Exercise 1: Use `@Override` Correctly

✅ **Problem Statement:**

Create a **parent** class `Animal` with a method `makeSound()`. Then, create a `Dog` class that **overrides** this method using `@Override`.

🔹 **Steps to Follow:**

1. Define a `makeSound()` method in `Animal` class.
2. Override it in `Dog` class with `@Override`.
3. Instantiate `Dog` and call `makeSound()`.

```
class Animal {
    void makeSound() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Bark");
    }
}

public class AnimalMain {
    public static void main(String[] args) {
        Dog obj = new Dog();
        obj.makeSound();
    }
}
```

Exercise 2: Use `@Deprecated` to Mark an Old Method

✅ Problem Statement:

Create a class `LegacyAPI` with an **old** method `oldFeature()`, which should not be used anymore. Instead, introduce a **new** method `newFeature()`.

🔹 Steps to Follow:

1. Define a class `LegacyAPI`.
2. Mark `oldFeature()` as `@Deprecated`.
3. Call both methods and observe the warning.

```java
class LegacyAPI {
    @Deprecated
    void oldFeature() {
        System.out.println("Old feature");
    }

    void newFeature() {
        System.out.println("New feature");
    }
}

public class LegacyAPIMain {
    public static void main(String[] args) {
        LegacyAPI api = new LegacyAPI();
        api.oldFeature();
        api.newFeature();
    }
}
```

Exercise 3: Suppress Unchecked Warnings

✅ **Problem Statement:**

Create an `ArrayList` without generics and use `@SuppressWarnings("unchecked")` to **hide** compilation warnings.

```
import java.util.*;

public class SuppressWarningsMain {

    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add("Test");
        System.out.println(list.get(0));
    }
}
```

Exercise 4: Create a Custom Annotation and Use It

✅ **Problem Statement:**

Create a custom annotation `@TaskInfo` to mark **tasks** with priority and assigned person.

🔹 **Steps to Follow:**

1. Define an annotation `@TaskInfo` with fields `priority` and `assignedTo`.
2. Apply this annotation to a method in `TaskManager` class.
3. Retrieve the annotation details using **Reflection API**.

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface TaskInfo {
    String priority();

    String assignedTo();
}

class TaskManager {
    @TaskInfo(priority = "High", assignedTo = "ABC")
    void task() {
        System.out.println("Task executed");
    }
}

public class TaskInfoMain {
    public static void main(String[] args) throws Exception {
        Method m = TaskManager.class.getMethod("task");
        TaskInfo info = m.getAnnotation(TaskInfo.class);
        System.out.println("Priority: " + info.priority());
        System.out.println("Assigned To: " + info.assignedTo());
    }
}
```

Exercise 5: Create and Use a Repeatable Annotation

✅ Problem Statement:

Define an annotation `@BugReport` that can be applied **multiple times** on a method.

🔹 **Steps to Follow:**

1. Define `@BugReport` with a `description` field.
2. Use `@Repeatable` to allow multiple bug reports.

3. Apply it twice on a method.
4. Retrieve and print all bug reports.

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Repeatable(BugReports.class)
@interface BugReport {
    String description();
}

@Retention(RetentionPolicy.RUNTIME)
@interface BugReports {
    BugReport[] value();
}

class Software {
    @BugReport(description = "Null pointer issue")
    @BugReport(description = "Memory leak detected")
    void process() {
        System.out.println("Processing");
    }
}

public class BugReportMain {
    public static void main(String[] args) throws Exception {
        Method m = Software.class.getMethod("process");
        BugReports reports = m.getAnnotation(BugReports.class);
        for (BugReport report : reports.value()) {
            System.out.println("Bug: " + report.description());
        }
    }
}
```

Practice Problems for Custom Annotations in Java

Beginner Level

1️⃣ Create an Annotation to Mark Important Methods

✅ **Problem Statement:**

Define a custom annotation `@ImportantMethod` that can be applied to methods to indicate their importance.

🔹 **Requirements:**

- Define `@ImportantMethod` with an optional `level` parameter (default: `"HIGH"`).
- Apply it to at least two methods.
- Retrieve and print annotated methods using **Reflection API**.

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface ImportantMethod {
    String level() default "HIGH";
}

class Utility {
    @ImportantMethod(level = "HIGH")
    void criticalTask() {
        System.out.println("Executing critical task");
    }

    @ImportantMethod(level = "MEDIUM")
    void regularTask() {
        System.out.println("Executing regular task");
    }
}

public class ImportantMethodMain {
    public static void main(String[] args) throws Exception {
        Method[] methods = Utility.class.getDeclaredMethods();
        for (Method m : methods) {
            if (m.isAnnotationPresent(ImportantMethod.class)) {
                ImportantMethod im = m.getAnnotation(ImportantMethod.class);
                System.out.println(m.getName() + " - Level: " + im.level());
            }
        }
    }
}
```

2️⃣ Create a @Todo Annotation for Pending Tasks

✅ Problem Statement:

Define an annotation @Todo to mark **pending** features in a project.

🔹 **Requirements:**

- The annotation should have fields:
  - task() (String) → **Description of the task**
  - assignedTo() (String) → **Developer responsible**
  - priority() (default: "MEDIUM")
- Apply it to multiple methods.
- Retrieve and print **all pending tasks** using Reflection.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface Todo {
    String task();

    String assignedTo();
    String priority() default "MEDIUM";
}

class Project {
    @Todo(task = "Implement authentication", assignedTo = "PQR", priority =
"HIGH")
    void loginFeature() {
        System.out.println("Login feature");
    }

    @Todo(task = "Optimize database queries", assignedTo = "XYZ")
    void optimizeDB() {
        System.out.println("Optimizing database");
    }
}

public class TodoMain {
    public static void main(String[] args) throws Exception {
        Method[] methods = Project.class.getDeclaredMethods();
        for (Method m : methods) {
            if (m.isAnnotationPresent(Todo.class)) {
                Todo todo = m.getAnnotation(Todo.class);
                System.out.println(m.getName() + " - Task: " + todo.task() + ",
Assigned To: " + todo.assignedTo()
                        + ", Priority: " + todo.priority());
            }
        }
    }
}
```

Intermediate Level

3️⃣ Create an Annotation for Logging Method Execution Time

✅ Problem Statement:

Define an annotation `@LogExecutionTime` to measure method execution time.

🔹 **Requirements:**

- Apply `@LogExecutionTime` to a method.
- Use `System.nanoTime()` before and after execution.
- Print execution time.
- Apply it on different methods and compare the time taken.

```java
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface LogExecutionTime {}

class Performance {
    @LogExecutionTime
    void task() {
        long start = System.nanoTime();
        for (int i = 0; i < 1000000; i++);
        long end = System.nanoTime();
        System.out.println("Execution Time: " + (end - start) + " ns");
    }
}

public class LogExecutionTimeMain {
    public static void main(String[] args) {
        new Performance().task();
    }
}
```

4️⃣ Create a `@MaxLength` Annotation for Field Validation

✅ Problem Statement:

Define a field-level annotation `@MaxLength(int value)` that restricts the **maximum length** of a `String` field.

🔹 **Requirements:**

- Apply it to a `User` class field (`username`).
- Validate length in the constructor.
- Throw `IllegalArgumentException` if the limit is exceeded.

```java
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MaxLength {
    int value();
}
class User {
    @MaxLength(5)
    String username;

    User(String username) {
        if (username.length() > 5) {
            throw new IllegalArgumentException("Username too long");
        }
        this.username = username;
    }
}

public class MaxLengthMain {
    public static void main(String[] args) {
        User user = new User("John");
        System.out.println("Username: " + user.username);
    }
}
```

Advanced Level

5️⃣ Implement a Role-Based Access Control with `@RoleAllowed`

✅ **Problem Statement:**

Define a class-level annotation `@RoleAllowed` to restrict method access based on roles.

🔹 **Requirements:**

- `@RoleAllowed("ADMIN")` should **only allow ADMIN users** to execute the method.
- Simulate user roles and validate access before invoking the method.
- If a non-admin tries to access it, print **Access Denied!**

```java
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@interface RoleAllowed {
    String value();
}

class SecureSystem {
    @RoleAllowed("ADMIN")
    void secureTask(String role) {
        if (!role.equals("ADMIN")) {
            System.out.println("Access Denied!");
            return;
        }
        System.out.println("Secure task executed");
    }
}
public class RoleAllowedMain {
    public static void main(String[] args) {
        SecureSystem obj = new SecureSystem();
        obj.secureTask("USER");
        obj.secureTask("ADMIN");
    }
}
```

6️⃣ Implement a Custom Serialization Annotation `@JsonField`

✅ Problem Statement:

Define an annotation `@JsonField` to mark **fields** for JSON serialization.

🔹 **Requirements:**

- `@JsonField(name = "user_name")` should **map field names** to custom JSON keys.
- Apply it on a `User` class.
- Write a method to **convert object to JSON string** by reading the annotations.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface JsonField {
    String name();
}

class Person {
    @JsonField(name = "user_name")
    String username = "ABC";
}

public class JsonFieldMain {
    public static void main(String[] args) throws Exception {
        Person person = new Person();
        Field field = person.getClass().getDeclaredField("username");
        JsonField annotation = field.getAnnotation(JsonField.class);
        System.out.println("{\"" + annotation.name() + "\": \"" +
field.get(person) + "\"}");
    }
}
```

7️⃣ Implement a Custom Caching System with `@CacheResult`

✅ **Problem Statement:**

Define `@CacheResult` to store method return values and **avoid repeated execution**.

🔹 **Requirements:**

- Apply `@CacheResult` to a computationally expensive method.
- Implement a **cache (HashMap)** to store previously computed results.
- If method is called with the same input, return cached result instead of re-computation.

```java
import java.lang.annotation.*;
import java.util.HashMap;

@Retention(RetentionPolicy.RUNTIME)
@interface CacheResult {
}

class Calculator {
    private HashMap<Integer, Integer> cache = new HashMap<>();

    @CacheResult
    int square(int num) {
        if (cache.containsKey(num)) {
            return cache.get(num);
        }
        int result = num * num;
        cache.put(num, result);
        return result;
    }
}

public class CacheResultMain {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.square(5));
        System.out.println(calc.square(5));
    }
}
```