MODULE: Java Programming

ASSIGNMENT 6 – Design Patterns, Custom Annotation & Java Reflection

Learning Objectives

By completing this assignment, learners will:

- Understand and implement the Singleton Design Pattern in multiple ways
- Handle common pitfalls in Singleton (cloning, serialization, reflection attacks)
- Create and use Custom Java Annotations
- Apply Java Reflection APIs to inspect annotations at runtime
- Strengthen understanding of metadata-driven programming
- Build developer-ready skills used in Spring, Hibernate, Jakarta EE, etc.

General Instructions

1. Write each question in a separate package or clearly separated section.
2. Follow Java naming conventions for classes, annotations, and methods.
3. For annotation exercises, ensure **RetentionPolicy.RUNTIME** is used.
4. Demonstrate all results clearly using a test class.
5. Handle exceptions gracefully when using reflection.
6. Keep code modular and well-commented for learning clarity.

Estimated Time & Difficulty

Question	Time	Difficulty
Q1	60–90 min	Intermediate → Advanced
Q2	45–60 min	Intermediate
Q3	45–60 min	Advanced

Evaluation Rubric

Criteria	Weight
Correct Implementation	40%
Understanding of Pattern Pitfalls	20%
Annotation Design & Usage	20%
Reflection Logic	15%

Criteria	Weight
Output Clarity	5%

Q1. Implement Singleton Pattern (All Variations + Pitfall Handling)

🌍 Real-World Context

Singleton is heavily used in frameworks like Spring, logging systems, DB connection pools, config loaders, etc.

Task Requirements

Implement the following **8 variations** of Singleton:

1. Eager Initialization Singleton

- Instance created during class loading
- Simple but not memory efficient

2. Static Block Initialization

- Similar to eager but allows exception handling

3. Lazy Initialization

- Instance created when requested
- Not thread-safe

4. Thread-Safe Singleton

Implement both:

- synchronized method
- synchronized block (double-checked locking)

5. Serialization Issue Singleton

- Demonstrate how serialization breaks singleton
- Fix it using `readResolve()` method

6. Cloning Issue

- Prevent cloning by overriding `clone()` and throwing exception

7. Reflection Attack Demonstration

- Show how using Reflection can create a second instance
- Fix it by adding guard in constructor

8. Enum Singleton

- The most robust form of singleton in Java
- Immune to serialization, reflection, cloning

Expected Output

- Print instance hashcodes for each variant

- Demonstrate where singleton breaks and how it is fixed

Example

```
Instance 1 hashcode: 12345
Instance 2 hashcode: 12345
Singleton preserved.
```

Common Mistakes to Avoid

- Forgetting private constructor
- Missing volatile keyword in double-checked locking
- Not fixing serialization & reflection issues
- Using cloning without override

Q2. Custom Annotations: Author & Version Metadata

🌎 Real-World Context

Frameworks (Spring, JPA, Hibernate, JUnit) rely heavily on annotations for metadata-driven behaviors.

Part A – Create Annotations

Create two annotations:

1. `@Author`

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Author {
    String name() default "unknown";
}
```

2. `@Version`

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Version {
    double number();
}
```

Part B – Apply Annotations

Create a class `AnnotatedClass`:

```
@Author(name="Johny")
@Version(number=1.0)
public class AnnotatedClass {

    @Author(name="Author1")
    @Version(number=2.0)
    public void annotatedMethod1() { }

    @Author(name="Author2")
    @Version(number=4.0)
    public void annotatedMethod2() { }
}
```

Q3. Read & Display Annotation Metadata Using Reflection

🌍 Real-World Context

Reflection is used by dependency injection containers, ORM mappers, test frameworks, and serialization tools.

Task Requirements

1. Create a utility method:

```
public static void readAnnotation(AnnotatedElement element)
```

Where `AnnotatedElement` may be:

- Class
- Method
- Constructor

2. Inside the method:

- Get all annotations using `element.getAnnotations()`
- Loop through the array
- Downcast to `Author` or `Version` as appropriate
- Print annotation metadata:

Example:

```
Author: Johny
Version: 1.0
```

3. Test reflection on:

- The class level
- Each annotated method

Expected Output Example

```
Class: AnnotatedClass
Author: Johny
Version: 1.0

Method: annotatedMethod1
Author: Author1
Version: 2.0

Method: annotatedMethod2
Author: Author2
Version: 4.0
```

Common Mistakes to Avoid

- Forgetting RUNTIME retention → reflection won't work
- Using wrong import (`java.lang.annotation.*`)
- Not checking annotation type before casting

- Missing handling for methods without annotations

BONUS CHALLENGES (Optional)

⭐ Bonus 1 — Scan Entire Package for Annotated Classes

Use `Reflections`-style logic (manual or library-free).

⭐ Bonus 2 — Build a Simple Dependency Injection Container

Use `@Inject` annotation + reflection.

⭐ Bonus 3 — Annotation Validation Logic

Create `@MandatoryField` and use reflection to enforce constraints.

⭐ Bonus 4 — Combine Singleton + Annotation

Use annotation metadata to generate different singleton strategies.

Reflection Questions

1. Why is Enum considered the safest Singleton implementation?
2. How do annotations reduce boilerplate in Java frameworks?
3. What are risks of using reflection excessively in production systems?
4. How does adding `readResolve()` preserve Singleton behavior?
5. How would you design a new annotation for API documentation?