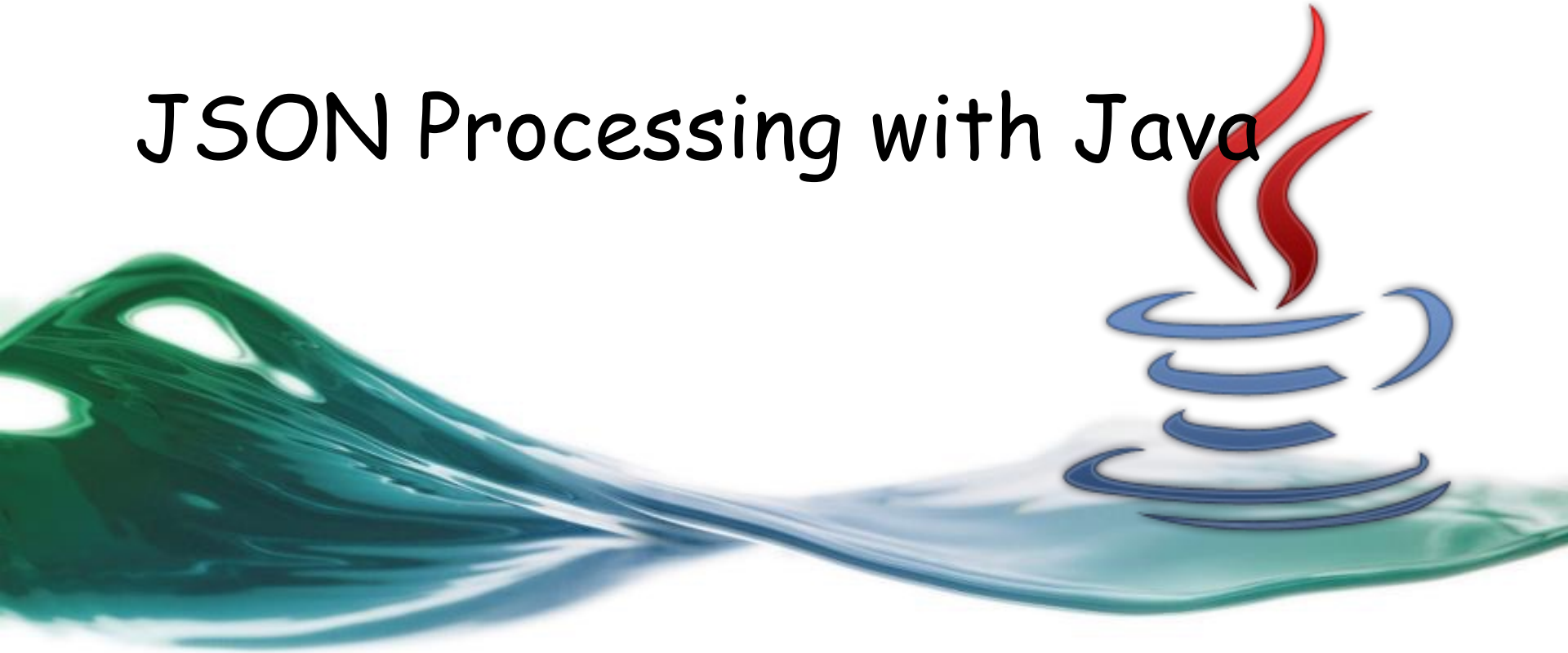
Java Programming Course

JSON Processing with Java



Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Session objectives

JSON Introduction

JSON structure

Java API for JSON Processing

- Object Model API
- Streaming API

Libraries:

- Jackson
- G-son
- Yasson

JSON Introduction

http://www.json.org/

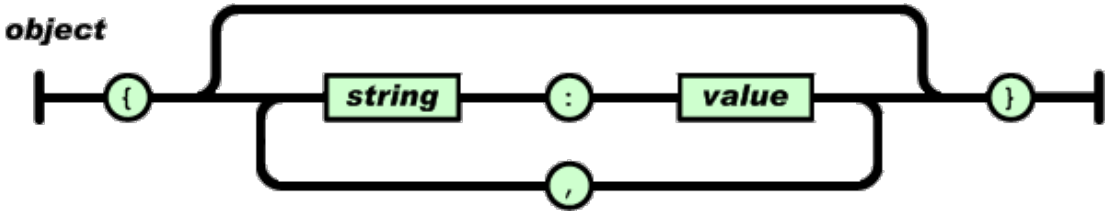
- **JSON** (JavaScript Object Notation) is a lightweight (nhẹ) data-interchange format.
  - It is easy for humans to read and write.
  - It is easy for machines to parse (phân tích) and generate (tạo).
  - It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999.
  - JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.
- JSON is often used in Ajax applications, configurations, databases, and RESTful web services. All popular websites offer JSON as the data exchange format with their RESTful web services.

JSON structure (1)

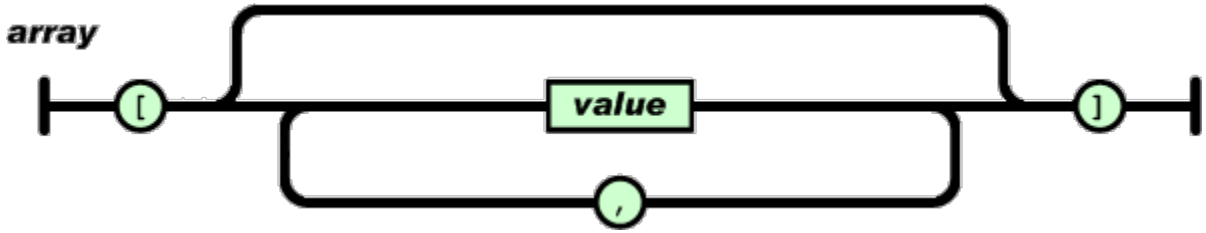
- JSON is built on two structures:
  - A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
  - An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

JSON structure (2)

- In JSON, they take on these forms:
  - An *object* is an unordered set of name/value pairs. An object begins with { (left brace) and ends with } (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (comma).

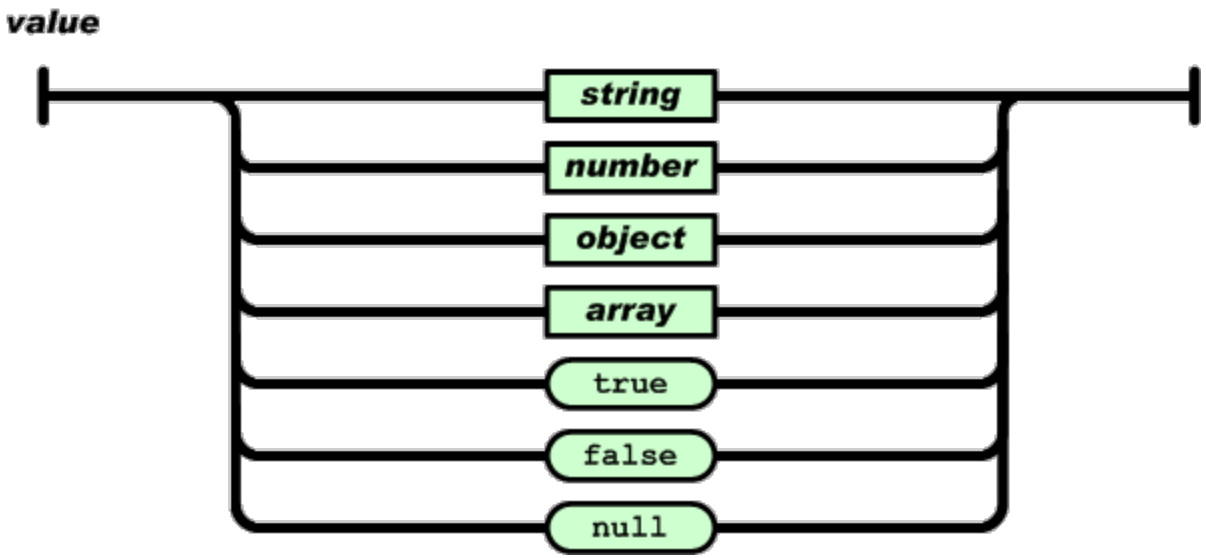


  - An *array* is an ordered collection of values. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by , (comma).



JSON structure (3)

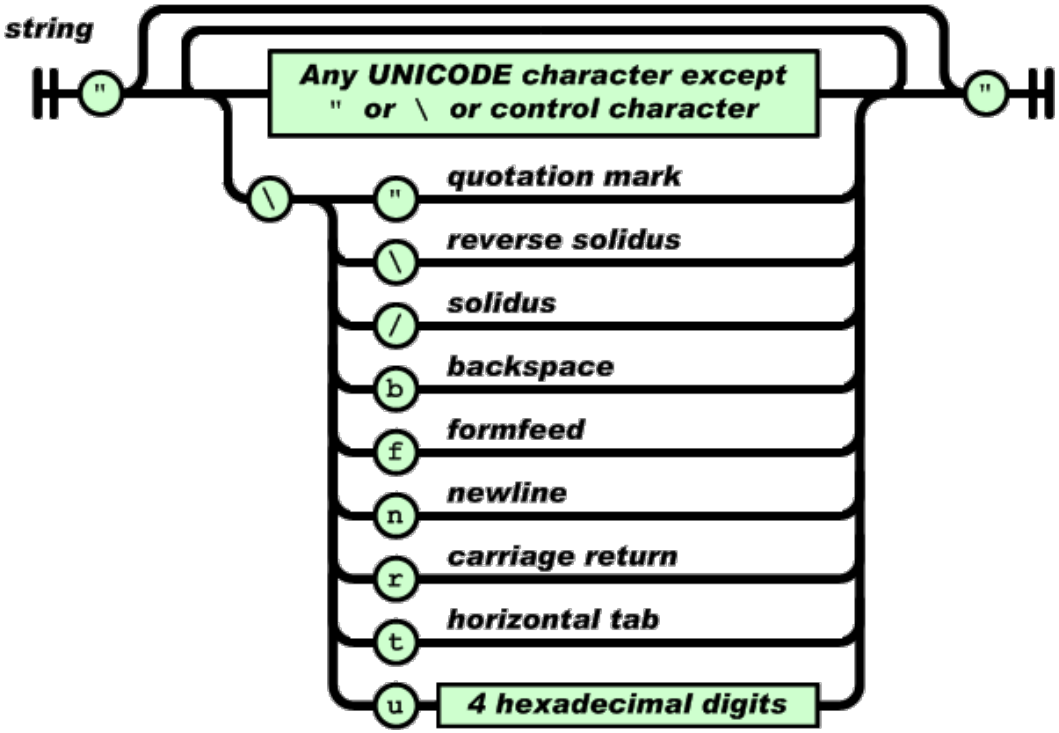
- A *value* can be a *string* in double quotes, or a *number*, or true or false or null, or an *object* or an *array*. These structures can be nested.



JSON structure (4)

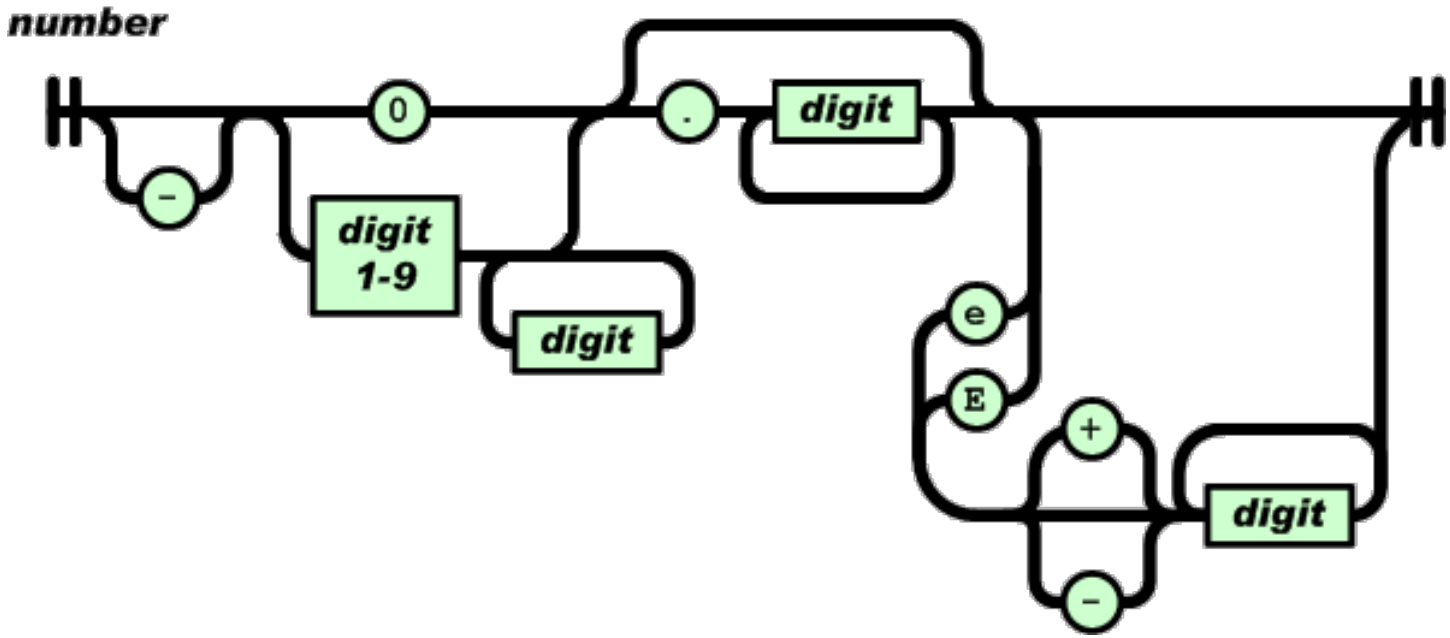
- A *string* is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash (\) escapes. A character is represented as a single character string. A string is very much like a C or Java string.

- \"
- \\
- \/
- \b
- \f
- \n
- \r
- \t
- \u



JSON structure (5)

- A *number* is very much like a C or Java number, except that the octal and hexadecimal formats are not used.



Sample json document & rule

cust.json

```
1  {
2      "firstName": "John",
3      "lastName": "Smith",
4      "age": 25,
5      "address": {
6          "streetAddress": "21 2nd Street",
7          "city": "New York",
8          "state": "NY",
9          "postalCode": 10021
10     },
11     "phoneNumbers": [
12         {
13             "type": "home",
14             "number": "212 555-1234"
15         },
16         {
17             "type": "fax",
18             "number": "646 555-4567"
19         }
20     ]
21 }
```

object
    { }
    { *members* }
members
    pair
    pair , *members*
pair
    string : *value*
array
    [ ]
    [*elements*]
elements
    value
    value , *elements*
value
    string
    number
    object
    array
    true
    false
    null

Java API for JSON Processing

- The Java API for JSON Processing (JSR 353) provides portable APIs to parse (phân tích), generate (tạo), transform (chuyển đổi), and query (truy vấn) JSON using Object model and Streaming APIs.
- It produces and consumes JSON text in a streaming fashion (similar to StAX API for XML) and allows to build a Java object model for JSON text using API classes (similar to DOM API for XML).
- Ref: https://jakarta.ee/specifications/jsonp/2.0/apidocs/

JSON Processing - The Object Model API (1)

- The Object Model API
  - The object model API creates a random-access, tree-like structure that represents the JSON data in memory. The tree can then be navigated and queried.
  - This programming model is the most flexible and enables processing that requires random access to the complete contents of the tree. However, it is often not as efficient as the streaming model and requires more memory.
  - The object model API is similar to the Document Object Model (DOM) API for XML.
  - It is a high-level API that provides immutable object models for JSON object and array structures. These JSON structures are represented as object models using the Java types JsonObject and JsonArray.

JSON Processing - The Object Model API (2)

- The main classes and interfaces in the object model API

Class or Interface	Description
Json	Contains static methods to create JSON readers, writers, builders, and their factory objects.
JsonGenerator	Writes JSON data to a stream one value at a time.
JsonReader	Reads JSON data from a stream and creates an object model in memory.
JsonObjectBuilder	Create an object model or an array model in memory by adding values from application code.
JsonArrayBuilder	
JsonWriter	Writes an object model from memory to a stream.
JsonValue	Represent data types for values in JSON data.
JsonObject	
JsonArray	
JsonString	
JsonNumber	

JSON Processing - The Object Model API (3)

- JsonObject, JsonArray, JsonString, and JsonNumber are subtypes of JsonValue. These are constants defined in the API for null, true, and false JSON values.
- The object model API uses builder patterns to create these object models from scratch. Application code can use the interface JsonObjectBuilder to create models that represent JSON objects. The resulting model is of type JsonObject. Application code can use the interface JsonArrayBuilder to create models that represent JSON arrays. The resulting model is of type JsonArray.
- These object models can also be created from an input source *(such as InputStream or Reader)* using the interface JsonReader. Similarly, these object models can be written to an output source *(such as OutputStream or Writer)* using the class JsonWriter.

Mapping between JSON and Java entities

JSON	Java
string	java.lang.String
number	java.lang.Number
true|false	java.lang.Boolean
null	null
array	java.util.List
object	java.util.Map

On decoding:

The default concrete class of *java.util.List* is *org.json.simple.JSONArray*

The default concrete class of *java.util.Map* is *org.json.simple.JSONObject*.

Encoding JSON in Java (1)

```java
public static void main(String[] args) {
    JsonObjectBuilder jsonBuilder = Json.createObjectBuilder();
    Employee employee = new Employee(1L, "John Nguyen", 1000d);
    JsonObject jo = jsonBuilder
            .add("id", employee.getId())
            .add("name", employee.getName())
            .add("salary", employee.getSalary())
            .build();

    // Get the JSON string from the JsonObject
    // Create a StringWriter to store the formated JSON
    StringWriter stringWriter = new StringWriter();

    // Create a JsonWriter with pretty printing (indentation) configuration
    JsonWriter jsonWriter = Json.createWriterFactory(Collections.singletonMap(JsonGenerator.PRETTY_PRINTING, true))
            .createWriter(stringWriter);
    jsonWriter.write(jo);
    jsonWriter.close();

    System.out.println(stringWriter);
}
```

```
{
    "id": 1,
    "name": "John Nguyen",
    "salary": 1000.0
}
```

```xml
<dependencies>
    <!-- https://mvnrepository.com/artifact/jakarta.json/jakarta.json-api -->
      <dependency>
          <groupId>jakarta.json</groupId>
          <artifactId>jakarta.json-api</artifactId>
          <version>2.1.3</version>
      </dependency>
      <!-- https://mvnrepository.com/artifact/org.eclipse.parsson/parsson -->
      <dependency>
          <groupId>org.eclipse.parsson</groupId>
          <artifactId>parsson</artifactId>
          <version>1.1.5</version>
      </dependency>
</dependencies>
```

Encoding JSON in Java (2)

```java
JsonObjectBuilder jsonBuilder = Json.createObjectBuilder();
JsonObject jsonObject = jsonBuilder
        .add("firstName", "John")
        .add("lastName", "Smith")
        .add("age", 25)
        .add("address", Json.createObjectBuilder()
                .add("streetAddress", "21 2nd Street")
                .add("city", "NewYork")
                .add("state", "NY")
                .add("postalCode", "10021"))
        .add("phoneNumber", Json.createArrayBuilder()
                .add(Json.createObjectBuilder()
                        .add("type", "home")
                        .add("number", "212 555-1234"))
                .add(Json.createObjectBuilder()
                        .add("type", "fax")
                        .add("number", "646 555-4567")))
        .build();
```

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "NewYork",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumber": [
        {
            "type": "home",
            "number": "212 555-1234"
        },
        {
            "type": "fax",
            "number": "646 555-4567"
        }
    ]
}
```

Decoding JSON in Java (1)

```java
public static void main(String[] args) {

    String json = "{\"id\":1,\"name\":\"John Nguyen\",\"salary\":1000.0}";
    JsonReader reader = Json.createReader(new StringReader(json));
    JsonObject jo = reader.readObject();

    Employee employee = new Employee(jo.getJsonNumber("id").longValue(),
            jo.getString("name"),
            jo.getJsonNumber("salary").doubleValue());
    System.out.println(employee);
}
```

```
Console ×  Problems  Debug Shell
<terminated> Exer01 [Java Application] C:\Java\jdk-17.0.2\bin\javaw.exe (Jul 14, 2022, 8:48:3
Employee [id=1, name=John Nguyen, salary=1000.0]
```

Decoding JSON in Java (2)

```
public static void main(String[] args) throws FileNotFoundException {
    String json = "[{\"id\":10001,\"name\":\"John Smith\",\"salary\":10000.0},"
            + "{\"id\":10002,\"name\":\"Tom Cruise\",\"salary\":20000.0}]";
    JsonReader reader = Json.createReader(new StringReader(json));
    List<Employee> employees = new ArrayList<>();
    JsonArray ja = reader.readArray();
    for(JsonObject jo:ja.getValuesAs(JsonObject.class)) {
        JsonNumber id = jo.getJsonNumber("id");
        String name = jo.getString("name");
        JsonNumber sal = jo.getJsonNumber("salary");
        employees.add(new Employee(id.longValue(), name, sal.doubleValue()));
    }
    System.out.println(employees);
}
```

```
[Person [id=10001, name=John Smith, salary=10000.0], Person [id=10002, name=Tom Cruise, salary=20000.0]]
```

JSON Processing - The Streaming API (1)

- The Streaming API
  - The streaming API provides a way to parse and generate JSON in a streaming fashion.
  - It hands over parsing and generation control to the programmer.
  - The streaming API provides an event-based parser and allows an application developer to ask for the next event rather than handling the event in a callback. This gives a developer more procedural control over the JSON processing. Application code can process or discard the parser event and ask for the next event (pull the event).

JSON Processing - The Streaming API (2)

- The streaming API is similar to the Streaming API for XML (StAX) and consists of the interfaces JsonParser and JsonGenerator.
- JsonParser contains methods to parse JSON data using the streaming model.
- JsonGenerator contains methods to write JSON data to an output source. Table 2 lists the main classes and interfaces in the streaming API.

JSON Processing - The Streaming API (3)

- The main classes and interfaces in the streaming API

Class or Interface	Description
Json	Contains static methods to create JSON parsers, generators, and their factory objects.
JsonParser	Represents an event-based parser that can read JSON data from a stream.
JsonGenerator	Writes JSON data to a stream one value at a time

JSON Processing - The Streaming API (4)

- JsonParser provides forward, read-only access to JSON data using the pull parsing programming model. In this model, the application code controls the thread and calls methods in the parser interface to move the parser forward or to obtain JSON data from the current state of the parser.

  Ref:

https://jakarta.ee/specifications/jsonp/2.0/apidocs/jakarta.json/jakarta/json/stream/jsonparser

- JsonGenerator provides methods to write JSON data to a stream. The generator can be used to write name/value pairs in JSON objects and values in JSON arrays.

  Ref:

https://jakarta.ee/specifications/jsonp/2.0/apidocs/jakarta.json/jakarta/json/stream/jsongenerator

- The streaming API is a low-level API designed to process large amounts of JSON data efficiently. Other JSON frameworks *(such as JSON binding)* can be implemented using this API.

Encoding JSON in Java – Stream API

```java
public static void main(String[] args) throws FileNotFoundException {

    StringWriter writer = new StringWriter();
    JsonGenerator jsonGenerator = Json.createGenerator(writer);

    // Create a post
    Post post = new Post();
    post.setTitle("JsonGenerator Demo");
    post.setId(1);
    post.setDescription("JsonGenerator Description");
    post.setContent("Markdown content here");

    String[] tags = {
        "Java",
        "JSON"
    };
    // Create some predefined tags
    post.setTags(tags);

    jsonGenerator.writeStartObject(); // {
    jsonGenerator.write("id", post.getId());
    jsonGenerator.write("title", post.getTitle());
    jsonGenerator.write("description", post.getDescription());
    jsonGenerator.write("content", post.getContent());

    jsonGenerator.writeStartArray("tags");
    for (String tag: post.getTags()) {
        jsonGenerator.write(tag);
    }

    jsonGenerator.writeEnd(); // end of tags array
    jsonGenerator.writeEnd(); // }

    jsonGenerator.close();

    System.out.println(writer.toString());
}
```

```json
{
  "id": 1,
  "title": "JsonGenerator Demo",
  "description": "JsonGenerator Description",
  "content": "Markdown content here",
  "tags": [
    "Java",
    "JSON"
  ]
}
```

Decoding JSON in Java – Stream API (1)

```java
public static void main(String[] args) {
    final String result = "{\"name\":\"John\",\"isRetired\":false,\"age\":42,\"skills\":[\"Java SE\", \"Java EE\"]}";
    final JsonParser parser = Json.createParser(new StringReader(result));

    while (parser.hasNext()) {
        Event event = parser.next();

        switch (event) {
            case KEY_NAME:
                System.out.println(parser.getString());
                break;
            case VALUE_STRING:
                System.out.println(parser.getString());
                break;
            case END_ARRAY:break;
            case END_OBJECT: break;
            case START_ARRAY:
                break;
            case START_OBJECT: break;
            case VALUE_TRUE:
                System.out.println(true);
                break;
            case VALUE_FALSE:
                System.out.println(false);
                break;
            case VALUE_NULL: break;
            case VALUE_NUMBER:
                System.out.println(parser.getInt());
                break;

            default: break;
        }
    }
    parser.close();
}
```

Problems | @ Javadoc | Console | Properties

```
<terminated> JsonStreamDemo [Java Application] /Users/thoah
name
John
isRetired
false
age
42
skills
Java SE
Java EE
```

Decoding JSON in Java – Stream API (2)

```
[
    {
        "id": 1,
        "title": "JSONP Demo 1",
        "description": "Post about JSONP 1",
        "content": "Markdown content here 1",
        "tags": [
            "Java 1",
            "JSON 1"
        ]
    },
    {
        "id": 2,
        "title": "JSONP Demo 2",
        "description": "Post about JSONP 2",
        "content": "Markdown content here 2",
        "tags": [
            "Java 2",
            "JSON 2"
        ]
    }
]
```

Decoding JSON in Java – Stream API (3)

```java
public static void main(String[] args) throws FileNotFoundException {
    InputStream is = new FileInputStream("json/posts.json");

    JsonParserFactory factory = Json.createParserFactory(null);
    JsonParser parser = factory.createParser(is, StandardCharsets.UTF_8);

    if (!parser.hasNext() && parser.next() != JsonParser.Event.START_ARRAY) {
        return;
    }

    // looping over object attributes
    while (parser.hasNext()) {
        Event event = parser.next();
        // starting object
        if (event == JsonParser.Event.START_OBJECT) {
            while (parser.hasNext()) {
                event = parser.next();
                if (event == JsonParser.Event.KEY_NAME) {
                    String key = parser.getString();
                    switch (key) {
                        case "id":
                            parser.next();
                            System.out.printf("id: %s%n", parser.getString());
                            break;
                        case "title":
                            parser.next();
                            System.out.printf("title: %s%n", parser.getString());
                            break;
                        case "description":
                            parser.next();
                            System.out.printf("description: %s%n%n", parser.getString());
                            break;
                        case "content":
                            parser.next();
                            System.out.printf("content: %s%n%n", parser.getString());
                            break;
                    }
                }
            }
        }
    }
}
```

Jackson (1)

- Jackson is a simple java based library to serialize java objects to JSON and vice versa.
- Feature:
  - Easy to use - provides a high level facade to simplify commonly used use cases.
  - No need to create mapping - provides default mapping for most of the objects to be serialized.
  - Performance - it is quiet fast and is of low memory footprint and is suitable for large object graphs or systems.
  - Clean JSON - creates a clean and compact JSON results which is easy to read.
  - No Dependency - it does not require any other library apart from jdk.
  - Open Source - it is open source and is free to use.

Jackson (2)

- ObjectMapper is the main actor class of Jackson library. ObjectMapper class ObjectMapper provides functionality for reading and writing JSON, either to and from basic POJOs (Plain Old Java Objects)
- Ref: https://fasterxml.github.io/jackson-databind/javadoc/2.7/com/fasterxml/jackson/databind/ObjectMapper.html

Jackson (3)

```java
private static String json;
public static void main(String[] args) {
    ObjectMapper objectMapper = new ObjectMapper();
    Employee employee=new Employee(1L, "John Smith", 1000d);

    try {
        json = objectMapper.writeValueAsString(employee);
        System.out.println(json);
    } catch (JsonProcessingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

```xml
<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.16.1</version>
</dependency>
```

Jackson (4)

```
private static Employee employee;
public static void main(String[] args) {
    String employeeJson ="{\"id\":1,\"name\":\"John Smith\",\"salary\":1000.0}";
    ObjectMapper objectMapper = new ObjectMapper();

    try {
        employee = objectMapper.readValue(employeeJson, Employee.class);
        System.out.println(employee.toString());
    } catch (JsonProcessingException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
```

Conclusion

- The Java API for JSON Processing provides the following capabilities:
  - Parsing input streams into immutable objects or event streams
  - Writing event streams or immutable objects to output streams
  - Programmatically navigating immutable objects
  - Programmatically building immutable objects with builders

FAQ

That's all for this session!

Thank you all for your attention and patient !