ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

Java APIs for JSON Processing

Phuc H. Duong

phuc@newai.vn

Textbook

- Bogunuva Mohanram Balachandar. RESTful Java Web Services - Third Edition (2017). Packt Publishing Limited.
  - Chapter 2: Java APIs for JSON Processing

Update Log

- 18/12/2019: release first version of this slide

Objective

- XML and JSON are the two popular formats used by RESTful web services today
- Within these two formats, JSON is widely adopted by many vendors because of its simplicity and light weight
- In this chapter, you will learn more about the JSON message format, how to represent real-life business data in the JSON format, and various processing tools and frameworks related to JSON

Objective

- The following topics are covered in this chapter:
  - A brief overview of JSON
  - Using the **JSR 353** – Java API for processing JSON
  - Using the **Jackson API** for processing JSON
  - Using the **Gson API** for processing JSON
  - **Java EE 8 Enhancements** for processing JSON

Outline

1. A brief overview of JSON
2. JSON Processing Tools and Frameworks
3. JSR 353 – Java API for processing JSON
4. Jackson API for processing JSON
5. Gson API for processing JSON
6. Java EE 8 Enhancements for processing JSON
7. Summary
8. Exercises

A brief overview of JSON

JSON

- JSON - JavaScript Object Notation - https://www.json.org
- JSON is a *lightweight*, *text-based*, *platform-neutral data interchange format* in which objects are represented in the attribute-value pair format

JSON data syntax

- JSON format is very simple by design
- It is represented by the following two data structures:
  - An **unordered collection** of the name-value pairs (representing an object)
  - An **ordered collection** of values (representing an array)

JSON data syntax

- An **unordered collection** of the name-value pairs (representing an object):
  - Attributes of an object and their values are represented in the **name-value pair** format
  - The name and the value in a pair are separated by a colon (:)
  - Names in an object are strings, and values may be of any of the valid JSON data types such as number, string, Boolean, array, object, or null
  - Each *name:value* pair in a JSON object is separated by a comma (,)
  - The entire object is enclosed in curly braces ({})

JSON data syntax

- An **unordered collection** of the name-value pairs (representing an object):
  - The example shows how you can represent the various attributes of a department, such as *departmentId*, *departmentName*, and *manager* in the JSON format

```
{
    "departmentId" : 10,
    "departmentName" : "IT",
    "manager" : "John Chen"
}
```

JSON data syntax

- An **ordered collection** of values (representing an array):
  - Arrays are enclosed in square brackets ([ ])
  - Their values are separated by a comma (,)
  - Each value in an array may be of a different type, including another array or an object

JSON data syntax

- An **ordered collection** of values (representing an array):
  - The following example illustrates the use of array notation to represent the employees working in a department
  - You can also see an array of locations in the example

```
{
    "departmentName":"IT",
    "employees":[
        {"firstName":"John", "lastName":"Chen"},
        {"firstName":"Ameya", "lastName":"Job"},
        {"firstName":"Pat", "lastName":"Fay"} ],
    "location":["New York", "New Delhi"]
}
```

JSON – Data types (1)

- **Number**
  - This type is used for storing a signed decimal number that may optionally contain a fractional part
  - Both integer and floating point numbers are represented by using this data type
  - Example: `{"totalWeight": 123.456}`

JSON – Data types (2)

- **String**
  - This type represents a sequence of zero or more characters
  - Strings are surrounded with double quotation marks and support a backslash escaping syntax
  - Example: **`{"firstName": "Jobinesh"}`**

JSON – Data types (3)

- **Boolean**
  - This type represents either a true or a false value
  - The Boolean type is used for representing whether a condition is true or false, or to represent two states of a variable (true or false) in the code
  - Example: **`{"isValidEntry": true}`**

JSON – Data types (4)

- **Array**
  - This type represents an ordered list of zero or more values, each of which can be of any type
  - In this representation, comma-separated values are enclosed in square brackets
  - Example: `{"fruits": ["apple", "banana", "orange"]}`

JSON – Data types (5)

- **Object**
  - This type is an unordered collection of comma-separated attribute-value pairs enclosed in curly braces
  - All attributes must be strings and should be distinct from each other within that object
  - Example:

```
{
    "departmentId":10,
    "departmentName":"IT",
    "manager":"John Chen"
}
```

JSON – Data types (6)

- **null**
  - This type indicates an empty value, represented by using the word *null*
  - The following example uses *null* as the value for the *error* attribute of an object: **`{"error": null}`**

Sample JSON File

Sample JSON file

- The ***emp-array.json*** contains the JSON array of the employee objects


```
[
    {"employeeId":100,"firstName":"John","lastName":"Chen",
    "email":"john.chen@xxxx.com","hireDate":"2008-10-16"},

    {"employeeId":101,"firstName":"Ameya","lastName":"Job",
    "email":"ameya.job@xxx.com","hireDate":"2013-03-06"},

    {"employeeId":102,"firstName":"Pat","lastName":"Fay",
    "email":"pat.fey@xxx.com","hireDate":"2001-03-06"}
]
```

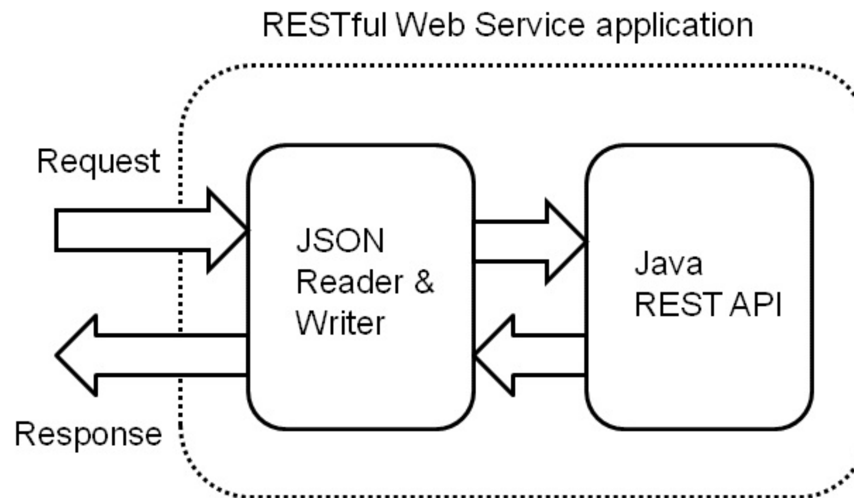
Processing JSON data

- Using Java RESTful web service frameworks, the **serialization** and **deserialization** of the request and response messages will be taken care of by the framework

- There are two important components in processing JSON data
  - JSON marshalling
  - JSON unmarshalling

Processing JSON data

- There are **2** important components in processing JSON data
  - JSON marshalling
    - converting Java object to JSON format
  - JSON unmarshalling
    - converting JSON format to Java Object



But… "Processing" what?

- The term "processing" means **reading**, **writing**, **querying**, and **modifying** JSON data
- There are **2** models for processing JSON data:
  - Object model
  - Streaming model

Processing JSON data

- **Object model**
  - The entire JSON data is read into memory in a tree format
  - This tree can be traversed, analyzed, or modified with appropriate APIs
  - As this approach loads the entire content in the memory first and then starts parsing, it ends up consuming more memory and CPU cycles
  - However, this model gives more flexibility while manipulating the content.

Processing JSON data

- **Streaming model**
  - Data can be read or written in blocks
  - Model does not read the entire JSON content into the memory to get started with parsing; rather, it reads one element at a time
  - Instead of letting the parser push the content to the client (push parser), the client can pull the information from the parser
  - The client can skip or stop reading contents in the middle of the process if it has finished reading the desired elements

Processing JSON data – Best pratice

- You may want to consider using streaming APIs in the following situations:
  - When the data is huge in size, and it is not feasible to load the entire content into the memory for processing the content
  - When partial processing is needed, and the data model is not fully available yet

JSON Processing Tools and Frameworks

JSON processing tools

- The **JSR 353** – Java API for processing JSON
- The **Jackson API** for processing JSON
- The **Gson API** for processing JSON
- **Java EE 8 Enhancements** for processing JSON

JSR 353 – Java API for processing JSON

JSR 353

- Java Specification Request (JSR) - **JSR 353** - Java API for JSON Processing
- **JSR 353 APIs** can be classified into two categories based on the processing model followed by the APIs:
  - Object model API
  - Streaming model API

JSR 353 – Object Model APIs

- This category of APIs generates an in-memory tree model for JSON data and then starts processing it as instructed by the client
- For some frequently used classes, refer to the textbook:
  - Chapter 2
  - Section: Processing JSON with JSR 353 object model APIs

http://docs.oracle.com/javaee/7/api/javax/json/package-summary.html

JSR 353 – Object Model APIs

- Practice section:
  - Refer to **Chapter 2** in textbook
  - Section: Processing JSON with JSR 353 object model APIs
  - We will practice how to generate object model from JSON, and vice versa

JSR 353 – Streaming APIs

- This model is designed to process a large amount of data in a more efficient way
  - Streaming APIs read and write the content serially at runtime in accordance with client calls, which makes them suitable for handling a large amount of data
- Streaming APIs are grouped in the *javax.json.stream* package in the JSR specification

JSR 353 – Streaming APIs

- Practice section:
  - Refer to **Chapter 2** in textbook
  - Section: Processing JSON with JSR 353 streaming APIs
  - We will practice how to parse and generate JSON data

Jackson API

Jackson API

- **Jackson** is a multipurpose data processing Java library
- It has additional modules for processing the data encoded in other popular formats:
  - Apache Avro (a data serialization system)
  - Concise Binary Object Representation (CBOR)
  - a binary JSON format
  - Smile (a binary JSON format)
  - XML
  - comma-separated values (CSV)
  - YAML

Jackson API

- Jackson provides the following **3** methods for processing JSON:
  - **Tree model APIs**
    - This method provides APIs for building a tree representation of a JSON document
  - **Data binding API**
    - This method provides APIs for converting a JSON document into and from Java objects
  - **Streaming API**
    - This method provides streaming APIs for reading and writing a JSON document

Jackson API

- Practice section:
  - Refer to **Chapter 2** in textbook
  - Section: Using the Jackson API for processing JSON
  - We will practice how to work with JSON data in **3** approaches (Tree model APIs, Data binding API, and Streaming API)

Gson API

Gson

- **Gson** is an open source Java library that can be used for converting Java objects into JSON representations, and vice versa
- The **Gson library** was originally developed by **Google** for its internal use and later open sourced under the terms of Apache License 2.0

Gson

- Practice section:
  - Refer to **Chapter 2** in textbook
  - Section: Using the Gson API for processing JSON
  - Like JSR 353, we will practice how to work with Gson object model APIs and Gson streaming APIs

Java EE 8 enhancements for processing JSON

Java EE 8

- Java EE 8 introduces JSR 374 – Java API for JSON Processing 1.1
- And then, JSR 367 Java API for JSON Binding (**JSON-B**) brings in the capability of marshalling/unmarshalling Java objects to JSON representation

Java EE 8

- Practice section:
  - Refer to **Chapter 2** in textbook
  - Section: Java EE 8 enhancements for processing JSON
  - We will use JSR 374 and JSR 367 for working JSON data

Summary

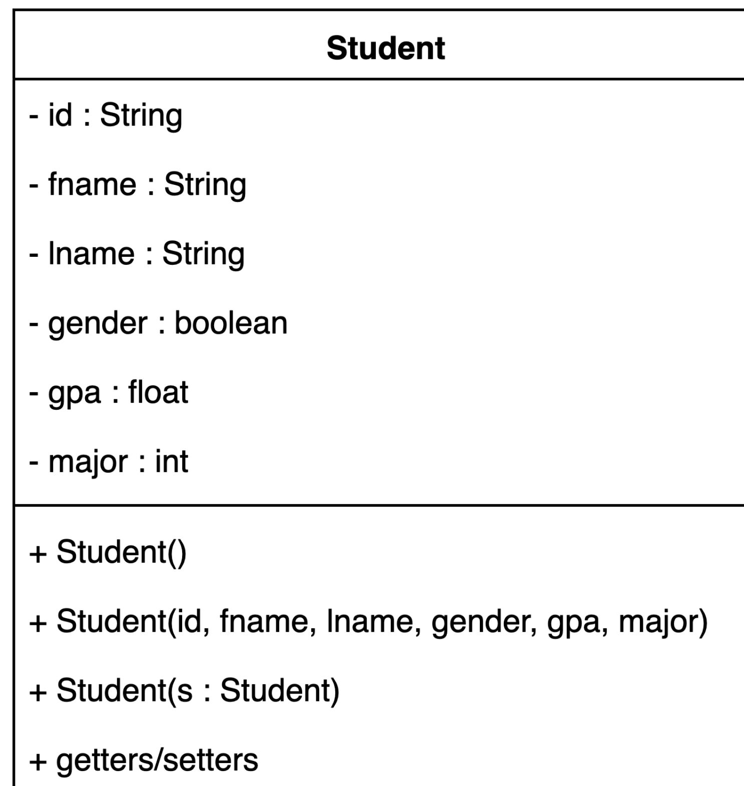
Summary

- You were introduced to the various processing models for the JSON content and some of the popular Java-based JSON processing frameworks
- This chapter is essential for understanding how the JSON-based request and response messages are bound to the Java model while building REST APIs
- *In the next chapter, we will build our first REST service by using JAX-RS APIs*

Exercises

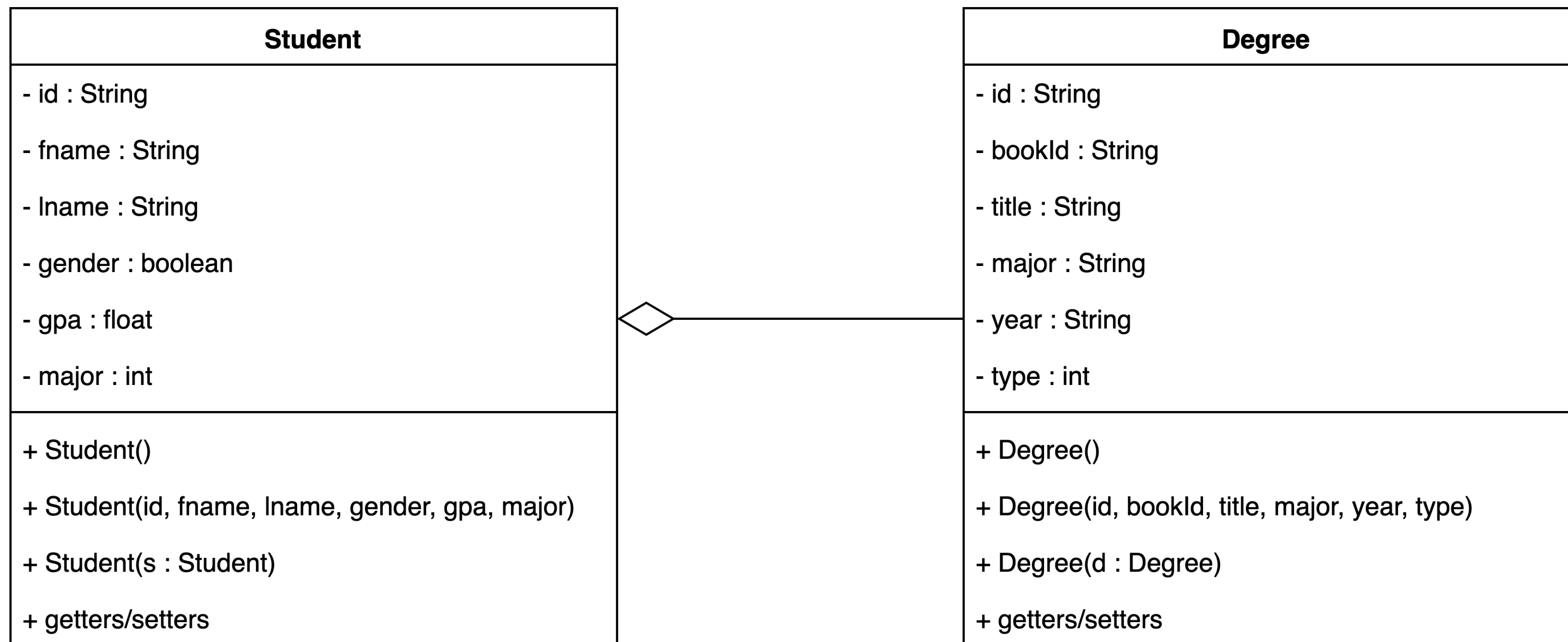
Question 1

- Give a class UML as below, you are required to apply JSON processing tools to generate and parse JSON data which contains a list of Student objects.



Question 2

- Give a class UML as below, you are required to apply JSON processing tools to generate and parse JSON data which contains a list of Student together with Degree objects.



Question 3

- Continuing Q1 and Q2, let's assume that you have a list of over 3K elements of <Student> and <Student, Degree>
- You are required to program an API to read a specific element given an *id* of Student
- And given an *id* of Degree, trace back the Student info

Question 4*

- Continuing Q3, how can you deploy your API as SaaS (Software as a Service) approach, so that a client can remotely access to your API.

END OF CHAPTER