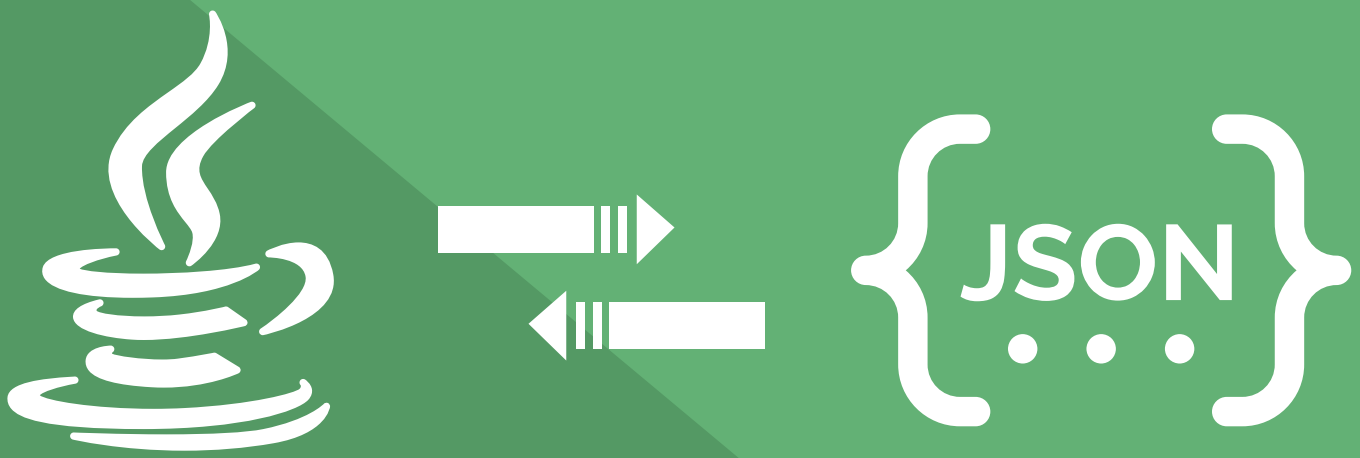
Baeldung

Do JSON with Jackson



1: Jackson Annotation Examples

2: Intro to the Jackson ObjectMapper

3: Jackson Ignore Properties on Marshalling

4: Ignore Null Fields with Jackson

5: Jackson – Change Name of Field

6: Jackson Unmarshalling JSON with Unknown Properties

1: Jackson Annotation Examples

1. Overview

In this chapter, we'll do a deep dive into Jackson Annotations.
We'll see how to use the existing annotations, how to create custom ones and finally – how to disable them.

2. Maven Dependencies

Let's first add the jackson-databind dependency to the pom.xml:

```
1. <dependency>
2.     <groupId>com.fasterxml.jackson.core</groupId>
3.     <artifactId>jackson-databind</artifactId>
4.     <version>2.9.8</version>
5. </dependency>
```

This dependency will also transitively add the following libraries to the classpath:

1. jackson-annotations-2.9.8.jar
2. jackson-core-2.9.8.jar
3. jackson-databind-2.9.8.jar

First, we'll take a look at the serialization annotations.

3.1. @*JsonAnyGetter*

The @JsonAnyGetter annotation allows the flexibility of using a *Map* field as standard properties.

Here's a quick example – the *ExtendableBean* entity has the *name* property and a set of extendable attributes in the form of key/value pairs:

```
1. public class ExtendableBean {
2.     public String name;
3.     private Map<String, String> properties;
4.
5.     @JsonAnyGetter
6.     public Map<String, String> getProperties() {
7.         return properties;
8.     }
9. }
```

When we serialize an instance of this entity, we get all the key-values in the *Map* as standard, plain properties:

```
1. {
2.     "name":"My bean",
3.     "attr2":"val2",
4.     "attr1":"val1"
5. }
```

And here how the serialization of this entity looks like in practice:

```
@Test
public void whenSerializingUsingJsonAnyGetter_thenCorrect()
  throws JsonProcessingException {

    ExtendableBean bean = new ExtendableBean("My bean");
    bean.add("attr1", "val1");
    bean.add("attr2", "val2");

    String result = new ObjectMapper().writeValueAsString(bean);

    assertThat(result, containsString("attr1"));
    assertThat(result, containsString("val1"));
}
```

We can also use optional argument *enabled* as *false* to disable @*JsonAnyGetter()*. In this case, the *Map* will be converted as JSON and will appear under *properties* variable after serialization.

3.2. @*JsonGetter*

The @*JsonGetter* annotation is an alternative to the @*JsonProperty* annotation to mark a method as a getter method.

In the following example – we specify the method *getTheName()* as the getter method of *name* property of *MyBeanentity*:

```
public class MyBean {
    public int id;
    private String name;

    @JsonGetter("name")
    public String getTheName() {
        return name;
    }
}
```

And here's how this works in practice:

```
@Test
public void whenSerializingUsingJsonGetter_thenCorrect()
  throws JsonProcessingException {

    MyBean bean = new MyBean(1, "My bean");

    String result = new ObjectMapper().writeValueAsString(bean);

    assertThat(result, containsString("My bean"));
    assertThat(result, containsString("1"));
}
```

3.3. @*JsonPropertyOrder*

We can use the @*JsonPropertyOrder* annotation to specify the **order of properties on serialization.**

Let's set a custom order for the properties of a *MyBean* entity:

```
@JsonPropertyOrder({ "name", "id" })
public class MyBean {
    public int id;
    public String name;
}
```

And here is the output of serialization:

```
{
    "name":"My bean",
    "id":1
}
```

And a simple test:

```
@Test
public void whenSerializingUsingJsonPropertyOrder_thenCorrect()
  throws JsonProcessingException {

    MyBean bean = new MyBean(1, “My bean”);

    String result = new ObjectMapper().writeValueAsString(bean);
    assertThat(result, containsString(“My bean”));
    assertThat(result, containsString(“1”));
}
```

We can also use *@JsonPropertyOrder(alphabetic=true)* to order the properties alphabetically. And in that case the output of serialization will be:

```
{
    “id”:1,
    “name”:”My bean”
}
```

3.4. *@JsonRawValue*

The *@JsonRawValue* annotation can **instruct Jackson to serialize a property exactly as is.**

In the following example, we use *@JsonRawValue* to embed some custom JSON as a value of an entity:

```
public class RawBean {
    public String name;

    @JsonRawValue
    public String json;
}
```

The output of serializing the entity is:

```
{
    “name”:”My bean”,
    “json”:{
        “attr”:false
    }
}
```

And a simple test:

```
@Test
public void whenSerializingUsingJsonRawValue_thenCorrect()
  throws JsonProcessingException {

    RawBean bean = new RawBean(“My bean”, “{“attr”:false}”);

    String result = new ObjectMapper().writeValueAsString(bean);
    assertThat(result, containsString(“My bean”));
    assertThat(result, containsString(“{“attr”:false}”));
}
```

We can also use the optional boolean argument *value* that defines whether this annotation is active or not.

3.5. *@JsonValue*

@JsonValue indicates a single method that the library will use to serialize the entire instance.

For example, in an enum, we annotate the *getName* with *@JsonValue* so that any such entity is serialized via its name:

```
public enum TypeEnumWithValue {
    TYPE1(1, "Type A"), TYPE2(2, "Type 2");

    private Integer id;
    private String name;

    // standard constructors

    @JsonValue
    public String getName() {
        return name;
    }
}
```

Our test:

```
@Test
public void whenSerializingUsingJsonValue_thenCorrect()
  throws JsonParseException, IOException {

    String enumAsString = new ObjectMapper()
      .writeValueAsString(TypeEnumWithValue.TYPE1);

    assertThat(enumAsString, is(""Type A""));
}
```

3.6. *@JsonRootName*

The *@JsonRootName* annotation is used – if wrapping is enabled – to specify the name of the root wrapper to be used.

Wrapping means that instead of serializing a *User* to something like:

```
{
    "id": 1,
    "name": "John"
}
```

It's going to be wrapped like this:

```
{
    "User": {
        "id": 1,
        "name": "John"
    }
}
```

So, let's look at an example – **we're going to use the *@JsonRootName* annotation to indicate the name of this potential wrapper entity:**

```
@JsonRootName(value = "user")
public class UserWithRoot {
    public int id;
    public String name;
}
```

By default, the name of the wrapper would be the name of the class – *UserWithRoot*. By using the annotation, we get the cleaner-looking *user*:

```
@Test
public void whenSerializingUsingJsonRootName_thenCorrect()
  throws JsonProcessingException {

    UserWithRoot user = new User(1, "John");

    ObjectMapper mapper = new ObjectMapper();
    mapper.enable(SerializationFeature.WRAP_ROOT_VALUE);
    String result = mapper.writeValueAsString(user);

    assertThat(result, containsString("John"));
    assertThat(result, containsString("user"));
}
```

Here is the output of serialization:

```
{
    "user":{
        "id":1,
        "name":"John"
    }
}
```

Since Jackson 2.4, a new optional argument *namespace* is available to use with data formats such as XML. If we add it, it will become part of the fully qualified name:

```
@JsonRootName(value = "user", namespace="users")
public class UserWithRootNamespace {
    public int id;
    public String name;

    // ...
}
```

If we serialize it with *XmlMapper* the output will be:

```
<user xmlns="users">
    <id xmlns="">1</id>
    <name xmlns="">John</name>
    <items xmlns=""/>
</user>
```

3.7. *@JsonSerialize*

***@JsonSerialize* indicates a custom serializer to use when marshalling the entity.**

Let's look at a quick example. We're going to use *@JsonSerialize* to serialize the *eventDate* property with a *CustomDateSerializer*:

```
1. public class Event {
2.     public String name;
3.
4.     @JsonSerialize(using = CustomDateSerializer.class)
5.     public Date eventDate;
6. }
```

Here's the simple custom Jackson serializer:

```
1. public class CustomDateSerializer extends StdSerializer<Date> {
2.
3.     private static SimpleDateFormat formatter
4.       = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
5.
6.     public CustomDateSerializer() {
7.         this(null);
8.     }
9.
10.    public CustomDateSerializer(Class<Date> t) {
11.        super(t);
12.    }
13.
14.    @Override
15.    public void serialize(
16.      Date value, JsonGenerator gen, SerializerProvider arg2)
17.      throws IOException, JsonProcessingException {
18.        gen.writeString(formatter.format(value));
19.    }
20. }
```

Let's use these in a test:

```
@Test
public void whenSerializingUsingJsonSerialize_thenCorrect()
  throws JsonProcessingException, ParseException {

    SimpleDateFormat df
      = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");

    String toParse = "20-12-2014 02:30:00";
    Date date = df.parse(toParse);
    Event event = new Event("party", date);

    String result = new ObjectMapper().writeValueAsString(event);
    assertThat(result, containsString(toParse));
}
```

Next – let's explore the Jackson deserialization annotations.

4.1. @JsonCreator

We can use the *@JsonCreator* annotation to tune the constructor/factory used in deserialization.

It's very helpful when we need to deserialize some JSON that doesn't exactly match the target entity we need to get.

Let's look at an example; say we need to deserialize the following JSON:

```
{
    "id":1,
    "theName":"My bean"
}
```

However, there is no *theName* field in our target entity – there is only a *name* field. Now, we don't want to change the entity itself – we just need a little more control over the unmarshalling process – by annotating the constructor with *@JsonCreator* and using the *@JsonProperty* annotation as well:

```
public class BeanWithCreator {
    public int id;
    public String name;

    @JsonCreator
    public BeanWithCreator(
      @JsonProperty("id") int id,
      @JsonProperty("theName") String name) {
        this.id = id;
        this.name = name;
    }
}
```

Let's see this in action:

```
@Test
public void whenDeserializingUsingJsonCreator_thenCorrect()
  throws IOException {
    String json = "{\"id\":1,\"theName\":\"My bean\"}";
    BeanWithCreator bean = new ObjectMapper()
      .readerFor(BeanWithCreator.class)
      .readValue(json);
    assertEquals("My bean", bean.name);
}
```

4.2. *@JacksonInject*

***@JacksonInject* indicates that a property will get its value from the injection and not from the JSON data.**

In the following example – we use *@JacksonInject* to inject the property *id*:

```
public class BeanWithInject {
    @JacksonInject
    public int id;

    public String name;
}
```

And here's how this works:

```
@Test
public void whenDeserializingUsingJsonInject_thenCorrect()
  throws IOException {
    String json = "{\"name\":\"My bean\"}";
    InjectableValues inject = new InjectableValues.Std()
      .addValue(int.class, 1);
    BeanWithInject bean = new ObjectMapper().reader(inject)
      .forType(BeanWithInject.class)
      .readValue(json);
    assertEquals("My bean", bean.name);
    assertEquals(1, bean.id);
}
```

4.3. *@JsonAnySetter*

@JsonAnySetter allows us the flexibility of using a *Map* as standard properties. On deserialization, the properties from JSON will simply be added to the map.

Let's see how this works – we'll use *@JsonAnySetter* to deserialize the entity *ExtendableBean:*

```
1. public class ExtendableBean {
2.     public String name;
3.     private Map<String, String> properties;
4.     @JsonAnySetter
5.     public void add(String key, String value) {
6.         properties.put(key, value);
7.     }
8. }
```

This is the JSON we need to deserialize:

```
1. {
2.     "name":"My bean",
3.     "attr2":"val2",
4.     "attr1":"val1"
5. }
```

And here's how this all ties in together:

```
1. @Test
2. public void whenDeserializingUsingJsonAnySetter_thenCorrect()
3.   throws IOException {
4.     String json
5.       = "{\"name\":\"My
6. bean\",\"attr2\":\"val2\",\"attr1\":\"val1\"}";
7.     ExtendableBean bean = new ObjectMapper()
8.       .readerFor(ExtendableBean.class)
9.       .readValue(json);
10.    assertEquals("My bean", bean.name);
11.    assertEquals("val2", bean.getProperties().get("attr2"));
12. }
```

4.4. *@JsonSetter*

@JsonSetter is an alternative to *@JsonProperty* – that marks the method as a setter method.

This is super useful when we need to read some JSON data but **the target entity class doesn't exactly match that data**, and so we need to tune the process to make it fit.

In the following example, we'll specify the method *setTheName()* as the setter of the *name* property in our *MyBeanentity*:

```
1. public class MyBean {
2.     public int id;
3.     private String name;
4.
5.     @JsonSetter("name")
6.     public void setTheName(String name) {
7.         this.name = name;
8.     }
9. }
```

Now, when we need to unmarshall some JSON data – this works perfectly well:

```
1. @Test
2. public void whenDeserializingUsingJsonSetter_thenCorrect()
3.   throws IOException {
4.
5.     String json = "{\"id\":1,\"name\":\"My bean\"}";
6.
7.     MyBean bean = new ObjectMapper()
8.       .readerFor(MyBean.class)
9.       .readValue(json);
10.    assertEquals("My bean", bean.getTheName());
11. }
```

4.5. *@JsonDeserialize*

***@JsonDeserialize* indicates the use of a custom deserializer.**

Let's see how that plays out – we'll use *@JsonDeserialize* to deserialize the *eventDate* property with the *CustomDateDeserializer*:

```
1.  public class Event {
2.      public String name;
3.
4.      @JsonDeserialize(using = CustomDateDeserializer.class)
5.      public Date eventDate;
6.  }
```

Here's the custom deserializer:

```
1.  public class CustomDateDeserializer
2.    extends StdDeserializer<Date> {
3.
4.      private static SimpleDateFormat formatter
5.        = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
6.
7.      public CustomDateDeserializer() {
8.          this(null);
9.      }
10.
11.     public CustomDateDeserializer(Class<?> vc) {
12.         super(vc);
13.     }
14.
15.     @Override
16.     public Date deserialize(
17.       JsonParser jsonparser, DeserializationContext context)
18.       throws IOException {
19.
20.         String date = jsonparser.getText();
21.         try {
22.             return formatter.parse(date);
23.         } catch (ParseException e) {
24.             throw new RuntimeException(e);
25.         }
26.     }
27. }
```

And here's the back-to-back test:

```
@Test
public void whenDeserializingUsingJsonDeserialize_thenCorrect()
  throws IOException {
    String json
      = "{"name":"party","eventDate":"20-12-2014 02:30:00"}";

    SimpleDateFormat df
      = new SimpleDateFormat("dd-MM-yyyy hh:mm:ss");
    Event event = new ObjectMapper()
      .readerFor(Event.class)
      .readValue(json);
    assertEquals(
      "20-12-2014 02:30:00", df.format(event.eventDate));
}
```

4.6. *@JsonAlias*

The *@JsonAlias* defines **one or more alternative names for a property during deserialization.** Let's see how this annotation works with a quick example:

```
public class AliasBean {
    @JsonAlias({ "fName", "f_name" })
    private String firstName;
    private String lastName;
}
```

Here, we have a POJO and we want to deserialize JSON with values such as *fName*, *f_name*, and *firstName* into the *firstName* variable of the POJO.
And here is a test making sure this annotation works as expected :

```
@Test
public void whenDeserializingUsingJsonAlias_thenCorrect() throws
IOException {
    String json = "{\"fName\": \"John\", \"lastName\":
\"Green\"}";
    AliasBean aliasBean = new ObjectMapper().readerFor(AliasBean.
class).readValue(json);
    assertEquals("John", aliasBean.getFirstName());
}
```

Next – let's take a look at Jackson polymorphic type handling annotations:

- *@JsonTypeInfo* – indicates details of what type information to include in serialization
- *@JsonSubTypes* – indicates sub-types of the annotated type
- *@JsonTypeName* – defines a logical type name to use for annotated class

Let's look at a more complex example and use all three – *@JsonTypeInfo*, *@JsonSubTypes*, and *@JsonTypeName* – to serialize/deserialize the entity *Zoo*:

```
1.  public class Zoo {
2.      public Animal animal;
3.
4.      @JsonTypeInfo(
5.        use = JsonTypeInfo.Id.NAME,
6.        include = As.PROPERTY,
7.        property = "type")
8.      @JsonSubTypes({
9.          @JsonSubTypes.Type(value = Dog.class, name = "dog"),
10.         @JsonSubTypes.Type(value = Cat.class, name = "cat")
11.     })
12.     public static class Animal {
13.         public String name;
14.     }
15.
16.     @JsonTypeName("dog")
17.     public static class Dog extends Animal {
18.         public double barkVolume;
19.     }
20.
21.     @JsonTypeName("cat")
22.     public static class Cat extends Animal {
23.         boolean likesCream;
24.         public int lives;
25.     }
26. }
```

When we do serialization:

```
@Test
public void whenSerializingPolymorphic_thenCorrect()
  throws JsonProcessingException {
    Zoo.Dog dog = new Zoo.Dog("lacy");
    Zoo zoo = new Zoo(dog);

    String result = new ObjectMapper()
      .writeValueAsString(zoo);

    assertThat(result, containsString("type"));
    assertThat(result, containsString("dog"));
}
```

Here's what serializing the *Zoo* instance with the *Dog* will result in:

```
{
    "animal": {
        "type": "dog",
        "name": "lacy",
        "barkVolume": 0
    }
}
```

Now for de-serialization – let's start with the following JSON input:

```
{
    "animal":{
        "name":"lacy",
        "type":"cat"
    }
}
```

And let's see how that gets unmarshalled to a Zoo instance:

```
@Test
public void whenDeserializingPolymorphic_thenCorrect()
throws IOException {
    String json =
"{\"animal\":{\"name\":\"lacy\",\"type\":\"cat\"}}";

    Zoo zoo = new ObjectMapper()
      .readerFor(Zoo.class)
      .readValue(json);

    assertEquals("lacy", zoo.animal.name);
    assertEquals(Zoo.Cat.class, zoo.animal.getClass());
}
```

Next – let's discuss some of Jackson more general annotations.

6.1. *@JsonProperty*

We can add **the *@JsonProperty* annotation to indicate the property name in JSON.**

Let's use *@JsonProperty* to serialize/deserialize the property *name* when we're dealing with non-standard getters and setters:

```
1.  public class MyBean {
2.      public int id;
3.      private String name;
4.      @JsonProperty("name")
5.      public void setTheName(String name) {
6.          this.name = name;
7.      }
8.      @JsonProperty("name")
9.      public String getTheName() {
10.         return name;
11.     }
12. }
```

Our test:

```
1.  @Test
2.  public void whenUsingJsonProperty_thenCorrect()
3.    throws IOException {
4.      MyBean bean = new MyBean(1, "My bean");
5.      String result = new ObjectMapper().writeValueAsString(bean);
6.
7.      assertThat(result, containsString("My bean"));
8.      assertThat(result, containsString("1"));
9.
10.     MyBean resultBean = new ObjectMapper()
11.       .readerFor(MyBean.class)
12.       .readValue(result);
13.     assertEquals("My bean", resultBean.getTheName());
14. }
```

6.2. @JsonFormat

The *@JsonFormat* annotation specifies a format when serializing Date/Time values.

In the following example – we use *@JsonFormat* to control the format of the property *eventDate*:

```
1.  public class Event {
2.      public String name;
3.
4.      @JsonFormat(
5.        shape = JsonFormat.Shape.STRING,
6.        pattern = "dd-MM-yyyy hh:mm:ss")
7.      public Date eventDate;
8.  }
```

And here's the test:

```
1.  @Test
2.  public void whenSerializingUsingJsonFormat_thenCorrect()
3.    throws JsonProcessingException, ParseException {
4.      SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy
5.  hh:mm:ss");
6.      df.setTimeZone(TimeZone.getTimeZone("UTC"));
7.
8.      String toParse = "20-12-2014 02:30:00";
9.      Date date = df.parse(toParse);
10.     Event event = new Event("party", date);
11.
12.     String result = new ObjectMapper().writeValueAsString(event);
13.
14.     assertThat(result, containsString(toParse));
15. }
```

6.3. *@JsonUnwrapped*

@JsonUnwrapped **defines values that should be unwrapped/flattened when serialized/deserialized.**

Let's see exactly how that works; we'll use the annotation to unwrap the property *name*:

```
1.  public class UnwrappedUser {
2.      public int id;
3.
4.      @JsonUnwrapped
5.      public Name name;
6.
7.      public static class Name {
8.          public String firstName;
9.          public String lastName;
10.     }
11. }
```

Let's now serialize an instance of this class:

```
1.  @Test
2.  public void whenSerializingUsingJsonUnwrapped_thenCorrect()
3.    throws JsonProcessingException, ParseException {
4.      UnwrappedUser.Name name = new UnwrappedUser.Name("John", "Doe");
5.      UnwrappedUser user = new UnwrappedUser(1, name);
6.
7.      String result = new ObjectMapper().writeValueAsString(user);
8.
9.      assertThat(result, containsString("John"));
10.     assertThat(result, not(containsString("name")));
11. }
```

Here's how the output looks like – the fields of the static nested class unwrapped along with the other field:

```
1.  {
2.      "id":1,
3.      "firstName":"John",
4.      "lastName":"Doe"
5.  }
```

6.4. *@JsonView*

***@JsonView* indicates the View in which the property will be included for serialization/deserialization.**

An example will show exactly how that works – we'll use *@JsonView* to serialize an instance of *Item* entity. Let's start with the views:

```
1. public class Views {
2.     public static class Public {}
3.     public static class Internal extends Public {}
4. }
```

And now here's the *Item* entity, using the views:

```
1. public class Item {
2.     @JsonView(Views.Public.class)
3.     public int id;
4.
5.     @JsonView(Views.Public.class)
6.     public String itemName;
7.
8.     @JsonView(Views.Internal.class)
9.     public String ownerName;
10. }
```

Finally – the full test:

```
1. @Test
2. public void whenSerializingUsingJsonView_thenCorrect()
3.   throws JsonProcessingException {
4.     Item item = new Item(2, "book", "John");
5.
6.     String result = new ObjectMapper()
7.       .writerWithView(Views.Public.class)
8.       .writeValueAsString(item);
9.
10.    assertThat(result, containsString("book"));
11.    assertThat(result, containsString("2"));
12.    assertThat(result, not(containsString("John")));
13. }
```

6.5. *@JsonManagedReference, @JsonBackReference*

The *@JsonManagedReference* and *@JsonBackReference* annotations can handle parent/child relationships and work around loops.

In the following example – we use *@JsonManagedReference* and *@JsonBackReference* to serialize our *ItemWithRefentity*:

```
1.  public class ItemWithRef {
2.      public int id;
3.      public String itemName;
4.
5.      @JsonManagedReference
6.      public UserWithRef owner;
7.  }
```

Our *UserWithRef* entity:

```
1.  public class UserWithRef {
2.      public int id;
3.      public String name;
4.
5.      @JsonBackReference
6.      public List<ItemWithRef> userItems;
7.  }
```

And the test:

```
1.  @Test
2.  public void whenSerializingUsingJacksonReferenceAnnotation_
3.  thenCorrect()
4.    throws JsonProcessingException {
5.      UserWithRef user = new UserWithRef(1, “John”);
6.      ItemWithRef item = new ItemWithRef(2, “book”, user);
7.      user.addItem(item);
8.      String result = new ObjectMapper().writeValueAsString(item);
9.
10.     assertThat(result, containsString(“book”));
11.     assertThat(result, containsString(“John”));
12.     assertThat(result, not(containsString(“userItems”)));
13. }
```

6.6. *@JsonIdentityInfo*

@JsonIdentityInfo indicates that Object Identity should be used when serializing/deserializing values – for instance, to deal with infinite recursion type of problems.

In the following example – we have an *ItemWithIdentity* entity with a bidirectional relationship with the *UserWithIdentity* entity:

```
@JsonIdentityInfo(
  generator = ObjectIdGenerators.PropertyGenerator.class,
  property = "id")

  public class ItemWithIdentity {

  public int id;
  public String itemName;
  public UserWithIdentity owner;
}
```

And the *UserWithIdentity* entity:

```
@JsonIdentityInfo(
  generator = ObjectIdGenerators.PropertyGenerator.class,
  property = "id")

  public class UserWithIdentity {

  public int id;
  public String name;
  public List<ItemWithIdentity> userItems;
}
```

Now, let's see how the infinite recursion problem is handled:

```
@Test
public void whenSerializingUsingJsonIdentityInfo_thenCorrect()
  throws JsonProcessingException {
    UserWithIdentity user = new UserWithIdentity(1, "John");
    ItemWithIdentity item = new ItemWithIdentity(2, "book", user);
    user.addItem(item);
    String result = new ObjectMapper().writeValueAsString(item);
    assertThat(result, containsString("book"));
    assertThat(result, containsString("John"));
    assertThat(result, containsString("userItems"));
}
```

Here's the full output of the serialized item and user:

```
{
    "id": 2,
    "itemName": "book",
    "owner": {
        "id": 1,
        "name": "John",
        "userItems": [
            2
        ]
    }
}
```

6.7. *@JsonFilter*

The *@JsonFilter* annotation specifies a filter to use during serialization.

Let's take a look at an example; first, we define the entity, and we point to the filter:

```
@JsonFilter("myFilter")
public class BeanWithFilter {
    public int id;
    public String name;
}
```

Now, in the full test, we define the filter – which excludes all other properties except *name* from serialization:

```
1.  @Test
2.  public void whenSerializingUsingJsonFilter_thenCorrect()
3.    throws JsonProcessingException {
4.      BeanWithFilter bean = new BeanWithFilter(1, “My bean”);
5.
6.      FilterProvider filters
7.        = new SimpleFilterProvider().addFilter(
8.          “myFilter”,
9.          SimpleBeanPropertyFilter.filterOutAllExcept(“name”));
10.
11.     String result = new ObjectMapper()
12.       .writer(filters)
13.       .writeValueAsString(bean);
14.
15.     assertThat(result, containsString(“My bean”));
16.     assertThat(result, not(containsString(“id”)));
17. }
```

Next, let's see how to create a custom Jackson annotation. **We can make use of the *@JacksonAnnotationsInside* annotation:**

```
@Retention(RetentionPolicy.RUNTIME)
    @JacksonAnnotationsInside
    @JsonInclude(Include.NON_NULL)
    @JsonPropertyOrder({ "name", "id", "dateCreated" })
    public @interface CustomAnnotation {}
```

Now, if we use the new annotation on an entity:

```
@CustomAnnotation
public class BeanWithCustomAnnotation {
    public int id;
    public String name;
    public Date dateCreated;
}
```

We can see how it does combine the existing annotations into a simpler, custom one that we can use as a shorthand:

```
@Test
public void whenSerializingUsingCustomAnnotation_thenCorrect()
  throws JsonProcessingException {
    BeanWithCustomAnnotation bean
      = new BeanWithCustomAnnotation(1, "My bean", null);
    String result = new ObjectMapper().writeValueAsString(bean);
    assertThat(result, containsString("My bean"));
    assertThat(result, containsString("1"));
    assertThat(result, not(containsString("dateCreated")));
}
```

The output of the serialization process:

```
{
    "name":"My bean",
    "id":1
}
```

Next – let's see how to use Jackson MixIn annotations.

Let's use the MixIn annotations to – for example – ignore properties of type *User*:

```
1. public class Item {
2.     public int id;
3.     public String itemName;
4.     public User owner;
5. }
6.
7. @JsonIgnoreType
8. public class MyMixInForIgnoreType {}
```

Let's see this in action:

```
1. @Test
2. public void whenSerializingUsingMixInAnnotation_thenCorrect()
3.   throws JsonProcessingException {
4.     Item item = new Item(1, "book", null);
5.
6.     String result = new ObjectMapper().writeValueAsString(item);
7.     assertThat(result, containsString("owner"));
8.
9.     ObjectMapper mapper = new ObjectMapper();
10.    mapper.addMixIn(User.class, MyMixInForIgnoreType.class);
11.
12.    result = mapper.writeValueAsString(item);
13.    assertThat(result, not(containsString("owner")));
14. }
```

Finally – let's see how we can **disable all Jackson annotations**. We can do this by disabling the *MapperFeature*.USE_ANNOTATIONS as in the following example:

```
@JsonInclude(Include.NON_NULL)
@JsonPropertyOrder({ "name", "id" })
public class MyBean {
    public int id;
    public String name;
}
```

Now, after disabling annotations, these should have no effect and the defaults of the library should apply:

```
@Test
public void whenDisablingAllAnnotations_thenAllDisabled()
  throws IOException {
    MyBean bean = new MyBean(1, null);

    ObjectMapper mapper = new ObjectMapper();
    mapper.disable(MapperFeature.USE_ANNOTATIONS);
    String result = mapper.writeValueAsString(bean);

    assertThat(result, containsString("1"));
    assertThat(result, containsString("name"));
}
```

The result of serialization before disabling annotations:

```
{"id":1}
```

The result of serialization after disabling annotations:

```
{
    "id":1,
    "name":null
}
```

10. Conclusion

This chapter has been a deep-dive into Jackson annotations, just scratching the surface of the kind of flexibility you can get using them correctly.

The implementation of all these examples and code snippets can be found in the GitHub project.

2: Intro to the Jackson ObjectMapper

1. Overview

This write-up focuses on understanding the Jackson ObjectMapper class – and how to serialize Java objects into JSON and deserialize JSON string into Java objects.

Let's start with the basic read and write operations.

The simple *readValue* API of the *ObjectMapper* is a good entry point. We can use it to parse or deserialize JSON content into a Java object.

Also, on the writing side of things, **we can use the *writeValue* API to serialize any Java object as JSON output.**

We'll use the following *Car* class with two fields as the object to serialize or deserialize throughout this chapter:

```
1.  public class Car {
2.
3.      private String color;
4.      private String type;
5.
6.      // standard getters setters
7.  }
```

2.1. Java Object to JSON

Let's see a first example of serializing a Java Object into JSON using the *writeValue* method of *ObjectMapper* class:

```
1.  ObjectMapper objectMapper = new ObjectMapper();
2.  Car car = new Car("yellow", "renault");
3.  objectMapper.writeValue(new File("target/car.json"), car);
```

The output of the above in the file will be:

```
1.  {"color":"yellow","type":"renault"}
```

The methods *writeValueAsString* and *writeValueAsBytes* of *ObjectMapper* class generates a JSON from a Java object and returns the generated JSON as a string or as a byte array:

```
1. String carAsString = objectMapper.writeValueAsString(car);
```

2.2. JSON to Java Object

Below is a simple example of converting a JSON String to a Java object using the *ObjectMapper* class:

```
1. String json = "{ \"color\" : \"Black\", \"type\" : \"BMW\" }";
2. Car car = objectMapper.readValue(json, Car.class);
```

The *readValue()* function also accepts other forms of input like a file containing JSON string:

```
1. Car car = objectMapper.readValue(new File("target/json_car.json"),
2. Car.class);
3. or a URL:
4. Car car = objectMapper.readValue(new URL("target/json_car.json"),
5. Car.class);
```

2.3. JSON to Jackson JsonNode

Alternatively, a JSON can be parsed into a *JsonNode* object and used to retrieve data from a specific node:

```
1. String json = "{ \"color\" : \"Black\", \"type\" : \"FIAT\" }";
2. JsonNode jsonNode = objectMapper.readTree(json);
3. String color = jsonNode.get("color").asText();
4. // Output: color -> Black
```

2.4. Creating a Java List from a JSON Array String

e can parse a JSON in the form of an array into a Java object list using a *TypeReference*:

```
String jsonCarArray =
  "[{ \"color\" : \"Black\", \"type\" : \"BMW\" }, { \"color\" :
\"Red\", \"type\" : \"FIAT\" }]";
List<Car> listCar = objectMapper.readValue(jsonCarArray, new
TypeReference<List<Car>>(){});
```

2.5. Creating Java Map from JSON String

Similarly, we can parse A JSON into a Java *Map*:

```
String json = "{ \"color\" : \"Black\", \"type\" : \"BMW\" }";
Map<String, Object> map
  = objectMapper.readValue(json, new
TypeReference<Map<String,Object>>(){});
```

One of the greatest strength of the Jackson library is the highly customizable serialization and deserialization process.

In this section, we'll go through some advanced features where the input or the output JSON response can be different from the object which generates or consumes the response.

3.1. Configuring Serialization or Deserialization Feature

While converting JSON objects to Java classes, in case the JSON string has some new fields, then the default process will result in an exception:

```
String jsonString
  = "{ \"color\" : \"Black\", \"type\" : \"Fiat\", \"year\" :
\"1970\" }";
```

The JSON string in the above example in the default parsing process to the Java object for the *Class Car* will result in the *UnrecognizedPropertyException* exception.

Through the *configure* method we can extend the default process to ignore the new fields:

```
objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_
PROPERTIES, false);
Car car = objectMapper.readValue(jsonString, Car.class);
JsonNode jsonNodeRoot = objectMapper.readTree(jsonString);
JsonNode jsonNodeYear = jsonNodeRoot.get("year");
String year = jsonNodeYear.asText();
```

Yet another option is based on the *FAIL_ON_NULL_FOR_PRIMITIVES* which defines if the *null* values for primitive values are allowed:

```
objectMapper.configure(DeserializationFeature.FAIL_ON_NULL_FOR_
PRIMITIVES, false);
```

Similarly, *FAIL_ON_NUMBERS_FOR_ENUM* controls if enum values are allowed to be serialized/deserialized as numbers:

```
objectMapper.configure(DeserializationFeature.FAIL_ON_NUMBERS_FOR_
ENUMS, false);
```

You can find the comprehensive list of serialization and deserialization features on the official site.

3.2. Creating Custom Serializer or Deserializer

Another essential feature of the *ObjectMapper* class is the ability to register custom serializer and deserializer. Custom serializer and deserializer are very useful in situations where the input or the output JSON response is different in structure than the Java class into which it must be serialized or deserialized.

Below is an example of custom JSON serializer:

```
public class CustomCarSerializer extends StdSerializer<Car> {

    public CustomCarSerializer() {
        this(null);
    }

    public CustomCarSerializer(Class<Car> t) {
        super(t);
    }

    @Override
    public void serialize(
      Car car, JsonGenerator jsonGenerator, SerializerProvider
serializer) {
        jsonGenerator.writeStartObject();
        jsonGenerator.writeStringField("car_brand", car.
getType());
        jsonGenerator.writeEndObject();
    }
}
```

This custom serializer can be invoked like this:

```
ObjectMapper mapper = new ObjectMapper();
SimpleModule module =
  new SimpleModule("CustomCarSerializer", new Version(1, 0, 0,
null, null, null));
module.addSerializer(Car.class, new CustomCarSerializer());
mapper.registerModule(module);
Car car = new Car("yellow", "renault");
String carJson = mapper.writeValueAsString(car);
```

Here's what the Car looks like (as JSON output) on the client side:

```
var carJson = {"car_brand":"renault"}
```

And here's an example of a custom JSON deserializer:

```
public class CustomCarDeserializer extends StdDeserializer<Car>
{

    public CustomCarDeserializer() {
        this(null);
    }

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

    @Override
    public Car deserialize(JsonParser parser,
DeserializationContext deserializer) {
        Car car = new Car();
        ObjectCodec codec = parser.getCodec();
        JsonNode node = codec.readTree(parser);

        // try catch block
        JsonNode colorNode = node.get("color");
        String color = colorNode.asText();
        car.setColor(color);
        return car;
    }
}
```

And here's an example of a custom JSON deserializer:

```
String json = "{ \"color\" : \"Black\", \"type\" : \"BMW\" }";
ObjectMapper mapper = new ObjectMapper();
SimpleModule module =
  new SimpleModule("CustomCarDeserializer", new Version(1, 0,
0, null, null, null));
module.addDeserializer(Car.class, new CustomCarDeserializer());
mapper.registerModule(module);
Car car = mapper.readValue(json, Car.class);
```

3.3. Handling Date Formats

The default serialization of *java.util.Date* produces a number i.e. epoch timestamp (number of milliseconds since January 1st, 1970, UTC). But this is not very human-readable and requires further conversion to be displayed in human-readable format. Let's wrap the *Car* instance we used so far inside the *Request* class with the *datePurchased* property:

```
public class Request
{
    private Car car;
    private Date datePurchased;

    // standard getters setters
}
```

To control the String format of a date, and set it to e.g. *yyyy-MM-dd HH:mm a z*, consider the following snippet:

```
ObjectMapper objectMapper = new ObjectMapper();
DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm a z");
objectMapper.setDateFormat(df);
String carAsString = objectMapper.writeValueAsString(request);
// output: {"car":{"color":"yellow","type":"renault"},
"datePurchased":"2016-07-03 11:43 AM CEST"}
```

To learn more about serializing dates with Jackson, read our more in-depth write-up.

3.4. Handling Collections

Another small but useful feature available through the *DeserializationFeature* class is the ability to generate the type of collection we want from a JSON Array response.

For example, we can generate the result as an array:

```
String jsonCarArray =
  "[{ \"color\" : \"Black\", \"type\" : \"BMW\" }, { \"color\" :
\"Red\", \"type\" : \"FIAT\" }]";
ObjectMapper objectMapper = new ObjectMapper();
objectMapper.configure(DeserializationFeature.USE_JAVA_ARRAY_FOR_
JSON_ARRAY, true);
Car[] cars = objectMapper.readValue(jsonCarArray, Car[].class);
// print cars
```

Or as a List:

```
String jsonCarArray =
  "[{ \"color\" : \"Black\", \"type\" : \"BMW\" }, { \"color\" :
\"Red\", \"type\" : \"FIAT\" }]";
ObjectMapper objectMapper = new ObjectMapper();
List<Car> listCar = objectMapper.readValue(jsonCarArray, new
TypeReference<List<Car>>(){});
// print cars
```

More information about handling collections with Jackson is available here.

4. Conclusion

Jackson is a solid and mature JSON serialization/deserialization library for Java. The *ObjectMapper* API provides a straightforward way to parse and generate JSON response objects with a lot of flexibility.

The chapter discusses the main features which make the library so popular. The source code that accompanies the chapter can be found over on GitHub.

3: Jackson Ignore Properties on Marshalling

1. Overview

This chapter will show how to **ignore certain fields when serializing an object to JSON** using Jackson 2.x.

This is very useful when the Jackson defaults aren't enough and we need to control exactly what gets serialized to JSON – and there are several ways to ignore properties.

We can ignore specific fields at the class level, using **the *@JsonIgnoreProperties* annotation and specifying the fields by name:**

```
@JsonIgnoreProperties(value = { "intValue" })
public class MyDto {

    private String stringValue;
    private int intValue;
    private boolean booleanValue;

    public MyDto() {
        super();
    }

    // standard setters and getters are not shown
}
```

We can now test that, after the object is written to JSON, the field is indeed not part of the output:

```
@Test
public void givenFieldIsIgnoredByName_whenDtoIsSerialized_
thenCorrect()
  throws JsonParseException, IOException {

    ObjectMapper mapper = new ObjectMapper();
    MyDto dtoObject = new MyDto();

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, not(containsString("intValue")));
}
```

To ignore any unknown properties in JSON input without exception, we can set *ignoreUnknown=true* of *@JsonIgnoreProperties* annotation.

3. Ignore Field at the Field Level

We can also ignore a field directly via the *@JsonIgnore* annotation directly on the field:

```
1.  public class MyDto {
2.
3.      private String stringValue;
4.      @JsonIgnore
5.      private int intValue;
6.      private boolean booleanValue;
7.
8.      public MyDto() {
9.          super();
10.     }
11.
12.     // standard setters and getters are not shown
13. }
```

We can now test that the intValue field is indeed not part of the serialized JSON output:

```
1.  @Test
2.  public void givenFieldIsIgnoredDirectly_whenDtoIsSerialized_
3.  thenCorrect()
4.    throws JsonParseException, IOException {
5.
6.      ObjectMapper mapper = new ObjectMapper();
7.      MyDto dtoObject = new MyDto();
8.
9.      String dtoAsString = mapper.writeValueAsString(dtoObject);
10.
11.     assertThat(dtoAsString, not(containsString("intValue")));
12. }
```

4. Ignore all Fields by Type

Finally, we can **ignore all fields of a specified type, using the *@JsonIgnoreType* annotation**. If we control the type, then we can annotate the class directly:

```
@JsonIgnoreType
public class SomeType { ... }
```

More often than not, however, we don't have control of the class itself; in this case, **we can make good use of Jackson mixins.**

First, we define a MixIn for the type we'd like to ignore, and annotate that with *@JsonIgnoreType* instead:

```
@JsonIgnoreType
public class MyMixInForIgnoreType {}
```

Then we register that mixin to replace (and ignore) all String[] types during marshaling:

```
mapper.addMixInAnnotations(String[].class, MyMixInForIgnoreType.
class);
```

At this point, all String arrays will be ignored instead of marshaled to JSON:

```
@Test
public final void givenFieldTypeIsIgnored_whenDtoIsSerialized_
thenCorrect()
  throws JsonParseException, IOException {

    ObjectMapper mapper = new ObjectMapper();
    mapper.addMixIn(String[].class, MyMixInForIgnoreType.class);
    MyDtoWithSpecialField dtoObject = new
MyDtoWithSpecialField();
    dtoObject.setBooleanValue(true);

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, containsString("intValue"));
    assertThat(dtoAsString, containsString("booleanValue"));
    assertThat(dtoAsString, not(containsString("stringValue")));
}
```

and here is our DTO:

```
public class MyDtoWithSpecialField {
    private String[] stringValue;
    private int intValue;
    private boolean booleanValue;
}
```

Note: Since version 2.5 – it seems that we can not use this method to ignore primitive data types, but we can use it for custom data types and arrays.

Finally, **we can also use filters to ignore specific fields** in Jackson. First, we need to define the filter on the Java object:

```
@JsonFilter("myFilter")
public class MyDtoWithFilter { ... }
```

Then, we define a simple filter that will ignore the *intValue* field:

```
SimpleBeanPropertyFilter theFilter = SimpleBeanPropertyFilter
  .serializeAllExcept("intValue");
FilterProvider filters = new SimpleFilterProvider()
  .addFilter("myFilter", theFilter);
```

Now we can serialize the object and make sure that the *intValue* field is not present in the JSON output:

```
@Test
public final void givenTypeHasFilterThatIgnoresFieldByName_
whenDtoIsSerialized_thenCorrect()
  throws JsonParseException, IOException {

    ObjectMapper mapper = new ObjectMapper();
    SimpleBeanPropertyFilter theFilter =
SimpleBeanPropertyFilter
      .serializeAllExcept("intValue");
    FilterProvider filters = new SimpleFilterProvider()
      .addFilter("myFilter", theFilter);

    MyDtoWithFilter dtoObject = new MyDtoWithFilter();
    String dtoAsString = mapper.writer(filters).
writeValueAsString(dtoObject);

    assertThat(dtoAsString, not(containsString("intValue")));
    assertThat(dtoAsString, containsString("booleanValue"));
    assertThat(dtoAsString, containsString("stringValue"));
    System.out.println(dtoAsString);
}
```

6. Conclusion

The chapter illustrated how to ignore fields on serialization – first by name, then directly, and finally – we ignored the entire java type with *MixIns* and we use filters for more control of the output.

The implementation of all these examples and code snippets can be found in my GitHub project.

4: Ignore Null Fields with Jackson

1. Overview

This quick chapter is going to cover how to set up **Jackson to ignore null fields when serializing** a java class.

Jackson allows controlling this behavior at either the class level:

```
@JsonInclude(Include.NON_NULL)
public class MyDto { ... }
```

Or – more granularly – at the field level:

```
public class MyDto {

    @JsonInclude(Include.NON_NULL)
    private String stringValue;

    private int intValue;

    // standard getters and setters
}
```

Now, we should be able to test that null values are indeed not part of the final JSON output:

```
@Test
public void givenNullsIgnoredOnClass_
whenWritingObjectWithNullField_thenIgnored()
  throws JsonProcessingException {
    ObjectMapper mapper = new ObjectMapper();
    MyDto dtoObject = new MyDto();

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, containsString("intValue"));
    assertThat(dtoAsString, not(containsString("stringValue")));
}
```

Jackson also allows **configuring this behavior globally on the *ObjectMapper***:

```
mapper.setSerializationInclusion(Include.NON_NULL);
```

Now any *null* field in any class serialized through this mapper will be ignored:

```
@Test
public void givenNullsIgnoredGlobally_
whenWritingObjectWithNullField_thenIgnored()
  throws JsonProcessingException {
    ObjectMapper mapper = new ObjectMapper();
    mapper.setSerializationInclusion(Include.NON_NULL);
    MyDto dtoObject = new MyDto();

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, containsString("intValue"));
    assertThat(dtoAsString, containsString("booleanValue"));
    assertThat(dtoAsString, not(containsString("stringValue")));
}
```

4. Conclusion

Ignoring *null* fields is such a common Jackson configuration because it's often the case that we need to have better control over the JSON output. This chapter shows how to do that for classes. There are however more advanced use cases, such as ignoring null values when serializing a Map.

The implementation of all these examples and code snippets can be found in my Github project.

5: Jackson – Change Name of Field

1. Overview

This quick chapter illustrates how to **change the name of a field to map to another JSON property** on serialization.

Working with a simple entity:

```
1.  public class MyDto {
2.      private String stringValue;
3.
4.      public MyDto() {
5.          super();
6.      }
7.
8.      public String getStringValue() {
9.          return stringValue;
10.     }
11.
12.     public void setStringValue(String stringValue) {
13.         this.stringValue = stringValue;
14.     }
15. }
```

Serializing it will result in the following JSON:

```
1.  {"stringValue":"some value"}
```

To customize that output so that, instead of *stringValue* we get – for example – *strVal*, we need to simply annotate the getter:

```
1.  @JsonProperty("strVal")
2.  public String getStringValue() {
3.      return stringValue;
4.  }
```

Now, on serialization, we will get the desired output:

```
1.  {"strVal":"some value"}
```

A simple unit test should verify the output is correct:

```
@Test
public void givenNameOfFieldIsChanged_whenSerializing_
thenCorrect()
  throws JsonParseException, IOException {
    ObjectMapper mapper = new ObjectMapper();
    MyDtoFieldNameChanged dtoObject = new
MyDtoFieldNameChanged();
    dtoObject.setStringValue("a");

    String dtoAsString = mapper.writeValueAsString(dtoObject);

    assertThat(dtoAsString, not(containsString("stringValue")));
    assertThat(dtoAsString, containsString("strVal"));
}
```

3. Conclusion

Marshaling an entity to adhere to a specific JSON format is a common task – and this chapter shows how to do is simply by using the *@JsonProperty* annotation.

The implementation of all these examples and code snippets can be found in my Github project.

6: Jackson Unmarshalling JSON with Unknown Properties

1. Overview

In this chapter, we're going to take a look at the unmarshalling process with Jackson 2.x – specifically at **how to deal with JSON content with unknown properties.**

JSON input comes in all shapes and sizes – and most of the time, we need to map it to predefined Java objects with a set number of fields. The goal is to simply **ignore any JSON properties that cannot be mapped to an existing Java field**.

For example, say we need to unmarshal JSON to the following Java entity:

```
1.  public class MyDto {
2.
3.      private String stringValue;
4.      private int intValue;
5.      private boolean booleanValue;
6.
7.      // standard constructor, getters and setters
8.  }
```

2.1. UnrecognizedPropertyException on Unknown Fields

Trying to unmarshal a JSON with unknown properties to this simple Java Entity will lead to a *com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException:*

```
1.  @Test(expected = UnrecognizedPropertyException.class)
2.  public void givenJsonHasUnknownValues_whenDeserializing_
3.  thenException()
4.    throws JsonParseException, JsonMappingException, IOException {
5.      String jsonAsString =
6.          "{"stringValue":"a"," +
7.          ""intValue":1," +
8.          ""booleanValue":true," +
9.          ""stringValue2":"something"}";
10.     ObjectMapper mapper = new ObjectMapper();
11.
12.     MyDto readValue = mapper.readValue(jsonAsString, MyDto.class);
13.
14.     assertNotNull(readValue);
15. }
```

This will fail with the following exception:

```
com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException:
Unrecognized field “stringValue2” (class org.baeldung.jackson.
ignore.MyDto),
not marked as ignorable (3 known properties: “stringValue”,
“booleanValue”, “intValue”])
```

2.2. Dealing with Unknown Fields on the ObjectMapper

We can now configure the full *ObjectMapper* to ignore unknown properties in the JSON:

```
new ObjectMapper()
  .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false)
```

We should then be able to read this kind of JSON into a predefined Java entity:

```
@Test
public void givenJsonHasUnknownValuesButJacksonIsIgnoringUnknowns_
whenDeserializing_thenCorrect()
  throws JsonParseException, JsonMappingException, IOException {

    String jsonAsString =
        “{“stringValue”:”a”,” +
        “”intValue”:1,” +
        “”booleanValue”:true,” +
        “”stringValue2”:”something”}”;
    ObjectMapper mapper = new ObjectMapper()
      .configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false);

    MyDto readValue = mapper.readValue(jsonAsString, MyDto.class);

    assertNotNull(readValue);
    assertThat(readValue.getStringValue(), equalTo(“a”));
    assertThat(readValue.isBooleanValue(), equalTo(true));
    assertThat(readValue.getIntValue(), equalTo(1));
}
```

2.3. Dealing with Unknown Fields on the Class

We can also mark a single class as accepting unknown fields, instead of the entire Jackson *ObjectMapper*:

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class MyDtoIgnoreUnknown { ... }
```

Now, we should be able to test the same behavior as before – unknown fields are simply ignored and only known fields are mapped:

```
@Test
public void givenJsonHasUnknownValuesButIgnoredOnClass_
whenDeserializing_thenCorrect()
  throws JsonParseException, JsonMappingException, IOException {
    String jsonAsString =
        "{"stringValue":"a"," +
        ""intValue":1," +
        ""booleanValue":true," +
        ""stringValue2":"something"}";
    ObjectMapper mapper = new ObjectMapper();
    MyDtoIgnoreUnknown readValue = mapper
      .readValue(jsonAsString, MyDtoIgnoreUnknown.class);
    assertNotNull(readValue);
    assertThat(readValue.getStringValue(), equalTo("a"));
    assertThat(readValue.isBooleanValue(), equalTo(true));
    assertThat(readValue.getIntValue(), equalTo(1));
}
```

Similarly to additional unknown fields, unmarshalling an incomplete JSON – a JSON that doesn't contain all the fields in the Java class – is not a problem with Jackson:

```
@Test
public void givenNotAllFieldsHaveValuesInJson_
whenDeserializingAJsonToAClass_thenCorrect()
  throws JsonParseException, JsonMappingException, IOException {
    String jsonAsString =
"{"stringValue":"a","booleanValue":true}";
    ObjectMapper mapper = new ObjectMapper();
    MyDto readValue = mapper.readValue(jsonAsString, MyDto.class);
    assertNotNull(readValue);
    assertThat(readValue.getStringValue(), equalTo("a"));
    assertThat(readValue.isBooleanValue(), equalTo(true));
}
```

4. Conclusion

This chapter covered deserializing a JSON with additional, unknown properties, using Jackson.

This is one of the most common things to configure when working with Jackson since it's often the case we need to **map JSON results of external REST APIs to an internal Java representation** of the entities of the API.

The implementation of all these examples and code snippets can be found in my Github project.