COMP 322: Parallel and Concurrent Programming

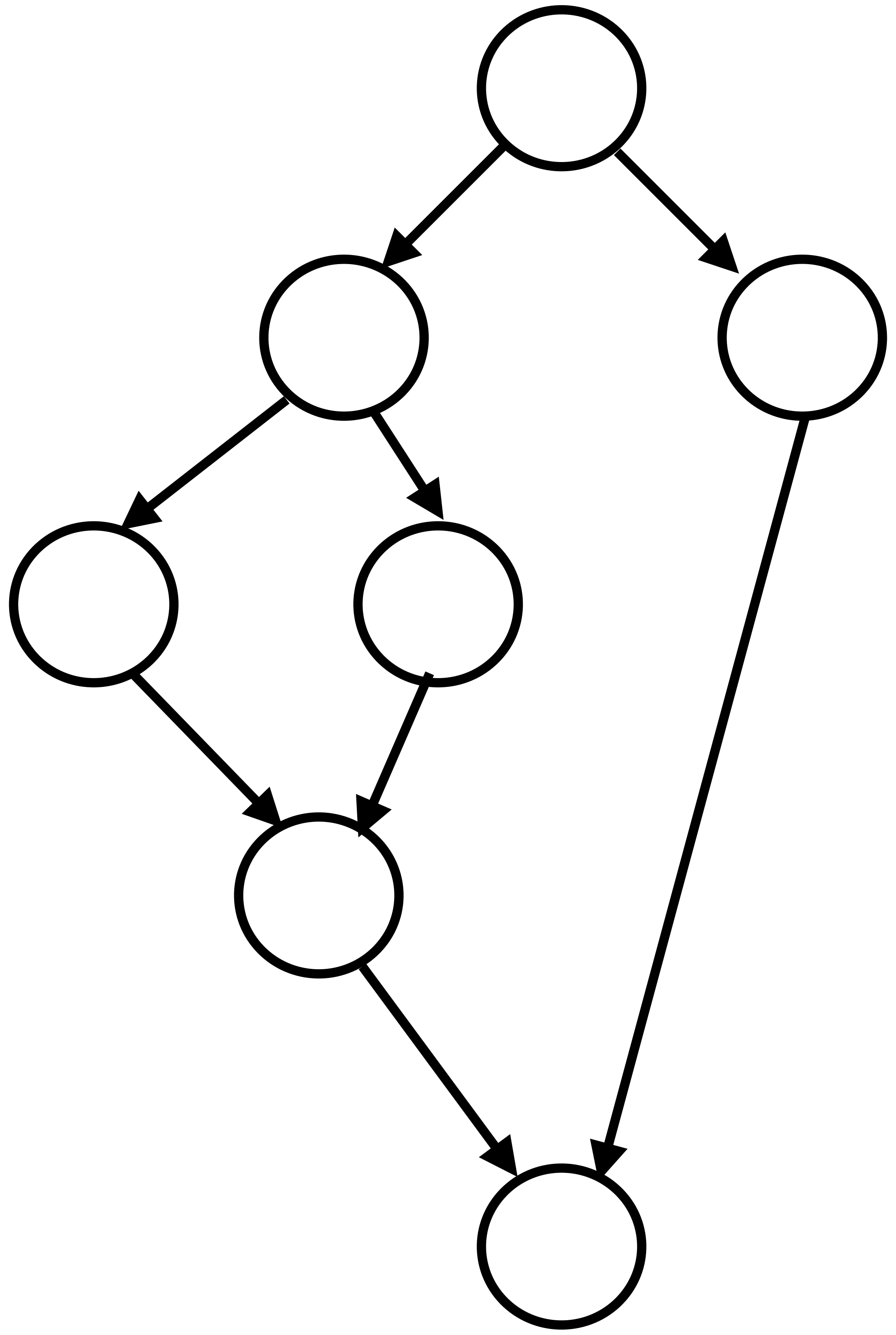
Lecture 11: Scheduling

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

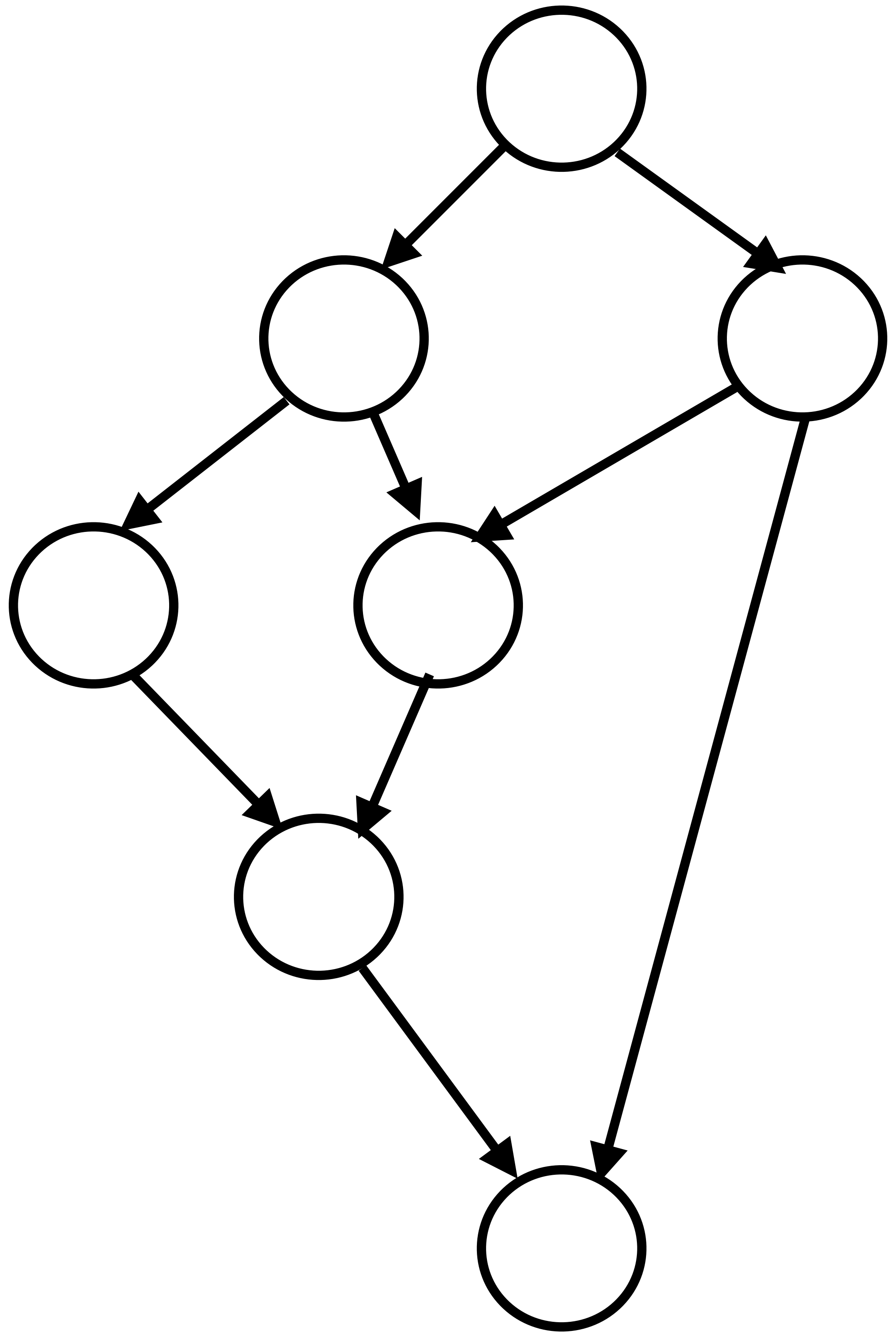
Computation Graphs

Structured Parallelism
(Finish/async)



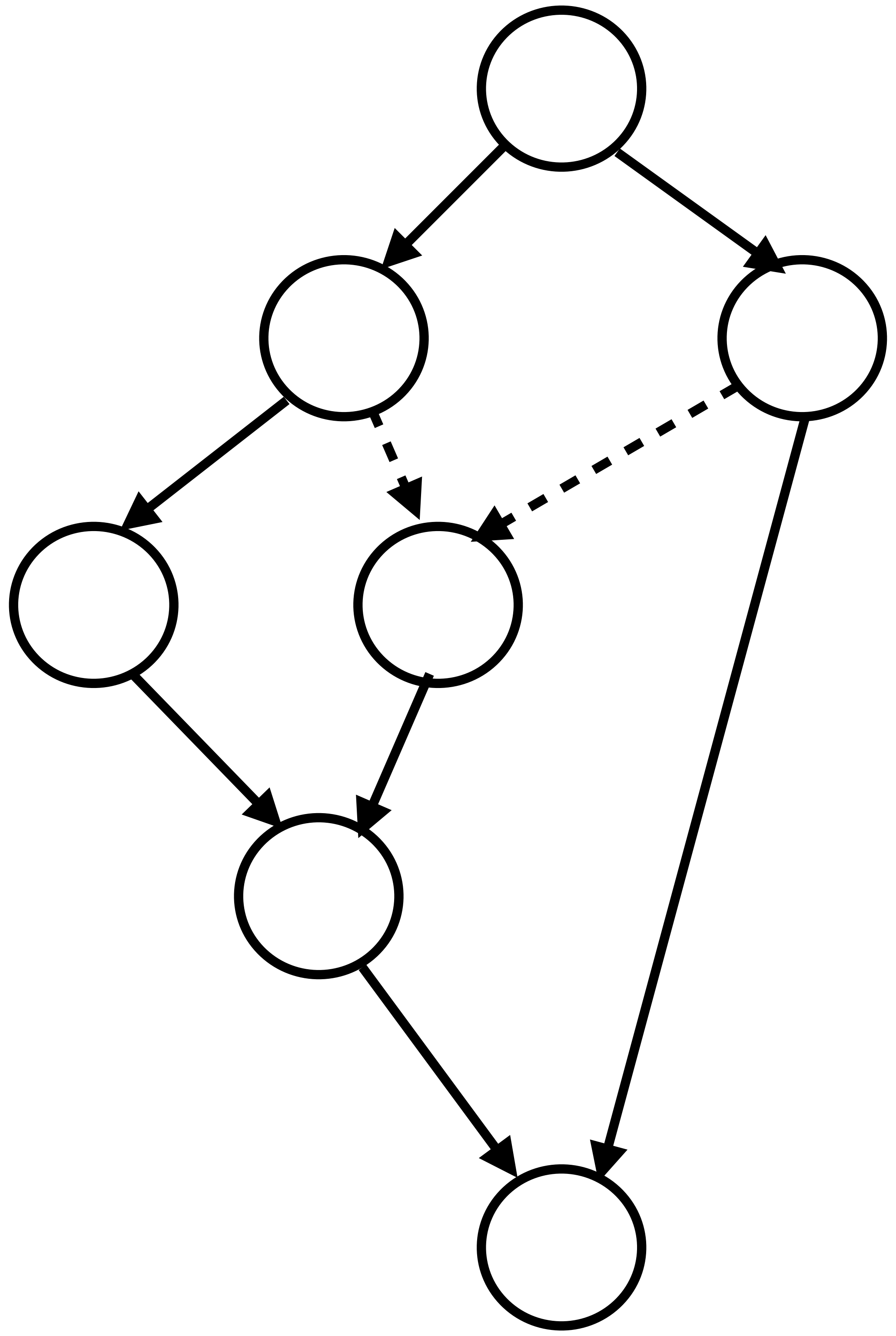
<

Futures and Future Tasks



<

Promises and Data-Driven Tasks



Computation Graphs

- Structured parallelism (finish/async):
  Create structured graphs (similar to what structured programming can create)
  No high-level data representation: have to share data
  Fast implementation, easy to synchronize large # of tasks
- Futures and future tasks:
  Easy to construct unstructured, arbitrary graphs
  Elegant, functional high-level data representation: futures
  Functional, “push” model: “where is the data going to, create futures for those”
  Large overhead when handling large # of tasks
- Promises and data-driven tasks:
  Easy to construct unstructured, arbitrary graphs with unknown task-promise association
  Data-driven, “pull” model: “what data does this DDT depend on, create promises for those”
  Can have a faster implementation than futures
  Large overhead when handling large # of tasks

Ordering Constraints and Transitive Edges in a Computation Graph

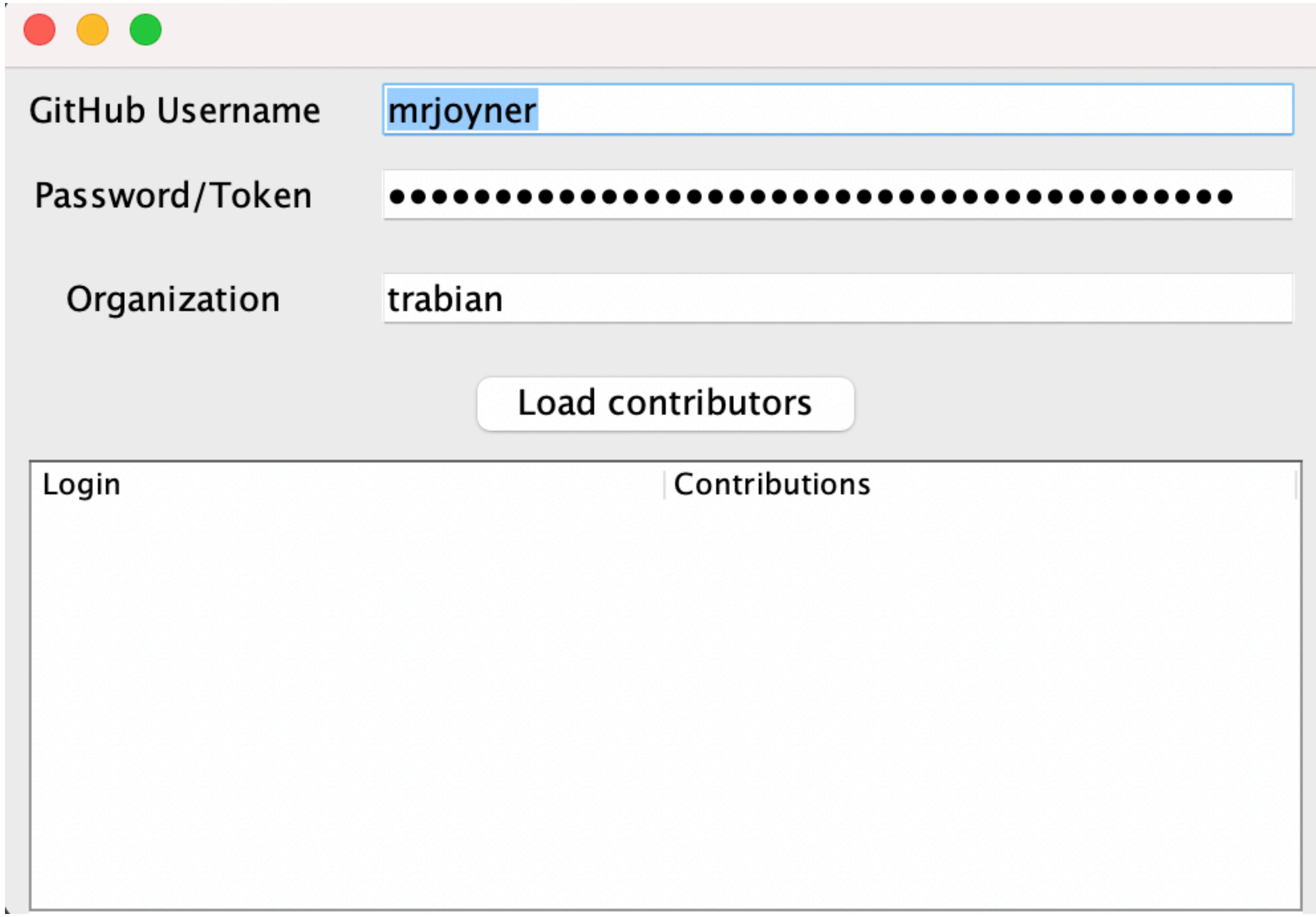
- The primary purpose of a computation graph is to determine if an ordering constraint exists between two steps (nodes)
  - Observation: Node A must be performed before node B if there is a path of directed edges from A and B

- An edge, X →Y, in a computation graph is said to be transitive if there exists a path of directed edges from X to Y that does not include the X →Y edge
  - Observation: Adding or removing a transitive edge does not change the ordering constraints in a computation graph

GUI Events with Java Swing

- Swing enables you to build a GUI in Java and respond to user events
- Containers (e.g. JFrame)
- Components
  - JButton
  - JLabel
  - JTextField
- Users interact with the GUI and trigger actions (events)
- ActionListeners are setup for a component to respond to the event

Homework 2: GitHub Contributors

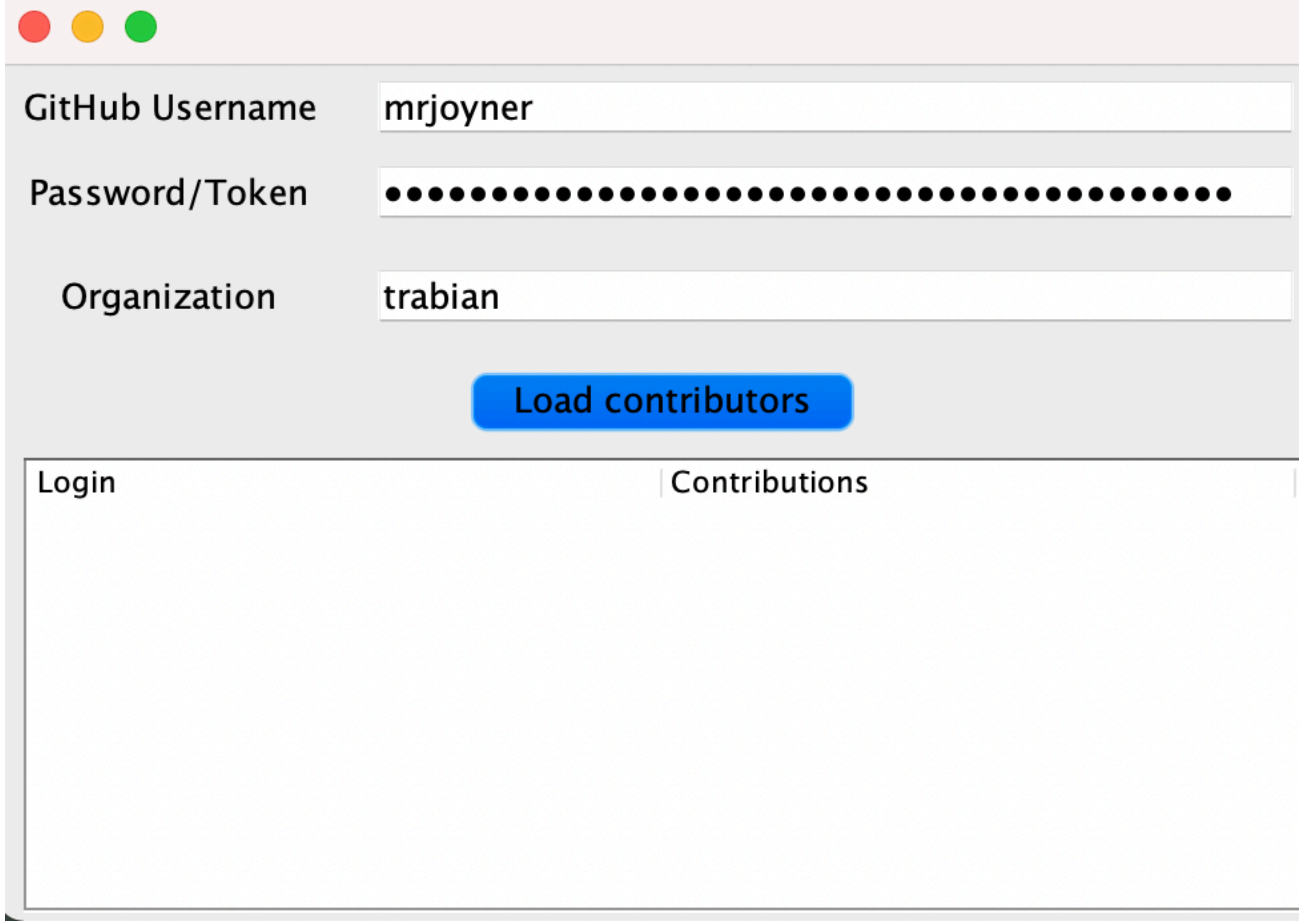


GitHub Username: mrjoyner

Password/Token: ●●

Organization: trabian

GitHub Contributors Event Handling with ActionListener

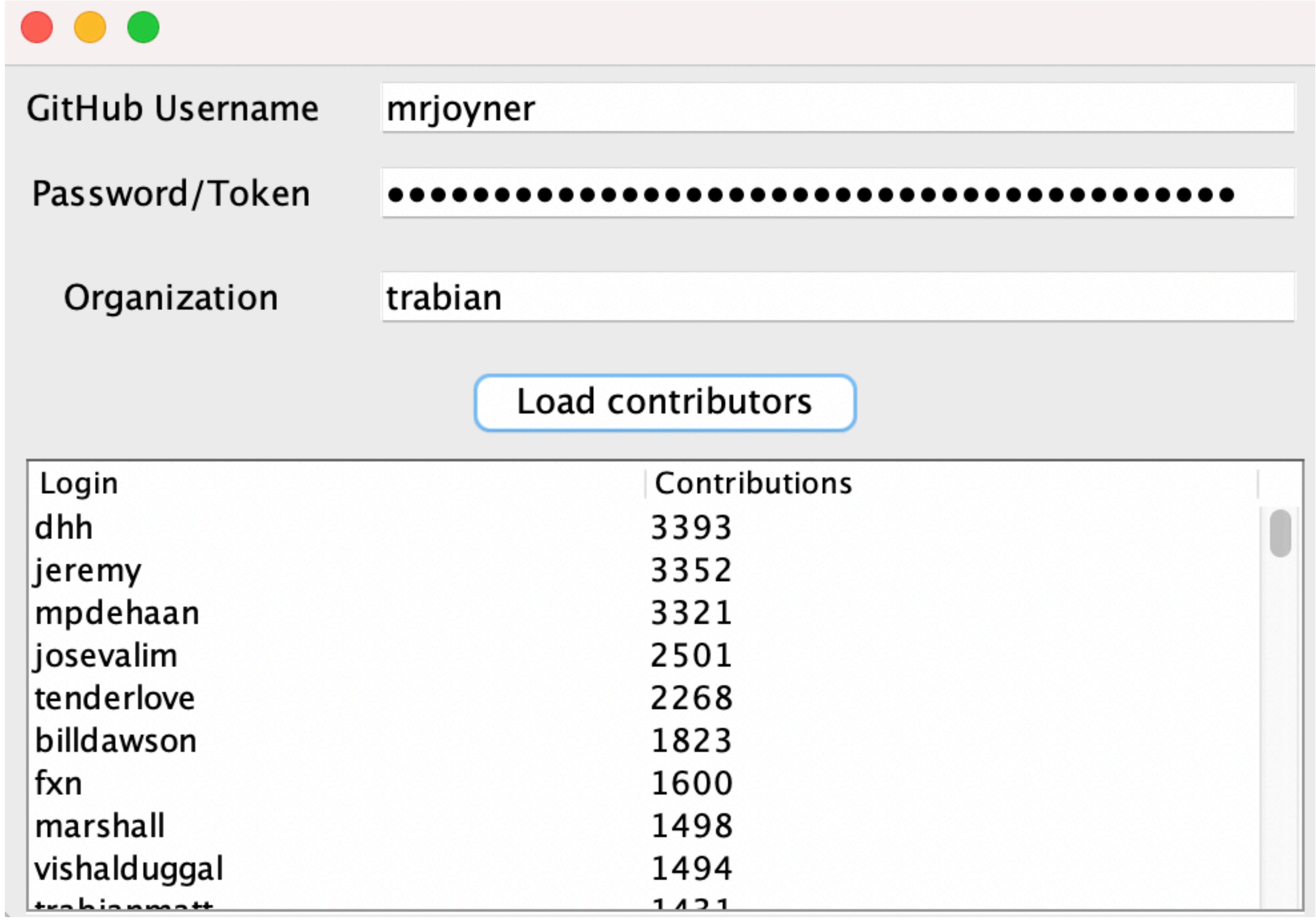


GitHub Username: mrjoyner

Password/Token: ••

Organization: trabian

GitHub Contributors



GitHub Username mrjoyner

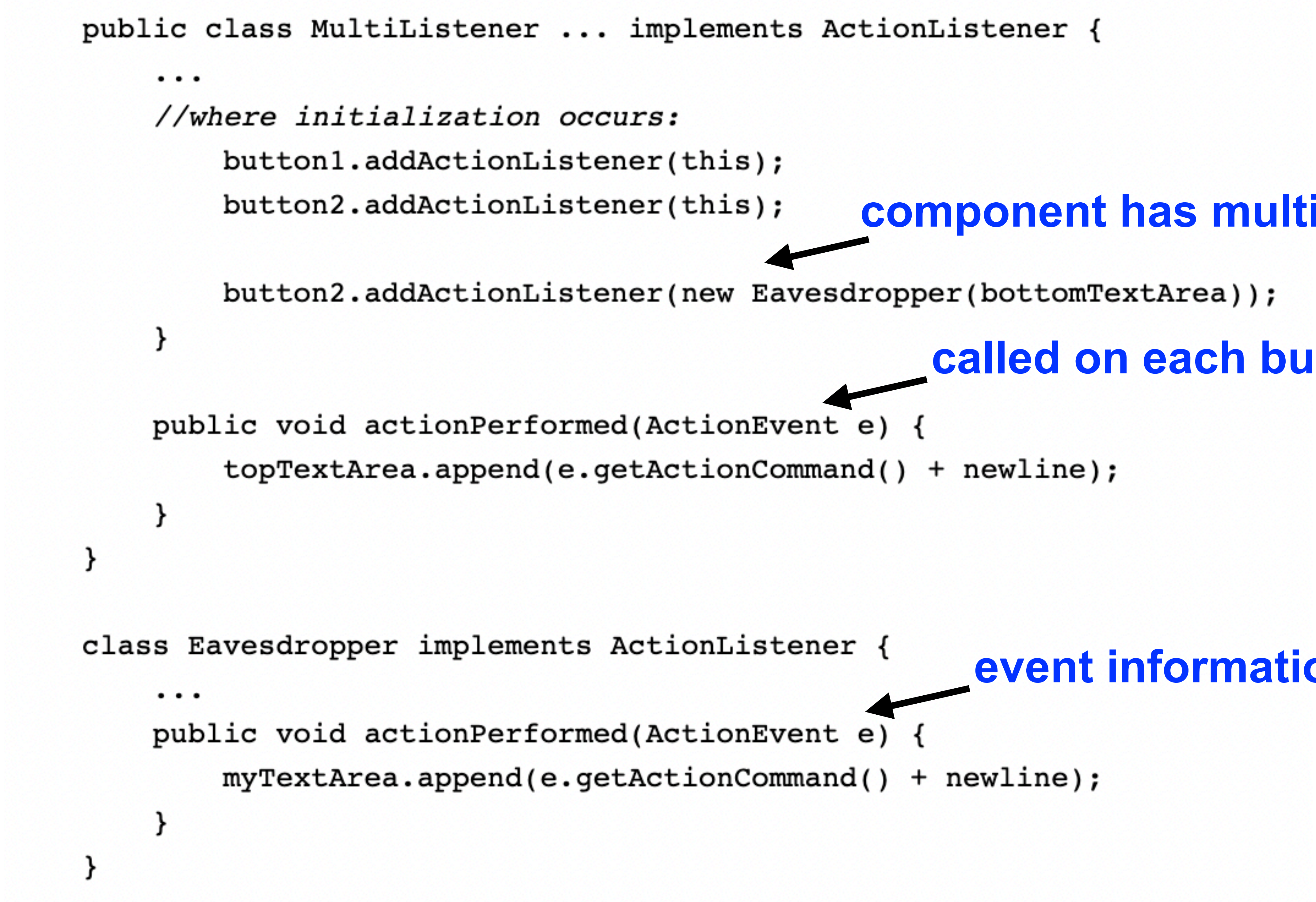
Password/Token ●●

Organization trabian

Login	Contributions
dhh	3393
jeremy	3352
mpdehaan	3321
josevalim	2501
tenderlove	2268
billdawson	1823
fxn	1600
marshall	1498
vishalduggal	1494

ActionListeners

Adding ActionListener without a lambda



```
public class MultiListener ... implements ActionListener {
    ...
    //where initialization occurs:
        button1.addActionListener(this);
        button2.addActionListener(this);

        button2.addActionListener(new Eavesdropper(bottomTextArea));
    }

    public void actionPerformed(ActionEvent e) {
        topTextArea.append(e.getActionCommand() + newline);
    }
}

class Eavesdropper implements ActionListener {
    ...
    public void actionPerformed(ActionEvent e) {
        myTextArea.append(e.getActionCommand() + newline);
    }
}
```

component has multiple listeners

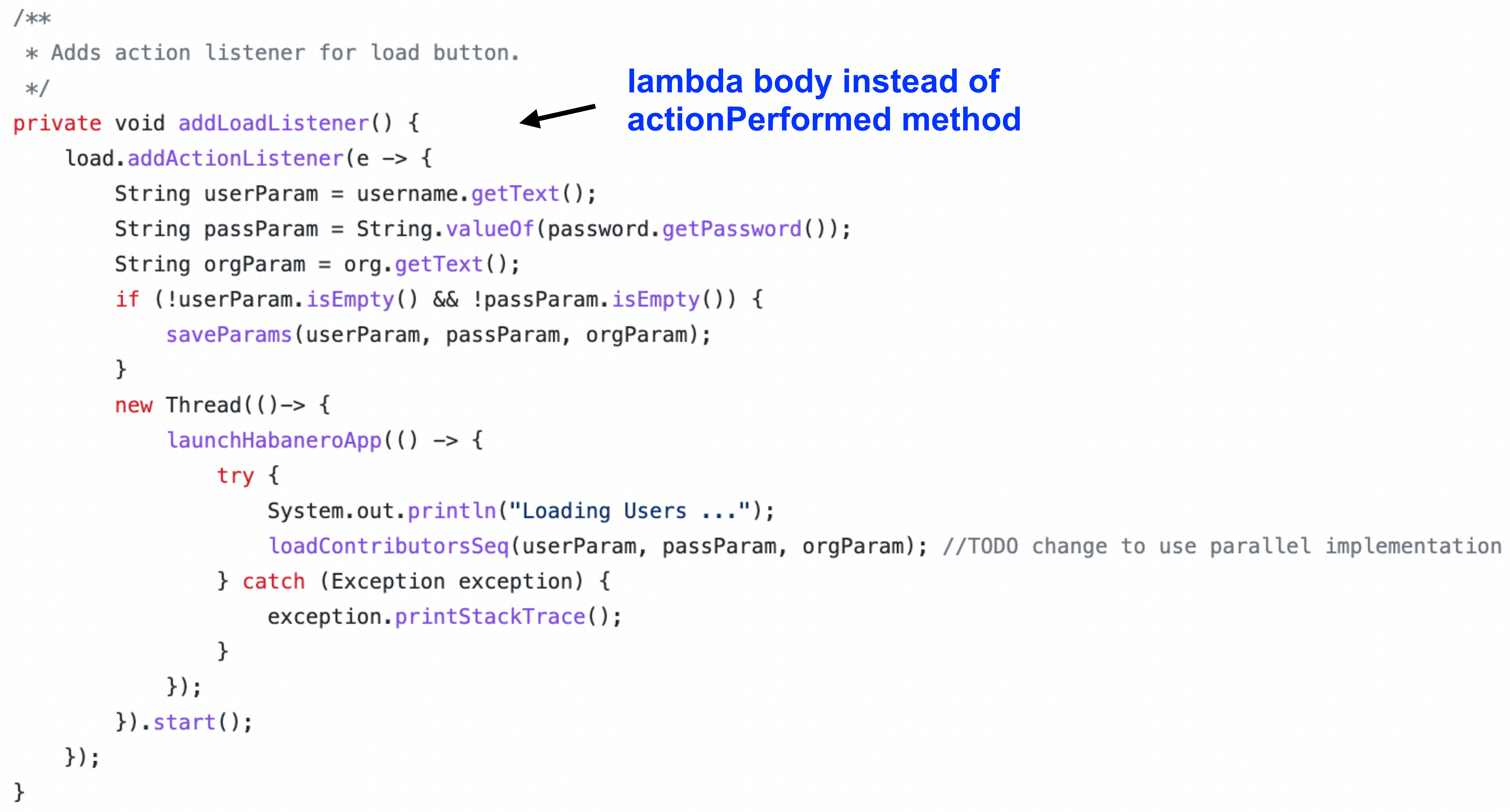
called on each button click

event information

See: https://docs.oracle.com/javase/tutorial/uiswing/events/intro.html

ActionListeners

Adding ActionListener with a lambda



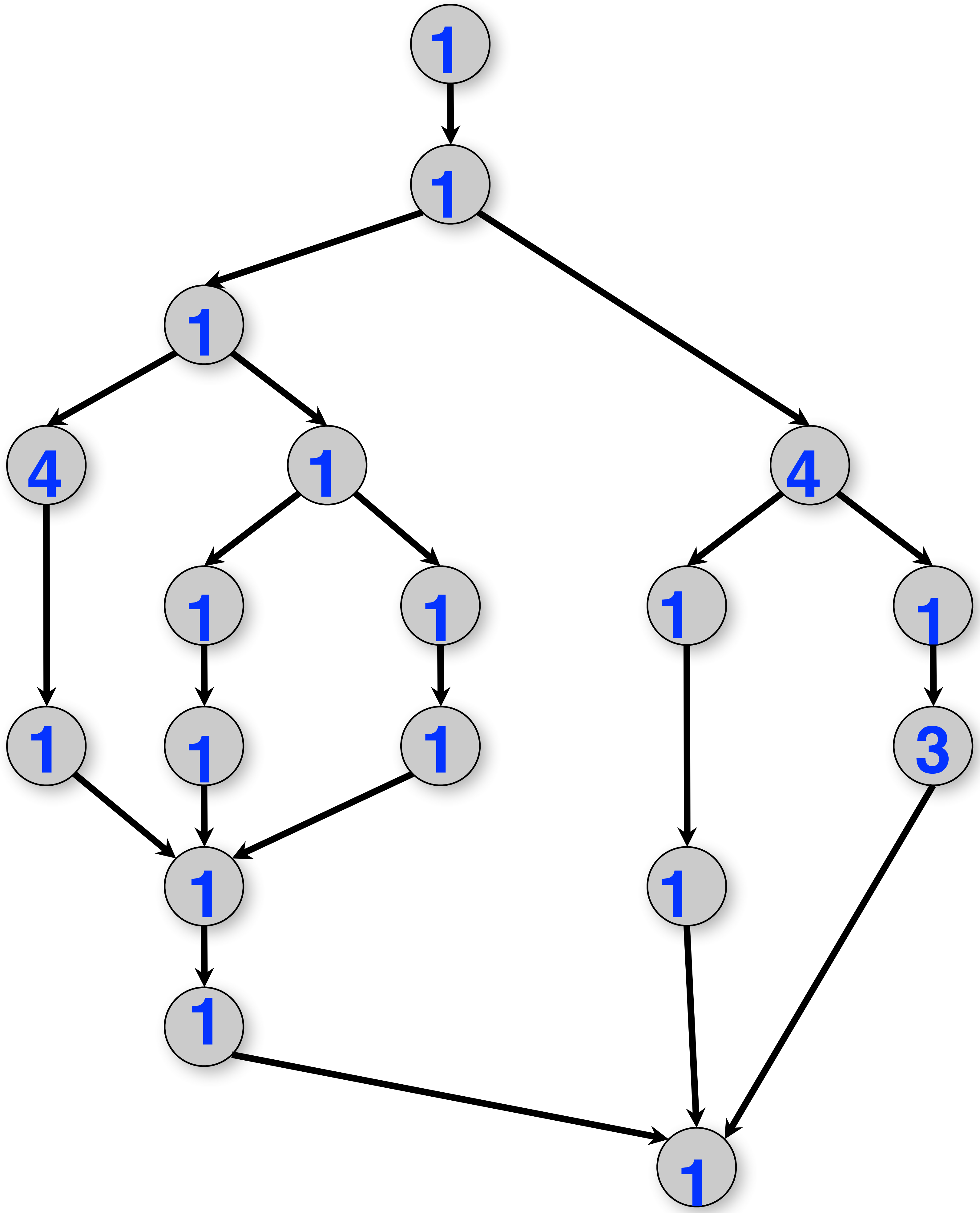
```
/**
 * Adds action listener for load button.
 */
private void addLoadListener() {
    load.addActionListener(e -> {
        String userParam = username.getText();
        String passParam = String.valueOf(password.getPassword());
        String orgParam = org.getText();
        if (!userParam.isEmpty() && !passParam.isEmpty()) {
            saveParams(userParam, passParam, orgParam);
        }
        new Thread(()-> {
            launchHabaneroApp(() -> {
                try {
                    System.out.println("Loading Users ...");
                    loadContributorsSeq(userParam, passParam, orgParam); //TODO change to use parallel implementation
                } catch (Exception exception) {
                    exception.printStackTrace();
                }
            });
        }).start();
    });
}
```

lambda body instead of actionPerformed method

Ideal Parallelism (Recap)

- Define ideal parallelism of Computation G Graph as the ratio, WORK(G)/CPL(G)

- Ideal Parallelism only depends on the computation graph, and is the speedup that you can obtain with an unbounded number of processors



__Example__:

WORK(G) = 26

CPL(G) = 11

Ideal Parallelism = WORK(G)/CPL(G) = 26/11 ~ 2.36

What is the critical path length of this parallel computation?

```
finish (() -> {        // F1
  async (() -> A);     // Boil water & pasta (10)
  finish (() -> {      // F2
    async (() -> B1);  // Chop veggies (5)
    async (() -> B2);  // Brown meat (10)
  });                  // F2
  B3;                  // Make pasta sauce (5)
})                     // F1
```

Step A

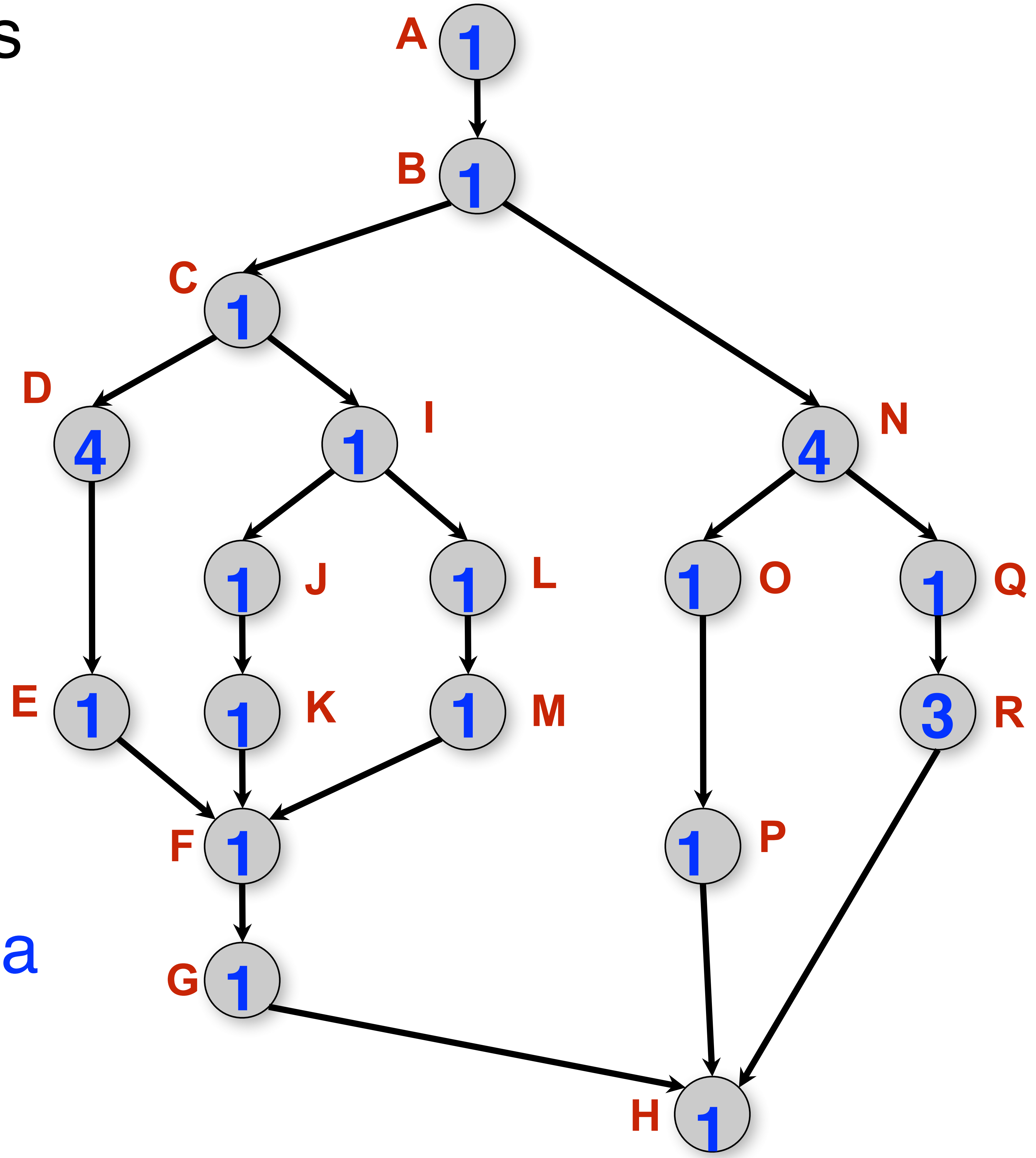
Step B1

Step B2

Step B3

Scheduling of a Computation Graph on a fixed number of processors

Node label = time(N), for all nodes N in the graph



Start time	Proc 1	Proc 2	Proc 3
0	A		
1	B		
2	C	N	
3	D	N	I
4	D	N	J
5	D	N	K
6	D	Q	L
7	E	R	M
8	F	R	O
9	G	R	P
10	H		
11	Completion time = 11		

NOTE: this schedule achieved a completion time of 11. Can we do better?

Scheduling of a Computation Graph on a fixed number of processors

- Assume that node N takes TIME(N) regardless of which processor it executes on, and that there is no overhead for creating parallel tasks

- A schedule specifies the following for each node
  - START(N) = start time
  - PROC(N) = index of processor in range 1…P

  such that
  - START(i) + TIME(i) <= START(j), for all CG edges from i to j (Precedence constraint)
  - A node occupies consecutive time slots in a processor (Non-preemption constraint)
  - All nodes assigned to the same processor occupy distinct time slots (Resource constraint)

Greedy Schedule

- A greedy schedule is one that never forces a processor to be idle when one or more nodes are ready for execution

- A node is ready for execution if all its predecessors have been executed
- Observations
  - T_1 = WORK(G), for all greedy schedules
  - T_∞ = CPL(G), for all greedy schedules

- $T_P(S)$ = execution time of schedule S for computation graph G on P processors

Lower Bounds on Execution Time of Schedules

- Let T_P = execution time of a schedule for computation graph G on P processors
  - T_P can be different for different schedules, for same values of G and P

- Lower bounds for all greedy schedules
  - Capacity bound: $T_P \geq WORK(G)/P$
  - Critical path bound: $T_P \geq CPL(G)$

- Putting them together
  - $T_P \geq \max(WORK(G)/P, CPL(G))$

Upper Bound on Execution Time of Greedy Schedules

Theorem [Graham ’66].
Any greedy scheduler achieves

$$T_P \leq WORK(G)/P + CPL(G)$$

Start time	Proc 1	Proc 2	Proc 3
0	A		
1	B		
2	C	N	
3	D	N	I
4	D	N	J
5	D	N	K
6	D	Q	L
7	E	R	M
8	F	R	O
9	G	R	P
10	H		
11			

Proof sketch:
Define a time step to be complete if P processors are scheduled at that time, or incomplete otherwise

complete time steps ≤ WORK(G)/P

incomplete time steps ≤ CPL(G)

Bounding the Performance of Greedy Schedulers

Combine lower and upper bounds to get

$\max(\mathrm{WORK}(G)/P, \mathrm{CPL}(G)) \leq T_P < \mathrm{WORK}(G)/P + \mathrm{CPL}(G)$

Corollary: Any greedy scheduler achieves execution time T_P that is within a factor of 2 of the optimal time (since max(a,b) and (a+b) are within a factor of 2 of each other, for any $a \geq 0, b \geq 0$).

Corollary 2: Lower and upper bounds approach the same value whenever:

- There's lots of parallelism, $\mathrm{WORK}(G)/\mathrm{CPL}(G) \gg P$
- Or there's little parallelism, $\mathrm{WORK}(G)/\mathrm{CPL}(G) \ll P$