香港科技大學
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

COMP 5111 Tutorial (1)

Rongxin Wu
Computer Science and Engineering
The Hong Kong University of Science and Technology
2014. Sep.11

Part I: Assignment 1

- Objective:
  - Random Testing
  - Test Coverage Criteria
- Tasks:
  - Using Randoop for Random Testing
  - Using EclEmma to Generate Coverage Report
  - Using Soot to Generate Statement Coverage
  - Using Soot to Generate Branch Coverage (Bonus)

Random Testing

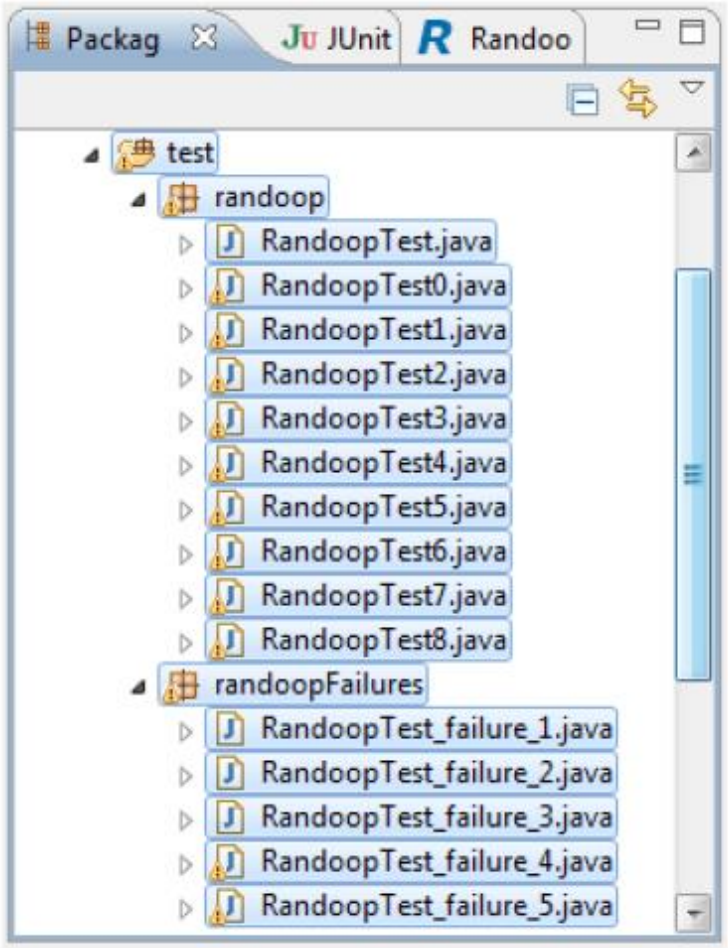
- Black-box Testing
- How does it work?
  - Random, independent inputs
  - Test Oracle
- Tool
  - Randoop

Randoop

- An automatic unit test generator for java
- Generate test input for:
  - Methods of non-abstract classes
  - Non-abstract classes
  - Enums
  - Compilation units that contain at least one of the above elements
  - Packages
  - Source folders

Randoop

- Output: test cases



Test Coverage Criteria

- To measure what percentage of code has been exercised by a test suite
- Basic coverage criteria
  - Statement Coverage
  - Branch Coverage
  - Call Coverage
  - Condition coverage
- Tool
  - Jacoco

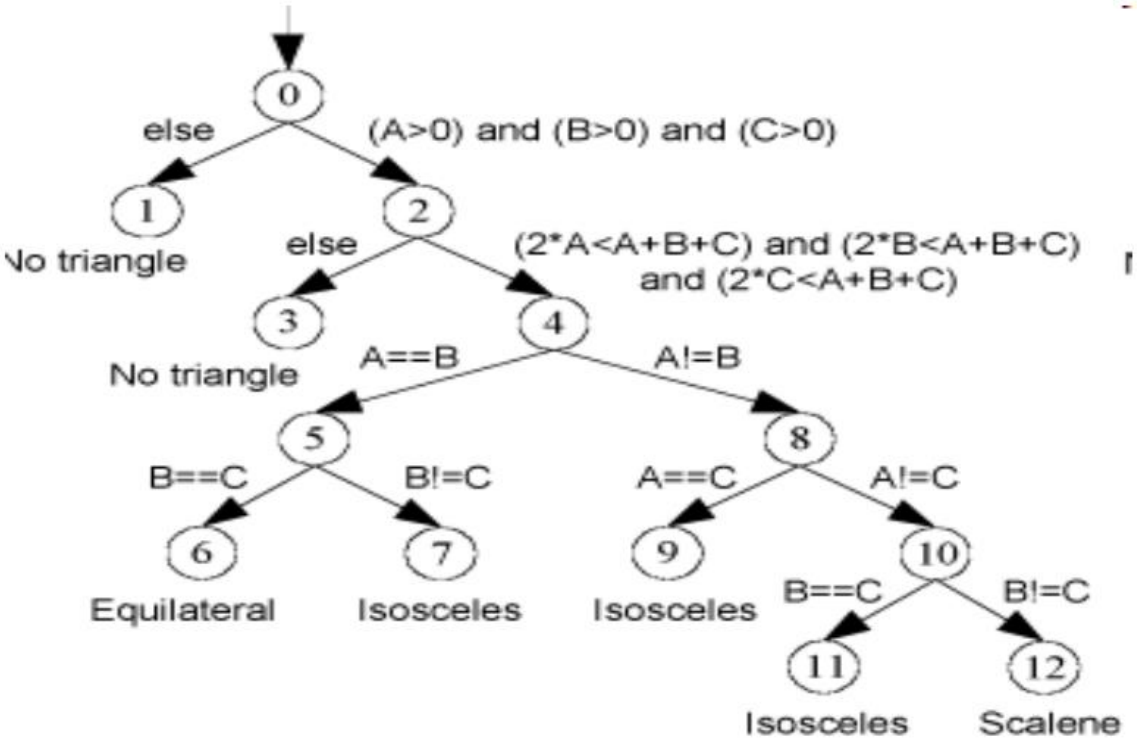
Statement Coverage

- Has each statement in the program been executed?

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x
  int count = 0;
  for (int i = 1; i < x.length; i++) {
      if (x[i] == 0) {
            count++;
      }
  }
  return count;
}
```

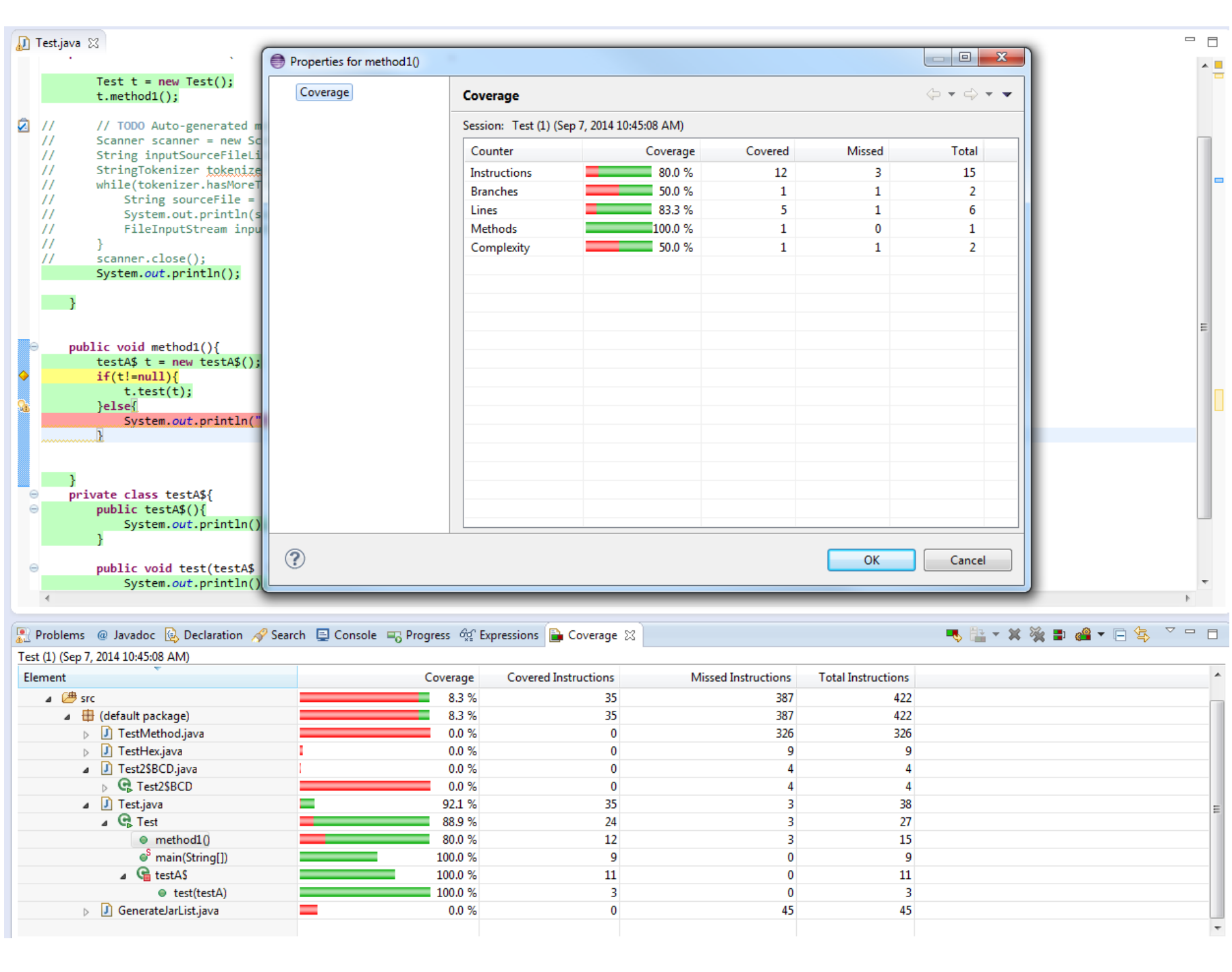
Branch Coverage

- Has each branch of each control structure been executed?



EclEmma

- A Free Java Code Coverage Tool
- EclEmma provides:
  - Method Coverage (Call Coverage)
  - Instruction Coverage (Statement Coverage)
  - Branch Coverage
  - Condition Coverage


Test.java
Test t = new Test();
t.method1();
public void method1(){
testA$ t = new testA$();
if(t!=null){
t.test(t);
}else{
System.out.println("
}
private class testA${
public testA$(){
System.out.println()
}
public void test(testA$
System.out.println()
Properties for method1()
Coverage
Coverage
Session: Test (1) (Sep 7, 2014 10:45:08 AM)
Counter Coverage Covered Missed Total
Instructions 80.0 % 12 3 15
Branches 50.0 % 1 1 2
Lines 83.3 % 5 1 6
Methods 100.0 % 1 0 1
Complexity 50.0 % 1 1 2
OK
Cancel
Problems
Javadoc
Declaration
Search
Console
Progress
Expressions
Coverage
Test (1) (Sep 7, 2014 10:45:08 AM)
Element Coverage Covered Instructions Missed Instructions Total Instructions
src 8.3 % 35 387 422
(default package) 8.3 % 35 387 422
TestMethod.java 0.0 % 0 326 326
TestHex.java 0.0 % 0 9 9
Test2$BCD.java 0.0 % 0 4 4
Test2$BCD 0.0 % 0 4 4
Test.java 92.1 % 35 3 38
Test 88.9 % 24 3 27
method1() 80.0 % 12 3 15
main(String[]) 100.0 % 9 0 9
testA$ 100.0 % 11 0 11
test(testA) 100.0 % 3 0 3
GenerateJarList.java 0.0 % 0 45 45

Part II: Use Soot for Program Instrumentation

- Objectives
  - Introduction to Soot
  - Introduction to program instrumentation
  - Learn how to use soot to instrument the Java programs

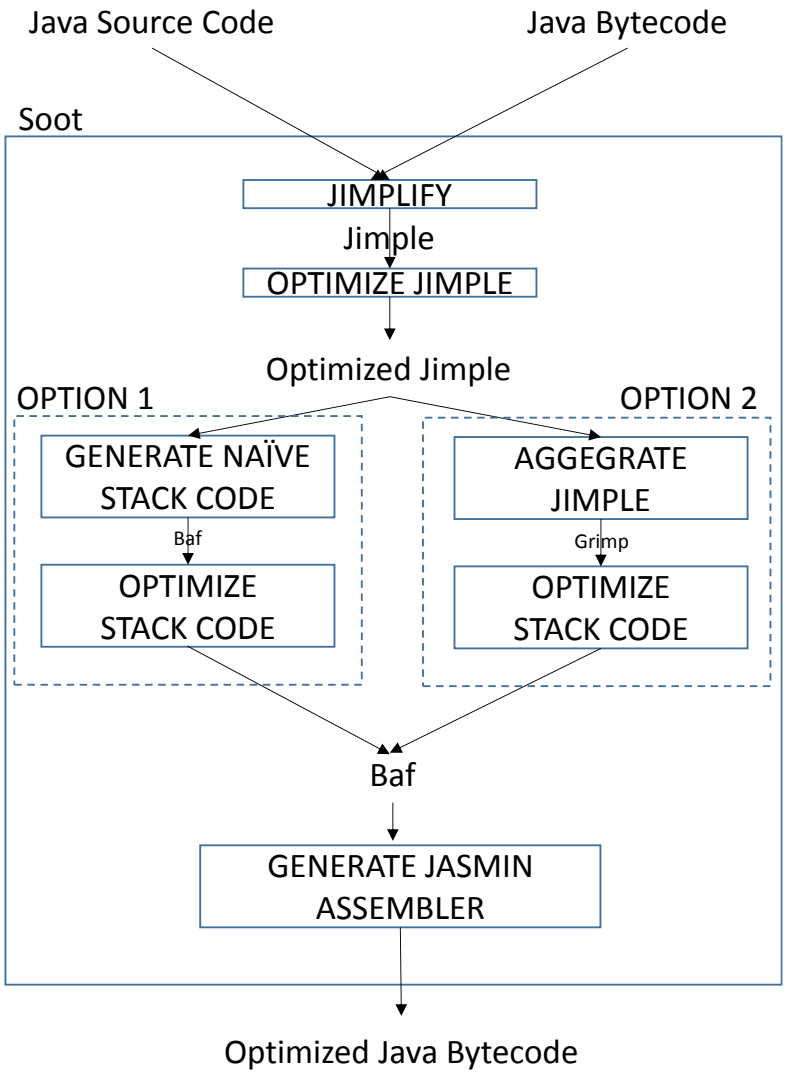
Introduction to Soot

- A Java Optimization Framework
- What we can do by Soot
  - Static Analysis (control flow analysis, call graph, point-to analysis, …)
  - Instrumentation
  - Optimization

Soot

- Input
  - Java Source Code
  - Java Bytecode
- Intermediate Representation
  - Jimple
  - Baf
  - Shimple
  - Grimp
- Output
  - Optimized Java Bytecode

Phase of the Optimization



Jimple

```
public static void main(String[] argv) throws Exception
{
    int x = 2, y = 6;

    System.out.println("Hi!");
    System.out.println(x * y + y);
    try
    {
        int z = y * x;
    }
    catch (Exception e)
    {
        throw e;
    }
}
```

```
public static void main(java.lang.String[]) throws java.lang.Exception
    {
        java.lang.String[] r0;
        int i0, i1, i2, $i3, $i4;
        java.io.PrintStream $r1, $r2;
        java.lang.Exception $r3, r4;

        r0 := @parameter0;
        i0 = 2;
        i1 = 6;
        $r1 = java.lang.System.out;
        $r1.println(``Hi!'');
        $r2 = java.lang.System.out;
        $i3 = i0 * i1;
        $i4 = $i3 + i1;
        $r2.println($i4);

     label0:
        i2 = i1 * i0;

     label1:
        goto label3;

     label2:
        $r3 := @caughtexception;
        r4 = $r3;
        throw r4;

     label3:
        return;

        catch java.lang.Exception from label0 to label1 with label2;
    }
```

How does Soot works?

- Soot's execution: a number of phases
  - e.g. JimpleBody are built in a phase named "jb"
  - Each phase conducts some tasks, e.g. transforms to IR code, generate call graph, generate control flow analysis
- Soot Phase options provide a way for you to customize your analysis
  - Configure the phases of Soot
  - Write your own subphases

Program Instrumentation

- Instrumentation means the ability of an application to incorporate
  - Code tracing
  - Debugging
  - Profiling
  - Computer data logging
  - …
- How to do program instrumentation?
  - Insert the code instructions that can monitor programs
  - Two types: source and binary instrumentation

Use Soot to Instrument Programs

- Goal: count how many InvokeStatic instructions executed in a sample program
- Sample Program

```
class TestInvoke {
    private static int calls = 0;

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            foo();
        }
        System.out.println("I made " + calls + " static calls");
    }

    private static void foo() {
        calls++;
        bar();
    }

    private static void bar() {
        calls++;
    }
}
```

```
// access flags 0x9
public static main([Ljava/lang/String;)V
 L0
  LINENUMBER 5 L0
  ICONST_0
  ISTORE 1
 L1
  GOTO L2
 L3
  LINENUMBER 6 L3
  FRAME APPEND [I]
  INVOKESTATIC TestInvoke.foo()V
```

How to Instrument

- Create Helper Class
  - To encapsulate the profiling function
- Customize our own phase in Soot
  - Program instrumentation will be done in our own phase
  - Leverage the Jimple code (thus, our phase should be after the Jimple code is created)
  - Insert the instruction using Jimple code

Create Helper Class

```
/* The counter class */
public class MyCounter {
/* the counter, initialize to zero */
private static int c = 0;

      /**
       * increases the counter by
       *
       * <pre>
       * howmany
       * </pre>
       *
       * @param howmany
       *            , the increment of the counter.
       */
      public static synchronized void increase(int howmany) {
            c += howmany;
      }

      /**
       * reports the counter content.
       */
      public static synchronized void report() {
            System.err.println("counter : " + c);
      }
}
```

Customize Our Own Phase

- The implementation of our own phase `InvokeStaticInstrumenter` should
  - Extend an abstract class `BodyTransformer`

```
public class InvokeStaticInstrumenter extends BodyTransformer{…}
```

  - Implement the method `internalTransform`

```
@override
protected void internalTransform(Body body, String phase, Map options)
{…}
```

Customize Our Own Phase

- Add our own phase into Soot
  - Add after Jimple code is created

```
/* add a phase to transformer pack by call Pack.add */
Pack jtp = PackManager.v().getPack("jtp");
jtp.add(new Transform("jtp.instrumenter",
        new InvokeStaticInstrumenter()));
```

Implementation of Our Own Phase

- Initialize the Helper Class
  - Initialize the profiling method
  - Initialize the report method

```
public class InvokeStaticInstrumenter extends BodyTransformer {
    /* some internal fields */
    static SootClass counterClass;
    static SootMethod increaseCounter, reportCounter;

    static {
        counterClass = Scene.v().loadClassAndSupport("MyCounter");
        increaseCounter = counterClass.getMethod("void increase(int)");
        reportCounter = counterClass.getMethod("void report()");
        Scene.v().setSootClassPath(null);
    }
    …
```

Implementation of Our Own Phase

- Insert the code for profiling the StaticInvoke instructions
  - Find out the StaticInvoke instruction

```
// get a snapshot iterator of the unit since we are going to mutate the chain when
iterating over it.
Iterator stmtIt = units.snapshotIterator();
// typical while loop for iterating over each statement
while (stmtIt.hasNext()) {
     // cast back to a statement.
     Stmt stmt = (Stmt) stmtIt.next();
     // there are many kinds of statements, here we are only
     // interested in statements containing InvokeStatic
     if (!stmt.containsInvokeExpr()) {
          continue;
     }
     // take out the invoke expression
     InvokeExpr expr = (InvokeExpr) stmt.getInvokeExpr();
     // now skip non-static invocations
     if (!(expr instanceof StaticInvokeExpr)) {
     continue;
     }
```

Implementation of Our Own Phase

- Insert the code for profiling the StaticInvoke instructions
  - Insert the profiling code

```
// now we reach the real instruction
// call Chain.insertBefore() to insert instructions
//
// 1. first, make a new invoke expression
InvokeExpr incExpr = Jimple.v().newStaticInvokeExpr(
increaseCounter.makeRef(), IntConstant.v(1));
// 2. then, make a invoke statement
Stmt incStmt = Jimple.v().newInvokeStmt(incExpr);

// 3. insert new statement into the chain
// (we are mutating the unit chain).
units.insertBefore(incStmt, stmt);
```

Implementation of Our Own Phase

- Report when main() method is returned
  - Find out the statement when main() is returned

```
// 1. check if this is the main method by checking signature
String signature = method.getSubSignature();
boolean isMain = signature.equals("void main(java.lang.String[])");

// re-iterate the body to look for return statement
if (isMain) {
stmtIt = units.snapshotIterator();

while (stmtIt.hasNext()) {
     Stmt stmt = (Stmt) stmtIt.next();

     // check if the instruction is a return with/without value
     if ((stmt instanceof ReturnStmt)
     || (stmt instanceof ReturnVoidStmt)) {
          …}
```

Implementation of Our Own Phase

- Report when main() method is returned
  - Report the profiling result

```
// 2. then, make a invoke statement
Stmt reportStmt = Jimple.v().newInvokeStmt(reportExpr);

// 3. insert new statement into the chain
// (we are mutating the unit chain).
units.insertBefore(reportStmt, stmt);
```

How to Run

- Instrument the program

```
java -cp bin;lib/soot-2.5.0.jar MainDriver -pp -soot-classpath
../Sample/bin TestInvoke
```

```
C:\Users\wurongxin\Dropbox\tutorials\Profiler>java -cp bin;lib/soot-2.5.0.jar Ma
inDriver -pp -soot-classpath  ../Sample/bin TestInvoke
-pp
-soot-classpath
../Sample/bin
TestInvoke
Soot started on Mon Sep 08 00:12:42 CST 2014
Transforming TestInvoke...
instrumenting method : <TestInvoke: void <clinit>()>
instrumenting method : <TestInvoke: void <init>()>
instrumenting method : <TestInvoke: void main(java.lang.String[])>
instrumenting method : <TestInvoke: void foo()>
instrumenting method : <TestInvoke: void bar()>
Writing to sootOutput\TestInvoke.class
Soot finished on Mon Sep 08 00:12:43 CST 2014
Soot has run for 0 min. 0 sec.
```

- Run the instrumented program

```
java -cp bin;sootOutput TestInvoke
```

```
C:\Users\wurongxin\Dropbox\tutorials\Profiler>java -cp bin;sootOutput TestInvoke

I made 20 static calls
counter : 20
```

Thanks!
Q & A