CodInClub

Powered By BridgeLabz

Practice Problems on Reflection in Java

- **Basic Level**

1. **Get Class Information:** Write a program to accept a class name as input and display its methods, fields, and constructors using Reflection.

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

import java.util.Scanner;



public class ClassInfo {

    public static void main(String[] args) throws ClassNotFoundException {



        Scanner sc = new Scanner(System.in);

        System.out.print("Enter class name: ");

        String className = sc.nextLine();

        Class<?> clazz = Class.forName(className);

        System.out.println("Methods: ");

        for (Method method : clazz.getDeclaredMethods()) {

            System.out.println(method.getName());

        }

        System.out.println("Fields: ");

        for (Field field : clazz.getDeclaredFields()) {

            System.out.println(field.getName());
```

```
        }

        System.out.println("Constructors: ");

        for (Constructor<?> constructor : clazz.getDeclaredConstructors())
{

            System.out.println(constructor.getName());

        }



        sc.close();

    }

}
```

2. **Access Private Field:** Create a class Person with a private field age. Use Reflection to modify and retrieve its value.

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



class Person {

    private int age = 25;



    public int getAge() {

        return age;

    }
```

```
}



public class PrivateField {

    public static void main(String[] args) throws Exception {

        Person p = new Person();

        Field field = Person.class.getDeclaredField("age");

        field.setAccessible(true);

        field.set(p, 30);

        System.out.println("Modified Age: " + field.get(p));

    }

}
```

3. **Invoke Private Method:** Define a class Calculator with a private method multiply(int a, int b). Use Reflection to invoke this method and display the result.

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



class Calculator {

    private int multiply(int a, int b) {

        return a * b;

    }

}
```

```
public class PrivateMethod {

    public static void main(String[] args) throws Exception {

        Calculator calc = new Calculator();

        Method method = Calculator.class.getDeclaredMethod("multiply",
int.class, int.class);

        method.setAccessible(true);

        int result = (int) method.invoke(calc, 5, 6);

        System.out.println("Result: " + result);

    }

}
```

4. **Dynamically Create Objects:** Write a program to create an instance of a Student class dynamically using Reflection without using the new keyword.

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

class Student {
    public Student() {
        System.out.println("Student object created");
    }
}

public class DynamicObject {
    public static void main(String[] args) throws Exception {
        Constructor<Student> constructor =
Student.class.getConstructor();
```

```
        Student student = constructor.newInstance();

        System.out.println(student);
    }
}
```

- **Intermediate Level**

5. **Dynamic Method Invocation:** Define a class MathOperations with multiple public methods (add, subtract, multiply). Use Reflection to dynamically call any method based on user input.

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

import java.util.Scanner;



class MathOperations {

    public int add(int a, int b) {

        return a + b;

    }



    public int subtract(int a, int b) {

        return a - b;

    }



    public int multiply(int a, int b) {
```

```
        return a * b;

    }

}



public class DynamicMethod {

    public static void main(String[] args) throws Exception {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter method name: ");

        String methodName = sc.nextLine();

        MathOperations obj = new MathOperations();

        Method method = MathOperations.class.getMethod(methodName,
int.class, int.class);

        System.out.println("Result: " + method.invoke(obj, 5, 3));



        sc.close();

    }

}
```

6. **Retrieve Annotations at Runtime:** Create a custom annotation @Author(name="Author Name"). Apply it to a class and use Reflection to retrieve and display the annotation value at runtime.

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

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

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

@Author(name = "John Doe")
class Document {
}

public class RetrieveAnnotation {
    public static void main(String[] args) {
        Author author = Document.class.getAnnotation(Author.class);
        System.out.println("Author: " + author.name());
    }
}
```

7. **Access and Modify Static Fields:** Create a Configuration class with a private static field API_KEY. Use Reflection to modify its value and print it.

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

class Configuration {
```

```
    private static String API_KEY = "123456";



}



public class StaticField {

    public static void main(String[] args) throws Exception {

        Field field = Configuration.class.getDeclaredField("API_KEY");

        field.setAccessible(true);

        field.set(null, "654321");

        System.out.println("Modified API_KEY: " + field.get(null));

    }

}
```

- **Advanced Level**

8. **Create a Custom Object Mapper:** Implement a method toObject(Class<T> clazz, Map<String, Object> properties) that uses Reflection to set field values from a given Map.

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

```
import java.util.*;



class ObjectMap {

    public static <T> T toObject(Class<T> clazz, Map<String, Object>
properties) throws Exception {

        T obj = clazz.getDeclaredConstructor().newInstance();

        for (Field field : clazz.getDeclaredFields()) {

            field.setAccessible(true);

            if (properties.containsKey(field.getName())) {

                field.set(obj, properties.get(field.getName()));

            }

        }

        return obj;

    }

}



class User {

    public String name;

    public int age;

}



public class ObjectMapper {

    public static void main(String[] args) throws Exception {
```

```
        Map<String, Object> properties = new HashMap<>();

        properties.put("name", "John");

        properties.put("age", 30);

        User user = ObjectMap.toObject(User.class, properties);

        System.out.println("User: " + user.name + ", Age: " + user.age);

    }

}
```

9. **Generate a JSON Representation:** Write a program that converts an object to a JSON-like string using Reflection by inspecting its fields and values.

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

// import java.util.*;



class JsonGenerator {

    public static String toJson(Object obj) throws Exception {

        StringBuilder json = new StringBuilder("{");

        Class<?> clazz = obj.getClass();

        Field[] fields = clazz.getDeclaredFields();

        for (int i = 0; i < fields.length; i++) {

            fields[i].setAccessible(true);
```

```
            json.append("\"" + fields[i].getName() + "\": \"" +
fields[i].get(obj) + "\"");

            if (i < fields.length - 1)

                json.append(", ");

        }

        json.append("}");

        return json.toString();

    }

}



public class JsonGeneratorMain {

    public static void main(String[] args) throws Exception {

        User user = new User();

        user.name = "ABC";

        user.age = 25;

        System.out.println(JsonGenerator.toJson(user));

    }

}
```

10. **Custom Logging Proxy Using Reflection:** Implement a **Dynamic Proxy** that intercepts method calls on an interface (e.g., Greeting.sayHello()) and logs the method name before executing it.

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



interface Greeting {

    void sayHello();

}



class GreetingImpl implements Greeting {

    public void sayHello() {

        System.out.println("Hello, world!");

    }

}



class LoggingHandler implements InvocationHandler {

    private final Object target;



    public LoggingHandler(Object target) {

        this.target = target;

    }



    public Object invoke(Object proxy, Method method, Object[] args)
throws Throwable {

        System.out.println("Executing method: " + method.getName());

        return method.invoke(target, args);
```

```
    }

}



public class LoggingProxy {

    public static void main(String[] args) {

        Greeting greeting = (Greeting) Proxy.newProxyInstance(

                Greeting.class.getClassLoader(),

                new Class[] { Greeting.class },

                new LoggingHandler(new GreetingImpl()));

        greeting.sayHello();

    }

}
```

11. **Dependency Injection using Reflection:** Implement a simple **DI container** that scans classes with @Inject annotation and injects dependencies dynamically.

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

// import java.util.*;



@interface Inject {

}
```

```
class Service {

}



class Client {

    @Inject

    private Service service;



    public Service getService() {

        return service;

    }

}



public class DependencyInjection {

    public static void main(String[] args) throws Exception {

        Client client = new Client();

        for (Field field : Client.class.getDeclaredFields()) {

            if (field.isAnnotationPresent(Inject.class)) {

                field.setAccessible(true);

                field.set(client, new Service());

            }

        }

        System.out.println("Dependency Injected: " + (client.getService()
!= null));
```

```
    }

}
```

12. **Method Execution Timing:** Use Reflection to measure the execution time of methods in a given class dynamically.

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



class TestClass {

    public void slowMethod() throws InterruptedException {

        Thread.sleep(500);

    }

}



public class ExecutionTiming {

    public static void main(String[] args) throws Exception {

        Method method = TestClass.class.getMethod("slowMethod");

        TestClass obj = new TestClass();

        long start = System.nanoTime();

        method.invoke(obj);

        long end = System.nanoTime();

        System.out.println("Execution time: " + (end - start) / 1_000_000
+ " ms");
```

```
        }

}
```