JsonToken.FIELD_NAME.equals(jsonToken)){ String fieldName = parser.getCurRentName(); System.out.println (school name); jsonToken = syntax analysis.nextToken(); if(brand.equals(fieldName)){ car.setBrand(parser.getValueAsString()); } else if (doors.equals(fieldName)){ car.setDoors(parser.getValueAsInt()); } } } return vehicles; } } Write JSON from Jackson ObjectMapper objects that can also be used to create JSON from an object. You do so using one of these methods: writeValue() writeValueAsString() writeValueAsBytes() This is an example of creating JSON from a car object, just like those used in previous examples: ObjectMapper objectMapper = new objectMapper (); Cars = new cars(); car.brand = BMW; car.doors = 4; objectMapper.writeValue (FileOutputStream(data/input-2.json), car); This example first creates an ObjectMapper, then a car case, and finally calls objectMapper's writeValue() method that converts vehicle objects to JSON and writes it into certain FileOutputStream. ObjectMapper's writeValueAsString() and writeValueAsBytes() both create JSON from an object, and return to JSON created as a string or as a byte array. Here's an example of how to call writeValueAsString(): ObjectMapper objectMapper = new objectMapper(); Cars = new cars(); car.brand = BMW; car.doors = 4; Json chain = objectMapper.writeValueAsString (car); System.out.println (json); The JSON input from this example would be: {brand:BMW,doors:4} Custom Serializer Sometimes you want to post a Java object to JSON other than what Jackson does by default. For example, you may want to use different field names in JSON than in Java objects, or you may want to leave out certain fields altogether. Jackson allows you to set up a custom serializer on the ObjectMapper. This serializer is registered for a certain class, and will then be called whenever ObjectMapper is required to serialize a car object. Here is an example showing how to register a custom serializer for the car class: CarSerializer carSerializer = new CarSerializer (Carserializer (Car.class); ObjectMapper objectMapper = new objectMapper(); SimpleModule Module = New SimpleModule (CarSerializer, New Version (2, 1, 3, null, null, null)); module.addSerializer (Car.class, carSerializer); objectMapper.registerModule (module); Cars = new cars(); car.setBrand (Mercedes); car.setDoors(5); CarJson chain = objectMapper.writeValueAsString (car); The chain produced by jackson custom serializer this example looks like this: {manufacturer:Mercedes,doorCount:5} The CarSerializer class looks like this: imported enter com.fasterxml.jackson.databind.SerializerProvider; </Car> </Car> enter java.io.IOException; carerializer public class extension StdSerializer<Car> { CarSerializer protection (Class<Car> t) { super(t); } public void serialize(Car, car, JsonGenerator jsonGenerator, SerializerProvider serializerProvider) throws IOException { jsonGenerator.writeStartObject(); jsonGenerator.writeStringField(manufacturer, car.getBrand()); jsonGenerator.writeNumberField(doorCount, car.getDoors()); jsonGenerator.writeEndObject(); } } Note that the second parameters transmitted to the serialize() method are a Jackson JsonGenerator version. You can use this case to serialize the object - in this case a car object. Jackson Date Format By default Jackson will serialize an object java.util.Date its long value, which is the number of milliseconds as of January 1, 1970. However, Jackson also supports date formatting as a series. In this section, we'll take a closer look at the Jackson day format. On to the first long I'll show you the default Jackson date format that serializes a date to milliseconds as of January 1, 1970 (its longest representation). Here is an example Java class that contains the Date: public class Transaction { private String type = null; private Date date = null; public Transaction() { } public Transaction(String type, Date date) { this.type = type; this.date = date; } public String getType() { return type; } public void setType (String type) { this.type = type; } public Date getDate() { return date; } void public setDate(Date date) { this.date = date; } Serializing an object dealing with ObjectMapper Jackson will be done just as you would serialize any other Java object. Here's what the code looked like: Transaction = New transaction (transfer, New Day()); ObjectMapper objectMapper = new objectMapper(); String input = objectMapper.writeValueAsString(transaction); System.out.println (input); The printed input from this example should be similar to: {type:transfer,date:1516442298301} Notice the format of the date field: It's a long number, as explained above. Date to String The long orderly format of a day is not very easy to read for humans. Jackson therefore supports a text date format too. You specify the correct Jackson date format to use by setting up a SimpleDateFormat on ObjectMapper. Here's an example of setting up a SimpleDateFormat on a Jackson ObjectMapper: SimpleDateFormat dateFormat = the new simpledateformat (yyyy-MM-dd); objectMapper2.setDateFormat(dateFormat); Output2 series = objectMapper2.writeValueAsString(transaction); System.out.println(out2); The printed input from this example should look like this: {type:transfer,date:2018-01-20} Notice how the date and time field is formatted as String. Jackson JSON Tree Model Jackson has an integrated tree model that can be used to represent a JSON object. Jackson's tree model is useful if you don't know how JSON you'll get looked at, or you for some reason can't (or just don't want to</Car> </Car> </Car> create a class to represent it. The Jackson Tree model is also useful if you need to manipulate the JSON before using or forwarding it. All these situations can easily occur in a data transfer scenario. Jackson tree model is represented by JsonNode class. You use Jackson ObjectMapper to analyze JSON into a JsonNode tree model, just like you did with your own class. The following sections will show how to read and write JsonNode versions with Jackson ObjectMapper. The Jackson JsonNode class itself is mentioned in more detail in its own guide to Jackson JsonNode. Example of the Jackson Tree Model Below is a simple Jackson tree model example: String carJson = { \brand\ : \Mercedes\, \doors\ : 5 }; ObjectMapper objectMapper = new objectMapper(); try { JsonNode jsonNode = objectMapper.readValue(carJson, JsonNode.class); catch (IOException e) { e.printStackTrace(); } As you can see, the JSON string is analyzed as a JsonNode object instead of a Car object, simply by passing through JsonNode.class as the second parameters for the readValue() method instead of car.class used in the previous example in this tutorial. The ObjectMapper class also has a special readTree() method that always returns JsonNode. Here's an example of parsing JSON to JsonNode with objectMapper readTree(): String carJson = { \brand\ : \Mercedes\, \doors\ : 5 }; ObjectMapper objectMapper = new objectMapper(); try { JsonNode jsonNode = objectMapper.readTree(carJson); catch (IOException e) { e.printStackTrace(); } The Jackson JsonNode Class JsonNode class allows you to navigate JSON as a Java object in a fairly flexible and dynamic way. As mentioned earlier, the JsonNode class is mentioned in more detail in its own tutorial, but I'll show you the basics of how to use it here. Once you have analyzed your JSON syntax into a JsonNode (or a tree of JsonNode cases), you can navigate the JsonNode tree model. Here's an example of JsonNode showing how to access JSON fields, arrays, and nested objects: CarJson chain = { \brand\ : \Mercedes\, \door\ : 5, + \owner\ : [\John\, \Jack\, \Jill\], + \cageObject\ : { \field\ ObjectMapper objectMapper = new objectMapper(); try { JsonNode jsonNode = objectMapper.readValue (carJson, JsonNode.class); JsonNode brandNode = jsonNode.get (brand); Brand chain = brandNode.asText(); System.out.println (brand = + brand); JsonNode doorsNode = jsonNode.get (doors); int door = doorsNode.asInt(); System.out.println(door = + door); JsonNode Array = jsonNode.get (owner); JsonNode jsonNode = array.get(0); John String = jsonNode.asText(); System.out.println(john = + john); JsonNode con = jsonNode.get(nestedObject); JsonNode childField = child.get(school); String field = childField.asText(); System.out.println(field = + field); } catch (IOException e) { e.printStackTrace(); } Get that the JSON chain now contains an array field called and a nested object field called nestedObject. Regardless of whether you're accessing a field, array, or nested object, you use the get() method of the JsonNode class. By providing a series as a parameters for the get(, you can access a JsonNode field. If JsonNode represents an array, you need to pass an index for the get() method instead. The index determines which elements in the array you want to receive. Convert object to JsonNode Can use Jackson ObjectMapper to convert a Java object to JsonNode with JsonNode as a JSON demonstration of converted Java objects. You convert a Java object into a JsonNode through the Jackson ObjectMapper valueToTree() method. Here's an example of converting a Java object to JsonNode using the ObjectMapper valueToTree() method: ObjectMapper objectMapper = new objectMapper(); Cars = new cars(); car.brand = Cadillac; car.doors = 4; JsonNode carJsonNode = objectMapper.valueToTree (car); Convert JsonNode to Object You can convert JsonNode into a Java object, using the Jackson ObjectMapper treeToValue() method. This is similar to the syntax analysis of a JSON string (or other source) into a Java object with Jackson ObjectMapper. The only difference is, that JSON source is a JsonNode. Here's an example of converting JsonNode into a Java object using the Jackson ObjectMapper treeToValue(): ObjectMapper objectMapper = new objectMapper(); CarJson chain = { \brand\ : \Mercedes\, \door\ : 5 }; JsonNode carJsonNode = objectMapper.readTree(carJson); Car = objectMapper.treeToValue (carJsonNode); The example above is a bit artificial in that we first convert a JSON string into JsonNode and then convert JsonNode into a Car object. Obviously, if we have a reference to a raw JSON string, you can also convert it directly into a Car object, without converting it to JsonNode first. However, the example above is built to show how to convert a JsonNode into a Java object. That's why. Read and write other data formats with Jackson ObjectMapper Can read and write data formats other than JSON with Jackson ObjectMapper. Jackson ObjectMapper can also read and write these data formats (and maybe more): Some of these data formats are more compact than JSON, and therefore take up less space when stored and faster to read and write than JSON. In the following sections, I'll show you how to read and write some of these data formats with Jackson ObjectMapper. Read and write CBOR With Jackson ObjectMapper CBOR is a binary data format compatible with JSON but more compact than JSON, and therefore faster to read and write. Jackson ObjectMapper can read and write CBOR the same way you read and write JSON. To read and write CBOR with Jackson, you need to add a dependency add to your project. More Jackson CBOR Maven dependency is mentioned in the Jackson Installation Guide. Here's an example of an object to CBOR with ObjectMapper Jackson: import com.fasterxml.jackson.core.JsonProcessingException; enter com.fasterxml.jackson.databind.ObjectMapper; imports com.fasterxml.jackson.dataformat.cbor.CBORFactory; public class CborJacksonExample { public static void main(String[] args) { ObjectMapper objectMapper = new ObjectMapper(new CBORFactory()); Employees = new employees (John Doe, john@doe.com); try { bytes[] cborBytes = objectMapper.writeValueAsBytes(employee); } catch (JsonProcessingException e) { e.printStackTrace(); } } } The cborBytes byte array contains an Employee object posted to the CBOR data format. Here's an example of reading the CBOR bytes back into an employee audience again: importing com.fasterxml.jackson.core.JsonProcessingException; enter com.fasterxml.jackson.databind.ObjectMapper; imports com.fasterxml.jackson.dataformat.cbor.CBORFactory; enter java.io.IOException; public class CborJacksonExample { public static void main(String[] args) { ObjectMapper objectMapper = new ObjectMapper(new CBORFactory()); Employees = new employees (John Doe, john@doe.com); bytes[] cborBytes = null; try { cborBytes = objectMapper.writeValueAsBytes(employee); } catch (JsonProcessingException e) { e.printStackTrace(); // usually, rethrow exception here - or don't catch it at all. } try { Employee employee2 = objectMapper.readValue(cborBytes, Employee.class); } catch (IOException e) { e.printStackTrace After running this code, employee2 will point to a different employee object but that is the equivalent of the employee object pointing to, because that object was posted to CBOR and deserialized back to employee2 again. Read and write MessagePack With Jackson ObjectMapper YAML is a text data format that is compatible with JSON but more compact, and therefore faster to read and write. Jackson ObjectMapper can read and write MessagePack the same way you read and write JSON. To read and write MessagePack with Jackson, you need to add an additional Maven dependency to your project. More Jackson MessagePack Maven dependency is mentioned in the Jackson Installation Guide. Here's an example of writing an object to MessagePack with ObjectMapper Jackson: import com.fasterxml.jackson.core.JsonProcessingException; enter com.fasterxml.jackson.databind.ObjectMapper; org.msgpack.jackson.dataformat.MessagePackFactory; enter java.io.IOException; public class MessagePackJacksonExample { public static void main(String[] args) { ObjectMapper objectMapper = new ObjectMapper(new MessagePackFactory()); Employees = new employees (John Doe, john@doe.com); bytes[] messagePackBytes = null; try { messagePackBytes = objectMapper.writeValueAsBytes(employee); } catch (JsonProcessingException e) { e.printStackTrace(); // usually, rethrow exception here - or don't catch it at all. } } MessagePackBytes bytes array containing Employee object serial to MessagePack data Here's an example of reading messagepack bytes back into an employee object again: importing com.fasterxml.jackson.core.JsonProcessingException; enter com.fasterxml.jackson.databind.ObjectMapper; org.msgpack.jackson.dataformat.MessagePackFactory; enter java.io.IOException; public class MessagePackJacksonExample { public static void main(String[] args) { ObjectMapper objectMapper = new ObjectMapper(new MessagePackFactory()); Employees = new employees (John Doe, john@doe.com); bytes[] messagePackBytes = null; try { messagePackBytes = objectMapper.writeValueAsBytes(employee); } catch (JsonProcessingException e) { e.printStackTrace(); // usually, remove the exception here - or don't catch it at all. } try { Employee employee2 = objectMapper.readValue(messagePackBytes, Employee.class); System.out.println(messagePackBytes = + messagePackBytes); } catch (IOException e) { e.printStackTrace(); } } } After running this code, employee2 will point to a different employee object but that is equal to the object the employee points to, because that object was posted to MessagePack and deserialized back to employee2 again. Read and Write YAML With Jackson ObjectMapper YAML is a text data format similar to JSON but uses a different syntax. Jackson ObjectMapper can read and write YAML the same way you read and write JSON. To read and write YAML with Jackson, you need to add an additional Maven dependency to your project. More re depends on the Jackson YAML Maven mentioned in the Jackson Installation Guide. Here's an example of writing an object to YAML with ObjectMapper Jackson: importing com.fasterxml.jackson.core.JsonProcessingException; enter com.fasterxml.jackson.databind.ObjectMapper; enter com.fasterxml.jackson.dataformat.yaml.YAMLFactory; enter java.io.IOException; public class YamlJacksonExample { public static void main(String[] args) { ObjectMapper objectMapper = new ObjectMapper(new YAMLFactory()); Employees = new employees (John Doe, john@doe.com); YamlString series = null; try { yamlString = objectMapper.writeValueAsString(employee); } catch (JsonProcessingException e) { e.printStackTrace(); // usually, rethrow exception here - or don't catch it at all. } } The yamlString variable contains the serialized object staff formatting the YAML data after implementing this code. Here's an example of reading YAML text to an Employee object again: type com.fasterxml.jackson.core.JsonProcessingException; enter com.fasterxml.jackson.databind.ObjectMapper; enter com.fasterxml.jackson.dataformat.yaml.YAMLFactory; enter java.io.IOException; public class YamlJacksonExample { public static void main(String[] args) { ObjectMapper objectMapper = new ObjectMapper(new YAMLFactory()); Employees = new employees (John Doe, john@doe.com); YamlString series = null; try { yamlString = objectMapper.writeValueAsString(employee); } catch (JsonProcessingException e) { usually, rethrow the exception here - or don't catch it at all. } try { Employee employee2 = objectMapper.readValue(yamlString, Employee.class); System.out.println(Done); } catch (IOException e) { e.printStackTrace(); } } } After running this code, employee2 will point to a different employee object but that is equal to the object the employee points to, because that object was posted to YAML and deserialized back to employee2 again. Again.