Curtin University — Department of Computing
Sri Lanka Institute of Information Technology

Software Metrics (ISAD4002)

Semester 1, 2017

Assignment 2

Due: Week 12 – Friday 19 May, 4:00 pm (UTC+5:30)

Weight: 35% of the unit mark.

Your task is to design, code and perform quality assurance on an application to build a call tree. Your application should parse compiled Java `.class` files.

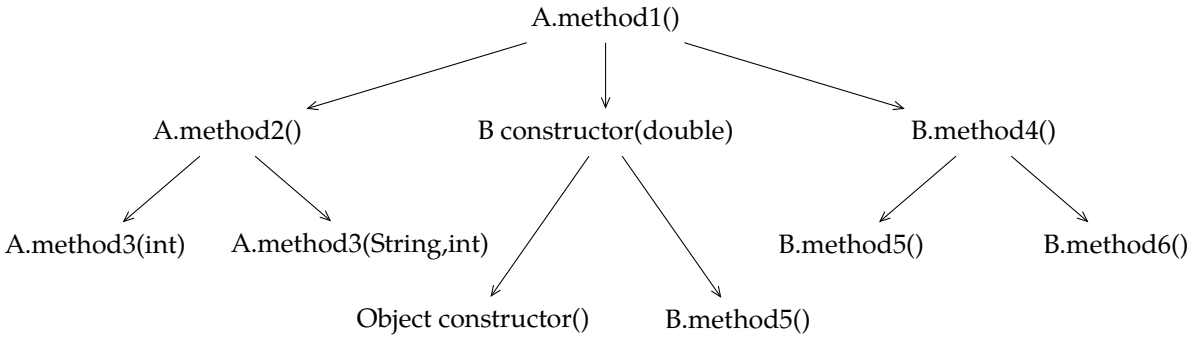
Background

A call tree is the hierarchy of methods/constructors called when a particular method or constructor runs. To illustrate, consider the following example code:

```
class A {
    public static void method1() {
        method2();
        B obj = new B(42.0);
        obj.method4();
    }

    public static void method2() {
        method3(42);
        method3("Hello", 42);
    }
    public static void method3(int x) {}
    public static void method3(String x, int y) {}
}

class B {
    public B(double x) {
        // Implied call to Object's constructor
        method5();
    }

    public void method4() {
        method5();
        method6();
    }
    public void method5() {}
```

```
    public void method6() {}
}
```

Given the above code, a call tree for `method1` would look something like this:



The same call tree can also be represented as follows:

```
A.method1()
    A.method2()
        A.method3(int)
        A.method3(String,int)
    B constructor(double)
        Object constructor()
        B.method5()
    B.method4()
        B.method5()
        B.method6()
```

(You could make it prettier with ASCII lines, but that's not necessary.)

Thus, a call tree consists of all the method/constructor calls made by a given method, directly and indirectly. (The same method/constructor can appear more than once – e.g. `method5` – if called from multiple places.)

JVM Instructions

The following JVM instructions are used to make calls:

invokestatic – used to call static methods:

```
SomeClass.method();
```

invokevirtual – used to call non-static methods, when the reference is a class:

```
SomeClass obj;
...
obj.method();
```

invokeinterface – used to call non-static methods, when the reference is an interface:

```
SomeInterface obj;
...
obj.method();
```

invokespecial – used to call constructors and also superclass methods:

```
new SomeClass(...);
```

```
super.method(...);
```

invokedynamic – ignore this one. It's used to call methods where the method and/or class are unknown at compile time. This is used for languages *other* than Java, and it's virtually impossible to analyse for our purposes.

Functionality

Your application must do the following:

(a) Have the user select a particular class and a particular method or constructor. You can use either command-line parameters or standard input. You must show the parameter types, and treat overloaded methods/constructors (i.e. with the same name but different parameters) as distinct from each other.

(b) Determine the call tree for the given method/constructor. You must do this by parsing the corresponding `.class` file, and any other `.class` files containing methods or constructors in the call tree.

(c) Output the call tree. The above textual format will suffice. Notes:

- You can attempt a more elaborate form if you like, provided it is readable, contains the same information, and is still actually a tree.
- The order in which methods are listed, at a particular level of the tree, doesn't matter. For instance, if method `apple()` calls methods `banana()` and `carrot()`, then you could output `banana()` and `carrot()` in either order.

(d) Print the total number of *unique* methods and constructors called, and the total number of classes involved. If a given method/constructor is called multiple times, count it only once.

In determining the call tree, you'll need to:

- Handle overloaded methods and constructors – those having the same name but different parameters (e.g. the two methods called "`method3`" in the above example). This implies that you'll need to output the parameter types for each method-/constructor.

- Indicate recursion – where a method calls itself *or* one of its ancestors higher up in the call tree. If you fail to detect recursion, you'll end up with an infinite call tree.

  Say that `methodA` calls itself, while `methodB` calls `methodC`, which calls `methodD`, which calls `methodB` again. We can represent this recursion as follows:

```
TheClass.methodA()
    TheClass.methodA() [recursive]
```

```
TheClass.methodB()
    TheClass.methodC()
        TheClass.methodD()
            TheClass.methodB() [recursive]
```

- Indicate abstract method calls. If you call an abstract method, there's no way to know at compile time which method you're actually calling. (In fact, this can also be true of non-abstract methods, because they too can be overridden. We'll conveniently overlook this, however.)

  Say that `methodA` calls abstract `methodB`. We can represent this as follows:

```
TheClass.methodA()
    TheClass.methodB() [abstract]
```

- Indicate and skip over missing classes – those for which your application can't locate the `.class` file (perhaps because it's a standard Java class).

```
TheClass.methodA()
    TheOtherClass.methodB() [missing]
```

- If a method/constructor makes multiple calls to a particular callee (another method-/constructor), ignore all but the first call.

  For instance, consider the following:

```
public void methodA() {
    methodB();
    ...
    methodB();
}

public void methodB() {
    for(int i = 0; i < 10; i++)
        methodC();
}

public void methodC() {}
```

We only care that `methodA` calls `methodB`, and `methodB` calls `methodC`. We don't care that these calls happen multiple times.

Coding

You may optionally make use of the already-partially-implemented ClassFileParser program (available on Blackboard). You will have to decide how best to adapt it for your purposes.

You may write your code in any language, provided you can demonstrate it working. ClassFileParser itself is written in Java.

Do not use any other pre-existing code (other than standard libraries).

Testing and Code Quality

To ensure your code is of high quality:

(a) Develop test cases to check all required functionality. Your tests must cover a broad range of situations, including boundary cases, complex cases and real-world cases. Obviously, your application should pass the tests!

For each test case, you must of course *manually* determine the expected metric results, to determine whether your code passes or fails the test. For this purpose, use the `javap` tool to see the JVM instructions.

Real-world test cases should involve typical (or better still, very complex) Java classes from open-source Java applications. For instance, you might consider classes from NetBeans, or from java.awt.swing, etc.

(b) Use RSM, PMD, or another tool approved by the lecturer to find and fix readability/maintainability issues. You may decide that some issues do not require fixing, but if so you must justify your decision.

Report

Prepare a report that includes the following:

Compiling and Running: Briefly explain how to compile and run your code from the command-line. If you have used Java and not used any 3rd-party libraries, this may be as simple as:

```
[user@pc]$ cd MySourceCodeDirectory

[user@pc]$ javac *.java

[user@pc]$ java MyAmazingApplication filenames ...
```

However, if it's more complicated for any reason, please let the marker know!

Functionality: State which functionality you have successfully implemented, to the best of your knowledge.

Design: Explain your design:

(a) structurally, describing each class you have developed or modified, and
(b) algorithmically, describing (in high level terms) the significant steps in your code and any important data structures it uses.

Testing: Describe your test cases, justify how you designed them, and give the results of your testing. Describe any bugs you are aware of.

Quality: State any code quality issues raised by RSM, PMD or the relevant tool used, and give your response to each.

Referencing: State precisely which code is yours (written solely by you without the help of anyone else) and which isn't.

As before (in Assignment 1), your report should:

- Have normal spacing and a sensible layout.
- Be logically structured, using headings where appropriate.
- Use good English, with proper spelling and grammar.
- Comprehensively demonstrate that you know what you are doing!

Submission

Submit the following electronically to the Assignment 2 area on Blackboard:

- A completed Declaration of Originality (*please read it carefully beforehand!*);
- The complete source code for your application, including any pre-existing code you use (so that the marker can easily compile and run your application);
- The complete source code for your test cases;
- Your report (in .pdf format);
- The output from RSM or PMD (or the relevant equivalent tool) in a single file called `quality.txt`, `quality.html` or `quality.zip` (if there are multiple files).

After submitting, please verify that your submission worked by downloading your submitted files and comparing them to the originals.

See the unit outline for the policy on late submissions. Additionally, if your Declaration of Originality is missing, incomplete, or in some way erroneous, your assignment will not be marked.

Marking

If you follow the submission guidelines, marks will be broadly allocated as follows:

- 50% – The functionality of your program, as determined by a series of official test cases (known only to the marker). These test cases will cover a range of different situations as per the requirements.
- 20% – The quality of your code.
- 30% – The quality of your design explanation and your own testing efforts.

Academic Integrity

As per the Declaration of Originality:

- The work you submit must be *entirely your own*, except where clearly indicated otherwise and correctly referenced.
- You must take all reasonable steps to ensure that your work is *not accessible* to any other students who may gain unfair advantage from it.
- You are prohibited from submitting work that you have previously submitted, including for prior attempts at this unit.

Please see Curtin's Academic Integrity website (and the unit outline) for further information on plagiarism and academic misconduct.

The teaching staff and/or unit coordinator may require you to provide an oral justification of, or to answer questions about, any piece of written work submitted in this unit. Your response(s) may be referred to as evidence in an Academic Misconduct inquiry. In addition, your assignment submission may be analysed by Turnitin and/or other systems to detect plagiarism and/or collusion.

End of Assignment 2