SPL Assignment2 Appendixes

Appendix A: GSON

GSON is a Java library developed by Google that facilitates converting Java objects to JSON (serialization) and JSON to Java objects (deserialization). It's powerful, easy to use, and handles complex data structures efficiently.

Adding GSON to Your Project

To use GSON, you need to add it as a dependency in your project. If you're using **Maven**, add the following to your `pom.xml`:

```
<dependencies>
    <!-- Other dependencies -->
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.8.9</version>
    </dependency>
</dependencies>
```

Reading JSON Files into Java Objects

Suppose you have a JSON file `employees.json` with the following content:

```
[
    {
        "employeeId": 101,
        "name": "John Doe",
        "department": "Engineering",
        "skills": ["Java", "C++", "Python"]
    },
    {
        "employeeId": 102,
        "name": "Jane Smith",
        "department": "Marketing",
        "skills": ["SEO", "Content Writing"]
    }
]
```

You want to parse this JSON file into a list of `Employee` objects.

Step 1: Define the Java Classes

Create a Java class that mirrors the structure of the JSON objects.

```
public class Employee {
    private int employeeId;
    private String name;
    private String department;
    private List<String> skills;

    // Default constructor
    public Employee() {
    }

    // Getters and setters (recommended for encapsulation)
    public int getEmployeeId() {
        return employeeId;
    }

    public void setEmployeeId(int employeeId) {
        this.employeeId = employeeId;
    }

    // Other getters and setters...

    @Override
    public String toString() {
        return "Employee{" +
               "employeeId=" + employeeId +
               ", name='" + name + '\'' +
               ", department='" + department + '\'' +
               ", skills=" + skills +
               '}';
    }
}
```

Step 2: Parse the JSON File

Use GSON to parse the JSON file into Java objects.

```
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;
import java.io.FileReader;
import java.io.IOException;
import java.lang.reflect.Type;
import java.util.List;
```

```
public class EmployeeParser {
    public static void main(String[] args) {
        Gson gson = new Gson();

        try (FileReader reader = new FileReader("employees.json")) {
            // Define the type for the list of employees
            Type employeeListType = new TypeToken<List<Employee>>()
{}.getType();

            // Deserialize JSON to list of employees
            List<Employee> employeeList = gson.fromJson(reader,
employeeListType);

            // Use the employee data
            for (Employee employee : employeeList) {
                System.out.println(employee);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- **Gson gson = new Gson();**: Creates an instance of the GSON parser.
- **TypeToken<List>() {}.getType();**: Captures the generic type `List<Employee>` for GSON.
- **gson.fromJson(reader, employeeListType);**: Parses the JSON content into a list of `Employee` objects.

Writing Java Objects to JSON Files

Suppose you have Java objects that you want to serialize to JSON.

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Arrays;

public class EmployeeWriter {
    public static void main(String[] args) {
        Employee emp1 = new Employee();
        emp1.setEmployeeId(103);
```

```
        emp1.setName("Alice Johnson");
        emp1.setDepartment("HR");
        emp1.setSkills(Arrays.asList("Recruitment", "Employee Relations"));

        Employee emp2 = new Employee();
        emp2.setEmployeeId(104);
        emp2.setName("Bob Brown");
        emp2.setDepartment("Finance");
        emp2.setSkills(Arrays.asList("Accounting", "Budgeting"));

        List<Employee> employees = Arrays.asList(emp1, emp2);

        // Create GSON instance with pretty printing
        Gson gson = new GsonBuilder().setPrettyPrinting().create();

        try (FileWriter writer = new FileWriter("new_employees.json")) {
            // Serialize Java objects to JSON file
            gson.toJson(employees, writer);
            System.out.println("Employees have been written to
new_employees.json");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explanation:

- **GsonBuilder().setPrettyPrinting().create();**: Configures GSON to output formatted JSON.
- **gson.toJson(employees, writer);**: Serializes the list of `Employee` objects to a JSON file.

Parsing Nested JSON Objects

If your JSON contains nested objects, GSON can handle that too.

Example JSON (`company.json`):

```
{
    "companyName": "Tech Solutions",
    "location": "New York",
    "employees": [
        {
            "employeeId": 101,
            "name": "John Doe",
            "department": "Engineering",
```

```
            "skills": ["Java", "C++", "Python"]
        },
        {
            "employeeId": 102,
            "name": "Jane Smith",
            "department": "Marketing",
            "skills": ["SEO", "Content Writing"]
        }
    ]
}
```

Corresponding Java Classes:

```
public class Company {
    private String companyName;
    private String location;
    private List<Employee> employees;

    // Getters and setters...

    @Override
    public String toString() {
        return "Company{" +
               "companyName='" + companyName + '\'' +
               ", location='" + location + '\'' +
               ", employees=" + employees +
               '}';
    }
}
```

Parsing the Nested JSON:

```
public class CompanyParser {
    public static void main(String[] args) {
        Gson gson = new Gson();

        try (FileReader reader = new FileReader("company.json")) {
            // Deserialize JSON to Company object
            Company company = gson.fromJson(reader, Company.class);

            // Use the company data
            System.out.println(company);

        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

Handling JSON Arrays and Collections

GSON can parse JSON arrays directly into Java collections.

Example JSON:

```
["Red", "Green", "Blue"]
```

Parsing the JSON Array:

```
public class ColorParser {
    public static void main(String[] args) {
        Gson gson = new Gson();
        String jsonArray = "[\"Red\", \"Green\", \"Blue\"]";

        Type colorListType = new TypeToken<List<String>>() {}.getType();
        List<String> colors = gson.fromJson(jsonArray, colorListType);

        System.out.println("Colors: " + colors);
    }
}
```

Custom Serialization and Deserialization

You can create custom serializers and deserializers for handling complex data types or formats.

Example: Custom Date Deserialization

Suppose you have dates in a specific format in your JSON:

```
{
    "employeeId": 105,
    "name": "Emma Wilson",
    "department": "Sales",
    "joiningDate": "15-03-2022"
}
```

Custom Deserializer for `Date`:

```
import com.google.gson.*;
import java.lang.reflect.Type;
import java.text.SimpleDateFormat;
import java.util.Date;
```

```
public class DateDeserializer implements JsonDeserializer<Date> {
    private static final SimpleDateFormat formatter = new
SimpleDateFormat("dd-MM-yyyy");

    @Override
    public Date deserialize(JsonElement jsonElement, Type typeOfT,
JsonDeserializationContext context)
            throws JsonParseException {
        String dateStr = jsonElement.getAsString();
        try {
            return formatter.parse(dateStr);
        } catch (Exception e) {
            throw new JsonParseException("Failed to parse Date: " + dateStr,
e);
        }
    }
}
```

Registering the Custom Deserializer:

```
public class EmployeeWithDateParser {
    public static void main(String[] args) {
        GsonBuilder gsonBuilder = new GsonBuilder();
        gsonBuilder.registerTypeAdapter(Date.class, new DateDeserializer());
        Gson gson = gsonBuilder.create();

        String json = "{ \"employeeId\": 105, \"name\": \"Emma Wilson\",
\"department\": \"Sales\", \"joiningDate\": \"15-03-2022\" }";

        EmployeeWithDate employee = gson.fromJson(json,
EmployeeWithDate.class);
        System.out.println(employee);
    }
}

public class EmployeeWithDate {
    private int employeeId;
    private String name;
    private String department;
    private Date joiningDate;

    // Getters and setters...
```

```
    @Override
    public String toString() {
        return "EmployeeWithDate{" +
               "employeeId=" + employeeId +
               ", name='" + name + '\'' +
               ", department='" + department + '\'' +
               ", joiningDate=" + joiningDate +
               '}';
    }
}
```

Error Handling

When parsing JSON, it's important to handle exceptions to prevent crashes due to invalid data.

```
try {
    Employee employee = gson.fromJson(jsonString, Employee.class);
} catch (JsonSyntaxException e) {
    System.err.println("JSON Syntax Error: " + e.getMessage());
} catch (JsonIOException e) {
    System.err.println("JSON IO Error: " + e.getMessage());
} catch (Exception e) {
    System.err.println("Unexpected Error: " + e.getMessage());
}
```

Tips and Best Practices

- **Field Naming:** Ensure that your Java class fields match the JSON keys. Use annotations like `@SerializedName` if they differ.

  ```
  public class Employee {
      @SerializedName("employeeId")
      private int id;

      @SerializedName("name")
      private String fullName;

      //...
  }
  ```

- **Null Handling:** GSON can handle null values gracefully. You can configure it to include or exclude null fields during serialization.

  ```
  Gson gson = new GsonBuilder().serializeNulls().create();
  ```

- **Transient Fields:** Fields marked as `transient` will be ignored by GSON during serialization and deserialization.

```
public class Employee {
    private int employeeId;
    private String name;
    private transient String password; // Will not be serialized
    //...
}
```

- **Custom Exclusion Strategies:** You can define custom rules to exclude fields.

```
Gson gson = new GsonBuilder()
    .setExclusionStrategies(new ExclusionStrategy() {
        @Override
        public boolean shouldSkipField(FieldAttributes f) {
            return f.getName().equals("password");
        }

        @Override
        public boolean shouldSkipClass(Class<?> clazz) {
            return false;
        }
    })
    .create();
```

Incorporating GSON into Your Assignment

In your assignment, you can guide students on how to use GSON to parse the input JSON files provided, such as configuration files or data from sensors.

Example Assignment Instruction:

Using GSON to Parse JSON Input Files

To handle the JSON input files in your assignment, follow these steps:

1. **Define Java Classes Corresponding to the JSON Structure**
   For example, if you have a configuration file `config.json`:

```
{
    "simulationTime": 1000,
    "serverSettings": {
        "port": 8080,
        "maxConnections": 100
    },
    "clients": [
        {"id": 1, "name": "ClientA"},
        {"id": 2, "name": "ClientB"}
```

```
    ]
}
```

Create corresponding Java classes:

```
public class Config {
    private int simulationTime;
    private ServerSettings serverSettings;
    private List<Client> clients;

    // Getters and setters...
}

public class ServerSettings {
    private int port;
    private int maxConnections;

    // Getters and setters...
}

public class Client {
    private int id;
    private String name;

    // Getters and setters...
}
```

2. **Parse the JSON File Using GSON**

```
public class ConfigParser {
    public static Config parseConfig(String filePath) {
        Gson gson = new Gson();

        try (FileReader reader = new FileReader(filePath)) {
            Config config = gson.fromJson(reader, Config.class);
            return config;
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

3. **Use the Parsed Data in Your Application**

```
public class Simulation {
    public static void main(String[] args) {
        Config config = ConfigParser.parseConfig("config.json");

        if (config != null) {
            System.out.println("Simulation Time: " +
config.getSimulationTime());
            System.out.println("Server Port: " +
config.getServerSettings().getPort());
            System.out.println("Clients: " + config.getClients());
            // Proceed with simulation setup...
        } else {
            System.err.println("Failed to load configuration.");
        }
    }
}
```

Appendix B: Coordinate Transformation

Objective: Transform the coordinates of detected objects from the robot's local coordinate system to the global coordinate system of the charging station.

Background

In robotics, it's common to work with multiple coordinate systems:

- **Local Coordinate System (Robot's Frame):** The coordinate system relative to the robot's current position and orientation. Measurements from sensors like LiDAR are typically in this frame.
- **Global Coordinate System (World Frame):** A fixed coordinate system used as a common reference point for all measurements, such as the coordinate system of the charging station.

To accurately map the environment and track objects over time, sensor data collected in the robot's local frame must be transformed into the global frame.

Transformation Process

The transformation from the local coordinate system to the global coordinate system involves two main steps:

1. **Rotation:** Adjusting for the robot's orientation (yaw) to align the axes.
2. **Translation:** Adjusting for the robot's position (x, y) to align the origins.

Mathematical Explanation

Let:

- $(x_{\text{local}}, y_{\text{local}})$ be the coordinates of a point in the robot's local frame.
- $(x_{\text{global}}, y_{\text{global}})$ be the coordinates of the point in the global frame.
- $(x_{\text{robot}}, y_{\text{robot}})$ be the position of the robot in the global frame.
- θ be the robot's orientation (yaw angle) in degrees (or radians, depending on implementation).

The transformation is performed using the following equations:

1. **Convert the yaw angle to radians** (if it's in degrees):
   $\theta_{\text{rad}} = \theta_{\text{degrees}} \times \left(\frac{\pi}{180}\right)$
2. **Compute the cosine and sine of the yaw angle:**
   $\cos(\theta_{\text{rad}}), \quad \sin(\theta_{\text{rad}})$
3. **Apply the rotation and translation:**
   $x_{\text{global}} = \cos(\theta_{\text{rad}}) \times x_{\text{local}} - \sin(\theta_{\text{rad}}) \times y_{\text{local}} + x_{\text{robot}}$
   $y_{\text{global}} = \sin(\theta_{\text{rad}}) \times x_{\text{local}} + \cos(\theta_{\text{rad}}) \times y_{\text{local}} + y_{\text{robot}}$

Appendix B: Coordinate Transformation

Objective: Transform the coordinates of detected objects from the robot's local coordinate system to the global coordinate system of the charging station.

Background

In robotics, it's common to work with multiple coordinate systems:

- **Local Coordinate System (Robot's Frame):** The coordinate system relative to the robot's current position and orientation. Measurements from sensors like LiDAR are typically in this frame.
- **Global Coordinate System (World Frame):** A fixed coordinate system used as a common reference point for all measurements, such as the coordinate system of the charging station.

To accurately map the environment and track objects over time, sensor data collected in the robot's local frame must be transformed into the global frame.

Transformation Process

The transformation from the local coordinate system to the global coordinate system involves two main steps:

1. **Rotation:** Adjusting for the robot's orientation (yaw) to align the axes.
2. **Translation:** Adjusting for the robot's position (x, y) to align the origins.

Mathematical Explanation

Let:

- $(x_{\text{local}}, y_{\text{local}})$ be the coordinates of a point in the robot's local frame.
- $(x_{\text{global}}, y_{\text{global}})$ be the coordinates of the point in the global frame.

- $(x_{\text{robot}}, y_{\text{robot}})$ be the position of the robot in the global frame.
- θ be the robot's orientation (yaw angle) in degrees (or radians, depending on implementation).

The transformation is performed using the following equations:

1. **Convert the yaw angle to radians** (if it's in degrees):
   $\theta_{\text{rad}} = \theta_{\text{degrees}} \times \left(\frac{\pi}{180}\right)$
2. **Compute the cosine and sine of the yaw angle:**
   $\cos(\theta_{\text{rad}}), \quad \sin(\theta_{\text{rad}})$
3. **Apply the rotation and translation:**
   $x_{\text{global}} = \cos(\theta_{\text{rad}}) \times x_{\text{local}} - \sin(\theta_{\text{rad}}) \times y_{\text{local}} + x_{\text{robot}}$
   $y_{\text{global}} = \sin(\theta_{\text{rad}}) \times x_{\text{local}} + \cos(\theta_{\text{rad}}) \times y_{\text{local}} + y_{\text{robot}}$

Implementation Details

- **Rotation Matrix:**
  The rotation of a point in 2D space is given by:
  $\begin{bmatrix} x_{\text{rotated}} \\ y_{\text{rotated}} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x_{\text{local}} \\ y_{\text{local}} \end{bmatrix}$
- **Translation:**
  After rotation, we translate the point by adding the robot's position:
  $x_{\text{global}} = x_{\text{rotated}} + x_{\text{robot}}$
  $y_{\text{global}} = y_{\text{rotated}} + y_{\text{robot}}$
- **Combined Transformation:**
  The combined rotation and translation can be applied directly using the equations provided.

Example

Suppose the robot's current pose is:

- Position: $x_{\text{robot}} = 5.0$, $y_{\text{robot}} = 10.0$
- Orientation: $\theta = 30^\circ$

A point detected by the LiDAR in the robot's local frame is:

- $x_{\text{local}} = 2.0$
- $y_{\text{local}} = 3.0$

Transformation Steps:

1. Convert θ to radians:
   $\theta_{\text{rad}} = 30^\circ \times \left(\frac{\pi}{180}\right) = \frac{\pi}{6} \approx 0.5236 \text{ radians}$
2. Compute $\cos(\theta_{\text{rad}})$ and $\sin(\theta_{\text{rad}})$:
   $\cos(0.5236) \approx 0.8660$
   $\sin(0.5236) \approx 0.5000$

3. Apply the transformation:
   $x_{\text{global}} = (0.8660 \times 2.0) - (0.5000 \times 3.0) + 5.0 = (1.7320 - 1.5000) + 5.0 = 5.2320$
   $y_{\text{global}} = (0.5000 \times 2.0) + (0.8660 \times 3.0) + 10.0 = (1.0000 + 2.5980) + 10.0 = 13.5980$

So, the point's coordinates in the global frame are approximately $(5.2320, 13.5980)$.

Implementation Details

- **Rotation Matrix:**

  The rotation of a point in 2D space is given by:
  $$\begin{bmatrix} x_{\text{rotated}} \\ y_{\text{rotated}} \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \times \begin{bmatrix} x_{\text{local}} \\ y_{\text{local}} \end{bmatrix}$$
- **Translation:**

  After rotation, we translate the point by adding the robot's position:

  $x_{\text{global}} = x_{\text{rotated}} + x_{\text{robot}}$

  $y_{\text{global}} = y_{\text{rotated}} + y_{\text{robot}}$
- **Combined Transformation:**

  The combined rotation and translation can be applied directly using the equations provided.

Example

Suppose the robot's current pose is:

- Position: $x_{\text{robot}} = 5.0$, $y_{\text{robot}} = 10.0$
- Orientation: $\theta = 30^\circ$

A point detected by the LiDAR in the robot's local frame is:

- $x_{\text{local}} = 2.0$
- $y_{\text{local}} = 3.0$

Transformation Steps:

1. Convert θ to radians:
   $\theta_{\text{rad}} = 30^\circ \times \left(\frac{\pi}{180}\right) = \frac{\pi}{6} \approx 0.5236 \text{ radians}$
2. Compute $\cos(\theta_{\text{rad}})$ and $\sin(\theta_{\text{rad}})$:
   $\cos(0.5236) \approx 0.8660$
   $\sin(0.5236) \approx 0.5000$
3. Apply the transformation:
   $x_{\text{global}} = (0.8660 \times 2.0) - (0.5000 \times 3.0) + 5.0 = (1.7320 - 1.5000) + 5.0 = 5.2320$
   $y_{\text{global}} = (0.5000 \times 2.0) + (0.8660 \times 3.0) + 10.0 = (1.0000 + 2.5980) + 10.0 = 13.5980$

So, the point's coordinates in the global frame are approximately $(5.2320, 13.5980)$.