ASSIGNMENT 2

CS 474: Object-Oriented Languages and Environments / Fall 2020

Description

In this assignment, you will have to write a utility to call methods using Java reflection.

For this assignment, you need to write a class that implements the interface `ReflectionUtils`, as shown below. You will be provided with startup code that you cannot modify (except as described in this document), and you only have to submit an implementation to the interface described below.

```
interface ReflectionUtils {
  Optional invokeMethod(String owner, String name, boolean force, Optional receiver, Object ... args);

  Optional invokeMethod(Class owner, String name, boolean force, Optional receiver, Object ... args);
}
```

Method `invokeMethod` invokes the method as described by the arguments:

- `owner`: The `java.lang.Class` that owns the method to invoke, or a `java.lang.String` with the name of the owner class
- `name`: The name of the method to invoke
- `force`: See Forced Execution below
- `receiver`: The receiver object
  - This argument may be absent or not for static methods, your implementation should follow the rules on the Static Methods section below
- `arg`: The arguments to pass to the method being invoked

Method `invokeMethod` returns the following:

- An optional with the value returned by the target method, if the target method does not return `null`.
- An optional with `ReflectionUtils.NULL` if the target method returns `null`.
- An optional with `ReflectionUtils.VOID` if the target method returns `void`.
- An empty optional in case it is not possible to invoke the target method (e.g., the target method is private or does not exist).

Boxing and Primitive Types

Java performs autoboxing of primitive types, which means that your code will be called with `java.lang.Integer` arguments instead of `int`. When this happens, you need to find the appropriate method that does not use autoboxing, and call that method.

For instance, `invokeMethod("C","m", ... , new C(), new Integer(10))` and `invokeMethod("C","m", ... , new C(),10)` should both print `I10` and not `Integer10` for class C defined below.

```
class C {
  void m(int i)     { print("I" + i); }
  void m(Integer i) { print("Integer" + i); }
}
```

Exception Handling

Method `invokeMethod` should behave as follows, in the presence of exceptions:

- Return `empty` when it is not possible to invoke the target method (e.g., method is private, method does not exist, wrong arguments, wrong receiver type, etc.)
- Return an `Optional` with an exception, if the target method throws a checked exception when executing.
- Throw an exception, if the target method throws an unchecked exception when executing. Method invokeMethod should throw the same exception as the target method threw.

Forced Execution

If the argument force is set to true, then the behavior of the method should ignore any Java protection mechanisms. For instance:

```
class Z { private void m() { print("Z"); } }

invokeMethod(Z.class, "m", new Z(), false, Optional.empty(), Optional.empty());
// Returns Optional.empty()

invokeMethod(Z.class, "m", new Z(), true, Optional.empty(), Optional.empty());
// Prints Z and returns Optional.of(ReflectionUtils.VOID)

invokeMethod(Z.class, "m", new Z(), true, Optional.empty(), Optional.empty());
// Returns Optional.empty()
```

Static Methods

Inheritance

Your submission should handle inheritance for static methods correctly, as shown below.

```
class A { public static void m () { print("inherited" } }
class B { }

invokeM(B.class, "m", Optional.empty()); // Prints inherited
```

Receiver

Static methods have no receiver, but your submission may be called with a receiver for static methods. When this happens, the receiver is the first argument and requires you to perform multiple dispatch on that argument. This means that you should invoke the static method with the same name and the type of the first argument closest to the receiver provided. For instance, consider the example below:

```
class M {
  static void m(A a) { print("A"); }
  static void m(C a) { print("C"); }
  static void m(D a) { print("D"); }
}

class A { }
class B extends A { }
class C extends B { }
class D extends C { }
class E extends D { }

invokeM(M.class, "m", Optional.of(new A()); // Prints A
invokeM(M.class, "m", Optional.of(new B()); // Prints A
invokeM(M.class, "m", Optional.of(new C()); // Prints C
invokeM(M.class, "m", Optional.of(new D()); // Prints D
invokeM(M.class, "m", Optional.of(new E()); // Prints D
```

Implementation Hint: Note that you can find the correct method to call by traversing the receiver's class hierarchy until you find an exact match with the first argument type. For instance, there is no `m(E)`, but by traversing `E`'s hierarchy (E -> D -> C -> B -> A), the first exact match we find for `m(D)` is the correct method to invoke.

Entry Point

You should create a new class, on a new file, where you will implement your solution. You should change method `Main.getReflectionUtility` so that it creates an instance the class you added. You cannot change any other part of the code that is provided to you.

```
public abstract class Main {
    static ReflectionUtils getReflectionUtility() {
        throw new Error("Not implemented");
    }
}
```

Due Date and Resubmission Policy

This assignment is due on **October 3 2020** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%**. You can resubmit your assignment until **October 10 2020** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **10% bonus points**, which you can earn by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment2` label, and non-anonymous to the instructors to count towards the bonus.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking Travis. The automatic grade is determined by 10 tests, that will check if your project outputs the expected result. Each test is worth 10%.

Graduate students should also submit a video explaining their solution. For graduate students, each test is worth 9%, for a total of 90%, and the video is worth 10%. **The maximum length for the video is 5 minutes.** This video should be a screencast of their IDE open on the code submitted, and the student should highlight the code and narrate the purpose of the highlighted code. You can record such a video without installing any software by using the following website: https://screenapp.io/#/

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

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 submission with `System.out.println("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. 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.