```
@JsonProperty
String firstName
```

```
@JsonCreator
public Portfolio
```

```
@JsonValue
public String toString()
```

```
{
  "title" : "Jackson Cookbook",
  "subtitle" : "JSON Recipes in Java",
  "author" : {
    "name" : {
      "first" : "Ted",
      "middle" : "M",
      "last" : "Young"
    },
    "twitterId" : "@jitterted",
    "website" : "http://about.me/tedmyoung"
  },
  "url" : "https://leanpub.com/jacksoncookbook",
  "language" : "English",
  "version" : "0.1"
}
```

```
ObjectMapper objectMapper = new ObjectMapper
objectMapper.writeValue(jsonWriter, portfol
```

The Jackson Cookbook

JSON Recipes in Java

Ted M. Young

This book is for sale at http://leanpub.com/jacksoncookbook

This version was published on 2013-09-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Ted M. Young by spreading the word about this book on Twitter!

The suggested hashtag for this book is #jacksoncookbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#jacksoncookbook

Contents

Release Notes

Version 0.3

Published on September 3, 2013.

- Added new chapter on Mix-Ins.
- Added content to the Configuring ObjectMapper chapter:
  - Renamed this chapter to be "Configuring JSON Formatting", since it now covers more than just globally configuring the ObjectMapper instance.
  - Changed the use of `objectMapper.configure()` to `.enable()` for configuring the `SerializationFeature.INDENT_OUTPUT`.
  - Added recipe for ordering properties.
  - Added ObjectMapper tip.

Version 0.2

Published on September 1, 2012.

- Updated the Configuring ObjectMapper chapter.
- Added chapter on Value Instantiators.

Version 0.1

The initial release of this book. Published on August 28, 2012.

- Has a chapter on Configuring ObjectMapper and two appendices.

Acknowledgments

Thanks to Tatu Saloranta for creating Jackson and being so responsive to questions on the Jackson forum. You can find information about him at his Cowtown Coder[1] blog.

Also, thanks to the LeanPub[2] folks (Scott, Peter, and Len) for being so responsive to requests and bug reports.

[1] http://cowtowncoder.com/author-cowtowncoder.html

[2] https://leanpub.com

Introduction

Jackson is a popular, high-performance JSON processor for Java. Jackson can support almost any JSON need that you might have, but that flexibility can make it complex to use. This book takes a problem and then shows, with lots of code and JSON examples, how to solve that problem using various Jackson features. Many of these problems were ones I encountered integrating JSON support with a "legacy" codebase, i.e., a codebase where I couldn't modify the code, using Data Binding, but I'll also cover some of the Streaming API that's used within custom serializers/deserializers. As a bonus, the Appendix contains a description of JSON tools that I've found useful along the way.

Problems Welcome!

If you're using Jackson, and have a pesky problem (or solution!), don't hesitate to post it in the book discussion group[3] and I may include it in the next release of this book.

Jackson Version:

Note that this book covers only the 2.x version of Jackson, not the earlier 1.x releases (2.x is the future!).

[3] https://groups.google.com/forum/#!forum/jackson-cookbook-discuss

Configuring JSON Formatting

This section covers customizing the serialization/deserialization of your object graph at a global level, i.e., per instance of `ObjectMapper`, as well as per-class or per-property using annotations.

Recipe: Make JSON Output More Readable

The `ObjectMapper` is the main class used to write out (serialize) an object graph as JSON. By default, the `ObjectMapper` (really the underlying `ObjectWriter`) will not put any extra whitespace in the generated JSON output. For example:

```
{"title":"The Jackson Cookbook","subtitle":"JSON Recipes in Java","author":
{"name":{"first":"Ted","middle":"M","last":"Young"},"twitterId":
"@jitterted","website":"http://about.me/tedmyoung"},"url":
"https://leanpub.com/jacksoncookbook","language":"English","version":"0.1"}
```

> **Note:**
>
> The actual JSON output doesn't have any newlines, the wrapping shown above is purely to fit this book's page margins.

To make the JSON easier to read, you can tell `ObjectMapper` to use a "pretty printer", which will insert spaces and newlines into the output (line 4 tells the mapper to use the default pretty printer using `writerWithDefaultPrettyPrinter()`):

```
1 ObjectMapper objectMapper = new ObjectMapper();
2 Book jacksonCookbook = new JacksonCookbook();
3 String json = objectMapper
4                 .writerWithDefaultPrettyPrinter()
5                 .writeValueAsString(jacksonCookbook);
6 System.out.println(json);
```

The result will be:

```
{
  "title" : "The Jackson Cookbook",
  "subtitle" : "JSON Recipes in Java",
  "author" : {
    "name" : {
      "first" : "Ted",
      "middle" : "M",
      "last" : "Young"
    },
    "twitterId" : "@jitterted",
    "website" : "http://about.me/tedmyoung"
  },
  "url" : "https://leanpub.com/jacksoncookbook",
  "language" : "English",
  "version" : "0.1"
}
```

Globally use Pretty Printer

If you always want to pretty-print the output, you can configure the `ObjectMapper` instance directly using the `SerializationFeature` class:

```
ObjectMapper objectMapper = new ObjectMapper();
objectMapper.enable(SerializationFeature.INDENT_OUTPUT);
```

The `enable` method here turns on the `INDENT_OUTPUT` *feature*, which tells Jackson to use the `DefaultPrettyPrinter` when serializing your objects. The code

```
objectMapper.configure(SerializationFeature.INDENT_OUTPUT, true)
```

does the same thing as using `enable`, but `enable` is easier to read than trying to interpret what the boolean `true` value means.

Custom Pretty Printer

If you don't like the way the default pretty printer works, e.g., you don't want a space before the colon, only after (which is a common alternative format), then you'll need to create a custom implementation of the `PrettyPrinter` interface. For a minor modification like this, you can subclass the `DefaultPrettyPrinter` class and override the method that writes out the name and value fields as follows:

```
1  package com.jitterpig.jacksoncookbook;
2
3  import com.fasterxml.jackson.core.util.DefaultPrettyPrinter;
4
5  public class ModifiedPrettyPrinter extends DefaultPrettyPrinter {
6    @Override
7    public void writeObjectFieldValueSeparator(JsonGenerator jg)
8                                                 throws IOException {
9      if (_spacesInObjectEntries) {
10       jg.writeRaw(": ");
11     } else {
12       jg.writeRaw(':');
13     }
14   }
15 }
```

The code above was copied from the `DefaultPrettyPrinter`'s implementation of this method, with the only change in line 10 where I removed the space before the colon.

To use this, pass an instance of the pretty printer to the mapper: line 4 below is using an instance of my `ModifiedPrettyPrinter`.

```
1 PrettyPrinter prettyPrinter = new ModifiedPrettyPrinter()
2 Book jacksonCookbook = new JacksonCookbook();
3 String json = objectMapper
4                 .writer(prettyPrinter)
5                 .writeValueAsString(jacksonCookbook);
6 System.out.println(json);
```

The output from the `ModifiedPrettyPrinter` looks like this:

```
{
  "title": "The Jackson Cookbook",
  "subtitle": "JSON Recipes in Java",
  "author": {
    "name": {
      "first": "Ted",
      "middle": "M",
      "last": "Young"
    },
    "twitterId": "@jitterted",
    "website": "http://about.me/tedmyoung"
  },
  "url": "https://leanpub.com/jacksoncookbook",
  "language": "English",
  "version": "0.1"
}
```

You can also subclass the compact or minimal pretty printer (i.e., the one that's used if you don't specify otherwise) by extending the `MinimalPrettyPrinter` class. If you want to make more substantial changes to how the JSON is formatted, you may want to implement the `PrettyPrinter` interface directly. However, I haven't come across the need to do that.

Recipe: Suppress `null` values globally

If you have property values in your data that can be `null`, but don't want them serialized when they are, you can do the following to make the setting global for all uses of this `objectMapper` reference:

```
objectMapper.setSerializationInclusion(JsonInclude.Include.NON_NULL);
```

If you only want to do this on a per-class basis, you can add the `@JsonInclude` annotation above the class or interface definition:

```
@JsonSerialize(include = Inclusion.NON_NULL)
public class HasNullValues {
  private String optionalSecondAddress; // nullable field
  ...
```

Options for this value are:

- `ALWAYS` - the default, which means all properties are serialized. This is the default.
- `NON_NULL` - as used above: only non-null properties are serialized.
- `NON_DEFAULT` - only serializes properties if they differ from the defaults for that object. Defaults are the values of properties when instantiated using the no-argument constructor. Note that this applies to arrays as well.
  - This is useful for saving bandwidth: if the defaults are the same on both sides of the serialization, there's no point in serializing that information.
- `NON_EMPTY` - only serializes properties that are not "empty", where empty includes `null` as well as:
  - **Arrays**: `length == 0`
  - **Collections**: `isEmpty()` is `true` (e.g., `Map`, `List`, etc.)
  - **Date**: the time stamp value is `0`
  - **String**: `length == 0`

Recipe: Ordering properties in JSON output

Sometimes you'll want the properties that in your JSON output to be in a specific order, perhaps because you have tests or clients that rely on the ordering (I'd try to avoid that, but sometimes it's not under your control), or you want it to be easier to find a certain property when looking at the JSON during debugging.

Ordering per-class

To order on a per-class basis, use the `@JsonPropertyOrder` annotation:

```
@JsonPropertyOrder({"id", "symbol"})
public class Currency {
  public String symbol;
  public String shortName;
  public String longName;
  public long id;
}
```

This will ensure that the `id` property will be first, followed by the `symbol` property, with the rest in an undetermined order. For example:

```
{
  "id" : 1239129,
  "symbol" : "$",
  "shortName" : "USD",
  "longName" : "U.S. Dollar"
}
```

If you wanted them simply in alphabetic order, you would use the `alphabetic` parameter in the annotation:

```
@JsonPropertyOrder(alphabetic = true)
public class Currency {
...
```

Results in:

```
{
  "id" : 1239129,
  "longName" : "U.S. Dollar",
  "shortName" : "USD",
  "symbol" : "$"
}
```

These two ordering parameters can be combined, so you can ensure some properties are first, with the rest alphabetically ordered:

```
@JsonPropertyOrder(value = {"id", "symbol"}, alphabetic = true)
public class Currency {
...
```

With the result:

```
{
  "id" : 1239129,
  "symbol" : "$",
  "longName" : "U.S. Dollar",
  "shortName" : "USD"
}
```

Globally ordering

If you always want the properties sorted alphabetically, configure the `ObjectMapper` instance directly:

```
objectMapper.enable(MapperFeature.SORT_PROPERTIES_ALPHABETICALLY);
```

Ordering keys in maps

Note that the above settings *won't* affect the order of the contents of any maps that you have (e.g., a `HashMap`). If you want the output of maps to be ordered by their keys, you'll need to turn on the `ORDER_MAP_ENTRIES_BY_KEYS` feature:

```
objectMapper.enable(SerializationFeature.ORDER_MAP_ENTRIES_BY_KEYS);
```

General `ObjectMapper` tip

> `ObjectMapper` is just an easy way to get access to `ObjectReaders` and `ObjectWriters`.

Quoting from Tatu (the creator of Jackson):

> "Any direct read/write methods in mapper are just for convenience.
>
> …
>
> ObjectWriter and ObjectReader are fully immutable and thread-safe; and they add minor optimizations for cases where types are known (i.e. they can pre-load root serializer/deserializer to use, just once, instead of per-call). So for most part I recommend trying to move to using ObjectReaders and ObjectWriters more, if possible."

Mix-Ins

This chapter will cover the way Mix-Ins can be used to solve certain problems using Jackson with legacy or third-party code.

Value Instantiators

This chapter will cover how to use Value Instantiators for instance creation when `@JsonCreator` *isn't enough.*

Handling Polymorphic Types

This chapter will cover how to handle deserializing JSON that contains information about the object's type in the JSON itself.

Appendix: JSON Information

JSON Defined

The JSON "standard" is defined at http://www.json.org. The syntax is clearly defined and there are links to JSON libraries in lots of different languages.

JSON Schema

JSON Schema is a proposed Internet draft defining a JSON media type (`application/schema+json`) with the following goals: Validation, Documentation, and Hyperlinking. For details, go to the web site at http://json-schema.org.

An example of a JSON schema for an Address is:

```
{
  "description": "An Address according to http://microformats.org/wiki/hcard",
  "type": "object",
  "properties": {
    "post-office-box": { "type": "string" },
    "extended-address": { "type": "string" },
    "street-address": { "type": "string" },
    "locality":{ "type": "string", "required": true },
    "region": { "type": "string", "required": true },
    "postal-code": { "type": "string" },
    "country-name": { "type": "string", "required": true}
  },
  "dependencies": {
    "post-office-box": "street-address",
    "extended-address": "street-address"
  }
}
```

Appendix: JSON Tools

JSON Lint

This site[4] checks that your JSON is valid. If not, it points out where the problem is. Note that it can only tell you about the **next** error, not all of the possible errors.

What happens if you forget an opening curly brace

[4] http://jsonlint.org/

JSON Editor Online

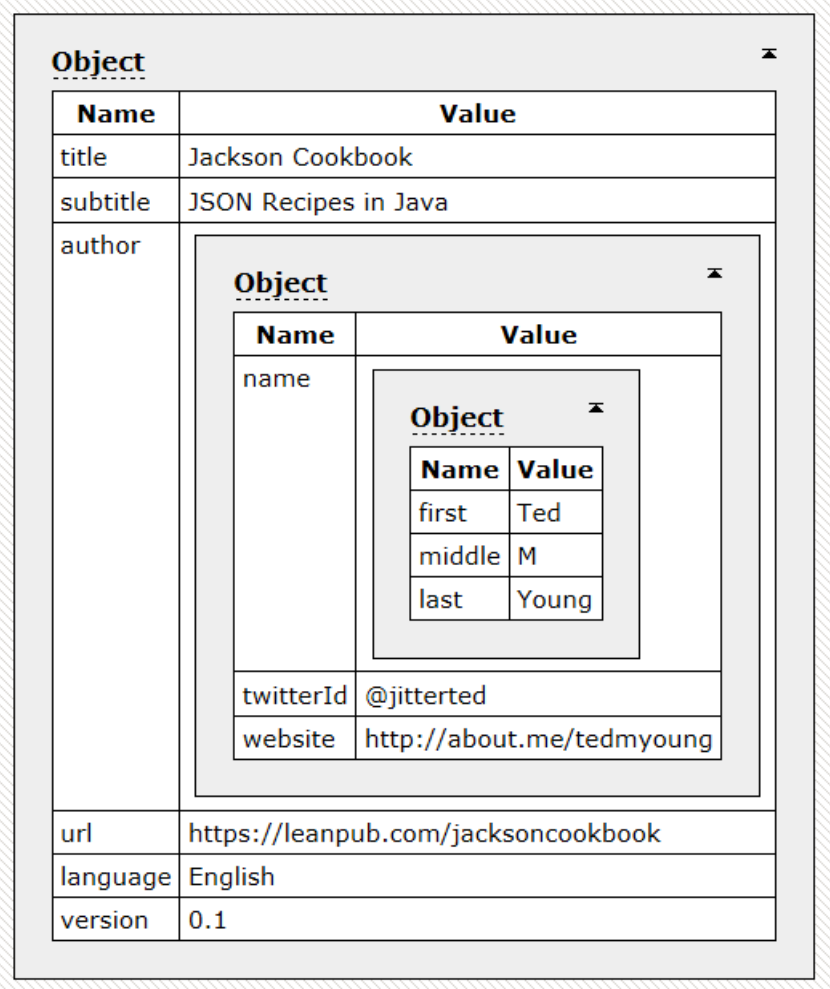
This site hosts a hierarchical JSON editor[5].

I used this to create the JSON for this book's cover

[5] http://jsoneditoronline.org

JSON2HTML

This site[6] will render your JSON as text in nested boxes. A nice way to view your JSON data.



Some JSON rendered by JSON2HTML

[6] http://json.bloople.net

Appendix: Java JSON Tools

JSON Schema to Java POJO

The idea of a schema for JSON is not new, but there are now tools to generate Java code from those schemas. One such tool is jsonschema2pojo[7].

[7] http://code.google.com/p/jsonschema2pojo/