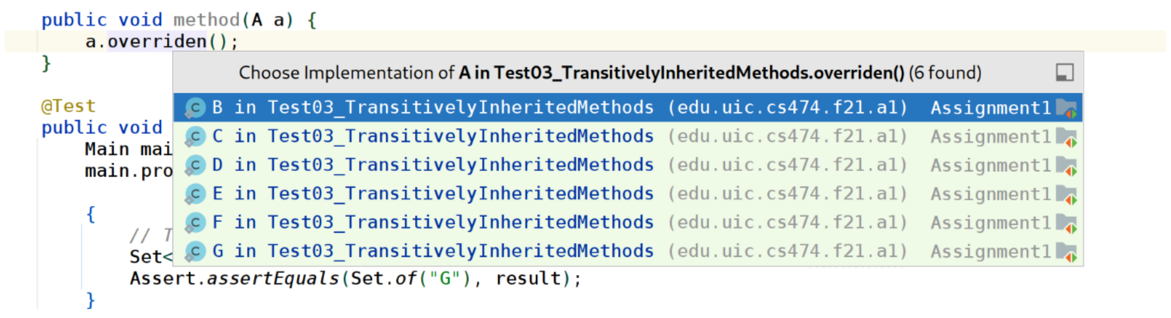
ASSIGNMENT 1

CS 474: Object-Oriented Languages and Environments / Fall 2021



```
public void method(A a) {
    a.overriden();
}

@Test
public void
    Main mai
    main.pro

    {
        // T
        Set<
        Assert.assertEquals(Set.of("G"), result);
    }
```

Choose Implementation of **A in Test03_TransitivelyInheritedMethods.overriden()** (6 found)

```
B in Test03_TransitivelyInheritedMethods (edu.uic.cs474.f21.a1)  Assignment1
C in Test03_TransitivelyInheritedMethods (edu.uic.cs474.f21.a1)  Assignment1
D in Test03_TransitivelyInheritedMethods (edu.uic.cs474.f21.a1)  Assignment1
E in Test03_TransitivelyInheritedMethods (edu.uic.cs474.f21.a1)  Assignment1
F in Test03_TransitivelyInheritedMethods (edu.uic.cs474.f21.a1)  Assignment1
G in Test03_TransitivelyInheritedMethods (edu.uic.cs474.f21.a1)  Assignment1
```

Description

In this assignment, you will have to write a tool to explain all methods that can be invoked at a particular call site, taking inheritance and dynamic dispatch into account. Your tool could be used to implement an IDE utility, as depicted above. You will use the Java Parser library[1] to access the structure of the Java source code provided to you.

For this assignment, you need to write a class that implements the interface `DynamicDispatchExplainer`, 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 DynamicDispatchExplainer {
  Set<String> explain(
    Map<String, ClassOrInterfaceDeclaration> classes,
    String receiverType,
    String methodName,
    String ... argumentTypes);
}
```

The single method takes the following arguments:

- **`classes`** is a Java map from `String` to `ClassOrInterfaceDeclaration`.[2]
    - The map contains all classes that you should process as entries.

[1] https://javaparser.org/

[2] https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.3.2/com/github/javaparser/ast/body/ClassOrInterfaceDeclaration.html

    - The **key** of each entry is the name of that class.
    - The **value** of each entry is a JavaParser object that allows you to access the structure of the target class (i.e., all methods, fields, super classes, etc.)
- **`receiverType`** is the name of the type of the receiver (e.g., "A" in the picture above)
- **`methodName`** is the name of the method being invoked (e.g., "overriden" in the picture above)
- **`argumentTypes`** are the names of the types of the arguments of the method being invoked

Method `explain` should return a set with the name of all the classes that have an implementation of the method being invoked, and that can be reached from that call site depending on the dynamic type of the receiver.

Method `explain` should consider only methods invoked through dynamic dispatch, and ignore methods invoked through static dispatch (e.g., private and static methods).

Examples

Consider the following Java class hierarchy:

```
class Top {
    void overridenAC(String s, Top t) { }
    void notOverriden() { }
}
class A extends Top { void overridenAC(String a, Top t) { } }
class B extends A   { }
class C extends B   { void overridenAC(String a, Top t) { } }
```

`explain(classes, "C", "overridenAC", "String", "Top")` should return the set `{"C"}`.

- This is equivalent to the following Java code:

```
C c = … ; Top top = … ; c.overridenAC("CS474", top);
```

- The only implementation that can be called is `C.overridenAC`

`explain(classes, "B", "overridenAC", "String", "Top")` should return the set `{"B","C"}`.

- This is equivalent to the following Java code:

```
B b = … ; Top top = … ; b.overridenAC("CS474", top);
```

- Class B inherits method `overridenAC` from class A, so the implementation `A.overridenAC` can be called when `B b = new B();`
- Class C defines its own implementation of method `overridenAC`, which can be called with

```
B b = new C(); …
```

`explain(classes, "Top", "overridenAC", "String", "Top")` should return the set `{ "Top", "A", "C" }`, as each of the classes listed in the set provides its own implementation of method `overridenAC`.

Reflection

This assignment cannot be solved with Java reflection. There will be a future assignment dedicated to using Java reflection. As such, you should refrain to use class `java.lang.Class` or any class belonging to the `java.lang.reflect` package.

`java.lang.Object`

Your solution should consider the class at the root of the Java hierarchy `java.lang.Object`. This includes considering methods inherited from `java.lang.Object`, and receivers of type `java.lang.Object`.

Entry Point

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

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

Due Date and Resubmission Policy

This assignment is due on **September 18 2021** (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 **September 25 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 `assignment1` 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 the autograder. 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.