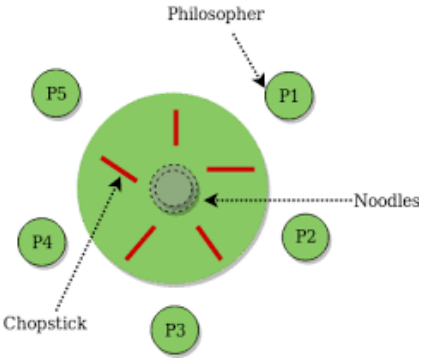
Start code for the first exercise (given day1): https://github.com/cph-dat-sem2/thread-samples.git

Friday Week1 (Concurrency, Threads, and Network intro)

The first and most important part of what you have to do here, is to complete all the daily exercises given throughout this week.

Deadlock - The Dining Philosophers Problem



The first part of this exercise should be relatively straightforward for all.

The code in the package `deadlock.dining philosophers` is a simulation of the famous Dining Philosophers problem. Five philosophers do nothing in life but eat and think. To eat, requires **two** chopsticks. Since any two adjacent philosophers must share a chopstick, this is a protected resource that must be used by only one philosopher at any given time.

1. Execute the code and see if "it fails". The code may need to run some time to provoke the problem. This is cool, since it visualizes how hard it often is to test concurrent software.

2. Add the necessary code to automatically detect if there is a deadlock. If this is the case, print it to the console and close the program (System.exit (1)).
   Hint: Use the DeadlockDetector class, running in a separate thread, for this part.

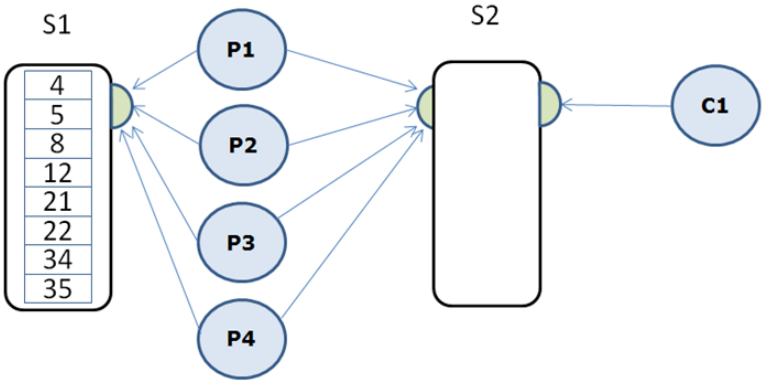
Producer - Consumer for Red students

For this exercise we will use the previously used Fibonacci algorithm to calculate a Fibonacci number.

```
int fib(int n) {
     if (n < 2) return n;
     return fib(n - 1) + fib(n - 2);
}
```

What a Fibonacci number is, is not interesting for this task, except, as we have seen, that the execution time for the given algorithm grows exponentially when the number to calculate is increased.

Implement the following design:



1. A thread (in this case the main thread) must initially fill a shared data structure S1 with numbers from which the corresponding Fibonacci numbers are desired to be calculated.

2. The main thread then starts four Producer threads (P1 - P4) which will all access the shared data structure S1 to retrieve a number, from which it must calculate the corresponding Fibonacci number.
3. When a thread has finished its calculation, the result is placed in another shared data structure S2, and it should fetch a new number to calculate from S1 or stop (leaves its run method) if S1 does not contain any more numbers to be calculated.
4. Immediately after the main thread has started the four Producer threads, it starts a Consumer thread C1, which continuously retrieves the calculated numbers, prints them to the console and calculates the total sum of numbers retrieved (for whatever reason).
5. When all threads have completed, the sum of all the calculated Fibonacci numbers must be printed.

Hints:

1. Initialize S1 with the following numbers: 4,5,8,12,21,22,34,35,36,37,42
2. Use a BlockingQueue implementation (ArrayBlockingQueue) for both shared data structures, then this task will be relatively simple (BlockingQueue collections are all Thread Safe)
3. For S1 you can use any of the possible insert methods, but use **`poll ()`** to remove elements. If this call returns null you know that the list is empty (there are no more numbers to calculate).
4. 5. For S2 you should use **`put()`** and **`take()`** to insert/retrieve elements as they automatically make sure to **wait** / **wake** when inserted in a list without more space, or retrieved from a list without elements.