How to run the code ?

1. `javac ./testcases/java/*.java`
2. `export CLASSPATH=./sootclasses-trunk-jar-with-dependencies.jar:$CLASSPATH`
3. `javac PA1.java`
4. `java PA1 <arg1> <arg2>`
    - **arg1 :** analysis_option [jap.dmt, jap.npc, jap.abc, ...]
    - **arg2 :** testcase_name [Testcase1, Testcase2 , ...]

Observation

Testcase 1

- Analysis : **jap.abc** (Array bound checker)
- The java code contains an array of int `Numbers`.
- In the `for` loop, all the access are within the correct index range and hence, the jimple file reports all these access as [safe lower bound] [safe upper bound]
- In line 14, however we accessed the value at a negative index `-2`. Since this is an out of bound access, jimple IR reports it as [potentially unsafe lower bound] [safe upper bound] at *line 14* .
- The array bound checker does a forward analysis. Its because before we conclude whether an access is out of bound or not, its important to know the size of the array first.

Testcase 2

- Analysis : **jap.abc** (Array bound checker)
- The java code contains an array of string `names`.
- For the `names` array, we make two different access, one at *line 10* (for index 2), other at *line 11* (for index 5) , jimple IR reports the first access as [safe lower bound] [safe upper bound], while the second access as [safe lower bound] [unsafe upper bound] becaues the names array has only 3 elements.
- The array bound checker does a forward analysis. Its because before we conclude whether an access is out of bound or not, its important to know the size of the array first.

Testcase 3

- Analysis : **jap.lvtagger** (Live variable analysis)
- Live variable analysis is a program analysis technique used to determine the set of variables that are live or still in use at various points in a program. A variable is considered "live" if its value may be used later in the program, meaning that there is a subsequent computation that depends on its current value. Identifying live variables is crucial for optimizing compilers and various program analyses.
- This test case has following snippet:

```
1.  int x = 5;
2.  int y = x + 3;
3.  int z = y * 2;
4.  int result = z;
5.  System.out.println(result);
```

1. **Itialize Sets:**
    - Create a set for each program point (line of code).
    - Initialize the set of live variables at each program point to be empty.
        2. **Iterative Analysis:**
            - Start from the end of the program and work backward.
            - For each line of code, update the live variable set based on the definition and use of variables.
2. **Data Flow Equations:**

    ```
    IN[n] = USE[n] U (OUT[n] - DEF[n])
    OUT[n] = U (IN[s]) for all successors s of n
    ```

    Here, `USE` is the set of variables used, `DEF` is the set of variables defined, and `U` represents the union of sets.

- **Analysis Steps:**
    - **Line 5** (`System.out.println(result);`):
        - OUT[5] = {}
        - IN[5] = OUT[5] ∪ USE[5] = {} ∪ {result} = {result}
    - **Line 4** (`int result = z;`):
        - OUT[4] = IN[5]
        - IN[4] = OUT[4] ∪ USE[4] - DEF[4] = {result} ∪ {z} - {result} = {z}
    - **Line 3** (`int z = y * 2;`):
        - OUT[3] = IN[4]
        - IN[3] = OUT[3] ∪ USE[3] - DEF[3] = {z} ∪ {y} - {z} = {y, z}
    - **Line 2** (`int y = x + 3;`):
        - OUT[2] = IN[3]
        - IN[2] = OUT[2] ∪ USE[2] - DEF[2] = {y, z} ∪ {x} - {y} = {x, y, z}
    - **Line 1** (`int x = 5;`):
        - OUT[1] = IN[2]
        - IN[1] = OUT[1] ∪ USE[1] - DEF[1] = {x, y, z} ∪ {} - {} = {x, y, z}

After the iterative analysis, the live variable sets stabilize:

- IN[1] = {x, y, z}

    - IN[2] = {x, y, z}
    - IN[3] = {y, z}
    - IN[4] = {z}
    - IN[5] = {result}

The jimple file also reports similar observation with some optimisation.
This analysis is a backward analysis because to know the variables that are going to be used in future, we need to analyse backwards.

Testcase 4

- Analysis: **jap.npc** (Null pointer checker)
- In this test case, we have defined two string variables viz str1 and str2.
- While str1 has beed initialised with a non null string value - "NonNullString", str2 can be non Null as well as Null depending on the output of the Math.random() function.
- As a result of this uncertainty for the value of str2. Jimple IR outputs str1 as [Non Null] at *line 7* while str2 as [Unknown] at *line 10*.
- This analysis is also a forward analysis, because to determine if a variable is null or not requires to know the statements that are before it. Hence analysis needs to be done in forward direction.

Testcase 5

- Analysis: **jap.dmt** (Tags dominator of statements)
- A node *d* of a control-flow graph **dominates** a node *n* if every path from the *entry node* to *n* must go through *d*. Notationally, this is written as *d* dom *n* (or sometimes $d \gg n$). By definition, every node dominates itself.
- To perform this analysis, the testcase contains a simple if else block. The jimple IR is observed to notice which statements dominate a given statement in the code.
- According to the Jimple IR:
    - Line 6 is sominated by Line: 6
    - Line 7 is dominated by Line : 6, 7
    - Line 9 (checking isCat() == 0) is dominated by Line: 6, 7, 9
    - Line 10 is dominated by: 6,7,9,10
    - Line 12 is dominated by: 6,7,9,12
    - Line 15 is dominated by: 6,7,9,15
    - Return is dominated by: 6,7,9,15, return
- This analysis is a forward analysis because we are only considered about the statements that comes before a given statement for determining whether it dominates or not.