**🌱 Spring REST — JSON (JavaScript Object Notation)**

**🔹 What is JSON?**

JSON stands for **JavaScript Object Notation**.  
It is a **lightweight**, **language-independent** format used to **transfer data** between client and server.

You can think of JSON as a **simple way to represent Java objects** (or data structures) in **text form** using **key–value pairs**.

**🔹 Basic Format**

JSON represents data in the form of:

"key" : "value"

Example:

{

"name": "Pavan",

"age": 25,

"city": "Hyderabad"

}

**🔹 Rules & Structure**

1️⃣ **Each data item** is represented as a **key–value pair**  
👉 Keys must always be inside **double quotes ("")**  
👉 Values can be:

* String → "Pavan"
* Number → 25
* Boolean → true or false
* Object → another {} block
* Array/List → [] block

2️⃣ **Curly braces {}**  
Represent **one complete object** or a **sub-object**.  
Example:

{

"student": {

"id": 101,

"name": "John"

}

}

👉 Here, "student" is the key, and its value is another **object**.

3️⃣ **Square brackets []**  
Represent a **collection (array/list/set)** of values.  
Example:

{

"languages": ["Java", "Python", "C++"]

}

👉 JSON treats all these types (array/list/set) simply as **arrays**.

4️⃣ **1-D Array (One-Dimensional Array)**  
Means a list of elements like:

["A", "B", "C"]

5️⃣ **2-D Array (Two-Dimensional Array)**  
Means a collection of key–value pairs — similar to a **Map** in Java.  
Example:

{

"101": {"name": "Pavan", "age": 25},

"102": {"name": "Nitin", "age": 24}

}

👉 Each key (like 101, 102) maps to another object — that’s why a **Map** is called a **2-D structure** in JSON.

6️⃣ **HAS-A Property / Nested Object**  
Sometimes one object contains another object — this is called a **HAS-A relationship**.  
Example:

{

"employee": {

"id": 1,

"name": "Ravi",

"address": {

"city": "Pune",

"state": "Maharashtra"

}

}

}

👉 employee HAS-A address.  
The inner {} block (address) is a **sub-object node**.

**🔹 Summary Table**

| **Concept** | **Symbol** | **Meaning** |
| --- | --- | --- |
| Object | {} | Represents a single object or sub-object |
| Array/List/Set | [] | Represents multiple elements (1-D array) |
| Map | { key : value, ... } | Acts like 2-D array |
| Key | Always in quotes | "key" |
| Value | Can be any type | "value", true, 123, {}, [] |

**🌱 Java Object → JSON → XML**

**🧩 In Java:**

We create an object like this —

Customer customer = new Customer(10, "Sachin", "IND", 53.4f);

Here:

* 10 → customer number
* "Sachin" → name
* "IND" → country
* 53.4f → average score

**🧩 Same Data in JSON (Lightweight)**

{

"cno": 10,

"cname": "Sachin",

"country": "IND",

"avg": 53.4

}

📘 Notes:

* Uses **key–value** pairs.
* Uses **less syntax** (no closing tags).
* Very **compact**, **readable**, and **easy to parse**.
* Commonly used in **web APIs** and **REST services** for data transfer.

**🧩 Same Data in XML (Heavyweight)**

<customer>

<cno>10</cno>

<cname>Sachin</cname>

<country>IND</country>

<avg>53.4</avg>

</customer>

📘 Notes:

* Uses **opening** and **closing tags**.
* More **verbose** (more characters → heavier).
* Used in **SOAP web services** and **configuration files**.
* Still human-readable but not as compact as JSON.

**⚖️ HTML vs JSON — Difference**

| **Feature** | **HTML** | **JSON** |
| --- | --- | --- |
| Purpose | **Display data** on web pages | **Describe/transfer data** between systems |
| Usage | Used with **CSS/JS** for UI | Used with **APIs** for backend communication |
| Format | Has **tags** like <p>, <div> | Has **key–value pairs** like "name": "Sachin" |
| Example | <h1>Hello</h1> | { "message": "Hello" } |
| Readability | Focused on **visual structure** | Focused on **data structure** |
| Heaviness | Relatively **heavier** | **Lightweight** |
| Main Use | Frontend / presentation layer | Data exchange between client & server |

**🧠 In Simple Words:**

* **HTML/CSS** → “How things look” (used to *display* data).
* **JSON/XML** → “What the data is” (used to *describe* and *transfer* data).

## ⚙️ Step-by-Step Fix for “ObjectMapper cannot be resolved”

### 🧩 1️⃣ Add the Correct Maven Dependency

Make sure your **pom.xml** has this dependency **inside <dependencies>**:

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.14.2</version>

</dependency>

✅ Also check that it’s **inside the <project>...</project> block** — not outside by mistake.

### 🧹 2️⃣ Clean and Rebuild Maven Project

In **Eclipse**, do the following:

1. Right-click on your project
2. Select 👉 **Maven → Update Project...**
3. Check **“Force Update of Snapshots/Releases”**
4. Click **OK**

Then again:

* Right-click project → **Run As → Maven Clean**
* Right-click project → **Run As → Maven Install**

This ensures all dependencies download properly.

### 🧰 3️⃣ Check Your Imports

At the top of your App.java, make sure you have:

import com.fasterxml.jackson.databind.ObjectMapper;

If this line gives an error even after updating Maven →  
then your **Maven repository cache** may be broken.

### 🔄 4️⃣ (If still not fixed) Clear Maven Cache Manually

Sometimes Eclipse doesn’t pick the jar even though it’s downloaded.  
Try this quick reset:

1. Close Eclipse
2. Go to your local Maven repository folder  
   👉 usually in C:\Users\<your-username>\.m2\repository\com\fasterxml\jackson
3. Delete the **jackson** folder
4. Reopen Eclipse
5. Again do **Maven → Update Project**

This will re-download fresh Jackson jars.

### ✅ 5️⃣ Verify by Checking "Referenced Libraries"

In your Project Explorer → Expand “**Maven Dependencies**” →  
You should see:

jackson-core-2.14.2.jar

jackson-databind-2.14.2.jar

jackson-annotations-2.14.2.jar

If these 3 jars appear, you’re good to go ✅

### 🧪 6️⃣ Sample Working Code for Final Verification

package in.orcas.JsonToPojo;

import java.io.File;

import java.io.IOException;

import com.fasterxml.jackson.databind.ObjectMapper;

public class App {

public static void main(String[] args) {

try {

ObjectMapper mapper = new ObjectMapper();

Customer customer = mapper.readValue(new File("data/sample-lite.json"), Customer.class);

System.out.println(customer);

} catch (IOException e) {

e.printStackTrace();

}

}

}

Would you like me to also show how to convert **Java → JSON** (the reverse process)  
using the same ObjectMapper — so you can test that Jackson works both ways once this is fixed?

##### You said:

now give me some notes for the example i will keep it for the future

###### ChatGPT said:

Perfect 👍 — here’s a **clean, simple, and complete note** you can save for your future reference.  
It explains everything about the **Jackson example** — why we use it, what happens internally, and how it connects with JSON/Java conversion.

## 🧾 Notes: Converting JSON ↔ Java using Jackson API

### 🔹 What is Jackson?

Jackson is a **Java library** (API) used to **convert JSON data into Java objects and vice versa**.  
It is lightweight, fast, and a standard choice in Spring Boot and REST APIs.

### 🔹 Commonly Used Jackson Modules

| **Module** | **Purpose** |
| --- | --- |
| **jackson-core** | Basic core functionality for reading/writing JSON |
| **jackson-databind** | Main module for converting JSON ↔ Java objects |
| **jackson-annotations** | Used for annotations like @JsonProperty, @JsonIgnore etc. |

✅ Usually, adding **jackson-databind** in your pom.xml automatically brings all 3.

### 🔹 Maven Dependencies

<!-- Jackson API -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.14.2</version>

</dependency>

<!-- Lombok (to avoid writing getters/setters) -->

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.30</version>

<scope>provided</scope>

</dependency>

### 🔹 Example Class (POJO)

package in.orcas.model;

import lombok.Data;

@Data

public class Customer {

private Integer cno;

private String cname;

private String country;

private Float avg;

}

Lombok’s @Data → automatically creates **getters, setters, toString(), equals(), and hashCode()** methods.

### 🔹 JSON File (sample-lite.json)

{

"cno": 10,

"cname": "Sachin",

"country": "IND",

"avg": 53.4

}

### 🔹 Java Code (Convert JSON → Java Object)

package in.orcas.JsonToPojo;

import java.io.File;

import java.io.IOException;

import com.fasterxml.jackson.databind.ObjectMapper;

import in.orcas.model.Customer;

public class App {

public static void main(String[] args) {

try {

// 1. Create ObjectMapper

ObjectMapper mapper = new ObjectMapper();

// 2. Read JSON file and convert to POJO

Customer customer = mapper.readValue(new File("data/sample-lite.json"), Customer.class);

// 3. Print the Java object

System.out.println(customer);

} catch (IOException e) {

e.printStackTrace();

}

}

}

### 🔹 Program Flow

1. **ObjectMapper** → acts as a converter/translator.
2. **readValue()** → reads JSON file and creates a Java object (Customer).
3. Jackson automatically maps JSON keys → Java field names (based on name matching).
4. Finally, System.out.println(customer) calls Lombok’s generated toString() method and prints the values.

### 🔹 Reverse Conversion (Java → JSON)

You can also write the Java object **back to a JSON file**:

Customer customer = new Customer();

customer.setCno(20);

customer.setCname("Virat");

customer.setCountry("IND");

customer.setAvg(65.8f);

// Convert Java object to JSON and save to file

mapper.writeValue(new File("data/output.json"), customer);

✅ This creates a JSON file like:

{

"cno": 20,

"cname": "Virat",

"country": "IND",

"avg": 65.8

}

### 🔹 Summary Table

| **Operation** | **Jackson Method** | **Description** |
| --- | --- | --- |
| JSON → Java | readValue() | Converts JSON data into POJO |
| Java → JSON | writeValue() | Converts POJO to JSON format |
| Format Output | writerWithDefaultPrettyPrinter() | Produces pretty-formatted JSON |

Example:

mapper.writerWithDefaultPrettyPrinter().writeValue(new File("data/output.json"), customer);

### 🔹 Real-Time Use Case

In REST APIs:

* When a client sends JSON → Jackson converts it into Java (Deserialization)
* When a server sends response → Jackson converts Java → JSON (Serialization)

Spring Boot automatically uses Jackson internally for this purpose.

Eg: JsonToPojo

**🧩 Internal Working of Jackson (Very Important Note)**

When Jackson converts JSON → Java Object (POJO),  
it **uses Java Bean naming conventions** internally.

**🔹 How it Works Internally**

1. Jackson reads each **key** from the JSON file.
2. For every key, it searches for a **matching Java property** in the class.
3. Then it looks for the **corresponding setter method** to assign that value.

**🔸 Example:**

**JSON Input:**

{

"id": 10,

"name": "Sachin"

}

**Java POJO:**

public class Customer {

private Integer id;

private String name;

// Getters and Setters

public void setId(Integer id) { this.id = id; }

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

}

**Internal Binding Process:**

"id" → finds → setId(10)

"name" → finds → setName("Sachin")

✅ So Jackson internally calls **setXXX(value)** for each key in JSON  
and assigns the corresponding value to the Java property.

**🧠 Remember**

| **JSON Key** | **Java Method Called** | **Java Property Updated** |
| --- | --- | --- |
| "id" | setId(10) | id = 10 |
| "name" | setName("Sachin") | name = "Sachin" |

💬 **In simple words:**

“To convert JSON → Java, Jackson reads JSON keys and calls the matching setter (setXXX()) for each property.”

Similarly,

“To convert Java → JSON, Jackson calls getter methods (getXXX()) to read property values and write them as JSON keys.”