Jackson ObjectMapper

Last update: 2018-01-20

The *Jackson ObjectMapper* class (`com.fasterxml.jackson.databind.ObjectMapper`) is the simplest way to parse JSON with Jackson. The Jackson `ObjectMapper` can parse JSON from a string, stream or file, and create a Java object or object graph representing the parsed JSON. Parsing JSON into Java objects is also referred to as to *deserialize Java objects from JSON*.

The Jackson ObjectMapper can also create JSON from Java objects. Generating JSON from Java objects is also referred to as to *serialize Java objects into JSON*.

The Jackson Object mapper can parse JSON into objects of classes developed by you, or into objects of the built-in JSON tree model explained later in this tutorial.

Jackson Databind

The `ObjectMapper` is located in the Jackson Databind project, so your application will need that project on its classpath to work. See the **Jackson Installation** tutorial for more information.

Jackson ObjectMapper Example

Here is a quick Java Jackson `ObjectMapper` example:

```
ObjectMapper objectMapper = new ObjectMapper();

String carJson =
    "{ \"brand\" : \"Mercedes\", \"doors\" : 5 }";

try {
    Car car = objectMapper.readValue(carJson, Car.class);

    System.out.println("car brand = " + car.getBrand());
    System.out.println("car doors = " + car.getDoors());
} catch (IOException e) {
    e.printStackTrace();
}
```

The `Car` class was made by me. As you can see, the `Car.class` is parsed as the second parameter to the `readValue()` method. The first parameter of `readValue()` is the source of the JSON (string, stream or file). Here is how the `Car` class looks:

```
public class Car {
    private String brand = null;
    private int doors = 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; }
}
```

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 a JSON object to the fields of a Java object, so I will explain how Jackson does that.

By default Jackson maps the fields of a JSON object to fields in a Java object by matching the names of the JSON field to the getter and setter methods in the Java object. Jackson removes the "get" and "set" part of the names of the getter and setter methods, and converts the first character of the remaining name to lowercase.

For instance, the JSON field named `brand` matches the Java getter and setter methods called `getBrand()` and `setBrand()`. The JSON field named `engineNumber` would match the getter and setter named `getEngineNumber()` and `setEngineNumber()`.

If you need to match JSON object fields to Java object fields in a different way, you need to either use a custom serializer and deserializer, or use some of the many **Jackson Annotations**.

Jackson Annotations

Jackson contains a set of Java annotations which you can use to modify how Jackson reads and writes JSON to and from Java objects. Jackson's annotations are explained in my **Jackson annotation tutorial**.

Read Object From JSON String

Reading a Java object from a JSON string is pretty easy. You have actually already seen an example of how. The JSON string is passed as the first parameter to the `ObjectMapper`'s `readValue()` method. Here is another simplified example:

```
ObjectMapper objectMapper = new ObjectMapper();

String carJson =
    "{ \"brand\" : \"Mercedes\", \"doors\" : 5 }";

Car car = objectMapper.readValue(carJson, Car.class);
```

Read Object From JSON Reader

You can also read an object from JSON loaded via a `Reader` instance. Here is an example of how to do that:

```
ObjectMapper objectMapper = new ObjectMapper();

String carJson =
        "{ \"brand\" : \"Mercedes\", \"doors\" : 4 }";
Reader reader = new StringReader(carJson);

Car car = objectMapper.readValue(reader, Car.class);
```

Read Object From JSON File

Reading JSON from a file can of course be done via a `FileReader` (instead of a `StringReader` - see previous section), but also with a `File` object. Here is an example of reading JSON from a file:

```
ObjectMapper objectMapper = new ObjectMapper();

File file = new File("data/car.json");

Car car = objectMapper.readValue(file, Car.class);
```

Read Object From JSON via URL

You can read an object from JSON via a `URL` (`java.net.URL`) like this:

```
ObjectMapper objectMapper = new ObjectMapper();

URL url = new URL("file:data/car.json");

Car car = objectMapper.readValue(url, Car.class);
```

This example uses a file URL, but you can use an HTTP URL too (similar to `http://jenkov.com/some-data.json`).

Read Object From JSON InputStream

It is also possible to read an object from JSON via an `InputStream` with the Jackson `ObjectMapper`. Here is an example of reading JSON from an `InputStream` :

```
ObjectMapper objectMapper = new ObjectMapper();
```

```
InputStream input = new FileInputStream("data/car.json");

Car car = objectMapper.readValue(input, Car.class);
```

Read Object From JSON Byte Array

Jackson also supports reading objects from a JSON `byte` array. Here is an example of reading an object from a JSON `byte` array:

```
ObjectMapper objectMapper = new ObjectMapper();

String carJson =
        "{ \"brand\" : \"Mercedes\", \"doors\" : 5 }";

byte[] bytes = carJson.getBytes("UTF-8");

Car 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 string. Here is an example of reading an object array from a JSON array string:

```
String jsonArray = "[{\"brand\":\"ford\"}, {\"brand\":\"Fiat\"}]";

ObjectMapper objectMapper = new ObjectMapper();

Car[] cars2 = objectMapper.readValue(jsonArray, Car[].class);
```

Notice how the `Car` array class is passed as the second parameter to the `readValue()` method to tell the `ObjectMapper` that you want to read an array of `Car` instances.

Reading arrays of objects also works with other JSON sources than a string. For instance, a file, URL, `InputStream`, `Reader` etc.

Read Object List From JSON Array String

The Jackson `ObjectMapper` can also read a **Java List** of objects from a JSON array string. Here is an example of reading a `List` of objects from a JSON array string:

```
String jsonArray = "[{\"brand\":\"ford\"}, {\"brand\":\"Fiat\"}]";

ObjectMapper objectMapper = new ObjectMapper();
```

```
List<Car> cars1 = objectMapper.readValue(jsonArray, new TypeReference<List<Car>>(){});
```

Notice the `TypeReference` parameter passed to `readValue()`. This parameter tells Jackson to read a `List` of `Car` objects.

Ignore Unknown JSON Fields

Sometimes you have more fields in the JSON than you do in the Java object you want to read from the JSON. By default Jackson throws an exception in that case, saying that it does not know field XYZ because it is not found in the Java object.

However, sometimes it should be allowed to have more fields in the JSON than in the corresponding Java object. For instance, if you are parsing JSON from a REST service which contains much more data than you need. In that case, Jackson enables you to ignore these extra fields with a Jackson configuration. Here is how configuring the Jackson `ObjectMapper` to ignore unknown fields looks:

```
objectMapper.configure(
    DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
```

Read Map from JSON String

The Jackson `ObjectMapper` can also read a **Java Map** from a JSON string. This can be useful if you do not know ahead of time the exact JSON structure that you will be parsing. Usually you will be reading a JSON object into a Java `Map`. Each field in the JSON object will become a key, value pair in the Java `Map`.

Here is an example of reading a Java `Map` from a JSON String with the Jackson `ObjectMapper`:

```
String jsonObject = "{\"brand\":\"ford\", \"doors\":5}";

ObjectMapper objectMapper = new ObjectMapper();
Map<String, Object> jsonMap = objectMapper.readValue(jsonObject,
    new TypeReference<Map<String,Object>>(){});
```

Custom Deserializer

Sometimes you might want to read a JSON string into a Java object in a way that is different from how the Jackson `ObjectMapper` does this by default. You can add a *custom deserializer* to the `ObjectMapper` which can perform the deserialization as you want it done.

Here is how you register and use a custom deserializer with the Jackson `ObjectMapper`:

```
String json = "{ \"brand\" : \"Ford\", \"doors\" : 6 }";

SimpleModule module =
        new SimpleModule("CarDeserializer", new Version(3, 1, 8, null, null, null));
module.addDeserializer(Car.class, new CarDeserializer(Car.class));

ObjectMapper mapper = new ObjectMapper();
mapper.registerModule(module);

Car car = mapper.readValue(json, Car.class);
```

And here is how the `CarDeserializer` class looks:

```
import com.fasterxml.jackson.core.JsonParser;
import com.fasterxml.jackson.core.JsonToken;
import com.fasterxml.jackson.databind.DeserializationContext;
import com.fasterxml.jackson.databind.deser.std.StdDeserializer;

import java.io.IOException;

public class CarDeserializer extends StdDeserializer<Car> {

    public CarDeserializer(Class<?> vc) {
        super(vc);
    }

    @Override
    public Car deserialize(JsonParser parser, DeserializationContext deserializer) throws IOExc
        Car car = new Car();
        while(!parser.isClosed()){
            JsonToken jsonToken = parser.nextToken();

            if(JsonToken.FIELD_NAME.equals(jsonToken)){
                String fieldName = parser.getCurrentName();
                System.out.println(fieldName);

                jsonToken = parser.nextToken();

                if("brand".equals(fieldName)){
                    car.setBrand(parser.getValueAsString());
                } else if ("doors".equals(fieldName)){
                    car.setDoors(parser.getValueAsInt());
                }
            }
        }
        return car;
    }
}
```

Write JSON From Objects

The Jackson `ObjectMapper` can also be used to generate JSON from an object. You do so using the one of the methods:

- `writeValue()`
- `writeValueAsString()`
- `writeValueAsBytes()`

Here is an example of generating JSON from a `Car` object, like the ones used in earlier examples:

```
ObjectMapper objectMapper = new ObjectMapper();

Car car = new Car();
car.brand = "BMW";
car.doors = 4;

objectMapper.writeValue(
    new FileOutputStream("data/output-2.json"), car);
```

This example first creates an `ObjectMapper`, then a `Car` instance, and finally calls the `ObjectMapper`'s `writeValue()` method which converts the `Car` object to JSON and writes it to the given `FileOutputStream`.

The `ObjectMapper`'s `writeValueAsString()` and `writeValueAsBytes()` both generate JSON from an object, and return the generated JSON as a `String` or as a `byte` array. Here is an example showing how to call `writeValueAsString()`:

```
ObjectMapper objectMapper = new ObjectMapper();

Car car = new Car();
car.brand = "BMW";
car.doors = 4;

String json = objectMapper.writeValueAsString(car);
System.out.println(json);
```

The JSON output from this example would be:

```
{"brand":"BMW","doors":4}
```

Custom Serializer

Sometimes you want to serialize a Java object to JSON differently than what Jackson does by default. For instance, you might want to use different field names in the JSON than in the Java object, or you might want to leave out certain fields altogether.

Jackson enables you to set a custom serializer on the `ObjectMapper`. This serializer is registered for a

Jackson enables you to set a custom serializer on the `ObjectMapper`. This serializer is registered for a certain class, and will then be called whenever the `ObjectMapper` is asked to serialize a `Car` object. Here is an example that shows how to register a custom serializer for the `Car` class:

```
CarSerializer carSerializer = new 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);

Car car = new Car();
car.setBrand("Mercedes");
car.setDoors(5);

String carJson = objectMapper.writeValueAsString(car);
```

The string produced by this Jackson custom serializer example looks like this:

```
{"producer":"Mercedes","doorCount":5}
```

The `CarSerializer` class looks like this:

```
import com.fasterxml.jackson.core.JsonGenerator;
import com.fasterxml.jackson.databind.SerializerProvider;
import com.fasterxml.jackson.databind.ser.std.StdSerializer;

import java.io.IOException;

public class CarSerializer extends StdSerializer<Car> {

    protected CarSerializer(Class<Car> t) {
        super(t);
    }

    public void serialize(Car car, JsonGenerator jsonGenerator,
                          SerializerProvider serializerProvider)
            throws IOException {

        jsonGenerator.writeStartObject();
        jsonGenerator.writeStringField("producer", car.getBrand());
        jsonGenerator.writeNumberField("doorCount", car.getDoors());
        jsonGenerator.writeEndObject();
    }
}
```

Notice that the second parameter passed to the `serialize()` method is a **Jackson JsonGenerator** instance. You can use this instance to serialize the object - in this case a `Car` object.

instance. You can use this instance to serialize the object - in this case a `Car` object.

Jackson Date Formats

By default Jackson will serialize a `java.util.Date` object to its `long` value, which is the number of milliseconds since January 1st 1970. However, Jackson also supports formatting dates as strings. In this section we will take a closer look at the Jackson date formats.

Date to long

First I will show you the default Jackson date format that serializes a `Date` to the number of milliseconds since January 1st 1970 (its `long` representation). Here is an example Java class that contains a `Date` field:

```
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;
    }

    public void setDate(Date date) {
        this.date = date;
    }
}
```

Serializing a `Transaction` object with the Jackson `ObjectMapper` would be done just as you would serialize any other Java object. Here is how the code looks:

```
Transaction transaction = new Transaction("transfer", new Date());

ObjectMapper objectMapper = new ObjectMapper();
String output = objectMapper.writeValueAsString(transaction);
```

```
System.out.println(output);
```

The output printed from this example would be similar to:

```
{"type":"transfer","date":1516442298301}
```

Notice the format of the `date` field: It is a long number, just as explained above.

Date to String

The `long` serialization format of a `Date` is not very readable for human beings. Therefore Jackson supports a textual date format too. You specify the exact Jackson date format to use by setting a `SimpleDateFormat` on the `ObjectMapper`. Here is an example of setting a `SimpleDateFormat` on a Jackson `ObjectMapper`:

```
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
objectMapper2.setDateFormat(dateFormat);

String output2 = objectMapper2.writeValueAsString(transaction);
System.out.println(output2);
```

The output printed from this example would look similar to this:

```
{"type":"transfer","date":"2018-01-20"}
```

Notice how the `date` field is now formatted as a String.

Jackson JSON Tree Model

Jackson has a built-in tree model which can be used to represent a JSON object. Jackson's tree model is useful if you don't know how the JSON you will receive looks, or if you for some reason cannot (or just don't want to) create a class to represent it.

The Jackson tree model is represented by the `JsonNode` class. You use the Jackson `ObjectMapper` to parse JSON into a `JsonNode` tree model, just like you would have done with your own class.

Jackson Tree Model Example

Here is a simple Jackson tree model example:

```
String carJson =
        "{ \"brand\" : \"Mercedes\", \"doors\" : 5 }";
```

```
ObjectMapper objectMapper = new ObjectMapper();

try {

    JsonNode node = objectMapper.readValue(carJson, JsonNode.class);

} catch (IOException e) {
    e.printStackTrace();
}
```

As you can see, the JSON string is parsed into a `JsonNode` object instead of a `Car` object, simply by passing the `JsonNode.class` as second parameter to the `readValue()` method instead of the `Car.class` used in the example earlier in this tutorial.

The Jackson JsonNode Class

Once you have parsed your JSON into a `JsonNode` (or a tree of `JsonNode` instances) you can navigate the `JsonNode` tree model. Here is a `JsonNode` example that shows how to access JSON fields, arrays and nested objects:

```
String carJson =
        "{ \"brand\" : \"Mercedes\", \"doors\" : 5," +
        "  \"owners\" : [\"John\", \"Jack\", \"Jill\"]," +
        "  \"nestedObject\" : { \"field\" : \"value\" } }";

ObjectMapper objectMapper = new ObjectMapper();


try {

    JsonNode node = objectMapper.readValue(carJson, JsonNode.class);

    JsonNode brandNode = node.get("brand");
    String brand = brandNode.asText();
    System.out.println("brand = " + brand);

    JsonNode doorsNode = node.get("doors");
    int doors = doorsNode.asInt();
    System.out.println("doors = " + doors);

    JsonNode array = node.get("owners");
    JsonNode jsonNode = array.get(0);
    String john = jsonNode.asText();
    System.out.println("john  = " + john);

    JsonNode child = node.get("nestedObject");
    JsonNode childField = child.get("field");
    String field = childField.asText();
    System.out.println("field = " + field);

} catch (IOException e) {
    e.printStackTrace();
}
```

Notice that the JSON string now contains an array field called `owners` and a nested object field called `nestedObject`.

Regardless of whether you are accessing a field, array or nested object you use the `get()` method of the `JsonNode` class. By providing a string as parameter to the `get()` method you can access a field of a `JsonNode`. If the `JsonNode` represents an array, you need to pass an index to the `get()` method instead. The index specifies what element in the array you want to get.

This concludes this *Jackson ObjectMapper tutorial*. I hope you found this tutorial useful!

Next: Jackson JsonParser

Share Tweet

Jakob Jenkov