Part 1: Designing and Implementing the Domain Classes

In today's lab you will create a working Restful Webservice for library system. You will implement different operations that can be executed on a Restful Webservice in a library management system. We will start creating this system in today's lab and will keep working on this in the upcoming labs. For today, we will implement following two key classes:

1. **Book**: Represents a book in the library.
2. **Library**: Represents the library that contains books.

To start working on this Lab, you may start from scratch by 1) creating a new Restful Webservice or 2) you may use the existing basic HelloWorld Restful Webservice that we created in Lab 6. I will prefer the second option to save time.

Requirements for the `Book` Class:

1. **Attributes**:

    - `id` (unique identifier for each book)
    - `title` (title of the book)
    - `author` (author of the book)
    - `isbn` (ISBN number of the book)

2. **Methods**:

    - Constructor for initializing a book object.
    - Getters and setters for all attributes.
    - Optional `toString()` method to display the book details in a readable format.

Requirements for the `Library` Class:

1. **Attribute**:

    - `books` (an ArrayList of type Book in the library)

2. **Methods**:

    - Constructor for initializing a library object with a name. The constructor will add 3-4 books in the books ArrayList. Just create the Book class objects in this constructor and add each object in ArrayList. Remember each book has an ISBN, title, author(s), and publishing year. The purpose of this constructor is to preload the sample data enables immediate testing of the RESTful endpoints without requiring external data setup.

By the end of Part 1, you will have functional `Library` and `Book` classes, which will serve as the foundation for the RESTful APIs.

Part 2: Developing the RESTful Web Service

In this part of lab you will convert your domain classes into a RESTful web service that provides endpoints to manage library and books. They will integrate the `Library` and `Book` classes from Part 1.

Following are the responsibilities of the Library class:

1. Maintains a collection of Book objects using an ArrayList.
2. Prepopulates the list with sample data for testing purposes.
3. Provides multiple RESTful endpoints to:

    - Retrieve books based on ISBN, title, and/or year.
    - Use both @PathParam and @QueryParam to handle URL parameters flexibly.
    - Demonstrate CRUD operations (though the PUT method is currently a placeholder)

Constructor of the Library class already created a list of sample data for testing purpose. Now we need to focus on providing multiple RESTful endpoints. As a first step we should define an end point for full Library resource using a @Path("library"). After that the following endpoints and their respective tags will be implemented:

1. Retrieve a Book by ISBN

- **Endpoint**: GET /library/books/{isbn}
- **Description**: Retrieves details of a book with the specified ISBN from the collection.
- **Implementation detail for respective method**:
    - Loops through the books list to find a matching ISBN.
    - If found, returns the book's details in plain text format.
    - If not found, returns a "NOT FOUND" message.

Note that you need to implement right JAX-RS tagging scheme to connect your endpoint with the methods. For example, you can implement the following notations in retrieving a Book object using ISBN:

```
@GET // because we are aiming to retrieve data
@Path("/books/{isbn}") // for mapping our method to the URI
@Produces(MediaType.TEXT_PLAIN) // for describing what MIME type our method will produce
PathParam("isbn") String id// for passing the parameter from URI to method. The name of the param "isbn" must be same as you are passing in endpoint @Path("/books/{isbn}")
```

2. Retrieve Books by Title and Year

- **Endpoint**: GET /library/books/{title}/{year}
- **Description**: Retrieves all books matching the specified title and year.
- **Implementation detail for respective method**:

- Loops through the books list to find matches based on title and year.
- Concatenates details of all matching books into a single response string.
- If no match is found, returns a "NOT FOUND" message.

3. Retrieve Books Using Query Parameters

- **Endpoint**: GET /library
- **Description**: Retrieves books based on either ISBN or a combination of title and year, provided via query parameters.
- **Implementation detail for respective method**
  - Checks the presence of query parameters (isbn, title, year).
  - If an ISBN is provided, returns the corresponding book details (if found).
  - If title and year are provided, returns all matching books.
  - If no query parameters match, returns a "Books NOT FOUND" message.

4. Update Library Resource

- **Endpoint**: PUT /library
- **Description**: A placeholder method for handling updates or resource creation.
- **Implementation detail for respective method**:
  - Currently does nothing but accepts plain text input.
  - Intended for future extensions, e.g., modifying the library name or bulk book updates.

4. Key Features

- **Use of** @PathParam: Allows dynamic URL path matching to fetch books by ISBN or title and year.
- **Use of** @QueryParam: Offers flexible querying for books based on multiple optional parameters.
- **Plain Text Responses**: Keeps the response format simple for demonstration, but can be extended to JSON/XML for real-world applications.
- **Preloaded Data**: Simplifies testing and ensures the endpoints can be verified immediately after implementation.

5. Potential Enhancements

1. **JSON Response**: Convert the responses to JSON using libraries like Jackson or Gson for better integration with modern clients.
2. **Validation**: Add checks to ensure valid inputs (e.g., ISBN format or year range).
3. **Pagination**: Implement pagination for endpoints returning multiple books to handle large datasets.
4. **Exception Handling**: Use custom exceptions and global exception handling to return standardized error messages.

This refined description provides students with a clear understanding of each method's functionality, implementation details, and how to extend the class for additional requirements.