ECE 448/528
Application Software Design

Lecture 12. JSON and Lambda Expression
Spring 2025

Won-Jae Yi, Ph.D.

Department of Electrical and Computer Engineering
Illinois Institute of Technology

JSON (JavaScript Object Notation)

JSON (JavaScript Object Notation)

- An open-standard to represent complex data structures as human-readable texts.
- Support primitive types, arrays, maps/dictionaries, and their compositions.
- Derived from JavaScript but supported by most languages now.
- As those data structures are language-independent.

JSON File Example

```
{
  "httpPort": 8080,
  "plugNames": [
    "a",
    "b.100",
    "cc",
    "dddd"],
  "mqttBroker": "tcp://127.0.0.1",
  "mqttClientId": "iot_sim",
  "mqttTopicPrefix": "iot_ece448"
}
```

- A configuration file used by our IoT simulator.
- JSON object: a map of key/value pairs enclosed in {}, with keys being strings.
- JSON array: an array of values enclosed in [].
- Values can be numbers, strings, Booleans, `null`, JSON objects, and JSON arrays.

Mapping between JSON and Java Objects

- Intuitively, one should be able to read the JSON file to create a corresponding Java data structure.
  - And to save a Java data structure as a JSON file.
- What data structure should we use?
- Option 1: use Java types directly
  - JSON objects are mapped to `Map<String, Object>`.
  - JSON arrays are mapped to `List<?>`.
- Option 2: use user-defined types
- In either case, numbers/strings/Booleans are mapped to boxed Java primitives.

The ObjectMapper

```
public class JsonMapping {
  ...
  private static final ObjectMapper mapper = new ObjectMapper();
  private static final Logger logger
    = LoggerFactory.getLogger(JsonMapping.class);
}
```

- In `src/main/ece448/lec12`, under branch `lec12-json`.
- **Jackson** is a popular library to support mapping between JSON and Java objects.
  - As well as many other file formats.
- Use an `ObjectMapper` object to access the Jackson library.
  - One per class, similar to the `logger`.

JSON and Java Types

```
// JSONfile -> Java types
Map<String, Object> configJava =mapper.readValue(
   new File("simConfig.json"),
   new TypeReference<Map<String, Object>>(){});

logger.info("httpPort {}", (Integer)configJava.get("httpPort"));
logger.info("mqttBroker {}", (String)configJava.get("mqttBroker"));

List<?> plugNames =(List<?>)configJava.get("plugNames");
for (Object name: plugNames)
   logger.info("plugName {}", (String)name);
```

- Use `readValue` to read a JSON file.
  - The tricky syntax with `TypeReference` is required to handle Java generics.
- Values in JSON objects and arrays are all `Object`, and you'll need to cast them into correct types.

JSON and Java Types

```
// Java types -> JSON string
String json = mapper.writeValueAsString(configJava);
logger.info("Java types->JSON {}", json);
```

- Use `writeValueAsString` to convert a Java object into a JSON string.
- Try to convert any Java objects and see how it works!

JSON and User Defined Types

```
public class SimConfig {
  private final int httpPort;
  private final List<String> plugNames;
  private final String mqttBroker;
  ...
  @JsonCreator
  public SimConfig(
    @JsonProperty(value ="httpPort", required =true) int httpPort,
    @JsonProperty(value ="plugNames", required =true) List<String> plugNames,
    @JsonProperty(value ="mqttBroker", required =false) String mqttBroker,
    ...) {
    this.httpPort =httpPort; this.plugNames =plugNames;
    this.mqttBroker =mqttBroker;
    ...
  }
  ... // getters omitted
```

- `SimConfig` corresponds to the structure of the JSON file.
- Use annotations to tell the Jackson library how to create an object from JSON.
  - Apply `@JsonCreator` to the constructor.
  - Use `@JsonProperty` to indicate what key/value pair should be used to initialize the parameter.

JSON and User Defined Types

```
// JSON string -> user type
SimConfig config = mapper.readValue(json, SimConfig.class);

// user type -> JSON file
mapper.writeValue(new File("user_type.json"), config);
```

- `readValue` can also read from a JSON string.
- Use `writeValue` to save a Java object into a JSON file.
  - It will refer to the getters to generate key/value pairs.

Lambda Expression

Lambda Expression

- A Java language feature to support anonymous functions.
  - For a methodology shift from complex interfaces to one-method interfaces that are closer to callback functions in other languages.
- As more one-method interfaces are used in Java programs for IoC, lambda expressions save time and effort to implement them and to NOT name the implementations.
- Since the implementations are not named *(anonymous)*, lambda expressions should be limited to short and straightforward ones.

One-Method Interface

```
interface Function {
  String call(String key, String value);
}
public class Lambda {
  public static void callFunction(String which, Function func) {
    logger.info("{}: {}", which, func.call("hello", "world"));
  }
  ...
}
```

- In `src/main/ece448/lec12`, under branch `lec12-lambda`.
- One may define multiple classes/interfaces in one Java file, as long as the ones with different names as the file are not `public`.
- The `Function` interface is used by `callFunction`.

Simple Implementation

```
public class Lambda {
  ...
  public static class Simple implements Function {
    @Override
    public String call(String key, String value) {
      return key+":"+value;
    }
  }
  public void runSimple() {
    callFunction("Simple", new Simple());
  }
}
```

- The `Simple` class just joins the two string parameters.

A Closure

```
public class Lambda {
  ...
  public static class Closure implements Function {
    private final String extra;
    public Closure(String extra) {
      this.extra =extra;
    }
    @Override
    public String call(String key, String value) {
      return key+":"+value+"-"+extra;
    }
  }
  public void runClosure() {
    callFunction("Closure", newClosure("extra for closure"));
  }
}
```

- We may create a closure to include extra data for the computation in addition to the parameters.

A Closure with This

```
public class Lambda {
  ...
  private final String context = "Lambda";
  public static class ClosureThis implements Function {
    private final Lambda that;
    private final String extra;
    public ClosureThis(Lambda that, String extra) {
      this.that = that; this.extra = extra;
    }
    @Override
    public String call(String key, String value) {
      return "["+that.context+"]"+key+":"+value+"-"+extra;
    }
  }
  public void runClosureThis() {
    callFunction("ClosureThis",
      new ClosureThis(this, "extra for closure and this"));
  }
}
```

- The closure could also refer to the outside object.

Use Lambda Expression

```
public class Lambda {
  ...
  private final String context ="Lambda";
  public void runLambda() {
    String extra ="extra for lambda";
    callFunction("Lambda", (key, value) -> {
      return "["+context+"]"+key+":"+value+"-"+extra;
    });
  }
}
```

- Less line of code than previous implementations.
- Use **`(params)->{body}`** to create a lambda expression as an anonymous implementation of one-method interface.
  - Params should match the parameters of the method.
  - Body should return what the method returns.
- Lambda expression can capture outside variables like `extra` so they can be used in the body.
  - As long as the variable is assigned only once.
- Lambda expression can capture outside `this` so members like `context` can be used in the body.