LAB 4: Reading from a file in Java

Task 1: Read from a file

To add the `students.csv` file to your Java project in IntelliJ, follow these steps:

Step 1: Create a `resources` Folder

1. In the **Project** view in IntelliJ, right-click on your project's root folder.
2. Select **New > Directory**.
3. Name the new directory **resources** (or **data** if you prefer).
4. Right-click on the `resources` folder, select **Mark Directory as > Resources Root**. This makes it a designated place for your project's files.

Step 2: Drag previously downloaded the CSV File and drop it in a created Folder

Step 3: Access the CSV File in Your Java Code

When the file is marked as a resource, you can load it using a relative path in your Java code. Here's how to do it:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.Objects;

public class Main {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new
InputStreamReader(

Objects.requireNonNull(Main.class.getResourceAsStream("/studen
ts.csv"))))) {
            String line;
            br.readLine(); // Skip the header line
            while ((line = br.readLine()) != null) {
                // Process each line from the CSV file
                System.out.println(line); // Just printing for
demonstration
            }
        } catch (IOException e) {
            e.printStackTrace();

        }
    }
}
```

Explanation

- **`Main.class.getResourceAsStream("/students.csv")`** loads the file as a resource from the `resources` directory. This method returns an `InputStream`, which allows us to read the contents of the file. **Objects.requireNonNull()**: Checks that the InputStream is not null (meaning the file was found and loaded). If it is null, an exception will be thrown immediately.
- The path `"/students.csv"` is relative to the resources folder.
- Using `BufferedReader` and `InputStreamReader` allows you to read each line of the file.
- **`InputStreamReader`**: Converts the `InputStream` from `getResourceAsStream()` into a character-based stream, so we can read the file as text.
- **`BufferedReader`**: Wraps `InputStreamReader` and provides efficient reading of lines of text. It also has a convenient method, `readLine()`, which reads one line of the file at a time.

This setup ensures that `students.csv` is bundled with your project, making it accessible for the lab tasks in any environment where the project is executed.

Task 2: Set Up the Student Class and Read from a File

1. **Create a Class Called Student**:

- Attributes: studentId (int), name (String), age (int), grade (double).
- **To Do**: Define these attributes and create a constructor to initialize them.

2. **Override the `toString()` Method**:

- **To Do**: Override `toString()` so that it returns a formatted string displaying each attribute of `Student`.

3. **Read Data from `students.csv`**:

- Use the provided code to read data from the CSV file. Complete the code to parse each line, split the data, and create a `Student` object for each line:

```
while ((line = br.readLine()) != null) {
    String[] data = line.split(",");
    int studentId = Integer.parseInt(data[0]);
    String name = data[1];
    int age = Integer.parseInt(data[2]);
    double grade = Double.parseDouble(data[3]);

    // Create a Student object and add it to the list
    students.add(new Student(studentId, name, age, grade));
}
```

4. **Print all students from the list of students**

Task 3: Implement equals() and Check for Duplicates

1. **Override the equals() method in Student**:

Objective: Define equality by studentId. If two students have the same studentId, they are considered equal.

2. **Check for Duplicate Students**:

To Do: Write code in `main` to check if there are duplicate students in the `students` list by comparing student IDs. You can create this using nested *for* loop where you will use the already implemented equals() method to check if there are identical students. If you find at least one, you can break, denoting that there are duplicate students.

Task 3: Sort Students by Grade and Name

1. **Implement Comparable<Student> for Sorting by grade**:

Objective: Sort students by grade in descending order.

2. **Write a `sortByName` Method to Sort Alphabetically**:

To Do: In the `Student` class, add a static method `sortByName` that uses `Comparator` to sort students alphabetically by `name`.

3. **Test Sorting**:

To Do: In `main`, call `Collections.sort(students)` to sort by `grade`, then print the list.
To Do: Call `Student.sortByName(students)` to sort by `name`, then print the list again.

Note:

In Java, both Comparable and Comparator are used to define sorting logic for objects, but they serve slightly different purposes and are used in different scenarios:

1. Comparable Interface

- **Purpose**: Defines a **natural ordering** for objects of a particular class, often based on one main attribute.
- **How It Works**:
  - You implement Comparable in the class and override the compareTo() method to specify the default sorting logic.
  - For example, if you want Student objects sorted by grade, you'd implement Comparable<Student> and override compareTo() in the Student class.
- **Use When**:
  - You have a primary or "default" way to sort objects.

  - Sorting logic is directly tied to the class (inherent sorting order).
- **Limitations**:
  - Only one sort order can be defined because compareTo() can only be implemented once in a class.
  - If you need multiple sorting criteria (e.g., sort by name sometimes, and by grade other times), Comparable alone won't suffice.

2. `Comparator` Interface

- **Purpose**: Defines **custom or alternate ordering** for objects, typically based on different or multiple criteria.
- **How It Works**:
  - `Comparator` is implemented outside the class (as an external sorting rule) and allows you to create as many comparison rules as needed by implementing the `compare()` method.
  - You pass a `Comparator` to `Collections.sort()` to sort objects by different attributes.
- **Use When**:
  - You need multiple sorting criteria for the same class.
  - You don't want or can't modify the class to add a natural ordering (for instance, if it's from an external library).
- **Advantages**:
  - Flexible: You can create multiple `Comparator` implementations for different sorting requirements.
  - Keeps the sorting logic separate from the actual class.

Task 4: Sorting Students by Multiple Criteria

1. **Define Natural Order with Comparable**:
   - Update the Student class to implement Comparable so that students are naturally sorted by grade in descending order. This will establish the primary sorting criterion.
2. **Add a Secondary Sorting Criterion with a Comparator**:
   - Create a Comparator that sorts students by grade first and, if two students have the same grade, sorts them by name alphabetically. This Comparator will allow flexible sorting criteria beyond the natural order.
3. **Sort and Print Using Both Criteria**:
   - In the main program:
     - First, sort and print students by the natural ordering (i.e., grade).
     - Then, use the custom Comparator to sort and print students by both grade and name.
4. **Expected Outcome**:
   - The first sorted list should display students ordered by grade (highest to lowest).
   - The second sorted list should show students ordered by grade and, within identical grades, by name alphabetically.