CS474 - Course Project

Team members

1. Maithreyi Rajagopalan
2. Sherryl Mathew George

Overview

The requirements of this project are:

1. An instrumentation program that takes syntactically correct source code of some Java application and using the Eclipse Java Abstract Syntax Tree (AST) parser parse this application into an AST
2. For each expression and statement in each scope, the program will insert an instrumenting statement to capture the values of the variables.
3. Once the application is instrumented, it will be compiled and run using a build script
4. Once instrumented and compiled, the rogram will run multiple instances of the instrumented application with different input values by starting the JVM. The trace information from the executing Java application is send to the launcher program where a hashtable is used to keep track of the variables, their bindings and values at different points of the program execution.

Prerequisites

1. JAVA SDK 11 or higher
2. Scala 2.13 or above

Note: The project is developed and tested on a Unix based OS.

Dependencies

Library	Version
config	v1.3.4
logback-classic	v1.2.3
scala-logging	v3.9.2
logback-config	v0.4.0
scalatest	v3.0.8
org.eclipse.jdt.core	v3.19.0
commons-io	v2.6
ant	v1.10.7
akka-http	v10.1.10
akka-stream	v2.6.0
play-json	v2.8.0

How to run

The project has two main files:

1. ServerLaunch - Launches the IPC server to listen to communications
2. InstrumLauch - Launches the instrumentation for a selected project

Below are the steps to be followed to run the project:

1. Run the file `ServerLauch.scala`. This should start a server at `127.0.0.1:8080`. Make sure that this port is free
   or else the server won't start. This file should be run first and has to be started only once for running any number of
   instrumentations. Donot close the terminal window in which this file is run
2. Run the file `InstrumLaunch.scala` to run the instrumentation. You will be provided with a list of project configuration

files available to the instrumenter. You can choose which project to run. Running the instrumenter for more than once on the
same source file may cause unexpected results. This project already has a `Pathfinder` and `Matrix Rotation` program which
can be used for test purposes.

3. After each run two timestamped files will be found in the `tracefiles` directory in the project root. The file with
`binding_` will have the formatted data of the hashtable for each variable and
`trace_` will have running data of each
instrum statement executed.

`sbt` command line can also be used to run the project. Running the project from the root will provide you with option to
select which class has to run. Run both the files in two separate terminals in the order described above.

What InstrumLaunch does apart from instrumentation

1. Before any instrumentation starts all java files in `src` folder are copied to a folder name `oldSrc` one level above `src` , thus
saving the files which can be reused again for instrumenting.
2. `TemplateClass.java` and `AP.java` are two files that is needed by the instrumented program to log the statements and also
to communicate back to the launcher program. These files are not instrumented and a copy is kept in `config/astparser` . Once the
isntrumentation is done and before launching JVM instance the entire
`config/astparser` is copied into the source folder of instrumented
program. Also a new import `import astparser.*` is added to all instrumented java files for resolving dependencies.
3. To enable the use of Websockets and JSON, two jars needs `nv-websocket-client-2.9.jar` and `json-20190722.jar` needs to be present in the `jars` folder of the instrumented
project. This is copied from `config/dependencyjar` .

How to add a new project for instrumentation

Each project used is based on Ant build. Ant is used for its simplicity to bootstrap a new project
quickly. Any tool build which gives out `.class` files as output can be swapped out instead of Any. Since using Ant
each project is expected to have a `build.xml` file with the below specification:

1. Default target is `compile`
2. A `javac` node is defined with the `destdir` property set to a folder.
3. Two dependency jars as defined below are specified in the classpath node. Only declaration for the `jars` in the build
   file is needed. The original files are copied automatically by the launcher program.

```
<pathelement path="<your_jar_folder>/nv-websocket-client-2.9.jar"/>
<pathelement path="<your_jar_folder>/json-20190722.jar"/>
```

Below is an example of a `build.xml` which can be used as a starting point

```
<project default="compile">
    <target name="compile">
        <mkdir dir="target"/>
        <javac srcdir="src" destdir="target">
        <classpath>
            <pathelement path="jars/stdlib(3).jar"/>
            <!--Below two lines are included as per specification-->
            <pathelement path="jars/nv-websocket-client-2.9.jar"/>
            <pathelement path="jars/json-20190722.jar"/>

        </classpath>
    </javac>
    </target>
</project>
```

To add a new project for instrumentation perform the below steps:

1. Make sure your project is Ant buildable.
2. Add a configuration file to the folder `config/instrum`. This a configuration file and should have the extension `.conf`.
   Each parameter in the configuration file is shown below: All parameters are

mandatory and the program cannot procced without
setting each of the below parameters

1. rootRelativetoInstrumDir - Set this to `true` if the new project foler is saved inside the instrumentation project
   directory. Else set this to `false`.
2. root - Specify path to the root of your project. If you set `true` for `rootRelativetoInstrumDir` this path should be
   relative else you need an absolute path to be specified.
3. srcDir - Specify path to top level directory which has `.java` code files. This is to be set relatively to the `root`
   parameter
4. targetDir - Specify where to find the `.class` files produced by the Ant builder. This is to be set relatively to the `root`
   parameter
5. jarFolder - Specify the folder where your `jars` are saved. This should be the folder part of your `classpath` specification
   in the `build.xml`. This is to be set relatively to the `root` parameter.
6. buildFile - Specify where your `build.xml` file is. This is to be set relatively to the `root` parameter.
7. mainClass - Give the qualified name including the package structure of your main class
8. arguments - This is a list of list. The inner list is separated by commas and each inner list consists of the arguments
   to be passed to the instrumented program.

See for a sample `pathfinder.conf` file for a project saved inside the instrumentation project in the path `projects/project1`

```
compile{
  # Set this to true if the project root directory
  # is inside the instrum project directory and the provide
  # relative path to the project folder
  rootRelativetoInstrumDir = true
  root = "projects/project1"
  # Everything below this line is relative to the
  # project root
  srcDir = "src"
  targetDir = "target"
```

```
  jarFolder = "jars"
  buildFile = "build.xml"
}

run {
  # Provide the qualified main class
  mainClass = "PathFindingOnSquaredGrid"
  # Provide input parameters
  arguments = [
    [10 0.3 0 0 9 9]
  ]
}
```

See for a sample `pathfinder.conf` file for a project saved at `/home/sherryl/Desktop/project1`

```
compile{
  # Set this to true if the project root directory
  # is inside the instrum project directory and the provide
  # relative path to the project folder
  rootRelativetoInstrumDir = false
  root = "/home/sherryl/Desktop/project1"
  # Everything below this line is relative to the
  # project root
  srcDir = "src"
  targetDir = "target"
  jarFolder = "jars"
  buildFile = "build.xml"
}

run {
  # Provide the qualified main class
  mainClass = "PathFindingOnSquaredGrid"
  # Provide input parameters
  arguments = [
    [10 0.3 0 0 9 9]
  ]
}
```

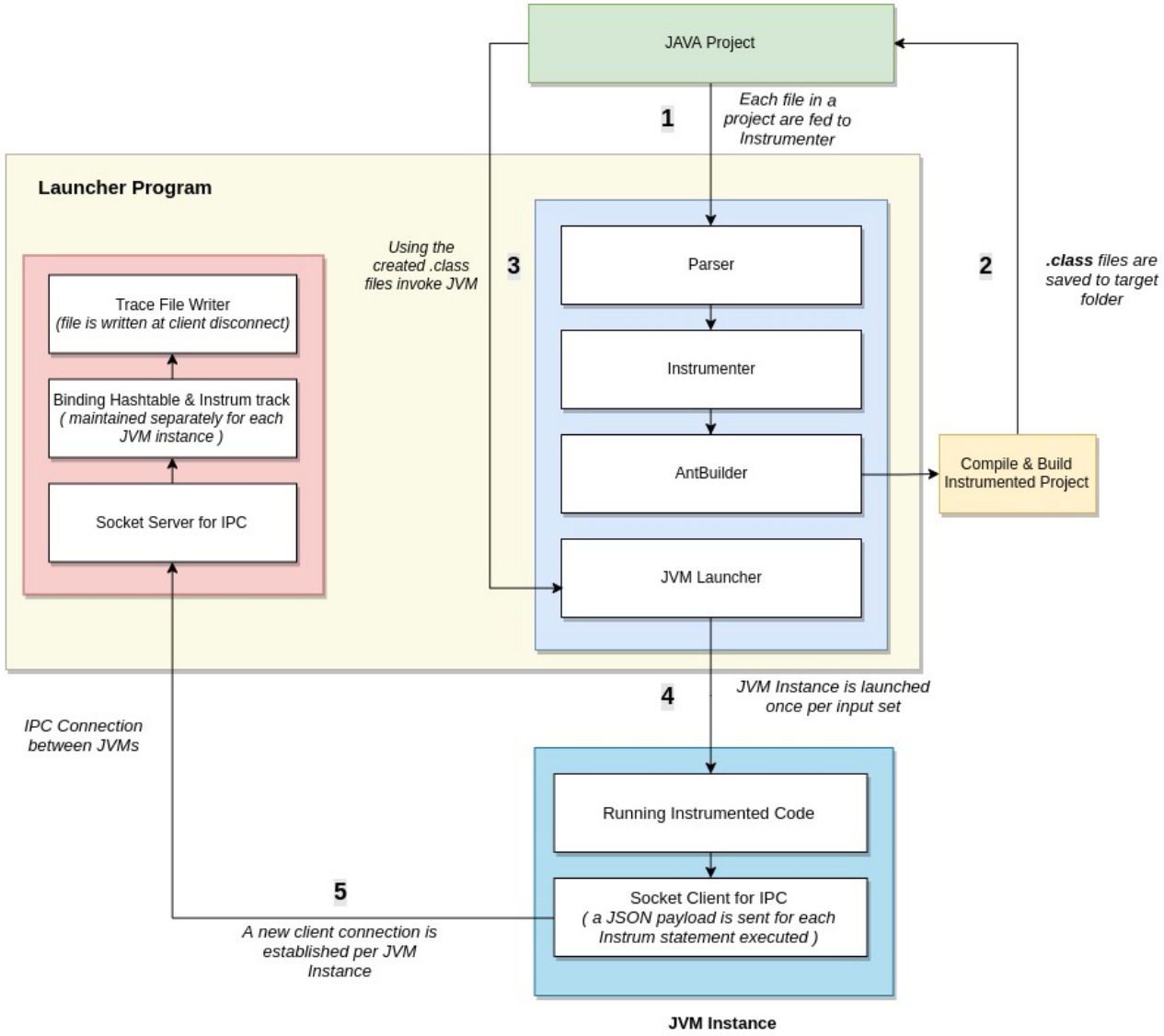
3. Make sure the project is in the path you specified as `root` and you are good to run the code.

Basic Troubleshooting

1. This project needs port `8080` to be free to run. If you get an `Address already in use` make sure to free the port.
   For a Unix system you can run the command `sudo lsof -i:8080` and then kill the PID blocking the port.
2. If you get a `Failed to connect to 'localhost:8080'` error when running the `InstrumLaunch` make sure you have run `ServerLaunch` and the same is running.

Technical Design

Below is the overall architecture of the Instrumenter system and overview of all components:



Components Overview

The Launcher Program

Launcher program is divided into two:

1. Instrumentor & JVMLauncher

This component deals with reading files in a JAVA project, instrumenting the code and launch JVM instances for each set
of input. Main components are:

- Parser : Parser will run over the code and rewrites certain control structures which has single statements to blocks.
- Instrumentor : Visitors are used to visitor over predefined statements and expressions, find their names, binding in
  the current scope and value currently held. The values found are then passed to `TemplateClass.instrum()` method for
  being sent to the launcher program.
- AntBuilder : Ant is used as the build tool. Each project passed should have a `build` file. Ant is used for its simplicity.
  Ant can be swapped out for any other build tool. The output of this step is `.class` files and that's all is needed from
  this step. The `.class` files will be saved into a predefined folder.
- JVMLauncher : JDI is used to launch separate instances of JVM for each set of inputs in the selected project

2. IPC Server & Trace Writer

This component has two main functions

- IPC Server - This server needs to be started before the instrumenter is run. Server listens to `127.0.0.1:8080` for
  incoming messages. Each message is in JSON format in the below schema:

```
{
  "line": "Int",
  "statementType": "String",
```

```
  "data": [{"type":  "String", "binding": "String", "value":  "String"}]
}
```

Each JSON thus obtained is parsed and a hastable is maintained with the unique identifier as `binding`. And the JSON is
saved. Also a running trace is maintained to store the unprocessed JSON data which comes from the JVM instance.
A hashtable and trace maintained for each connected client and the data is moved to timestamp named text file when each
client is disconnected.

- JSON Parser & File Writer : Has code to parse each JSON coming to the system and write files at the disconnection of
  each client.

The JVM Instance (Client)

Each project is passed to a JVM instace. The instrumented code has two main components:

- Original Code with Instrumentation statements added and each instrumented statement will be of the form
  `TemplateClass.instrum(int line, String typeofStatement, AP... parameters)`.
- A `TemplateClass` with static members. When the static class is initialized a unique web socket connection is made
  to the launcher program. This connection will be used for IPC. Every `instrum` method executed will send a JSON payload
  to the launcher program for processing and saving.

Component Details

1. The Launcher Program

AST Parsing and Instrumentation -

The parsing and instrumentation is done in multiple steps.

1. AST Parsing
    1. Block Rewrite
    2. Code Rewrite
2. Instrumentation

1. AST Parsing -

This consists of Block rewriting and code rewriting. This is done as a first step to transform the code, before we begin instrumentation. This is handled in files BlockConverter, DoStatementCon, ForStatementCon, WhileStatementCon and FinalConverter classes under the converters module.

i. Block Rewrite

As part of block rewrite step, all singled statements inside control structures , are converted to blocks.

This is done for all control statements including, *for*, *do-while*, *while*, *for-each*, *if-else if-else*. This is done to ensure that when we need to add an additional logging statement, we do not need to handle the absence of blocks.

Example -

```
int i = 3;
if(i < 2)
    System.out.println("hello");
else
    System.out.println("hi");
```

is transformed to

```
int i = 3;
if(i < 2) {
System.out.println("hello");
}
else {
System.out.println("hi");
}
```

ii. Code Rewrite

In the code rewrite step, we transform the method invocations in the expressions in looping constructs, *for*, *while* and *do-while* into
single assignments.

This is only done for simple infix expressions with a single method invocation on either side of the operand; i.e. statements of the below nature, are not handled.

```
while(x() + x() + x() < 10)
```

But for simple expressions involving one method invocation, we account for nested loops as well.

Example -

```
while(x() < 5) {
    do {
        System.out.println("Hi");
    } while(x()< 2);
}

int x() {
    return 2;
}
```

is transformed to

```
int wh1 = x();
while(wh1 < 5) {
    int do1 = 0;
    do {
        System.out.println(i);
        do1 = x();
    } while(do1 < 2);
    wh1 = x();
}

int x() {
    return 2;
}
```

In addition to the loop construct transformation, we also add imports to include the Template class package (*This class includes the instrumentation method that is inserted into the original source*) and an additional last statement in the main() method to disconnect the socket (*This is part of the IPC to communicate back with the launcher program*).

2. Instrumentation

Instrumentation is added to capture the variables, the bindings and their values in a Java source program. As part of this implementation, we handle the following constructs through visitors.
The visitors are under the visitors module and the instrumentation is done in Instrum, AssignmentInstrum, ControlInstrum, MethodDeclarationInstrum, ReturnInstrum and VDSInstrum classes under the instrumentation module.

Constructs Handled

1. Control structures - *for*, *while*, *do-while*, *for-each*, *if-else-if-else*, *switch*

   Sample statement -

   ```
   for(int i=0; i < 10 ; i++) {
       /*Do something here.*/
   }
   ```

   Instrumentation Statement inserted -

   ```
   for (int x = 0; x < matrix.length; x++)
   TemplateClass.instrum(39, "ForStatement", new AP("SimpleName", "PathFindi
   ```

   Resulting trace with details of Location, Statement type, Binding and variable value -

   ```
   Line: 39, SeenAt: "ForStatement"
   Name: "PathFindingOnSquaredGrid.generateHValue(boolean[][], int, int, int
   Name: "()", Type: "QualifiedName", Value: "10"
   ```

2. Return statements - Return statements at the end of a function call.

   Sample statement -

   ```
   return 2;
   ```

Instrumentation Statement inserted -

```
return a;
TemplateClass.instrum(21, "ReturnStatement", new AP("SimpleName", "PathFi
```

Resulting trace with details of Location, Statement type, Binding and variable value -

```
Line: 21, SeenAt: "ReturnStatement"
Name: "PathFindingOnSquaredGrid.random(int, double).a", Type: "SimpleName
```

3. Method Declarations - All method declarations along with the parameters in the method signatures. We only call out the formal parameters along with their bindings and values in the trace. The bindings include the name of the method themselves.
Sample statement -

```
public static void main(String[] args) {
    /* Do something here */
}
```

Instrumentation Statement inserted -

```
public static void main(String[] args)
TemplateClass.instrum(389, "MethodDeclaration", new AP("SimpleName", "Pat
```

Resulting trace with details of Location, Statement type, Binding and variable value -

```
Line: 389, SeenAt: "MethodDeclaration"
Name: "PathFindingOnSquaredGrid.main(String[]).args", Type: "SimpleName",
```

4. Variable Declaration Statements - All variable declaration statements, including multiple declarations with initializers.

Sample statement -

```
int i = 0;
int i,j;
int i = 0, j = 0;
```

Instrumentation Statement inserted -

```
gridSize = Integer.parseInt(args[0]);
TemplateClass.instrum(393, "VariableDeclaration", new AP("SimpleName", "P
```

Resulting trace with details of Location, Statement type, Binding and variable value -

```
Line: 393, SeenAt: "VariableDeclaration"
Name: "PathFindingOnSquaredGrid.main(String[]).gridSize", Type: "SimpleNa
Name: "Integer.parseInt(String)", Type: "MethodInvocation", Value: ""
Name: "PathFindingOnSquaredGrid.main(String[]).args", Type: "inner Simple
Name: "", Type: " inner NumberLiteral", Value: "0"

// This translates to gridSize = Integer.parseInt(args[0]). inner SimpleN
```

5. Expression statements - We handle instrumentation of a lot of expression statement constructs as provided by the Java language. Below is a list of constructs that we handle in our instrumentation.
   Sample Statements -

```
i = x();                    //Assignment (with a method invocation).
i = i++;                    //Assignment (postfix expression).
i = -2;                     //Assignment (prefix expression).
i = x() + 2;                //Assignment (infix expression).
cell[y][x] = new Node(y, x); //Class Instance creation.
cell[y] = Node.y            //Field Access.
if (node == cell[Bi][Bj])   //Array access.
cell[0][1] = 2;             //Array initialization.
```

Sample Instrumentation Statement inserted -

```
cell[y][x] = new Node(y, x);
TemplateClass.instrum(41, "Assign", new AP("Array Begin", "", "{"), new A
```

Resulting trace with details of Location, Statement type, Binding and variable value -

```
Line: 41, SeenAt: "Assign"
Name: "", Type: "Array Begin", Value: "{"
Name: "PathFindingOnSquaredGrid.cell", Type: "inner SimpleName", Value: "
Name: "PathFindingOnSquaredGrid.generateHValue(boolean[][], int, int, int
```

```
Name: "PathFindingOnSquaredGrid.generateHValue(boolean[][], int, int, int
Name: "", Type: "Array End", Value: "}"
Name: "Node", Type: "ClassInstanceCreation", Value: "Node.Node(int, int)"
Name: "PathFindingOnSquaredGrid.generateHValue(boolean[][], int, int, int
Name: "PathFindingOnSquaredGrid.generateHValue(boolean[][], int, int, int

This shows that an array is assigned a value new Node(y,x)
```

Compilation & Execution (JVM Launcher)

The launcher program is responsible for compiling and executing multiple instances of the client application (JVM instances) against multiple inputs.
This also receives IPC messages through the web socket server and stores them in a hash table. This is then dumped into files, which are available for viewing.
The launcher comprises of multiple components -

######1. The ANT builder -
This is responsible for compiling the source application (client), after adding instrumentation statements and generate `.class` files.

The driver method in the AntBuilder class, under the launcher module is `compileAndLaunchJVM()`. This reads the `build.xml` config file from the source application and builds the application.

We use the `buildFinished()` hook to determine if the build completed successfully and launch the JVM from this method.

On build failure, we log the error messages and abort as we cannot proceed further with execution.

######2. The JVM launcher -
The `executeJava()` method under the AntBuilder class is responsible for invoking the JVM by reading the run configuration parameters. This method runs a JVM instance for every parameter as mentioned in the `run.arguments` in the project configuration file.

The JVM is initialized with details of the Main class, program arguments, class path and any dependent jars and started. This is done in the JVMLauncher class under the launcher module.

When, the JVM launch fails, we print the error and continue execution for the next set of inputs.

IPC Server

Files: `WSServer.scala` and `BindingData.scala`
To enable IPC communication between the launcher program and JVM instances running code a `akka-websocket` connection is used.
Each new JVM instance requested connects to the server as a new client. The sever listens to `127.0.0.1:8000/instrumserver` for
data from the clients. The data accepted is JSON format.

The server is responsible for building the hashtable and keeping a trace of all the instrumentation data received. This means
that each client needs to keep its own copy of the hastable and the trace. For this purpose `akka-actor` is used as a sink
for each client. A new actor instance will be created for each client and separate hastable and trace are maintained. `InstrumActor`
object is used to keep a track of the various events that occur during the communication starting from `Init` till `Completed`.

The `Flow` used groups the incoming data, and passes it to the method `parseJSONMessage(jsonString: String)` to parse each
JSON object obtained. `BindingData` class is used to de-serialize the JSON and provide convenience methods of easy printing.
During parsing each JSON is saved to the hashtable using `binding` of each variable as the unique identifier.
Also the original JSON is saved to variable to be written to a tracefile later.

Method `writeData()` will be invoked when a client disconnects, ie. the JVM instance has finished execution of all the statements
in the code. At this point the hashtable and trace data is written into `.txt` files and saved in the folder `tracefiles`. Each file
will be time stamped. Hashtable data has the prefix `binding_` and trace data has the prefix `trace`.

The main class in `ServerLaunch` is used to invoke the server. Not that there is no need to start the server every time a new instrumentation

is done. It would be enough to make sure that the server is still running at all time that the instrumenter is used.

Trace writer

This writes trace and binding files under `tracefiles` folder.

The JVM Instance (Client)

This consists of the modified source with instrumented statements.

Template Class

This is a static java class that holds the `instrum()` method. Logging statements are added in the source application (client), which are invocations of this method.

- `instrum(int lineNumber, String typeOfStatement, AP... args)` - This method takes in a line number, type of statement and a set of arguments with details on the variables used in that statement.
  A JSON object is constructed and sent back to the launcher program through a web socket.
- `WebSocket connect()` - This method is invoked when the TemplateClass is initialized, during the very first invocation of `instrum()`. This initializes a web socket to send messages to the launcher program.
- `finalizeInstrum()` - This method is invoked prior to exiting the client application (the last statement in `main()`). This method disconnects the socket.

AP Class

Multiple instances of the AP class are passed as arguments to the `instrum()` method. This holds information about variables including the type, binding and the value of the variable, for every variable used in a statement.
Below is the structure of the AP class.

```
String type;
String name;
String value;
```

Results (Trace & Binding Files)

When execution completes, two files are created per execution instance (per input) and are stored under `tracefiles` folder.
The two files are created with names `trace_<timestamp>` and `bindings_<timestamp>`.

The files have the below content and structure.

- Trace -
  This file gives detailed information of the type of statement, the line number and the various components included in the statement with their values.

```
Line: 394, SeenAt: "VariableDeclaration"
Name: "PathFindingOnSquaredGrid.main(String[]).probability", Type: "SimpleName
Name: "Double.Double.parseDouble(String)", Type: "MethodInvocation", Value: ""
Name: "PathFindingOnSquaredGrid.main(String[]).args", Type: "inner SimpleName"
Name: "", Type: " inner NumberLiteral", Value: "1".
========
Line: 16, SeenAt: "ForStatement"
Name: "PathFindingOnSquaredGrid.random(int, double).i", Type: "SimpleName", Va
Name: "PathFindingOnSquaredGrid.random(int, double).N", Type: "SimpleName", Va
========
Line: 401, SeenAt: "MethodInvocation"
Name: "PathFindingOnSquaredGrid.menu(int, double, int, int, int, int)", Type:
Name: "PathFindingOnSquaredGrid.main(String[]).gridSize", Type: "args SimpleNa
Name: "PathFindingOnSquaredGrid.main(String[]).probability", Type: "args Simpl
Name: "PathFindingOnSquaredGrid.main(String[]).y1", Type: "args SimpleName", V
Name: "PathFindingOnSquaredGrid.main(String[]).x1", Type: "args SimpleName", V
Name: "PathFindingOnSquaredGrid.main(String[]).y2", Type: "args SimpleName", V
Name: "PathFindingOnSquaredGrid.main(String[]).x2", Type: "args SimpleName", V
========
```

- Bindings -
  This file mentions all instances of where a variable is used, with details of its binding, type, value , location of occurence and the type of statement it is used in.

```
========= "PathFindingOnSquaredGrid.menu(int, double, int, int, int, int).n" =
Name: "PathFindingOnSquaredGrid.menu(int, double, int, int, int, int).n", Type
Name: "PathFindingOnSquaredGrid.menu(int, double, int, int, int, int).n", Type
Name: "PathFindingOnSquaredGrid.menu(int, double, int, int, int, int).n", Type
Name: "PathFindingOnSquaredGrid.menu(int, double, int, int, int, int).n", Type
Name: "PathFindingOnSquaredGrid.menu(int, double, int, int, int, int).n", Type

========= "PathFindingOnSquaredGrid.main(String[]).y1" =========
```

```
Name: "PathFindingOnSquaredGrid.main(String[]).y1", Type: "SimpleName", Value:
Name: "PathFindingOnSquaredGrid.main(String[]).y1", Type: "args SimpleName", V
Name: "PathFindingOnSquaredGrid.main(String[]).y1", Type: "args SimpleName", V
```

ServerLaunch Output

```
                                        Java Instrumentation Server
*******************************************************************************
Keep this server running until your use of Instrumentation is done
Server runs at 127.0.0.1:8080

*******************************************************************************

Server online at http://localhost:8080/
Press RETURN to stop...
WebSocket terminated
```

InstrumLaunch Output (truncated). Run on `projects/project2`

```
*******************************************************************************
                                        Java Instrumentation Application
*******************************************************************************
Choose a config file from below to run the instrumentation

To add a new project to the list add a new .conf file to the resources/project
specified in the documentation

*******************************************************************************

1: pathfinder.conf
2: rotatematrix.conf
0: Exit
2
Detected Java version: 11 in: /usr/lib/jvm/java-11-openjdk-amd64
Detected OS: Linux
parsing buildfile /media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mathewgeor
Project base dir set to: /media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mat
Build sequence for target(s) `compile' is [compile]
Complete build sequence is [compile, ]

compile:
parsing buildfile jar:file:/home/sherryl/.cache/coursier/v1/https/repo1.maven.
```

```
    [mkdir] Created dir: /media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mat
    [javac] /media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mathewgeorge_cou
    [javac] astparser/AP.java added as astparser/AP.class doesn't exist.
    [javac] astparser/TemplateClass.java added as astparser/TemplateClass.clas
    [javac] com/uic/edu/App.java added as com/uic/edu/App.class doesn't exist.
    [javac] Compiling 3 source files to /media/01D3908E9C0056A0/code/IdeaProje
    [javac] Using modern compiler
    [javac] Compilation arguments:
    [javac] '-d'
    [javac] '/media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mathewgeorge_cc
    [javac] '-classpath'
    [javac] '/media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mathewgeorge_cc
    [javac] '-sourcepath'
    [javac] '/media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mathewgeorge_cc
    [javac] '-g:none'
    [javac]
    [javac] The ' characters around the executable and arguments are
    [javac] not part of the command.
    [javac] Files to be compiled:
    [javac]     /media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mathewgeorge
    [javac]     /media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mathewgeorge
    [javac]     /media/01D3908E9C0056A0/code/IdeaProjects/sherryl_mathewgeorge
BUILD FINISHED
{home=home=/usr/lib/jvm/java-11-openjdk-amd64, options=options=-classpath proj
[JDI: Sending Command(id=8) JDWP.EventRequest.Set]
[JDI: Sending:                eventKind(byte): 6]
[JDI: Sending:                suspendPolicy(byte): 2]
[JDI: Sending:                modifiers(Modifier[]): ]
[JDI: Sending Command(id=10) JDWP.EventRequest.Set]
[JDI: Sending:                eventKind(byte): 7]
[JDI: Sending:                suspendPolicy(byte): 2]
[JDI: Sending:                modifiers(Modifier[]): ]
[JDI: Sending Command(id=12) JDWP.VirtualMachine.Dispose]
[JDI: Target VM interface thread exiting]

Starting Matrix:
 5 64 51
 50 81 36
 49 59 55

Rotated Matrix:
 51 36 55
 64 81 59
 5 50 49
```

```
{home=home=/usr/lib/jvm/java-11-openjdk-amd64, options=options=-classpath proj
[JDI: Sending Command(id=21) JDWP.EventRequest.Set]
[JDI: Sending:                 eventKind(byte): 6]
[JDI: Sending:                 suspendPolicy(byte): 2]
[JDI: Sending:                 modifiers(Modifier[]): ]
[JDI: Sending Command(id=23) JDWP.EventRequest.Set]
[JDI: Sending:                 eventKind(byte): 7]
[JDI: Sending:                 suspendPolicy(byte): 2]
[JDI: Sending:                 modifiers(Modifier[]): ]
[JDI: Sending Command(id=25) JDWP.VirtualMachine.Dispose]
[JDI: Target VM interface thread exiting]
```