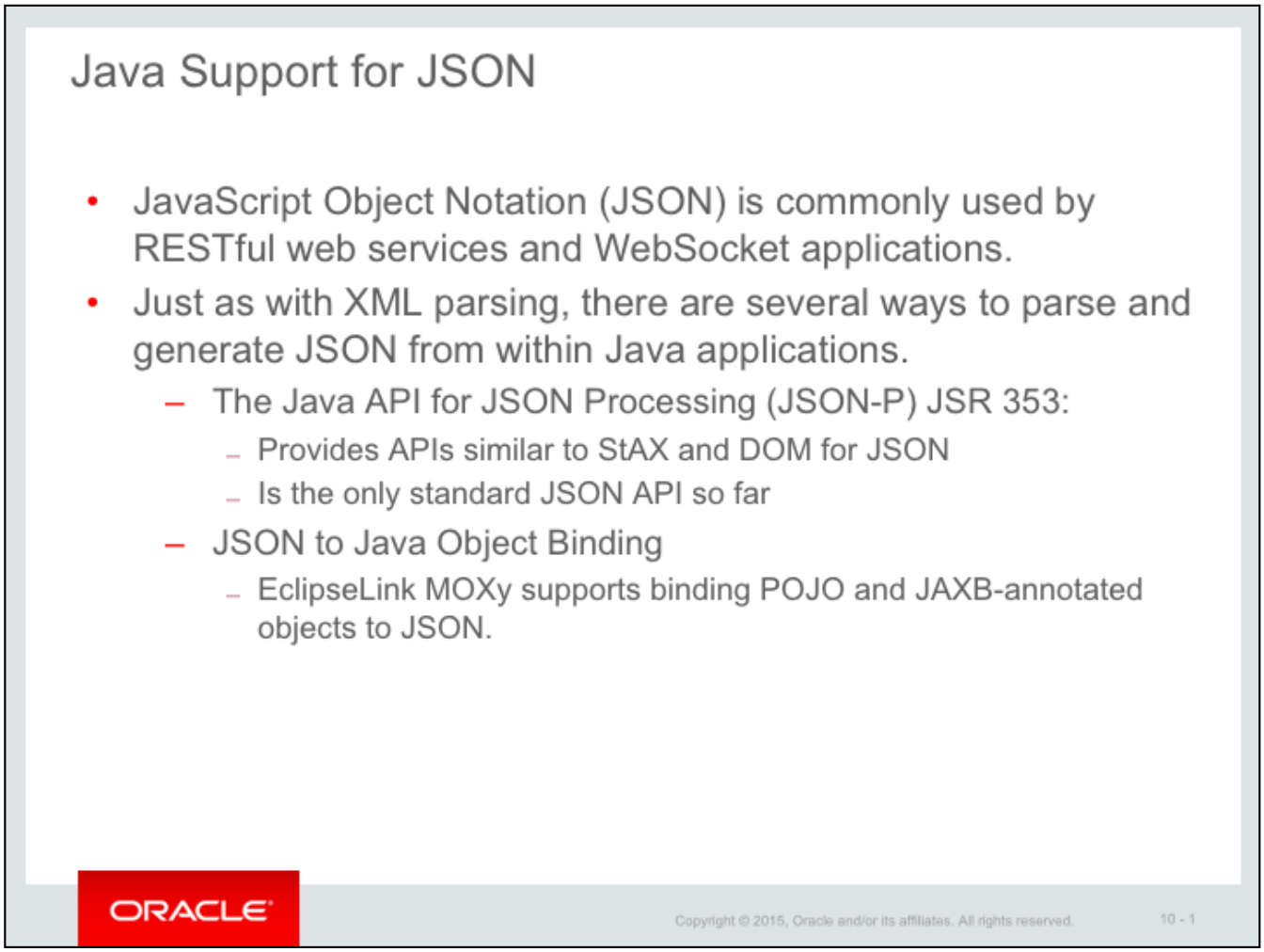
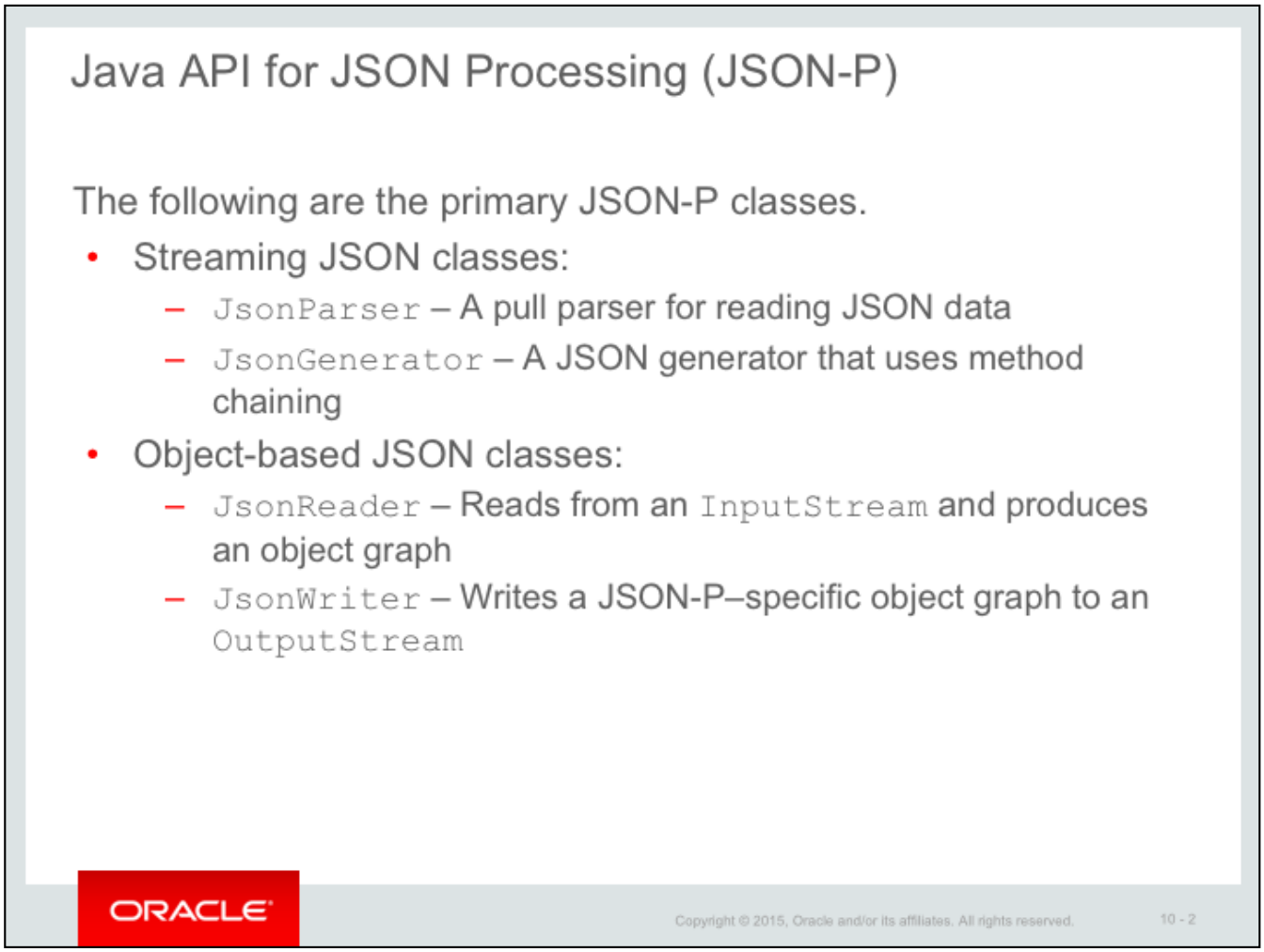


Third-party Java JSON libraries, such as Gson, have been around for years. However, JSON APIs are now starting to become the official components of the Java platform by using the JEP/JSR process. JSON-P was released as part of Java EE 7 and several other JSON standards are under development:

- **JSR 367: Java API for JSON Binding (JSON-B):** This API will standardize JSON to Java binding similar to what JAXB does for XML. It is currently scheduled for release as a Java EE 8 component. For more, see https://jcp.org/en/jsr/detail?id=367.
- **JEP 198: Light-Weight JSON API:** This API provides a lightweight JSON library under `java.util` and is under consideration for Java SE 9. For more, see http://openjdk.java.net/jeps/198.



The streaming classes parse or generate JSON as the data streams through your application eliminating the need to have all of the JSON data in memory as objects.

The object-based JSON reader or writer classes use JSON-P class types to represent JSON constructs.

- `JsonStructure` – The common JSON object supertype
- `JsonObject` – A JSON object that uses string keys to uniquely identify values
- `JsonArray` – A JSON object that uses `int` index values to uniquely identify values

JsonGenerator

```
JsonGenerator jgen = Json.createGenerator(System.out);
jgen.writeStartObject()
        .writeStartArray("persons")
            .writeStartObject()
                .write("firstName", "Sherlock")
                .write("lastName", "Holmes")
                .writeStartObject("address")
                    .write("street", "221b Baker Street")
                    .write("city", "London")
                    .write("country", "England")
                .writeEnd()
            .writeEnd()
        .writeEnd()
.writeEnd();
jgen.flush();
```

A `JsonGenerator` builds JSON text by using method chaining and writes it to an `OutputStream` or a `Writer`.

JsonParser

```
JsonParser jp =
Json.createParser(new FileInputStream("data.json"));
while(jp.hasNext()) {
    Event e = jp.next();
    switch(e) {
        case START_OBJECT:
        case END_OBJECT:
        case START_ARRAY:
        case END_ARRAY:
          break;
        case KEY_NAME:
        case VALUE_STRING:
          System.out.print(jp.getString());
    }
}
```

A `JsonParser` converts JSON text into a sequence of events. It is a fast, low-level method for parsing JSON without loading the JSON data into an object graph.

JsonWriter

```
JsonWriter jw = Json.createWriter(System.out);
JsonObject address = Json.createObjectBuilder()
        .add("street", "221b Baker Street")
        .build();
JsonObject sherlock = Json.createObjectBuilder()
        .add("firstName", "Sherlock")
        .add("address", address)
        .build();
JsonArray persons = Json.createArrayBuilder()
        .add(sherlock)
        .build();
JsonObject root = Json.createObjectBuilder()
        .add("persons", persons)
        .build();
jw.write(root);
```

A `JsonWriter` builds an object graph of JSON-specific objects, and then writes that object graph as JSON text to an `OutputStream` or a `Writer`.

A `JsonReader` converts JSON text into an object graph. The object types in the graph are the same as those used by `JsonReader`.

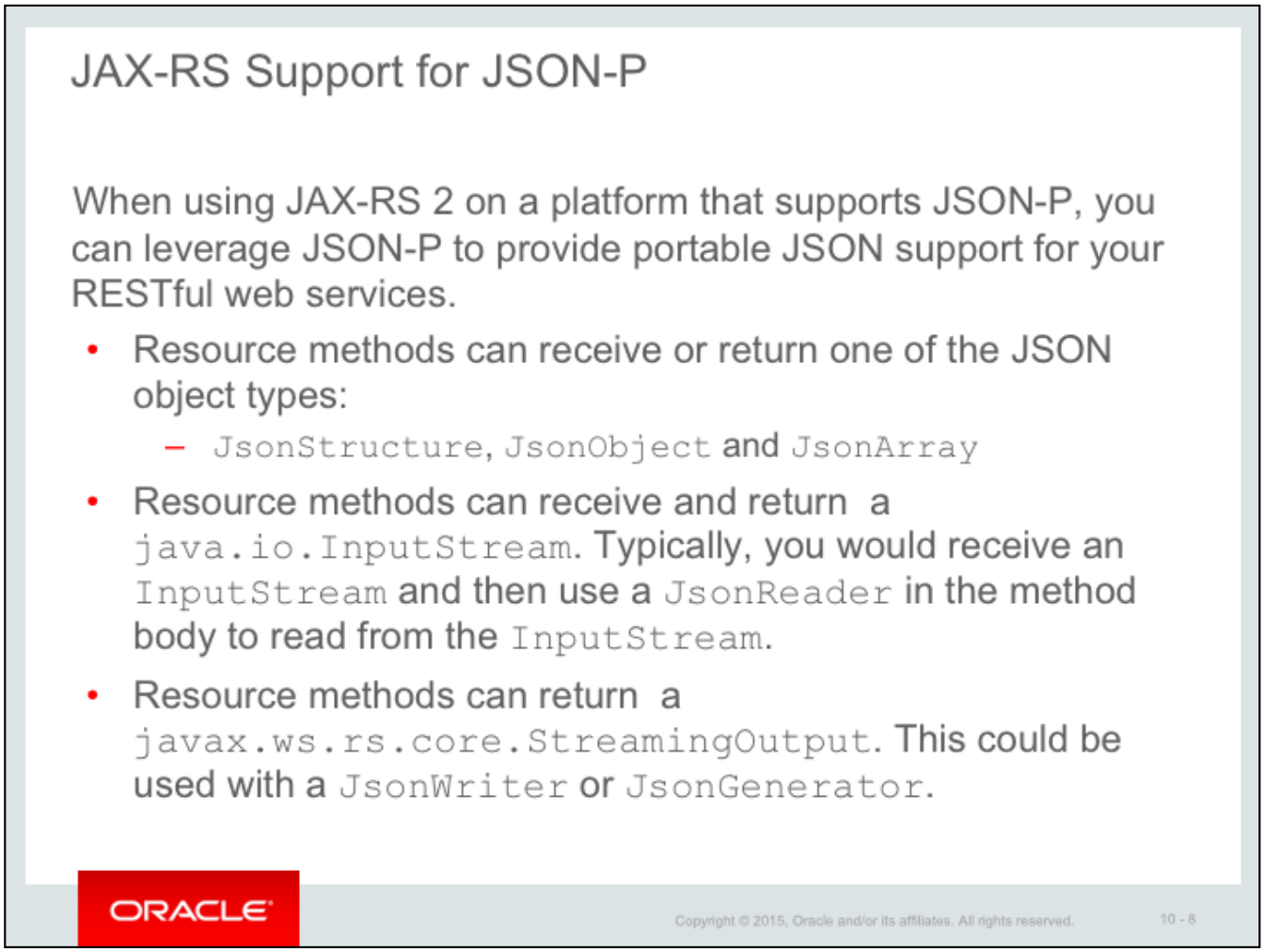
Java API for WebSocket Support for JSON

Use `java.io.StringReader` and `java.io.StringWriter` along with JSON-P to support JSON in Java WebSocket endpoints.

```
Async remote = session.getAsyncRemote();
StringWriter sw = new StringWriter();
JsonWriter jw = Json.createWriter(sw);
JsonObject stock = Json.createObjectBuilder()
        .add("symbol", sym)
        .add("price", price)
        .build();
jw.writeObject(stock);
remote.sendText(sw.toString());
```

The Java API for WebSocket supports custom encoder and decoder classes for endpoints. With a custom encoder and parameter type, JSON application class types can also be supported. A custom decoder can be used to return any custom class type. JSON-P code would be placed in the custom encoder and decoder classes.

```
@ServerEndpoint(value="/ep", decoders=JsonPersonCodec.class,
  encoders=JsonPersonCodec.class)
public class MyServerEndpoint {
  @OnMessage public Person onMessage(Person p) {
    return new Person();
  }
}
public class JsonPersonCodec implements Encoder.Text<Person>,
  Decoder.Text<Person> {
  //...
}
```



Using JSON-P with JAX-RS is currently the only standards-based approach to produce and consume JSON. Many JAX-RS implementations, such as Jersey, support JSON-POJO and JSON-JAXB binding; however, support for these types of bindings is not yet standardized. You might have to add lines of code to your `Application` subclass to enable the feature- or place-specific classes on your CLASSPATH. If you are using JSON with JAX-RS and you are not using JSON-P, then you must reference the documentation for your JAX-RS implementation to understand the JSON features available to you.

JAX-RS StreamingOutput

The `javax.ws.rs.core.StreamingOutput` interface type can be used as a return type for JAX-RS resource methods. It enables you to stream the response body as it is created. You can use `StreamingOutput` to stream large JSON responses.

```
StreamingOutput stream = new StreamingOutput() {
  @Override
  public void write(OutputStream os) throws IOException,
    WebApplicationException {
     //JsonGenerator jgen = Json.createGenerator(os);
  }
};
return Response.ok(stream).build();
```

Imagine sending responses that are hundreds of megabytes in size. If you had to load all that data into memory before you could create a response, then each client request would consume hundreds of megabytes of RAM.

Where Can I Learn More?

Resource	Website
JSON Processing	https://docs.oracle.com/javaee/7/tutorial/jsonp.htm#GLRBB
JSR 353: Java API for JSON Processing	https://jcp.org/en/jsr/detail?id=353

Summary

In this lesson, you should have learned how to:

- Explain the benefits of WebSockets
- Create WebSocket server endpoints with Java
- Create WebSocket client endpoints with Java
- Create WebSocket client endpoints with JavaScript
- Consume JSON with Java
- Produce JSON with Java

Practice 10: Overview

This practice covers the following topics:

- Developing a WebSocket Endpoint and Client
- Using the JSON-P API to Generate JSON