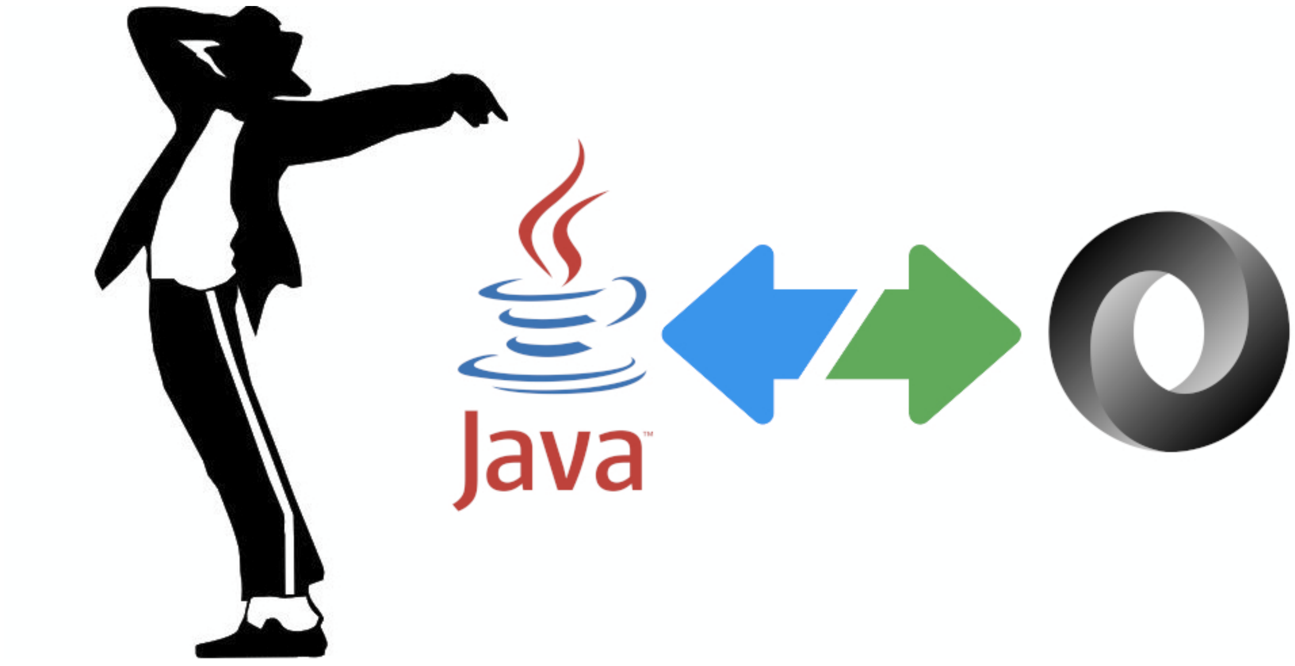
The Miners

Codeminer42's Engineering Blog



POSTS

Serialization and Deserialization in Java using Jackson

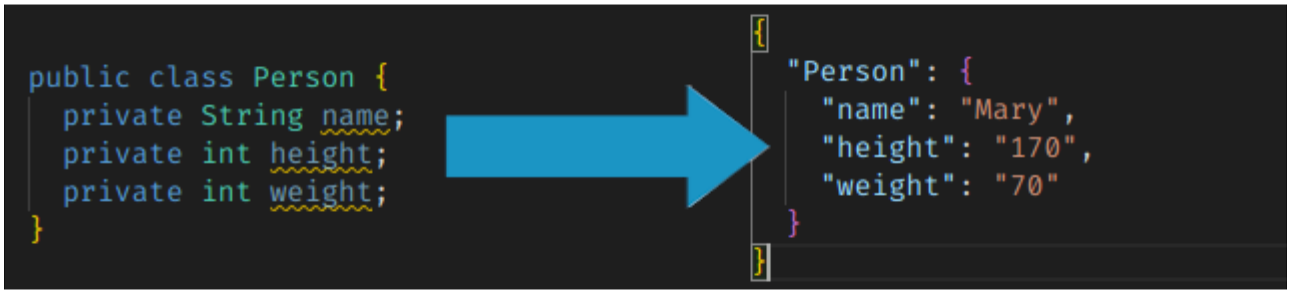
A practical guide on how to serialize and deserialize objects to JSON in Java using Jackson.

by Arthur Silva 13/07/2022

Before we start working with serialization and deserialization in Java we need to understand what these terms actually mean.

Serialization is the process of converting the state of an object to a byte stream in a way the byte stream can be reverted into a copy of the original object. Deserialization, as you may already have inferred, is the process of converting the serialized form of an object back into a copy of the original object.

For this post, we're gonna be serializing our objects to JSON using the Jackson library. So our serialized objects are going to look like this:



```
public class Person {
  private String name;
  private int height;
  private int weight;
}
```

```
{
  "Person": {
    "name": "Mary",
    "height": "170",
    "weight": "70"
  }
}
```

Now let's dive into how to use Jackson to serialize Java objects. It is very simple to serialize an object using Jackson, all you need to do is add it to your project using a dependency manager like Maven or add it manually.

Once we added it to our project, let's create two classes that we are going to use to instantiate our objects and test our serialization. They are going to look like this:

Person class:

```
 package com.artcruz.cm42.jackson_mapper;

import java.util.ArrayList;
import java.util.List;

public class Person {
  private String name;
  private int height;
  private int weight;
  private List<Car> cars = new ArrayList<Car>();

  public Person() {

  }
```

```java
public Person(String name, int height, int weight) {
  this.name = name;
  this.height = height;
  this.weight = weight;
}

public String getName() {
  return this.name;
}

public void setName(String name) {
  this.name = name;
}

public int getHeight() {
  return this.height;
}

public void setHeight(int height) {
  this.height = height;
}

public int getWeight() {
  return this.weight;
}

public void setWeight(int weight) {
  this.weight = weight;
}

public List<Car> getCars() {
  return this.cars;
}

public void setCars(List<Car> cars) {
  this.cars = cars;
}
```

```
  }
```

Car class:

```
 package com.artcruz.cm42.jackson_mapper;

public class Car {

  private String color;
  private Integer maxSpeed;
  private Integer weightCapacity;

  public Car() {
  }

  public Car(String color, Integer maxSpeed, Integer weightCapacity) {
    this.color = color;
    this.maxSpeed = maxSpeed;
    this.weightCapacity = weightCapacity;
  }

  public String getColor() {
    return this.color;
  }

  public void setColor(String color) {
    this.color = color;
  }

  public Integer getMaxSpeed() {
    return this.maxSpeed;
  }

  public void setMaxSpeed(Integer maxSpeed) {
    this.maxSpeed = maxSpeed;
  }

  public Integer getWeightCapacity() {
    return this.weightCapacity;
  }

  public void setWeightCapacity(Integer weightCapacity) {
    this.weightCapacity = weightCapacity;
  }

}
```

Now, to serialize our objects let's create instances and use Jackson's writeValueAsString method, as follows in the example below:

```
package com.artcruz.cm42.jackson_mapper;

import static org.junit.Assert.assertTrue;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import org.junit.Test;

public class AppTest {

  @Test
  public void shouldSerialize() throws JsonProcessingException {

    Person person = new Person("Peter", 180, 75);

    Car ferrari = new Car("Red", 400, 500);
    Car porsche = new Car("Green", 600, 350);

    person.getCars().add(ferrari);
```

```
        person.getCars().add(porsche);

        String jsonString = new ObjectMapper().writeValueAsString(person);

        System.out.println(jsonString);

        assertTrue(jsonString != null && !jsonString.isEmpty());

      }

    }
```

The output will be the following:

```
{
  "name": "Peter",
  "height": 180,
  "weight": 75,
  "cars": [
    { "color": "Red", "maxSpeed": 400, "weightCapacity": 500 },
    { "color": "Green", "maxSpeed": 600, "weightCapacity": 350 }
  ]
}
```

Now let’s take a look at deserialization. To deserialize our objects, let’s use the same output above and Jackson’s readValue method, as follows in the example below:

```
package com.artcruz.cm42.jackson_mapper;

import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertTrue;

 port com.fasterxml.jackson.core.JsonProcessingException;
```

```
import com.fasterxml.jackson.databind.JsonMappingException;
import com.fasterxml.jackson.databind.ObjectMapper;

import org.junit.Test;

public class AppTest {

 @Test
 public void shouldDeserialize() throws JsonMappingException, JsonProcessingExc

   String jsonString = "{\"name\":\"Peter\",\"height\":180,\"weight\":75,\"cars

   Person person = new ObjectMapper().readValue(jsonString, Person.class);

   System.out.println(person);

   assertNotNull(person);
   assertTrue(person.getCars().size() > 0);

 }

}
```

That's what serialization and deserialization with Jackson are. As you can see it is very simple to serialize and deserialize objects in Java using Jackson. Now let's take a look at some of Jackson's annotations.

@JsonProperty

The @JsonProperty annotation is used for mapping an attribute with a JSON key name in both serialization and deserialization. By default when we serialize an object, Jackson will map JSON keys matching the name of the object's attributes. To change this behavior we use @JsonProperty on the fields. We will pass a string value to the annotation to specify h name should be mapped to JSON.

Let’s see an example:

I mapped our name attribute to “fullname” and our cars attribute to “vehicles”:

```
public class Person {

 @JsonProperty("fullname")
 private String name;
 private int height;
 private int weight;

 @JsonProperty("vehicles")
 private List<Car> cars = new ArrayList<Car>();

  .
  .
  .
}
```

This is what our serialized object will look like now:

```
 {
   "height": 180,
   "weight": 75,
   "fullname": "Peter",
   "vehicles": [
     { "color": "Red", "maxSpeed": 400, "weightCapacity": 500 },
     { "color": "Green", "maxSpeed": 600, "weightCapacity": 350 }
   ]
}
```

[illegible] name attribute is now being mapped to the fullname JSON key, and the cars attribute [illegible]eing mapped to vehicles key.

To deserialize this object we must pass the JSON string with the attributes exactly as they are above. That is, “fullname” instead of “name” and “vehicles” instead of “cars”:

```
 @Test
public void shouldDeserialize() throws JsonMappingException, JsonProcessingExce

  String jsonString = "{\"fullname\":\"Peter\",\"height\":180,\"weight\":75,\"v

  Person person = new ObjectMapper().readValue(jsonString, Person.class);

  System.out.println(person);

  assertNotNull(person);
  assertTrue(person.getCars().size() > 0);

}
```

@JsonIgnore

@JsonIgnore is used at field level to mark a property or list of properties to be ignored.

Let’s see an example:

```
 public class Person {

  @JsonIgnore
  private String name;
  .
  .
  .
}
```

I annotated the name property with @JsonIgnore, now we are going to see that it will not appear in our JSON:

```
{
  "height": 180,
  "weight": 75,
  "cars": [
    { "color": "Red", "maxSpeed": 400, "weightCapacity": 500 },
    { "color": "Green", "maxSpeed": 600, "weightCapacity": 350 }
  ]
}
```

As you can see, the name property was ignored and doesn't appear in the serialized object anymore.

@JsonPropertyOrder

The @JsonPropertyOrder annotation is used to define an order for property serialization.

Let's see an example of its usage:

```
@JsonPropertyOrder(value = {"cars", "weight"}, alphabetic = true)
public class Person {
.
.
.
}
```

In the example above, our serialized object will be ordered with "cars" appearing first, then "weight" and the rest of the keys will be alphabetically ordered. The JSON keys order ›e the following:

1. cars
2. weight
3. height
4. name

@JsonRawValue

The @JsonRawValue annotation allows serializing a text without escaping or without any decoration.

Let’s see an example:

Let’s create an attribute called jobs and initialize it in the constructor with a JSON string, that as you can see, is escaped:

```
public class Person {

  private String jobs;

  private String name;
  private int height;
  private int weight;

  private List<Car> cars = new ArrayList<Car>();

  public Person() {

  }

  public Person(String name, int height, int weight) {
    this.name = name;
    this.height = height;
    this.weight = weight;
    jobs = "{ \"name\": [ \"Programmer\", \"Teacher\" ] }";
```

```
      }

      .
      .
      .

    }
```

If we serialize the object as it is, the output will be the following:

```
    {
      "jobs": "{ \"name\": [ \"Programmer\", \"Teacher\" ] }",
      "name": "Peter",
      "height": 180,
      "weight": 75,
      "cars": [
        { "color": "Red", "maxSpeed": 400, "weightCapacity": 500 },
        { "color": "Green", "maxSpeed": 600, "weightCapacity": 350 }
      ]
    }
```

As you can see, jobs value was outputted as an escaped string.

Now let’s annotate the jobs attribute with @JsonRawValue:

```
public class Person {

  @JsonRawValue
  private String jobs;
  .
  .
  .
}
```

Now the output will be the following:

```
{
  "jobs": { "name": ["Programmer", "Teacher"] },
  "name": "Peter",
  "height": 180,
  "weight": 75,
  "cars": [
    { "color": "Red", "maxSpeed": 400, "weightCapacity": 500 },
    { "color": "Green", "maxSpeed": 600, "weightCapacity": 350 }
  ]
}
```

Now the jobs attribute value is raw JSON.

@JsonTypeInfo and @JsonTypeName

@JsonTypeInfo is used to indicate details of type information that is to be included in serialization and deserialization. @JsonTypeName is used to set type names to be used for annotated class. Our purpose here with these two annotations is to show how to customize how our class name will appear in JSON after we serialize our object.

[illegible] see an example:

Our type has the name of the class by default, in this case, “person”. We annotate our class with @JsonTypeInfo and set the options to show the object type name, then we rename our type name to character.

```
 @JsonTypeInfo(include = As.WRAPPER_OBJECT, use = Id.NAME)
@JsonTypeName("character")
public class Person {
.
.
.
}
```

When we serialize our object, its name will be represented by “character”:

```
 {
   "character": {
     "name": "Peter",
     "height": 180,
     "weight": 75,
     "cars": [
       { "color": "Red", "maxSpeed": 400, "weightCapacity": 500 },
       { "color": "Green", "maxSpeed": 600, "weightCapacity": 350 }
     ]
   }
}
```

@JsonValue

When we annotate a method with @JsonValue, its content will be returned in the object serialization.

Let's see an example:

I created this method inside the Person class and annotated it with @JsonValue:

```
public class Person {
  .
  .
  .
  @JsonValue
  public Map<String, String> toJson() {
    Map<String, String> values = new HashMap<String, String>();

    values.put("name", "Peter Parker");

    return values;
  }
 .
 .
 .
}
```

When the serialization is run, the output will be the following:

```
{ "name": "Peter Parker" }
```

@JsonAnyGetter

@JsonAnyGetter allows a getter method to return Map which is then used to serialize the additional properties of JSON similarly to other properties.

Let’s see an example:

ated a method in the Person class called moreInfo and annotated it with
sonAnyGetter:

```
public class Person {
  .
  .
  .
  @JsonAnyGetter
  public Map<String, String> moreInfo() {
    Map<String, String> values = new HashMap<String, String>();

    values.put("last_name", "Parker");
    values.put("age", "26");

    return values;
  }
  .
  .
  .
}
```

Let’s see what will be our output:

```
{
 "name": "Peter",
 "height": 180,
 "weight": 75,
 "cars": [
   { "color": "Red", "maxSpeed": 400, "weightCapacity": 500 },
   { "color": "Green", "maxSpeed": 600, "weightCapacity": 350 }
 ],
 "last_name": "Parker",
 "age": "26"
}
```

ıu can see, the map properties were added to the serialized object.

@JsonAnySetter

@JsonAnySetter allows a setter method to use Map which is then used to deserialize the additional properties of JSON similarly to other properties.

In the person class, I created a map called info, its getter, and one method to add new info and annotated it with @JsonAnySetter:

```
public class Person {

  private Map<String, String> info = new HashMap<String, String>();
  .
  .
  .
  public Map<String, String> getInfo() {
    return info;
  }

  @JsonAnySetter
  public void addInfo(String s1, String s2) {
    info.put(s1, s2);
  }

  .
  .
  .
}
```

Now we can run our test and we will see that it passes:

```
package com.artcruz.cm42.jackson_mapper;

 nport static org.junit.Assert.assertEquals;
```

```
import static org.junit.Assert.assertNotNull;

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

import org.junit.Test;

public class AppTest {

  @Test
  public void shouldDeserialize() throws JsonMappingException, JsonProcessingEx

    String jsonString = "{\"nationality\":\"Brazilian\",\"age\":25,\"passion\":

    Person person = new ObjectMapper().readValue(jsonString, Person.class);

    System.out.println("abc");
    System.out.println(person.getInfo());

    assertNotNull(person);
    assertNotNull(person.getInfo().get("nationality"));
    assertNotNull(person.getInfo().get("age"));
    assertNotNull(person.getInfo().get("passion"));
    assertEquals(person.getInfo().get("nationality"), "Brazilian");
    assertEquals(person.getInfo().get("age"), "25");
    assertEquals(person.getInfo().get("passion"), "Java");

  }

}
```

And that's it. We saw what serialization is, and how to serialize and deserialize with Jackson and we learned the most useful annotations when we are starting to work with it.

erences:

https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html
https://www.baeldung.com/jackson-annotations
https://spring.io/blog/2014/12/02/latest-jackson-integration-improvements-in-spring
https://dzone.com/articles/jackson-annotations-for-json-part-4-general
https://www.tutorialspoint.com/jackson_annotations

> We want to work with you! Check out our "What We Do" page.

PREVIOUS POST
Redux Basics

NEXT POST
Adding a new technology or concept in your team (the correct way)

Arthur Silva

View all posts by Arthur Silva →

YOU MIGHT ALSO LIKE

Should I Migrate to Yarn?

19/05/2017

A microfrontend for non-propitious environments

09/02/2023

Cracking the Box Open with Module Factories

About the lesser-known Ruby Module Factories

19/12/2016

CONTENT TYPE

Beginner Intermediate Advanced

RECENT POSTS

Implementing high-performance multiple sort-rules query in SQL

Component-Driven UI Patterns Part ll

LocalStorage and Cookies under the hoodies

The reasons behind the “why”

ꞏminer42 Tour 2023/1: RailsConf & Reactathon

TAGS

Activerecord Android API C Clean Architecture Commit Components Crystal Lang CSS
Database Dependency Injection Design Patterns DevOps DI Docker Docker Compose Elixir
Erlang ES6 Flutter front-end Front End Development Functional Programming Git git-stash
JavaScript Node.js Nodejs Performance Programming Rails React Redux Refactoring
Ruby Ruby on Rails Software Architecture Software Development Testing Tutorial Typescript
Vue Vuejs Vuex Web Development