Dining Philosophers

A classical concurrent programming problem, introduced by Edsger Dijkstra. Philosophers, sitting at a round table, spend their time thinking and eating. At the center of the table is an enormous dish of spaghetti. When a philosopher wishes to eat, he (she) must take the forks available to his (her) immediate left and right in order to eat the spaghetti.

The pseudo code of a philosopher is

```
while (true) {
   "think"
   "get forks"
   "eat"
   "release forks"
}
```

A solution of the dining philosophers problem in pseudo-code (usually, N is 5):

```
Semaphore forks[N] = {1, 1, 1, 1, 1};

process Philosopher ((int id=0; id < N; id++)) {
   int right = id, left = (id+1) == N ? 0 : id+1;

    while (true) {
      System.out.println("Philisopher " +id+" is thinking");
      P(forks[left]); P(forks[right]);
      System.out.println("Philisopher " +id+" is eating");
      V(forks[left]); V(forks[right]);
   }
}
```

Is there anything wrong with this solution ?

Compare with this solution:

```
process Philosopher ((int id=0; id < N; id++)) {
   int right = id, left = (id+1) == N ? 0 : id+1;

   if (id == 0) {
      int temp=left; left=right; right=temp;
   }

   while (true) {
      System.out.println("Philisopher " +id+ " is thinking");
      P(forks[left]); P(forks[right]);
      System.out.println("Philisopher " +id+ " is eating");
      V(forks[left]); V(forks[right]);
   }
}
```

Another solution possible: limit the number of philosophers who are allowed to eat at a given time.

Points to remember:

- For any solution to work, there must be the same number of P() operations as V() operations
- It is the implementation of the semaphores that will guarantee that a philosopher will not starve (i.e. the semaphores must be fair)

Below is an example implementation of the dining philosopher's problem in Java:

```
import java.util.concurrent.*;

// A class to generate the number of fork semaphores for the dining philosphers
public class Forks {
   Semaphore[] forks;

   // create a fork for each philosopher
   public Forks(int n) {
      forks = new Semaphore[n];
      for (int i=0; i < n; i++) forks[i]=new Semaphore(1, true);
   }

   // called by a philosopher to get a particular fork
   public void getfork(int f) {
      try {
         forks[f].acquire();
      }
      catch (InterruptedException e) {
         System.err.println("in getfork, f=" + f + e.getMessage());
         e.printStackTrace();
      }
   }

   // called by philosophers when they are done eating
   public void relfork(int f) {
      forks[f].release();
   }
}
```

```
// a class to implement philosophers in the famous dining philosophers problem
public class Philosopher implements Runnable {

   static final int EATING=10;     // time for eating
   static final int THINKING=10;   // time for sleeping
   static final int NUMPHIL=5;     // standard number of philosophers
   static final int ITERATIONS=15; // number of time a philosopher eats/thinks

   int id,                  // id of this philosopher
       right,               // his(hers) right fork id
       left,                // left fork id
       turns;               // and the number of times to loop
   Forks forks;             // the semaphores representing the forks

   // a Philosopher constructor
   public Philosopher(int id, int max, int turns, Forks forks) {
      right=this.id=id;
      left = (id+1) == max ? 0 : id +1;
      if (id == 0) { int temp=left; left=right; right=temp; }
      this.turns=turns;
      this.forks=forks;
   }

   // the only useful things a philosopher does ?
   public void run() {
      try {
         for (int i=1; i<= turns; i++) {
            forks.getfork(left); forks.getfork(right);
            System.out.println("Philisopher " +id+" is eating");
            Thread.sleep(0, Math.round(Math.random() * EATING));
            forks.relfork(left); forks.relfork(right);
            System.out.println("Philisopher " +id+" is thinking");
            Thread.sleep(0, Math.round(Math.random() * THINKING));
         }
      }
      catch (InterruptedException e) {
         System.err.printf("Oops !  Philosopher %d bit the dust\n", id);
         System.err.println(e.getMessage());
         e.printStackTrace();
      }
      finally {
         System.out.printf("Philosopher %d is leaving now\n", id);
      }
   }

   // a main that ignores arguments for now
   public static void main(String [] args) {

      System.out.println("Simulation start");
      Philosopher[] p = new Philosopher[NUMPHIL];  // our table of philosophers
      Forks forks = new Forks(NUMPHIL);            // one fork per philosopher...
      Thread []t = new Thread[NUMPHIL];            // and each is a thread of execution

      for (int i=0; i < NUMPHIL; i++) {
         p[i]=new Philosopher(i, NUMPHIL, ITERATIONS, forks);
         t[i]=new Thread(p[i]);
         t[i].start();
      }

      try {
         for (int i=0; i < NUMPHIL; i++)
            t[i].join();
      }
      catch(InterruptedException e) {
         System.err.printf("Got an exception waiting for philosophers, bailing\n");
         e.printStackTrace();
      }

      System.out.println("Simulation finished.");
   }
}
```