Homework 2

CS474 - Fall 2019

Description

In this homework, you will use Java's reflection to create an object and class inspector. Your inspector should answer correctly a number of queries, specified by implementing the provided `Inspector` interface. This document describes the interface in detail.

Due Date and Late Policy

- This homework is due on **October 19** (Saturday) by **5pm CST**.
- Homeworks delivered by **October 20** (Sunday) by **5pm CST** will have a 10% penalty.
- Homeworks delivered by **October 21** (Monday) by **5pm CST** will have a 25% penalty.
- **No homeworks will be accepted after October 21 by 5pm CST.**

Homework 2 will use **Github** to keep your code and deliver the homework. Your grade is the grade of the most recent commit in your repository, subject to penalties as described above. The date of the most recent commit will be used to determine the penalty to apply, if any.

Java Reflection

To complete this homework, you must use the Java reflection framework, located in package java.lang.reflect. How to use the Java reflection was covered in class. The startup code provided is listed in this document under Appendix A, Appendix B, Appendix C, Appendix D, and Appendix E.

Inspector Interface

For this homework, you will have to implement the Inspector interface; which will be provided for you to start the homework. **You can only modify the file that implements the Inspector interface.** This interface consists of 5 methods:

1. `findClass`: Finds a class and returns an instance of `ClassInfo` if the class exists. Otherwise returns `empty`.
2. `fillInFields`: Fills in all the information for fields on the provided `ClassInfo` instance.
3. `fillInMethodsAndConstructors`: Fills in all the information about methods and constructors on the provided `ClassInfo` instance.

(continues on the next page)

4. `readWriteField`: Reads or writes a field, specified by the given instance of `FieldInfo`.
    - An optional object where to read/write fields
    - An optional new value to write (empty means read)
    - A flag to force the operation on fields without sufficient permission (e.g., writing to a final field or reading from a private field).
    - Special return values:
        - Reading null: The Inspector should return an optional with Inspector.NULL inside
        - Successful write: The Inspector should return an optional with Inspector.SUCCESS inside
5. `invokeMethodOrConstructor`: Invokes a method or a constructor
    - The optional receiver contains the object on which the method will be invoked on (empty means either static field or constructor).
    - A list of objects for the arguments
    - A flag to force the invocation on methods/constructors without sufficient permission (e.g., private methods or private constructors).
    - For methods that return void, the Inspector should return an optional with Inspector.NULL inside

Format of Class Names

All class names should be represented by their **fully qualified name**. For instance, class `String` should be represented as `java.lang.String`.
Primitive types should be represented as follows: `int, long, float, double, short, char, byte, boolean`.

Overridden Methods

Overridden methods should only appear once in `fillInMethods`. For instance, if a class defines a method `toString`, then the method `Object.toString` should not be present in the methods of the class. A method is overridden if it has the same name, and the same number and type of arguments as another method on any super class.

Failures and Exceptions

The first four methods of JavaInspector shall never throw any exception:

- If a class does not exist, the inspector should return empty.
- If a field does not have enough permissions and the flag force is not set, the inspector should return empty.
- If an attempt to write an incompatible type is made, the inspector should return empty.

(continues on the next page)

The `invokeMethodOrConstructor` throws exceptions only if the invocation is successful and the invoked method threw an exception. It should throw the same exception as the invoked method threw. It should return empty instead of throwing exceptions if:

- Any argument has the wrong type.
- The method does not have enough permissions and the flag force is not set.

How Much Data to Fill In

The enumerate HowMuchData controls how much data the inspector should fill in:

- `DECLARED`
    - findClass: Only the class name and parent name
    - fillInField: Only the fields declared in the owner class
    - fillInMethodsAndConstructors: Only the methods and constructors declared in the owner class
- `ALL`
    - findClass: All the parents and interfaces immediately implemented by the parents
    - fillInField: All the fields in all the parent classes, and all the fields in all the interfaces implemented directly by the owner class or any parent

(continues on the next page)

    - fillInMethodsAndConstructors: All the methods and constructors in all the parent classes, and all the methods in all the interfaces implemented directly by the owner class or any parent

- `ALL_INTERFACES`
    - findClass: `ALL` plus all the interfaces implemented by the interfaces in `ALL`
    - fillInField: `ALL` plus all the fields in interfaces implemented by the interfaces in `ALL`
    - fillInMethodsAndConstructors: `ALL` plus all the methods in interfaces implemented by the interfaces in `ALL`

Submission and Grading

This homework should be submitted through Github, and it is automatically graded. You can check your current grade at any point simply by running all the tests. The project is graded through 25 tests, that will check if your homework behaves as described in this document. Each test is worth 4%, and you will be provided with all 25 tests.

(continues on the next page)

You are encouraged to **make (and push) frequent commits** with small changes between each. This ensures that:

1. Your code is backed-up in case something happens to your computer;
2. You can revert changes that break something without much effort;
3. The instructors can track the progress of the class and provide hints for free for everyone in a timely manner;
4. The instructors can look into your code for a particular question you ask.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your homework with `System.out.println("expected result")`.

The academic integrity policy described in the syllabus applies to this homework. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.

Appendix A: `Inspector` Interface

```
// You should not change this file
package edu.uic.cs474.hw2;

import java.util.*;

public interface Inspector {
    public static final Object NULL = new Object();
    public static final Object SUCCESS = new Object();

    public Optional<ClassInfo> findClass(String fullyQualifiedName, HowMuchData howMuch);

    public void fillInFields(ClassInfo info, HowMuchData howMuch);

    public void fillInMethodsAndConstructors(ClassInfo info, HowMuchData howMuch);

    public Optional<Object> readWriteField(FieldInfo field, Optional<Object> o,
Optional<Object> newValue, boolean force);

    public Optional<Object> invokeMethodOrConstructor(MethodInfo method, Optional<Object>
receiver, LinkedList<Object> arguments, boolean force) throws Throwable;
}
```

Appendix B: Enumerate HowMuchData

```
// You should not change this file
package edu.uic.cs474.hw2;

public enum HowMuchData {
    DECLARED,
    ALL,
    ALL_INTERFACES,
}
```

Appendix C: Helper Class `ClassInfo`

```
// You should not change this file
package edu.uic.cs474.hw2;

import java.util.*;

public class ClassInfo {
    public String name;
    public Optional<ClassInfo> parent;
    public HashSet<ClassInfo> interfaces = new HashSet<>();

    public HashSet<FieldInfo> fields = new HashSet<>();
    public HashSet<MethodInfo> constructors = new HashSet<>();
    public HashSet<MethodInfo> methods = new HashSet<>();

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        ClassInfo classInfo = (ClassInfo) o;
        return Objects.equals(name, classInfo.name) &&
                Objects.equals(parent, classInfo.parent) &&
                Objects.equals(interfaces, classInfo.interfaces) &&
                Objects.equals(fields, classInfo.fields) &&
                Objects.equals(constructors, classInfo.constructors) &&
                Objects.equals(methods, classInfo.methods);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, parent, interfaces, fields, constructors, methods);
    }
```

```
   @Override
    public String toString() {
        return "ClassInfo{" +
                "name='" + name + '\'' +
                ", parent=" + (parent.isPresent() ? parent.get().name : "N/A") +
                ", interfaces=" + interfaces +
                ", fields=" + fields +
                ", constructors=" + constructors +
                ", methods=" + methods +
                '}';
    }
}
```

Appendix D: Helper Class `MethodInfo`

```
// You should not change this file
package edu.uic.cs474.hw2;

import java.util.*;

public class MethodInfo {
    public String declarerClassName;
    public Optional<String> name;
    public Optional<String> returnType;
    public LinkedList<String> argumentTypes = new LinkedList<>();

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        MethodInfo that = (MethodInfo) o;
        return Objects.equals(declarerClassName, that.declarerClassName) &&
                Objects.equals(name, that.name) &&
                Objects.equals(returnType, that.returnType) &&
                Objects.equals(argumentTypes, that.argumentTypes);
    }
    @Override
    public int hashCode() {
        return Objects.hash(declarerClassName, name, returnType, argumentTypes);
    }

    @Override
    public String toString() {
        return "MethodInfo{" +
                "declarerClassName='" + declarerClassName + '\'' +
                ", name=" + name +
                ", returnType='" + returnType + '\'' +
                ", argumentTypes=" + argumentTypes +
                '}';
    }
}
```

Appendix E: Helper Class FieldInfo

```
// You should not change this file
package edu.uic.cs474.hw2;

import java.util.Objects;

public class FieldInfo {
    public String declarerClassName;
    public String name;
    public String type;

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        FieldInfo fieldInfo = (FieldInfo) o;
        return Objects.equals(declarerClassName, fieldInfo.declarerClassName) &&
                Objects.equals(name, fieldInfo.name) &&
                Objects.equals(type, fieldInfo.type);
    }

    @Override
    public int hashCode() {
        return Objects.hash(declarerClassName, name, type);
    }
    @Override
    public String toString() {
        return "FieldInfo{" +
                "declarerClassName='" + declarerClassName + '\'' +
                ", name='" + name + '\'' +
                ", type='" + type + '\'' +
                '}';
    }
}
```