Program Analysis, University of Stuttgart, Winter 2021/2022

Exercise 4: Information Flow and Call Graphs

Deadline for uploading solutions via Ilias:
January 26, 2022, 11:59pm Stuttgart time

Task 1 Information Flow Analysis [28 points]

This task is about dynamic information flow analysis. Consider the following JavaScript code to analyze:

```
1  let id = getId({"name": "Alice", "pass": "Password1"});
2  let accessCredentials = getAccess(id);
3
4  if (accessCredentials[0] > 1) {
5      console.error("Required access level lower than 2!")
6      console.error("Found access level = " + accessCredentials[0]);
7
8  } else {
9      let dataLength = api.getDataLength(accessCredentials[1]);
10     if (dataLength > 15) {
11         alert("The length of data surpasses max capacity");
12
13     } else {
14         let data = api.getData(id, accessCredientials[1]);
15         for (let i=0; i<data.length; i++){
16             console.log(encryptMessage({"data": data[i], "key": data[dataLength]}))
17         }
18     }
19 }
```

There are four security classes for this program which are presented in the lattice below (Figure 1).

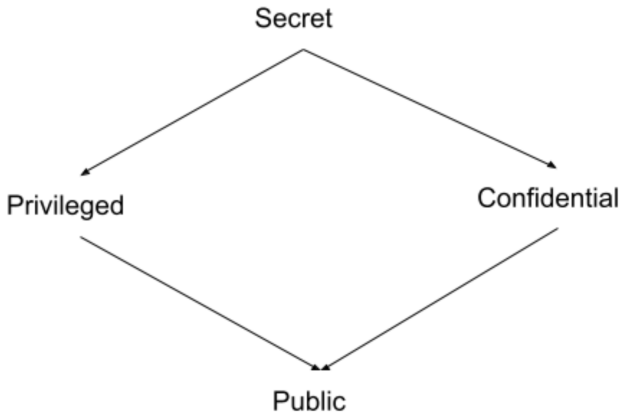


Figure 1: Lattice of security labels for Task 1.

Note that passing an argument to a function should be handled like an assignment to the formal parameter of the function. The returned values of functions `getId` and `getAccess` are presented in Tables 1 and 2 (respectively). The function `getAccess` returns an array of two values. The elements of an array can have different security labels. The first value of the array returned by `getAccess` is returned from the column `AccessGroup` in Table 2. The second value is returned from the column `Token` (in Table 2), for each given Id. As for the function `getDataLength`, it returns the value 15. The values returned by the used functions are labeled as follows:

- getId: Secret
- getAccess: Returns an array of two elements. The first element (`AccessGroup`) is Confidential while the second element (`Token`) is Secret.
- getDataLength: Privileged
- getData: Confidential
- encryptMessage: Public

The functions `console.error`, `console.log` and `alert` are untrusted sinks that should be reached by public information only.

Table 1: List of ids corresponding to each username and password.

Username	Password	Id
Alice	Password1	1
Bob	Password2	2

Table 2: Table of tokens and access groups for each given id.

Id	AccessGroup	Token
1	0	Token1
2	2	Token2

Subtask 1.1 Execution 1: Alice [14 points]

Consider a dynamic information flow analysis that considers both explicit and implicit flows. Suppose an execution where the user passes the values "Alice" and "Password1" as username and password, respectively, to the function getId.

- What are the security labels of variables and expressions during the execution? Use the following template to provide your answer. For unreachable lines of code during this execution, fill the security label with *Unreachable*.

Line	Variable or expression	Security label of variable or expression (after executing the line)
1	`id`	
2	`accessCredentials`	
6	`accessCredentials[0]`	
9	`datalength`	
10	`dataLength > 15`	
14	`data`	
16	`{"data":d, "key":  data[dataLength]}`	

- Does the execution violate the information flow policy? Explain your answer.

- If there is a leakage (through untrusted sinks) during this execution, how can you modify the line(s) of code causing the leakage so that you reduce information leakage.

- Based on the information that you can get from the untrusted sinks during this execution, does the token of Alice allow access to data? Consider both cases where you have the source code and the other case where you don't have the source code.

Subtask 1.2 Execution 2: Bob [14 points]

Consider a dynamic information flow analysis that considers both explicit and implicit flows. Suppose an execution where the user passes the values "Bob" and "Password2" as username and password, respectively, to the function getId.

- What are the security labels of variables and expressions during the execution? Use the following template to provide your answer. For unreachable lines of code during this execution, fill the security label with *Unreachable*.

Line	Variable or expression	Security label of variable or expression (after executing the line)
1	`id`	
2	`accessCredentials`	
6	`accessCredentials[0]`	
9	`datalength`	
10	`dataLength > 15`	
14	`data`	
16	`{"data":d, "key":  data[dataLength]}`	

- Does the execution violate the information flow policy? Explain your answer.

- If there is an information leakage (through untrusted sinks) during this execution, how can you modify the line(s) of code causing the leakage so that you reduce information leakage.

- Based on the information that you can get from the untrusted sinks during this execution, does the token of Bob allow access to data? Consider both cases where you have the source code and the other case where you don't have the source code.

Task 2 Universally Bounded Lattice [7 points]

Consider a policy defined with the following ordering rules: `A > B`, `A > C`, `A > D`, `B > E`, `C > F`, `D > E`, `D > F`, `E > G`, `F > G`, where A, B, C, D, E, F and G are corresponding security labels.

- Draw the graph of the previously defined Lattice.

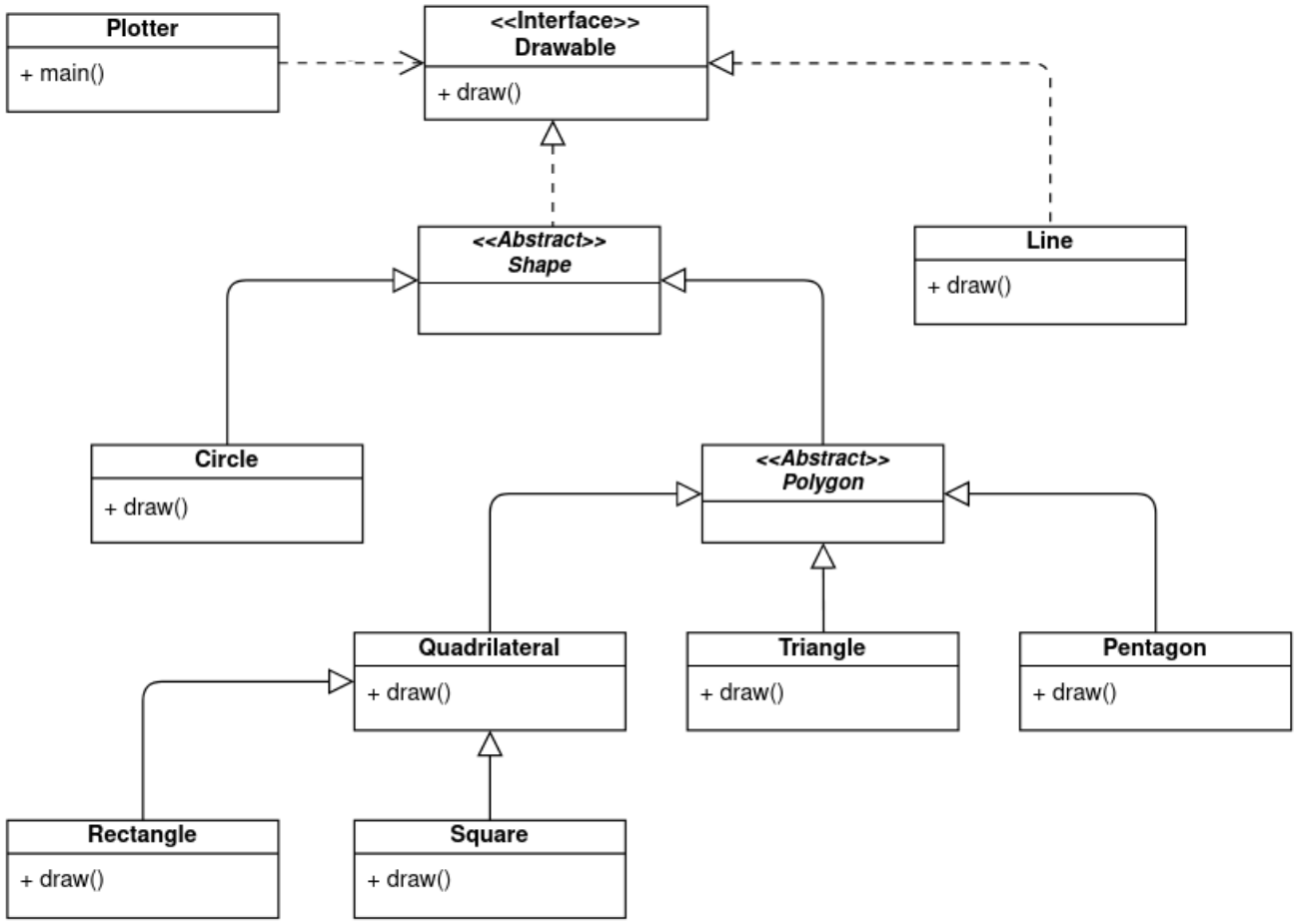
- Is it a universally bounded lattice (Explain)?

- Consider a program with a policy that only uses the labels A, B, D, E, G (with same previous ordering rules). Is the lattice of this program universally bounded (Explain)?

Task 3 Call Graphs: CHA, RTA and VTA [30 points]

Consider the following class diagram of a Java program:

Figure 2: Class Diagram



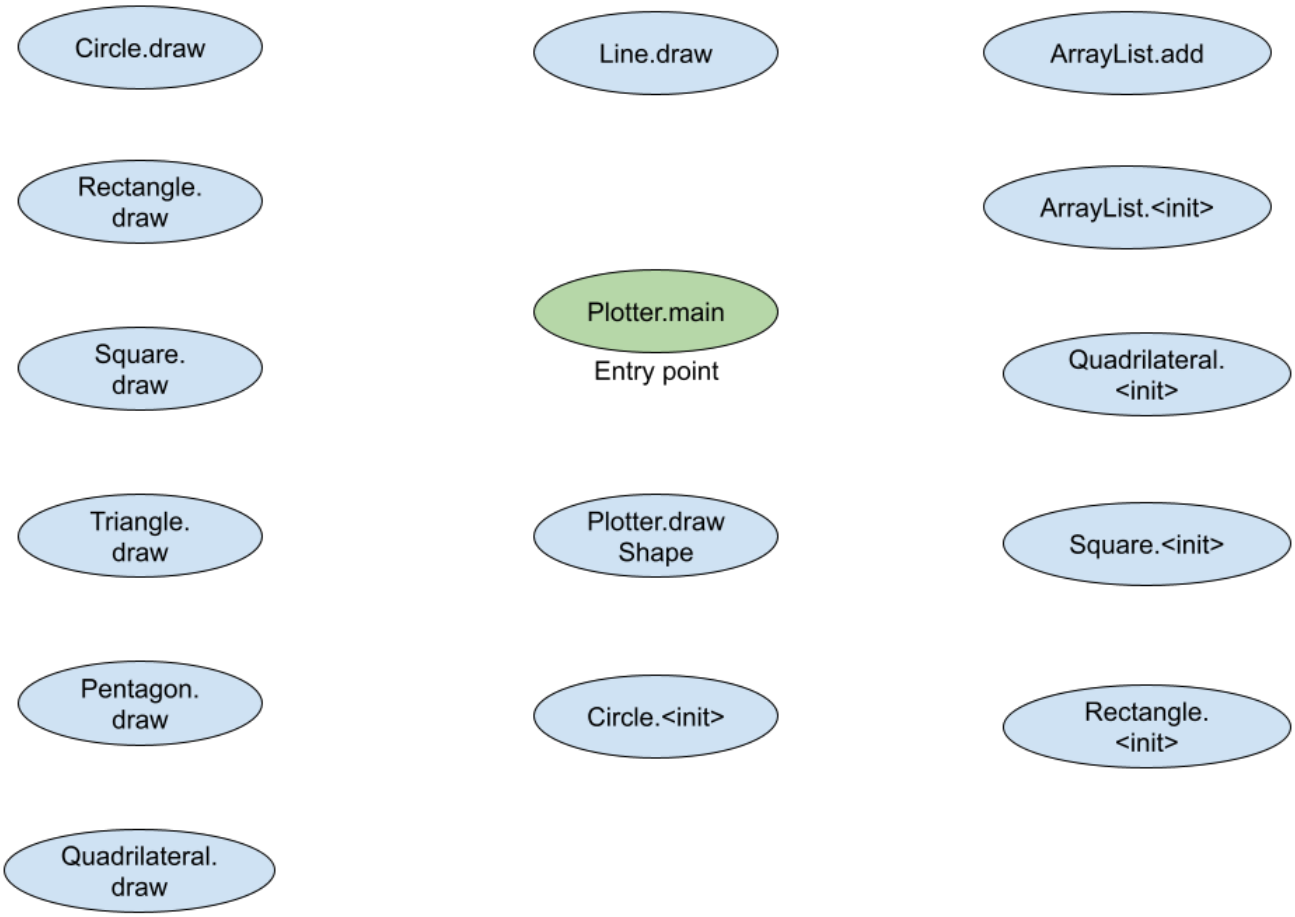
The implementation of the class `Plotter` is presented in the snippet of code below. All the classes and interfaces presented in the diagram are in a package called model. Thus, line 3 (in the code) imports all of them.

```
1  import java.util.ArrayList;
2  import java.util.List;
3  import model.*
4
5  class Plotter {
6      public static void main(String[] args) {
7          Quadrilateral quad1 = new Quadrilateral();
8          Quadrilateral quad2 = new Quadrilateral();
9          Polygon square = new Square();
10         Polygon rec = new Rectangle();
11         quad1 = (Quadrilateral)rec;
12
13         List<Polygon> polygons = new ArrayList<Polygon>();
```

```
14          polygons.add(rec);
15          polygons.add(square);
16
17          quad1.draw();
18          rec.draw();
19
20          drawShape();
21      }
22      public static void drawShape(){
23          Shape c = new Circle();
24          Shape r = new Rectangle();
25          r = c;
26          r.draw();
27      }
28  }
```

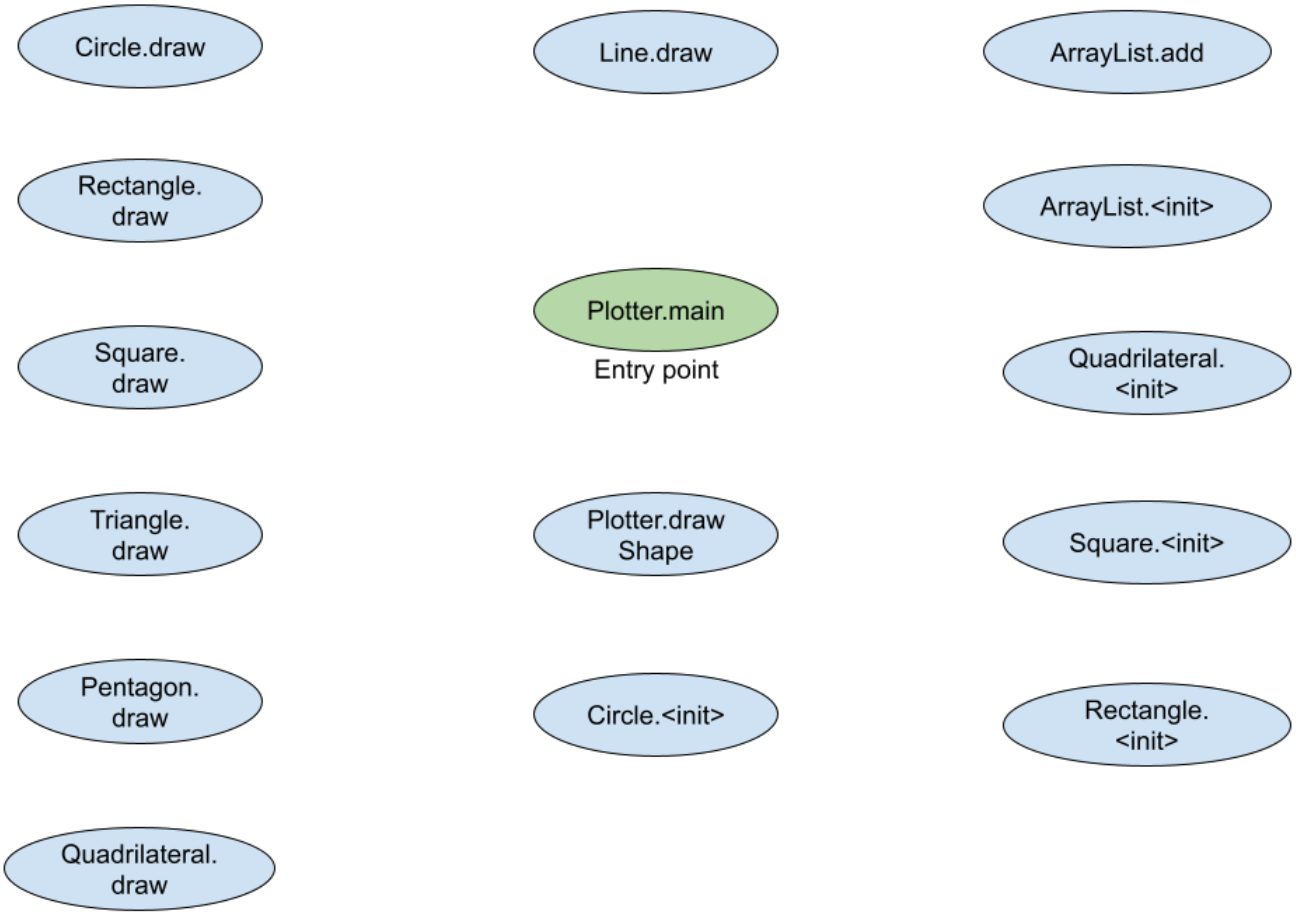
Subtask 3.1 CHA Graph [5 points]

- Considering the previous class diagram in Figure 2 and the snippet of code, provide the call graph computed by the CHA (Class Hierarchy Analysis) algorithm.



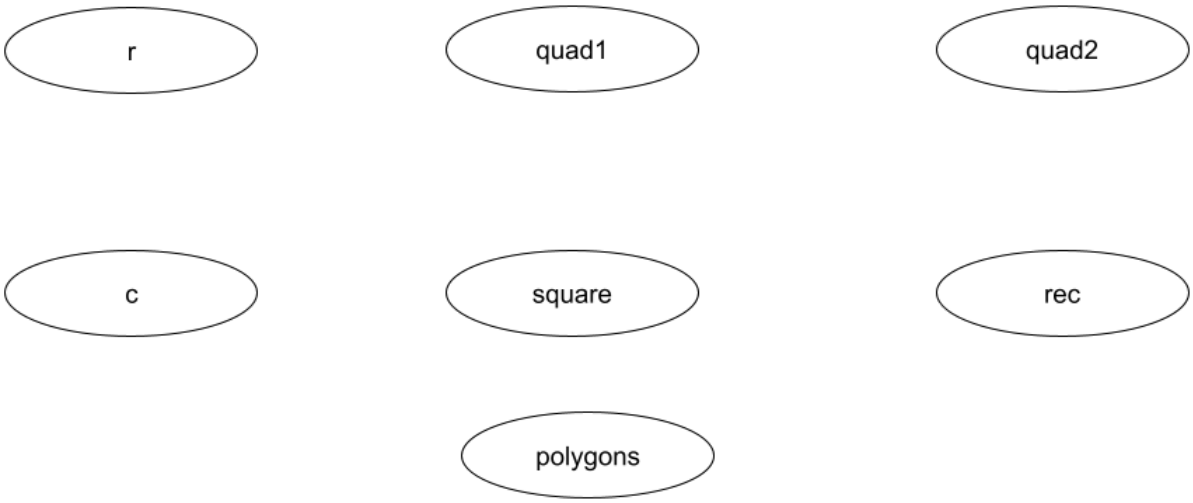
Subtask 3.2 RTA Graph [5 points]

- Considering the previous class diagram in Figure 2 and the snippet of code, provide the call graph computed by the RTA (Rapid Type Analysis) algorithm.

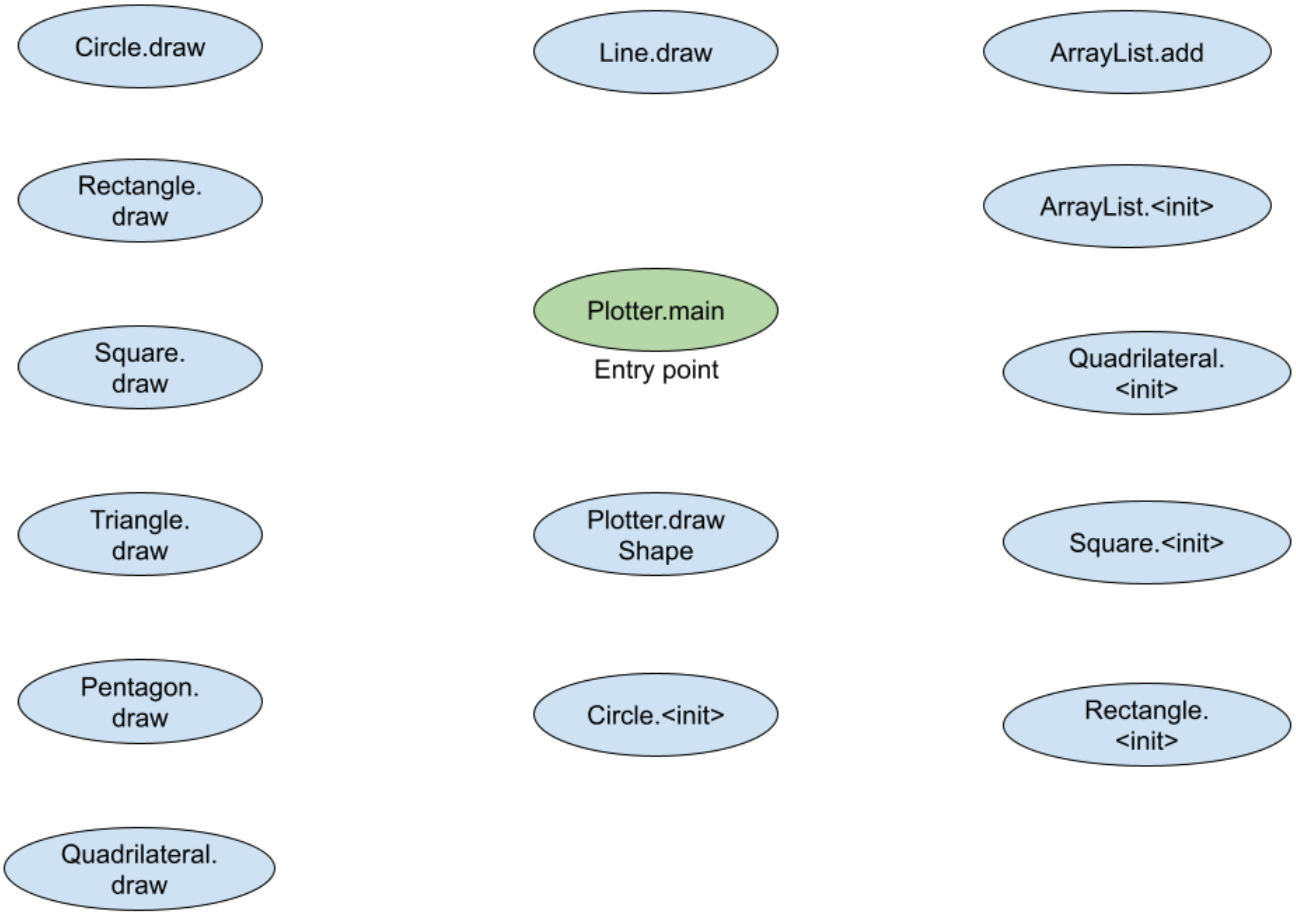


Subtask 3.3 VTA Graph [10 points]

- Considering the previous class diagram in Figure 2 and the snippet of code, provide the type propagation graph computed by VTA (Variable Type Analysis).

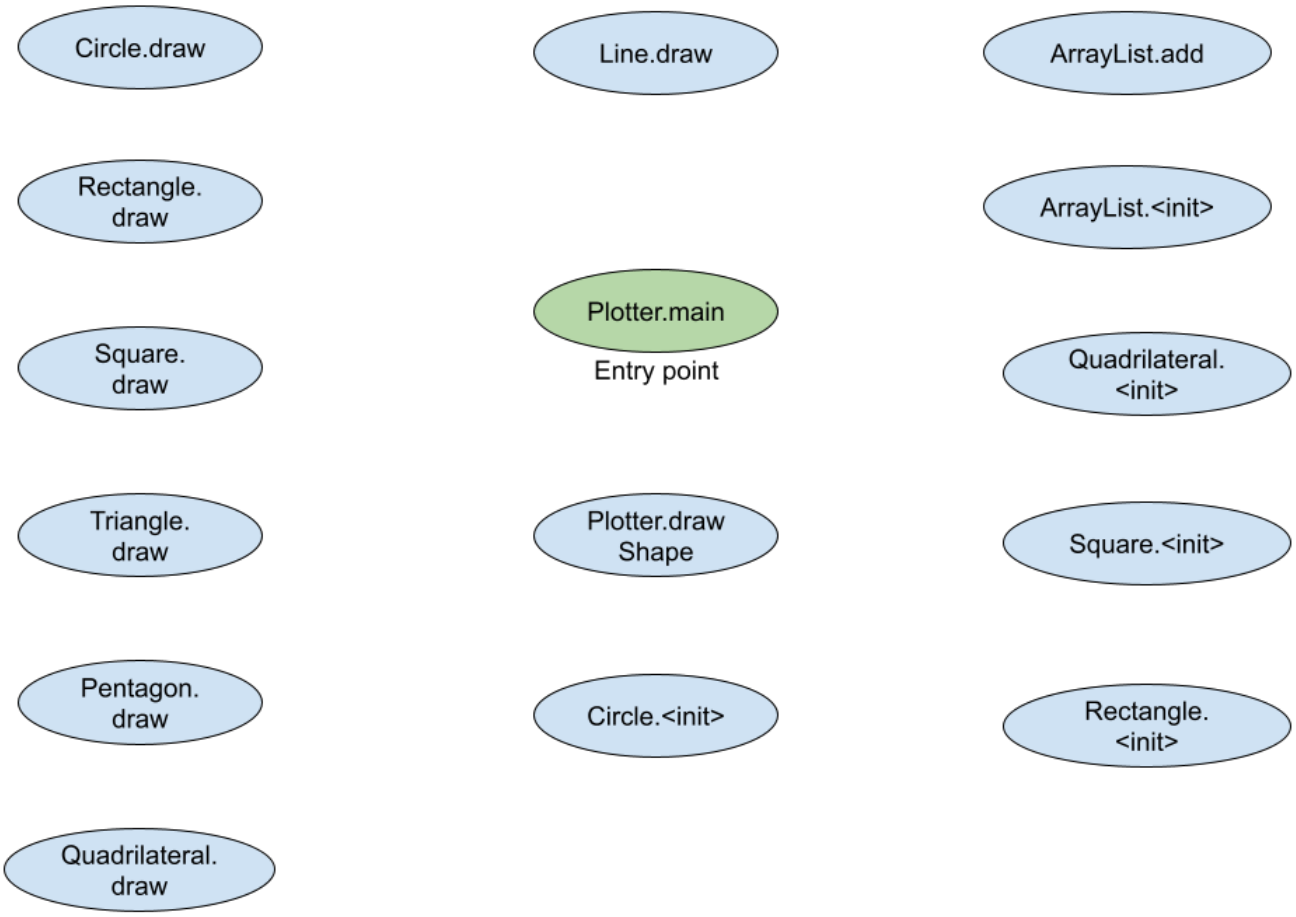


- Based on the types computed by VTA, give the call graph that VTA produces starting from the RTA graph.



Subtask 3.4 Dynamic Execution Call Graph [5 points]

- By performing a dynamic execution of the previous program, provide the call graph representing only the calls that happen during the dynamic execution.



Subtask 3.5 Comparision Between Algorithms [5 points]

- Using previously computed graphs, fill in the following table (Useless edges are edges that don't appear in the graph computed from dynamic execution):

Algorithm	Total number of edges	Number of useless edges
CHA		
RTA		
VTA		

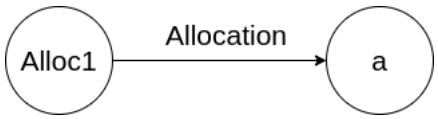
Task 4 Call Graphs: Pointer Analysis [35 points]

Consider the following Java program:

```
1  //classes definition
2  class A{
3      public String f(){
4          return "A";
5      }
6  }
7
8  class B extends A{
9      public String f(){
10         return "B";
11     }
12 }
13
14 class C{
15     public A a;
16     public String f(A x){
17         return x.f();
18     }
19 }
20 //main...
21 public static void main(String[] args){
22     A a = new A();
23     A b = new B();
24     C c = new C();
25
26     c.a = a;
27     a = b;
28     int i = 0;
29     if (i==1) b = c.a;
30
31     String r1 = a.f();
32     String r2 = b.f();
33     String r3 = c.f(c.a);
34 }
```

Subtask 4.1 PAG: Subset-based Analysis [20 points]

- Fill in the following table by specifying for each line of code the nodes and the edge connecting them as well as the type of the edge (allocation, assignment, field load or field store). For example, line 22 would produce the following:



Line	Code	Representation (nodes and edges)
22	`A a = new A();`	
23	`B b = new B();`	
24	`C c = new C();`	
26	`c.a = a;`	
27	`a = b;`	
28	`int i = 0;`	
29	`b = c.a;`	
31	`String r1 = a.f();`	
32	`String r2 = b.f();`	
33	`String r3 = c.f(c.a);`	

- Draw the pointer assignment graph for the previous program.

- Using subset based analysis and with the help of the previously drawn graph, calculate the following points-to sets (at their final state considering the entire code):

  `pts(a) =`

  `pts(b) =`

  `pts(c) =`

  `pts(c.a) =`

- Give the value stored in `r1, r2, r3` if it's possible to know based on the performed analysis. Otherwise explain why we can't conclude their values considering this analysis.

  `r1 :`

  `r2 :`

  `r3 :`

Subtask 4.2 PAG: Ordered Equality-Based Analysis [15 points]

In this part, we will introduce a variation of the subset-based analysis algorithm. The new algorithm is given below. This algorithm calculates points-to sets edge by edge. The input of the algorithm is the list of edges ordered by their order of appearance in code which is the same order presented in the table of question one in Subtask 4.1. The second difference to the algorithm presented in the lecture is that the algorithm uses equality-based propagation[1] of pointers which means instead of adding pts(a) to pts(b), pts(b) will be assigned pts(a).

```
1 for each edge in the ordered list of edges:
2     if allocation edge Alloc -> a: pts(a) = {Alloc}
3     if assignment edge a -> b : pts(b) = pts(a)
4     if load edge a.f -> b: pts(b) = pts(a.f)
5     if store edge a -> b.f : pts(b.f) = pts(a)
```

- Using equality based analysis and with the help of the previously drawn graph, calculate the following points-to sets (at their final state considering the entire code):

  `pts(a) =`

  `pts(b) =`

  `pts(c) =`

  `pts(c.a) =`

- Give the value stored in `r1, r2, r3` if it's possible to know based on the performed analysis. Otherwise explain why we can't conclude their values considering this analysis.

  `r1 :`

  `r2 :`

  `r3 :`

[1] Equality-based algorithms exist in the literature, but may be different from the one presented in this task.

- Is the concluded value of r2 the same as the value producing during execution? (Explain):

- What's the limitation illustrated by this example (The limitation of the Equality-Based propagation)?