

Define

Mapping در SW یعنی تغییر ساختار

Serializing نوعی mapping است. Obj به file of bytes

یکی از ابزارهای Serialize کردن Jackson است.

Serialization & Deserialization

Serialization converts a Java object into a stream of byte

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-dataformat-xml</artifactId>
  <version>2.14.2</version>
</dependency>
```

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

1. jackson-annotations
2. jackson-core

Standard Serialization (marshal/ write) (Obj to Json/ string)

```
public class Car {

    private String color;
    private String type;

    // standard getters setters
}
```

```
ObjectMapper objectMapper = new ObjectMapper();
Car car = new Car("yellow", "renault");
```

```
objectMapper.writeValue(new File("target/car.json"), car);
```

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

The methods `writeValueAsString` and `writeValueAsBytes` of `ObjectMapper` class generate a JSON from a Java object and return the generated JSON as a string or as a byte array:

*** @JsonValue Annotation

@JsonValue: ensure Jackson uses this custom **toString ()** when serializing

```
public class Fruit {  
    public String variety;  
  
    @JsonKey  
    public String name;  
  
    public Fruit(String variety, String name) {  
        this.variety = variety;  
        this.name = name;  
    }  
  
    @JsonValue  
    public String getFullName() {  
        return this.variety + " " + this.name;  
    }  
}
```

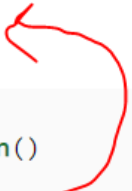
```
private static final Fruit FRUIT1 = new Fruit("Alphonso", "Mango");  
private static final Fruit FRUIT2 = new Fruit("Black", "Grapes");  
private static final ObjectMapper OBJECT_MAPPER = new ObjectMapper();
```

the @JsonValue property is used for serialization.

```

@Test
public void givenObject_WhenSerialize_ThenUseJsonValueForSerialization()
    throws JsonProcessingException {
    String serializedValueForFruit1 = OBJECT_MAPPER.writeValueAsString(FRUIT1);
    Assertions.assertEquals("\"Alphonso Mango\"", serializedValueForFruit1);
    String serializedValueForFruit2 = OBJECT_MAPPER.writeValueAsString(FRUIT2);
    Assertions.assertEquals("\"Black Grapes\"", serializedValueForFruit2);
}

```



Standard Deserialization (unmarshal/ read) (Json to Obj)

```

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

```

```

Car car = objectMapper.readValue(new File("src/test/resources/json_car.json"), Car.class);

```

```

Car car =
    objectMapper.readValue(new URL("file:src/test/resources/json_car.json"), Car.class);

```

```

public class User {
    public int id;
    public String name;
}

public class Item {
    public int id;
    public String itemName;
    public User owner;
}

```

```

Item itemWithOwner = new ObjectMapper().readValue(json, Item.class);

```

```
{
  "id": 1,
  "itemName": "theItem",
  "owner": {
    "id": 2,
    "name": "theUser"
  }
}
```

ok.

خود readValue فقط وقتی درست کار میکند که json دقیقاً مثل Class آمده باشد، اگر این json بیاید درست کار نمیکند:

```
{
  "id": 1,
  "itemName": "theItem",
  "createdBy": 2
}
```

When unmarshalling this to the exact same entities, by default, this will of course fail:

JSON to Jackson JsonNode

retrieve data from a specific node:

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

**Deserialization Collections- TypeReference

مثلاً در ورودی سرویس، بعنوان یک String ساده دیتا را بگیریم و بعد :

DeserializationFeature class is the ability to **generate** the type of **collection** we want from a “JSON Array req”.

Jackson's `TypeReference`

```
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
```

We 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>>(){});
```

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

<https://www.baeldung.com/jackson-collection-array>

****Serialization Date Formats**

default **serialization** of `java.util.Date` produces a number, i.e., epoch timestamp (number of milliseconds)

```

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

```

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"}

```

***Configuring object mapper

در استفاده از jackson، برای زمانی که نمیخواهیم همه ی اطلاعات obj را در response یا file بگنجانیم.

- برخی از تغییرات با **Configure کردن jackson** قابل انجام است.
- برخی از تغییرات با **Configure کردن قابل انجام نیست و باید Custom Serializer** با **jackson** بنویسیم.
- **convert** نوع انجام بگیرد و نوع جدید کاملاً **serialize** شود. **responseClass** بنویسیم.

If ignore the extra fields:

```

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

```

```
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();
```

if the null values for primitive values are allowed:

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

if enum values are allowed to be serialized/deserialize as numbers:

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

***Creating Custom Serializer / extend std

فرض می‌خواهیم به یک third party ریک بزنیم که روی json دریافتی اش حساس است، نمیتوانیم همین obj ای که داریم را serialize کنیم و بفرستیم، نیاز داریم یک **Serializer** مخصوص برایش بنویسیم.

Mapper → Module → Serializer (or Deserializer)

در استفاده از jackson، برای زمانی که نمیخواهیم همه ی اطلاعات obj را در response یا file بگنجانیم.

- برخی از تغییرات با **Configure کردن jackson** قابل انجام است.
- برخی از تغییرات با **Configure** کردن قابل انجام نیست و باید **Custom Serializer** با **jackson** بنویسیم.
- **convert** نوع انجام بگیرد و نوع جدید کاملاً **serialize** شود.

Custom serializers and deserializers 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.

```
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();  
    }  
}
```

```
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);
```




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

***Creating Custom Deserializer / extend std

مثلا برای وقتی مناسب (واجب) است که json رسیده منطبق بر class نباشد و خطا میخوریم. بخشی از اینکار با Configuration قابل انجام است و برای بقیش باید Custom بنویسیم.

```
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;  
    }  
}
```

```
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);
```



یک مثال دیگر:

```
public class ItemDeserializer extends StdDeserializer<Item> {

    public ItemDeserializer() {
        this(null);
    }

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

    @Override
    public Item deserialize(JsonParser jp, DeserializationContext ctxt)
        throws IOException, JsonProcessingException {
        JsonNode node = jp.getCodec().readTree(jp);
        int id = (Integer) ((IntNode) node.get("id")).numberValue();
        String itemName = node.get("itemName").asText();
        int userId = (Integer) ((IntNode) node.get("createdBy")).numberValue();

        return new Item(id, itemName, new User(userId, null));
    }
}
```

Introducing Deserializer on the Class

Reduce writing codes

Alternatively, we can also **register the deserializer directly on the class**:

```
@JsonDeserialize(using = ItemDeserializer.class)
public class Item {
    ...
}
```

With the deserializer defined at the class level, there is **no need to register it** on the *ObjectMapper* — a default mapper will work fine:

```
Item itemWithOwner = new ObjectMapper().readValue(json, Item.class);
```

This type of per-class configuration is very useful in situations in which we may not have direct access to the raw *ObjectMapper* to configure.

Deserializer for a Generic Type

```
public class Item {
    public int id;
    public String itemName;
    public Wrapper<User> owner;
}
```

```
public class Wrapper<T> {

    T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

```

public class WrapperDeserializer extends JsonSerializer<Wrapper<?>> implements
ContextualDeserializer {

    private JavaType type;

    @Override
    public JsonSerializer<?> createContextual(DeserializationContext ctxt, BeanProperty property) {
        this.type = property.getType().containedType(0);
        return this;
    }

    @Override
    public Wrapper<?> deserialize(JsonParser jsonParser, DeserializationContext
deserializationContext) throws IOException {
        Wrapper<?> wrapper = new Wrapper<>();
        wrapper.setValue(deserializationContext.readValue(jsonParser, type));
        return wrapper;
    }
}

```

Activate Windows
Go to Settings to activate Windows.

```

ObjectMapper mapper = new ObjectMapper();
SimpleModule module = new SimpleModule();
module.addDeserializer(Wrapper.class, new WrapperDeserializer());
mapper.registerModule(module);

Item readValue = mapper.readValue(json, Item.class);

```

****jackson in SpringBootWeb-restful

In rest-full SpringBoot uses a Jackson.ObjectMapper instance to serialize responses and deserialize requests.

وقتی rest هستیم (کاری که @RequestBody و @ResponseBody میکنند) یعنی model در req و res جابجا نمیشود، row-type جابجا میشود
دیفالت SpringBoot برای row-type، json است

عمل read و write در دل Spring MVC انجام میشود.

تنظیمات serialize و deserialize در Rest Spring-mvc میتواند :

■ تنظیمات ابتدایی در Spring-mvc میتواند در **@RequestMapping** انجام شود (در حد تغییر به xml) مثلا **Consume**: تنظیمات اینکه سرویس چه type ای را بعنوان دیتای ورودی میگیرد و req طبق آن درخواست بفرستد.

■ برخی از تغییرات با **Configure** کردن **jackson** قابل انجام است.

در **spring-boot** در **فایل Jackson** تغییرات بزرگتر قابل انجام است. **ObjectMapper** بعضی تغییرات را خود **SpringBoot** بصورت **default** کانفیگ کرده مثلا اگر **فیلدی نیامد خطا نخوریم** ولی **تغییرات بیشتر** را میتوانیم اعمال کنیم مثلا **null**ها نروند و فرمت تاریخ **millisecond** نباشد.

By default, the Spring Boot configuration will **disable** the following:

- *MapperFeature.DEFAULT_VIEW_INCLUSION*
- *DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES*
- *SerializationFeature.WRITE_DATES_AS_TIMESTAMPS*

■ برخی از تغییرات با **Configure** کردن قابل انجام نیست و باید **Custom Serializer** با **jackson** بنویسیم. در فایل **my note jackson** آمده.

راه دیگر: **convert** نوع هم میتواند انجام بگیرد و نوع جدید **serialize** شود. میتوانیم **obj** ای که میخواهیم با **client** رد و بدل کنیم را **convert** کنیم و **converted obj** را برای **serialize/deserialize** به **spring-mvc rest** بسپاریم. **responseClass** بنویسیم.

```
public class Coffee {

    private String name;
    private String brand;
    private LocalDateTime date;

    //getters and setters
}
```

```
@GetMapping("/coffee")
public Coffee getCoffee(
    @RequestParam(required = false) String brand,
    @RequestParam(required = false) String name) {
    return new Coffee()
        .setBrand(brand)
        .setDate(FIXED_DATE)
        .setName(name);
}
```

If:

GET <http://localhost:8080/coffee?brand=Lavazza>

بخاطر تنظیمات دیفالتی که SpringBoot روی Jackson گذاشته، با وجود اینکه json رسیده منطبق بر objای که سرویس انتظار دارد نیست، **Deserialize** خطا نمیخورد.

و **Response** ای که به کلاینت میدهد این است:

```
{
  "name": null,
  "brand": "Lavazza",
  "date": "2020-11-16T10:21:35.974"
}
```

But

We would like to exclude *null* values and to have a custom date format (dd-MM-yyyy HH:mm).
response:

```
{  
  "brand": "Lavazza",  
  "date": "04-11-2020 10:34"  
}
```

پس باید تنظیمات بیشتری به **SpringBoot** بدهیم:

When using **Spring Boot**, we have the option to customize the default *ObjectMapper* or to override it. We'll cover both options in the next sections.

****Configuring the ObjectMapper

Date Serialize in SpringBoot & Dont serialize null value:

روش اول Customizing :

application properties:

```
spring.jackson.<category_name>.<feature_name>=true,false
```

Dont serialize null value:

```
spring.jackson.default-property-inclusion=always, non_null, non_absent, non_default, non_empty
```

At this point, we'll obtain this result:

```
{
  "brand": "Lavazza",
  "date": "2020-11-16T10:35:34.593"
}
```

Date Serialize in SpringBoot

```
@Configuration
@PropertySource("classpath:coffee.properties")
public class CoffeeRegisterModuleConfig {

    @Bean
    public Module javaTimeModule() {
        JavaTimeModule module = new JavaTimeModule();
        module.addSerializer(LOCAL_DATETIME_SERIALIZER);
        return module;
    }
}
```

```
spring.jackson.serialization.write-dates-as-timestamps=false
```

```
{
  "brand": "Lavazza",
  "date": "16-11-2020 10:43"
}
```


روش دوم : Customizing

Jackson2ObjectMapperBuilderCustomizer

create configuration beans

```
@Bean
public Jackson2ObjectMapperBuilderCustomizer jsonCustomizer() {
    return builder -> builder.serializationInclusion(JsonInclude.Include.NON_NULL)
        .serializers(LOCAL_DATETIME_SERIALIZER);
}
```

The configuration beans are applied in a specific order, which we can control using the @Order annotation. This elegant approach is suitable if we want to configure the *ObjectMapper* from different configurations or modules.

Custom: Overriding the ObjectMapper

Date Serialize in SpringBoot & Dont serialize null value:

If we want to have full control over the configuration, there are several options that will disable the auto-configuration and allow only our custom configuration to be applied.

روش اول :Overriding

The simplest way to **override** the default configuration is to define an **ObjectMapper** bean and to mark it as **@Primary**.

```
@Bean
@Primary
public ObjectMapper objectMapper() {
    JavaTimeModule module = new JavaTimeModule();
    module.addSerializer(LOCAL_DATETIME_SERIALIZER);
    return new ObjectMapper()
        .setSerializationInclusion(JsonInclude.Include.NON_NULL)
        .registerModule(module);
}
```

روش دوم :Overriding

jackson2ObjectMapperBuilder

```
@Bean
public Jackson2ObjectMapperBuilder jackson2ObjectMapperBuilder() {
    return new Jackson2ObjectMapperBuilder().serializers(LOCAL_DATETIME_SERIALIZER)
        .serializationInclusion(JsonInclude.Include.NON_NULL);
}
```

روش سوم :Overriding

MappingJackson2HttpMessageConverter

```
@Bean
public MappingJackson2HttpMessageConverter mappingJackson2HttpMessageConverter() {
    Jackson2ObjectMapperBuilder builder = new
    Jackson2ObjectMapperBuilder().serializers(LOCAL_DATETIME_SERIALIZER)
        .serializationInclusion(JsonInclude.Include.NON_NULL);
    return new MappingJackson2HttpMessageConverter(builder.build());
}
```

It will configure two options by default:

- disable MapperFeature.DEFAULT_VIEW_INCLUSION
- disable DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES

Map Serialization/ Deserialization with Jackson

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.13.3</version>
</dependency>
```

Map<String, String> Serialization

ObjectMapper is Jackson's serialization mapper

```
Map<String, String> map = new HashMap<>();
map.put("key", "value");

ObjectMapper mapper = new ObjectMapper();
String jsonResult = mapper.writerWithDefaultPrettyPrinter()
    .writeValueAsString(map);
```

```
{
  "key" : "value"
}
```

Map<Object, String> Serialization: **@JsonValue / extend JsonSerializer and @Override Serialize()**

we annotate **toString()** with **@JsonValue** to ensure **Jackson uses** this custom **toString()** **when serializing**:

```

public class MyPair {

    private String first;
    private String second;

    @Override
    @JsonValue
    public String toString() {
        return first + " and " + second;
    }

    // standard getter, setters, equals, hashCode, constructors
}

```

```

public class MyPairSerializer extends JsonSerializer<MyPair> {

    private ObjectMapper mapper = new ObjectMapper();

    @Override
    public void serialize(MyPair value,
        JsonGenerator gen,
        SerializerProvider serializers)
        throws IOException, JsonProcessingException {


        StringWriter writer = new StringWriter();
        mapper.writeValue(writer, value);
        gen.writeFieldName(writer.toString());
    }
}

```

```

@JsonSerialize(keyUsing = MyPairSerializer.class)
Map<MyPair, String> map;

```



```
map = new HashMap<>();  
MyPair key = new MyPair("Abbott", "Costello");  
map.put(key, "Comedy");  
  
String jsonResult = mapper.writerWithDefaultPrettyPrinter()  
    .writeValueAsString(map);
```

```
{  
  "Abbott and Costello" : "Comedy"  
}
```

Map<Object, Object> Serialization

The most complex case is serializing a `Map<Object, Object>`, but most of the work is already done. Let's use Jackson's `MapSerializer` for our map, and `MyPairSerializer`, from the previous section, for the map's key and value types:

```
@JsonSerialize(keyUsing = MapSerializer.class)  
Map<MyPair, MyPair> map;  
  
@JsonSerialize(keyUsing = MyPairSerializer.class)  
MyPair mapKey;  
  
@JsonSerialize(keyUsing = MyPairSerializer.class)  
MyPair mapValue;
```



```
mapKey = new MyPair("Abbott", "Costello");
mapValue = new MyPair("Comedy", "1940s");
map.put(mapKey, mapValue);

String jsonResult = mapper.writerWithDefaultPrettyPrinter()
    .writeValueAsString(map);
```

```
{
  "Abbott and Costello" : "Comedy and 1940s"
}
```

@JsonValue Annotation

@JsonValue: ensure Jackson uses this custom toString() when serializing the @JsonValue property is used for serialization.

```
public class Fruit {
    public String variety;

    @JsonKey
    public String name;

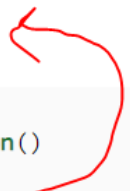
    public Fruit(String variety, String name) {
        this.variety = variety;
        this.name = name;
    }

    @JsonValue
    public String getFullName() {
        return this.variety + " " + this.name;
    }
}
```

```
private static final Fruit FRUIT1 = new Fruit("Alphonso", "Mango");
private static final Fruit FRUIT2 = new Fruit("Black", "Grapes");
private static final ObjectMapper OBJECT_MAPPER = new ObjectMapper();
```

the @JsonValue property is used for serialization.

```
@Test
public void givenObject_WhenSerialize_ThenUseJsonValueForSerialization()
    throws JsonProcessingException {
    String serializedValueForFruit1 = OBJECT_MAPPER.writeValueAsString(FRUIT1);
    Assertions.assertEquals("\"Alphonso Mango\"", serializedValueForFruit1);
    String serializedValueForFruit2 = OBJECT_MAPPER.writeValueAsString(FRUIT2);
    Assertions.assertEquals("\"Black Grapes\"", serializedValueForFruit2);
}
```



@JsonKey & @JsonValue in maps Serialization

```
@Test
public void givenMapWithObjectKeys_WhenSerialize_ThenUseJsonKeyForSerialization()
    throws JsonProcessingException {
    // Given
    Map<Fruit, String> selectionByFruit = new LinkedHashMap<>();
    selectionByFruit.put(FRUIT1, "Hagrid");
    selectionByFruit.put(FRUIT2, "Hercules");
    // When
    String serializedValue = OBJECT_MAPPER.writeValueAsString(selectionByFruit);
    // Then
    Assertions.assertEquals("{\"Mango\":\"Hagrid\",\"Grapes\":\"Hercules\"}", serializedValue);
}
```

```
@Test
public void givenMapWithObjectValues_WhenSerialize_ThenUseJsonValueForSerialization()
    throws JsonProcessingException {
    // Given
    Map<String, Fruit> selectionByPerson = new LinkedHashMap<>();
    selectionByPerson.put("Hagrid", FRUIT1);
    selectionByPerson.put("Hercules", FRUIT2);
    // When
    String serializedValue = OBJECT_MAPPER.writeValueAsString(selectionByPerson);
    // Then
    Assertions.assertEquals("{\"Hagrid\":\"Alphonso Mango\",\"Hercules\":\"Black Grapes\"}",
        serializedValue);
}
```

Map<String, String> Deserialization

```
private ObjectMapper mapper = new ObjectMapper();

String jsonInput = "{\"key\": \"value\"}";
TypeReference<HashMap<String, String>> typeRef
    = new TypeReference<HashMap<String, String>>() {};
Map<String, String> map = mapper.readValue(jsonInput, typeRef);
```

Jackson's **TypeReference**, which we'll use in all of our **deserialization** examples to describe the type of our destination Map

Map<Object, String> Deserialization: **Extend KeyDeserializer**

```
String jsonInput = "{\"Abbott and Costello\" : \"Comedy\"}";

TypeReference<HashMap<MyPair, String>> typeRef
    = new TypeReference<HashMap<MyPair, String>>() {};
Map<MyPair, String> map = mapper.readValue(jsonInput, typeRef);
```


Constructor:

```
public MyPair(String both) {  
    String[] pairs = both.split("and");  
    this.first = pairs[0].trim();  
    this.second = pairs[1].trim();  
}
```

```
public class MyPairDeserializer extends KeyDeserializer {  
  
    @Override  
    public MyPair deserializeKey(  
        String key,  
        DeserializationContext ctxt) throws IOException,  
        JsonProcessingException {  
  
        return new MyPair(key);  
    }  
}
```

```
public class ClassWithAMap {  
  
    @JsonProperty("map")  
    @JsonDeserialize(keyUsing = MyPairDeserializer.class)  
    private Map<MyPair, String> map;  
  
    @JsonCreator  
    public ClassWithAMap(Map<MyPair, String> map) {  
        this.map = map;  
    }  
  
    // public getters/setters omitted  
}
```

```
String jsonInput = "{\\\"Abbott and Costello\\\":\\\"Comedy\\\"}";  
  
ClassWithAMap classWithMap = mapper.readValue(jsonInput,  
    ClassWithAMap.class);
```

Map<Object,Object> Deserialization

همه مراحل مثل سکشن قبل است و در آخر:

```
String jsonInput = "{\\\"Abbott and Costello\\\" : \\\"Comedy and 1940s\\\"}";  
TypeReference<HashMap<MyPair, MyPair>> typeRef  
    = new TypeReference<HashMap<MyPair, MyPair>>() {};  
Map<MyPair, MyPair> map = mapper.readValue(jsonInput, typeRef);
```