Deadlock Victim

by Dr Heinz Kabutz && Olivier Croisier
The Java Specialists Newsletter && The Coder's Breakfast
heinz@javaspecialists.eu && olivier.croisier@zenika.com
twitter: @heinzkabutz && twitter: @oliviercroisier

You discover a race condition

Data is being corrupted

What ya gonna do?

Who ya gonna call?

Ghostbusters?

you do what we all do

you use **synchronized**

synchronized

synchronized

synchronized can be bad medicine

deadlock awaits you!

Deadlock Victim

Correctness fix can lead to liveness fault

lock ordering deadlock

at least two locks

and at least two threads

locked in different orders

can cause a *deadly embrace*

requires global analysis to fix

resource deadlocks

bounded queues

bounded thread pools

semaphores

class loaders

lost signals

The Drinking Philosophers

@ the Java Specialist Symposium in Crete

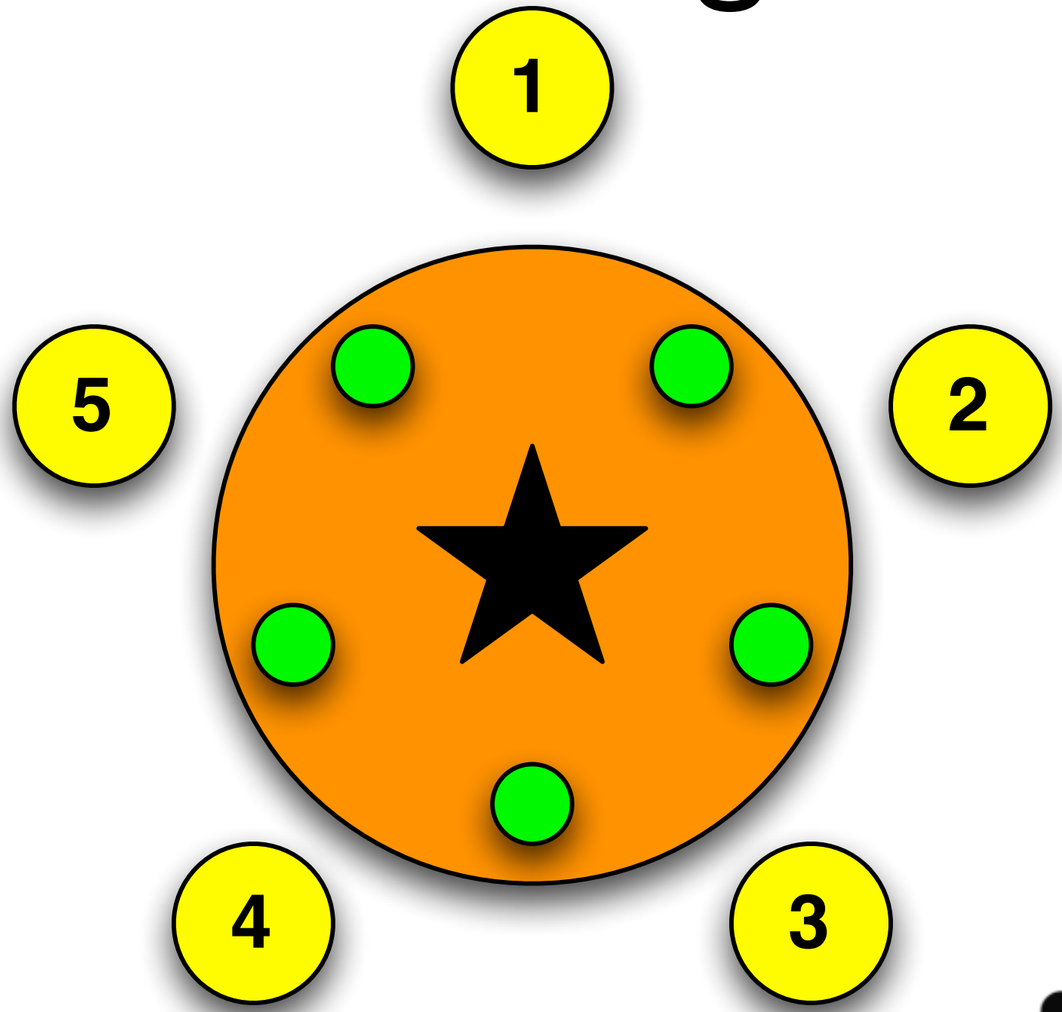
Symposium == sym + poto

Literally - to drink together

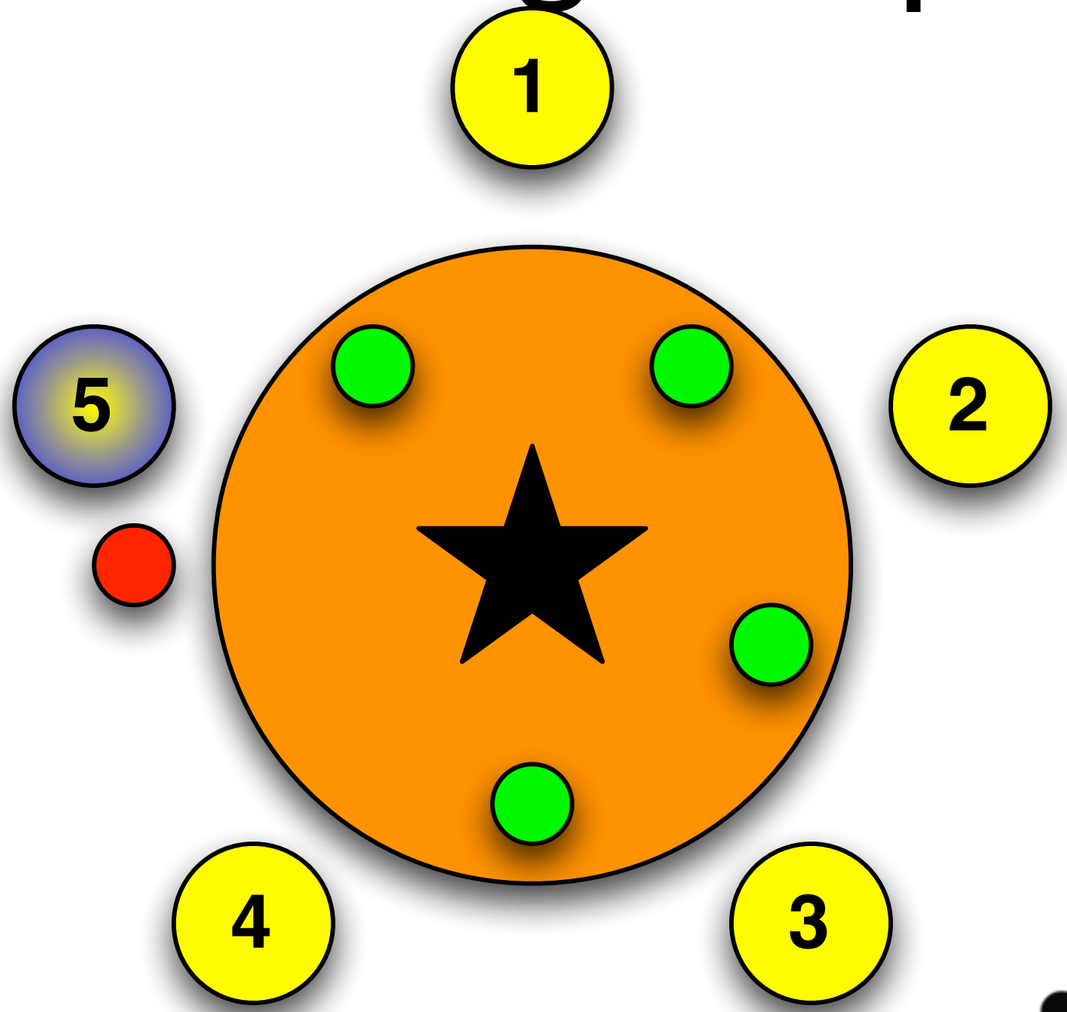
our philosophers need two cups

they either want to drink or think

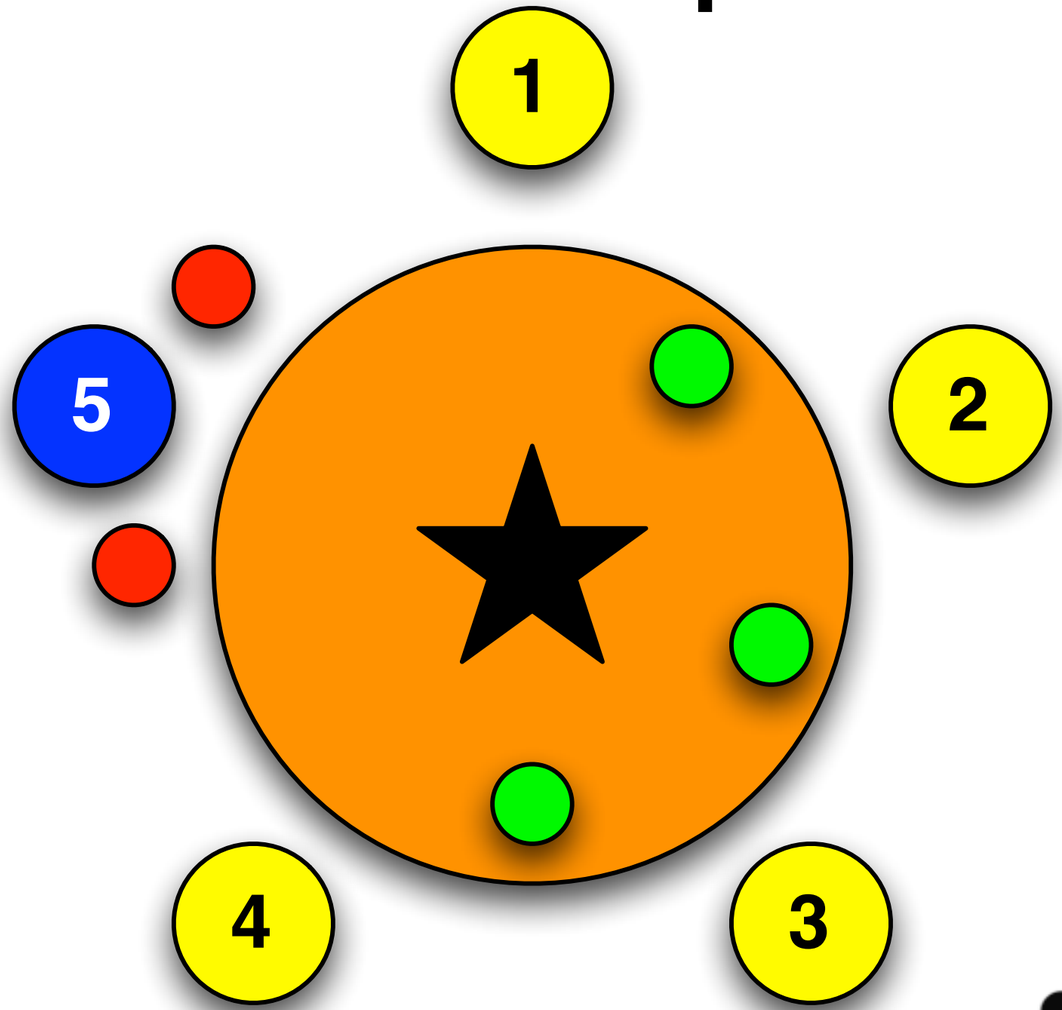
Table is ready, all philosophers are thinking



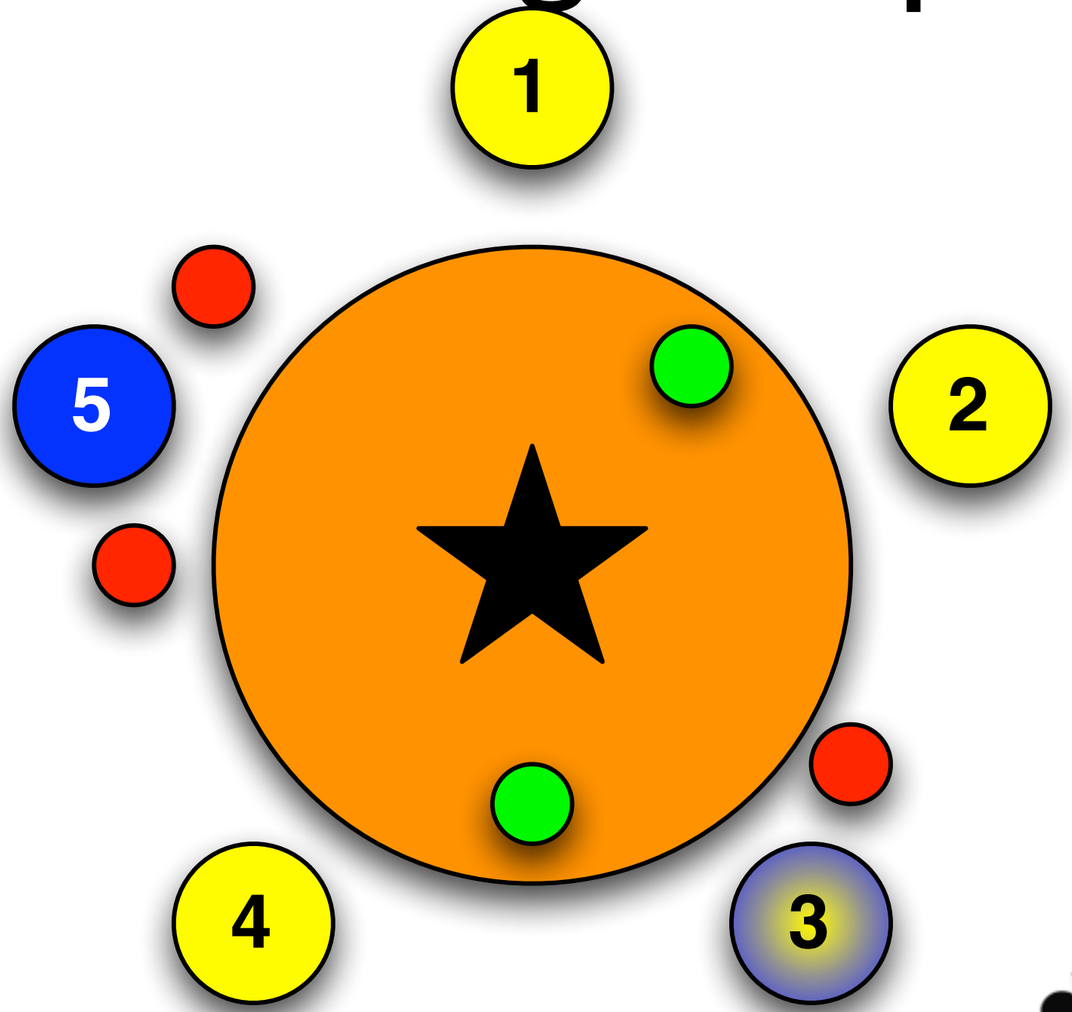
philosophers 5 wants to drink, takes right cup



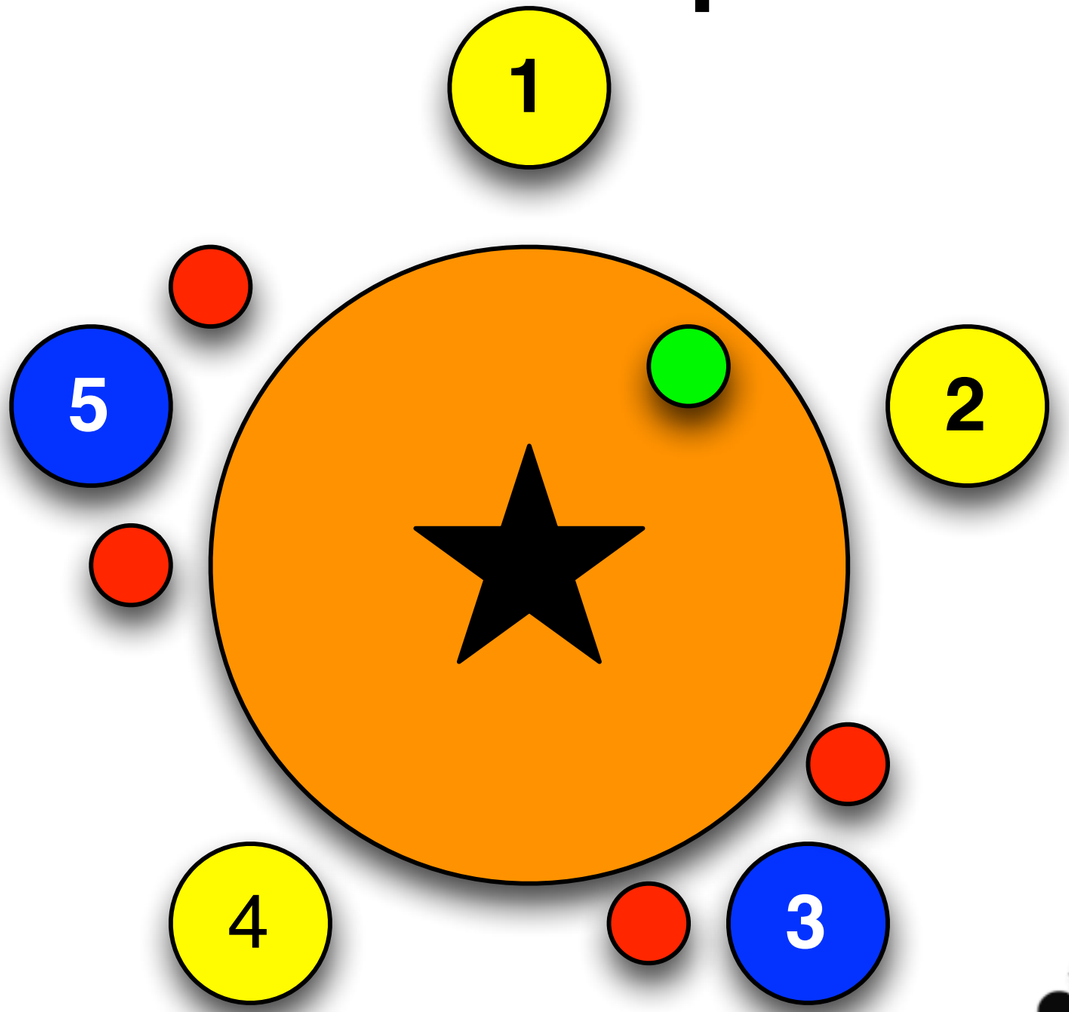
Philosopher 5 is now drinking with both cups



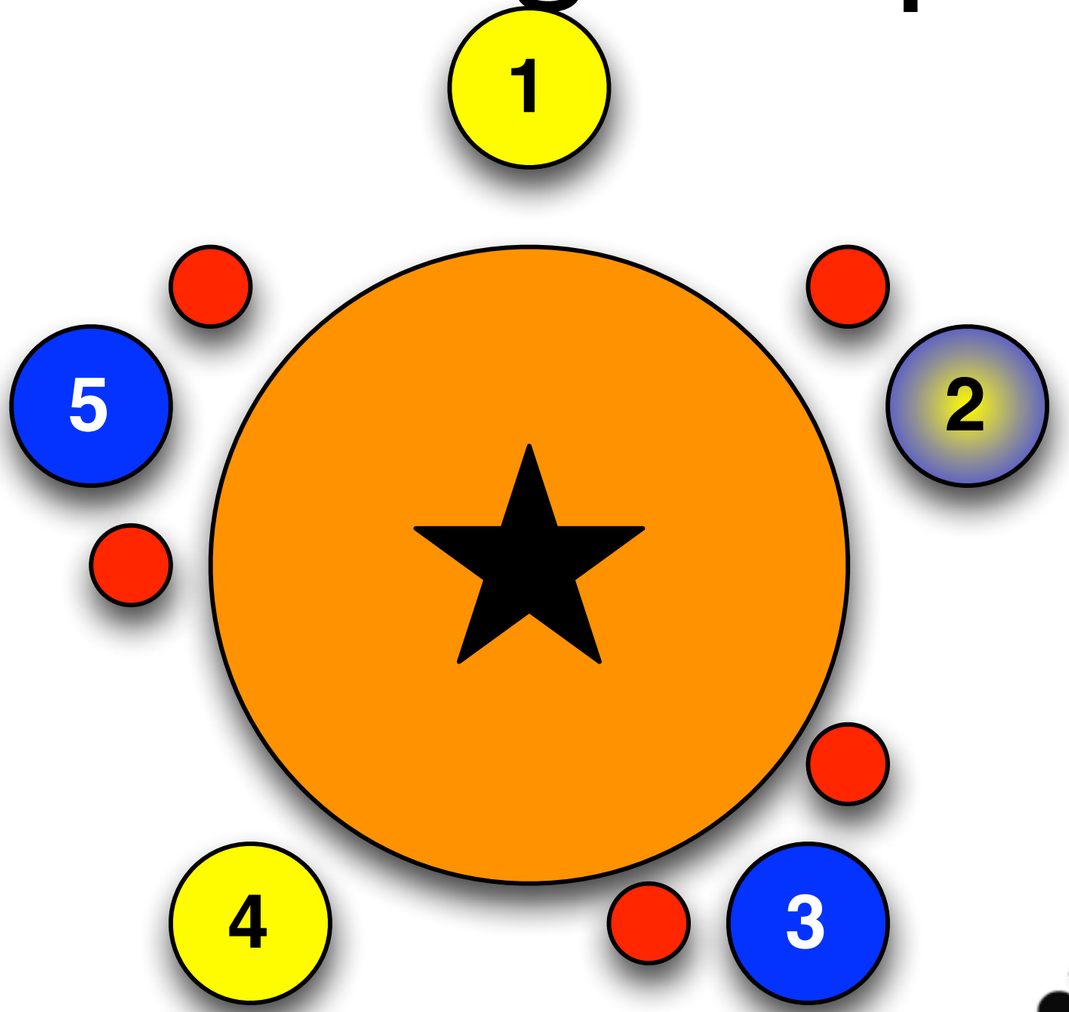
philosophers 3 wants to drink, takes right cup



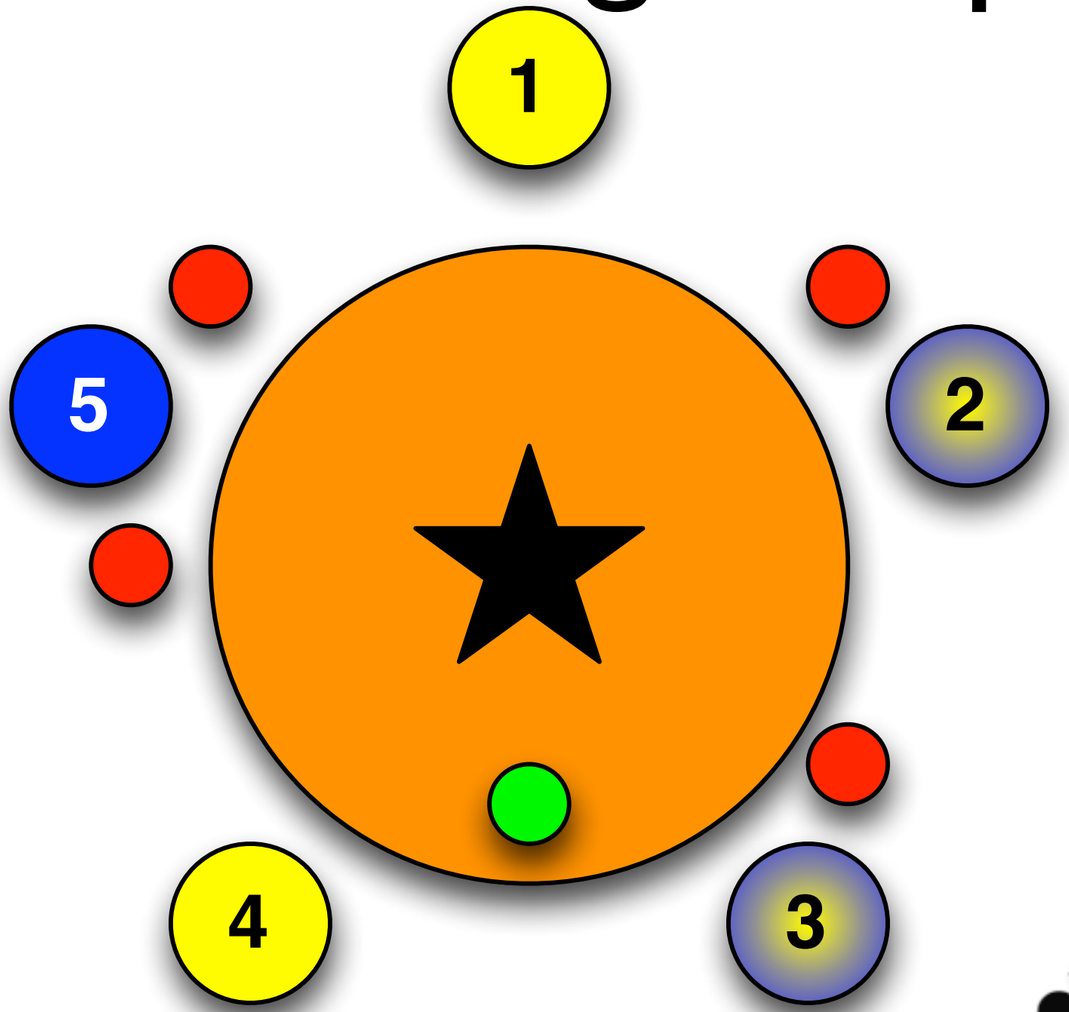
Philosopher 3 is now drinking with both cups



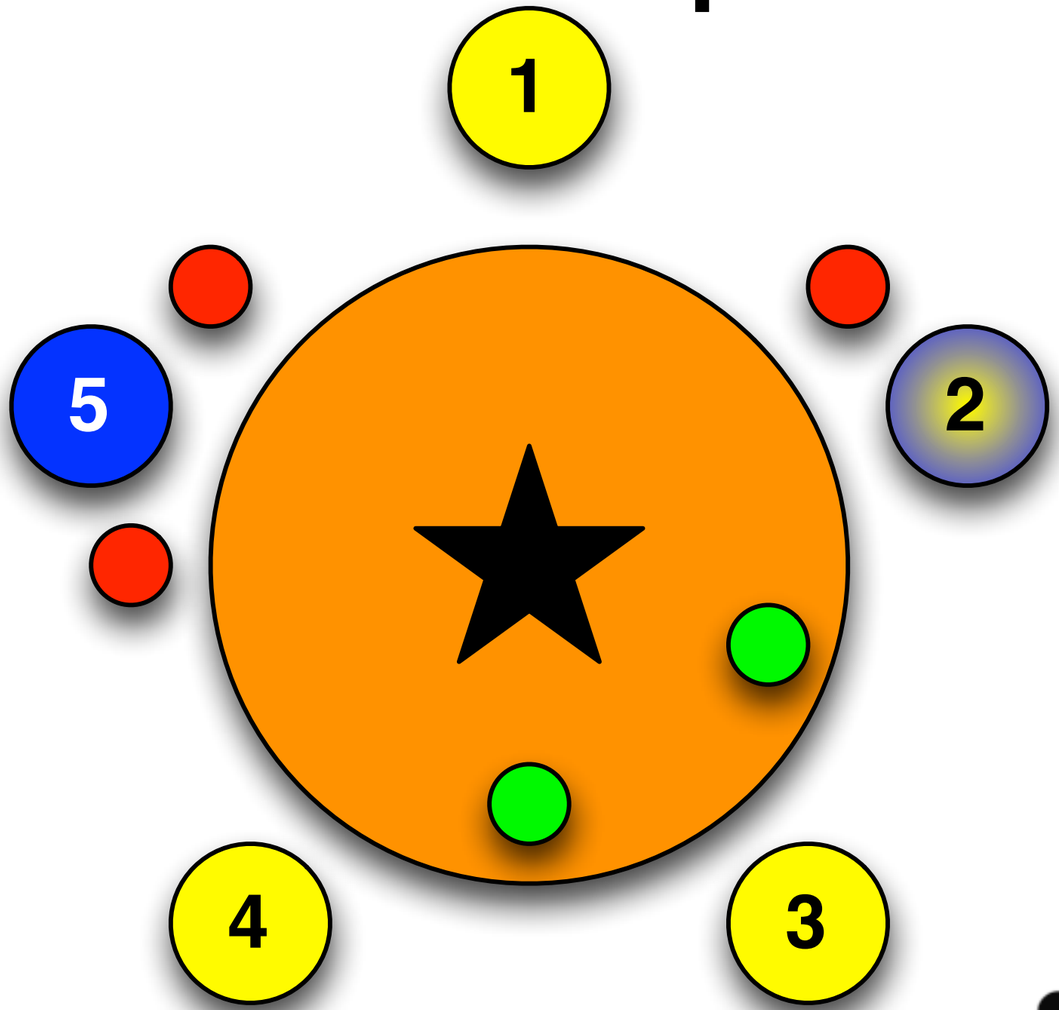
philosophers 2 wants to drink, takes right cup



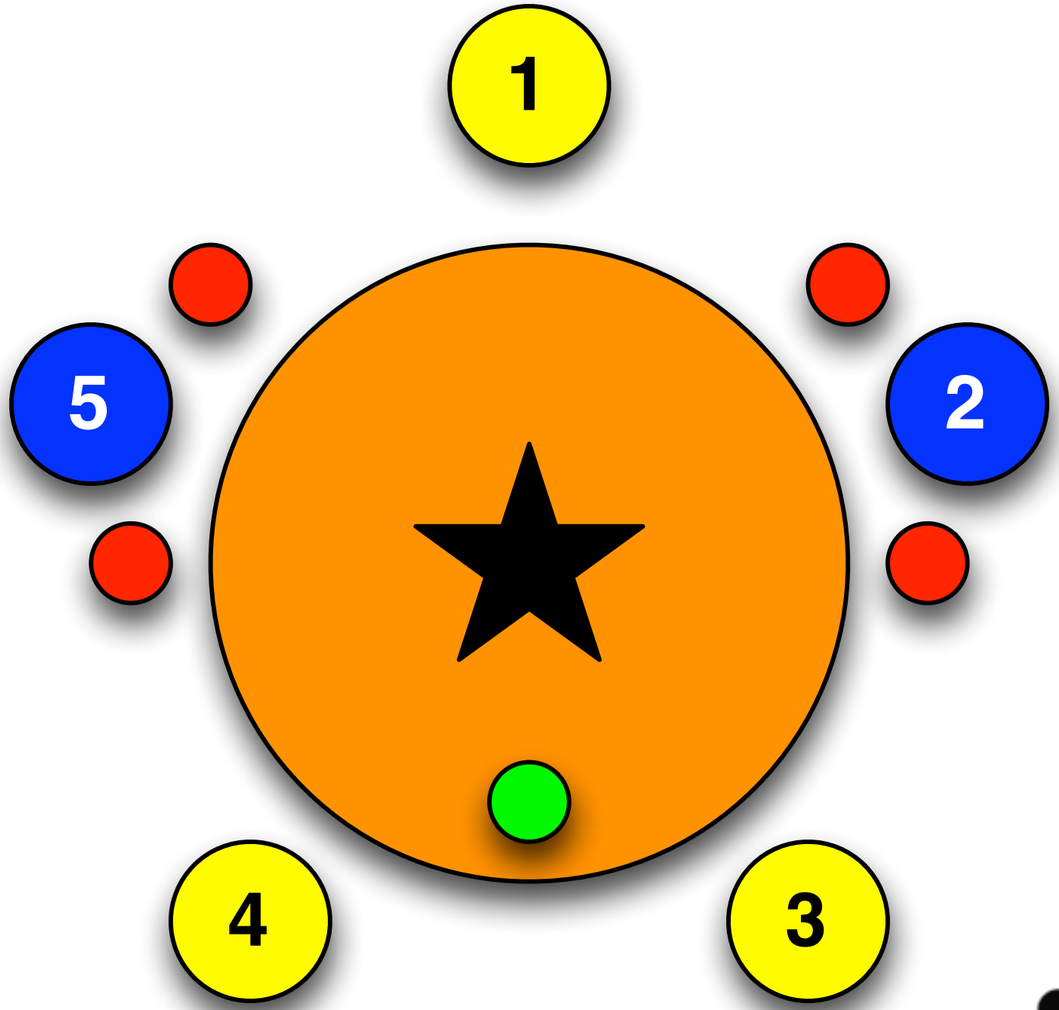
Philosopher 3 finished drinking, returns right cup



Philosopher 2 is now drinking with both cups



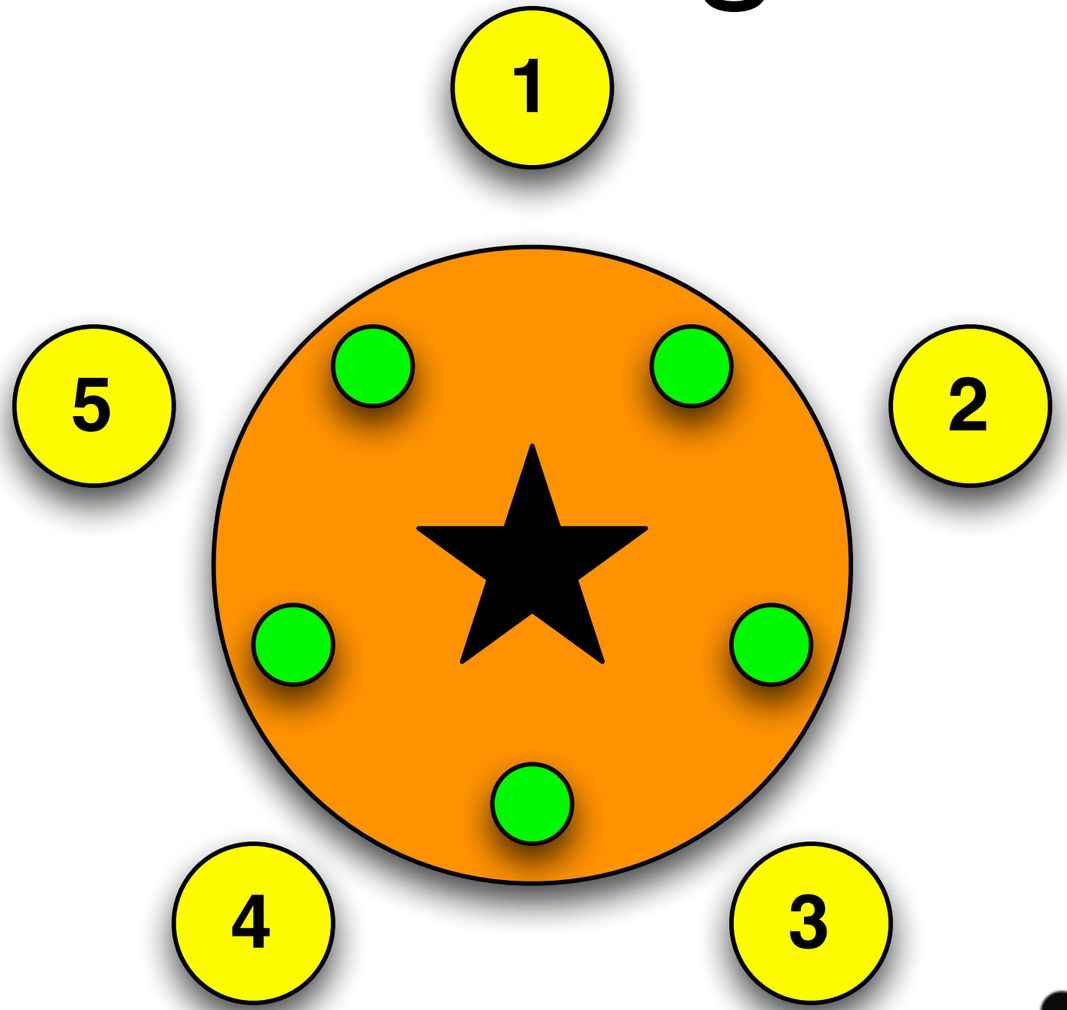
Philosopher 3 returns Left cup



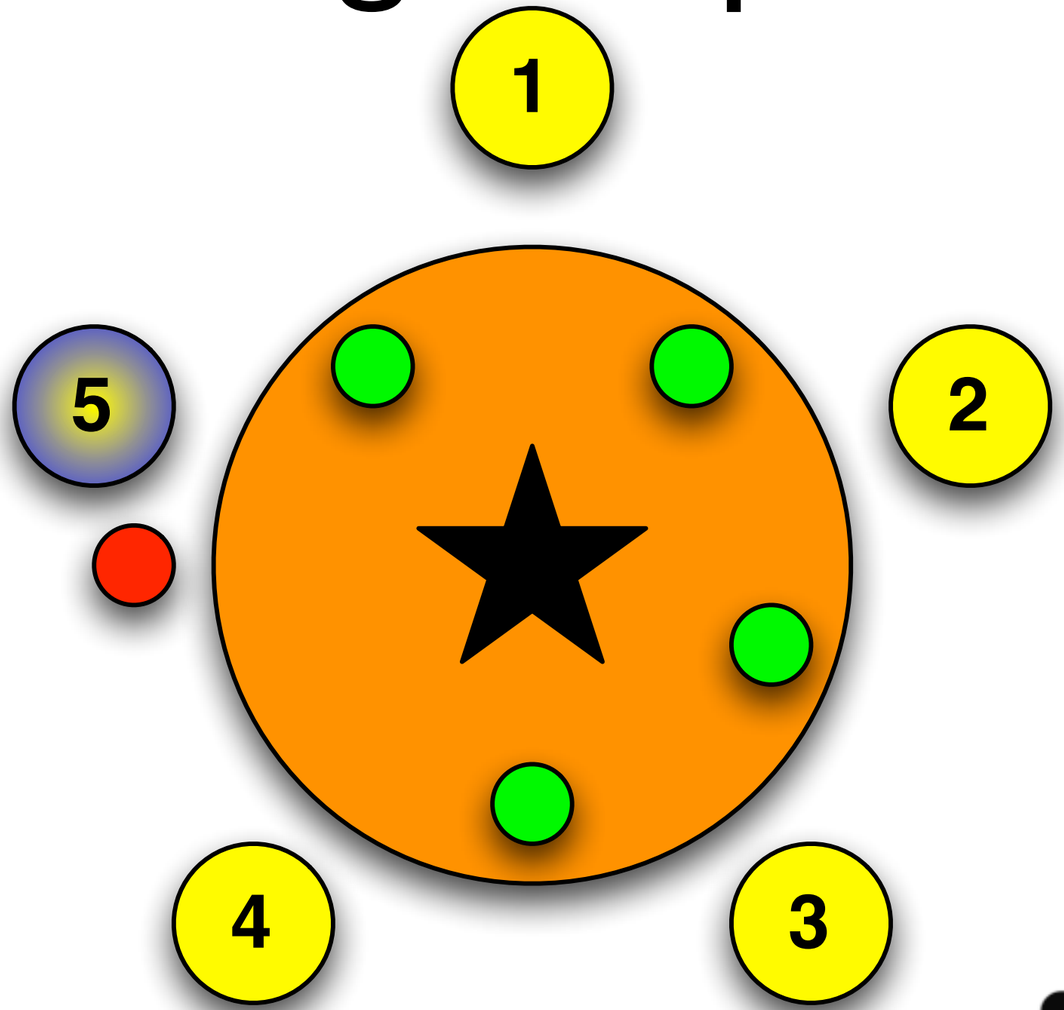
philosophers can get stuck

if they all pick up the right cup

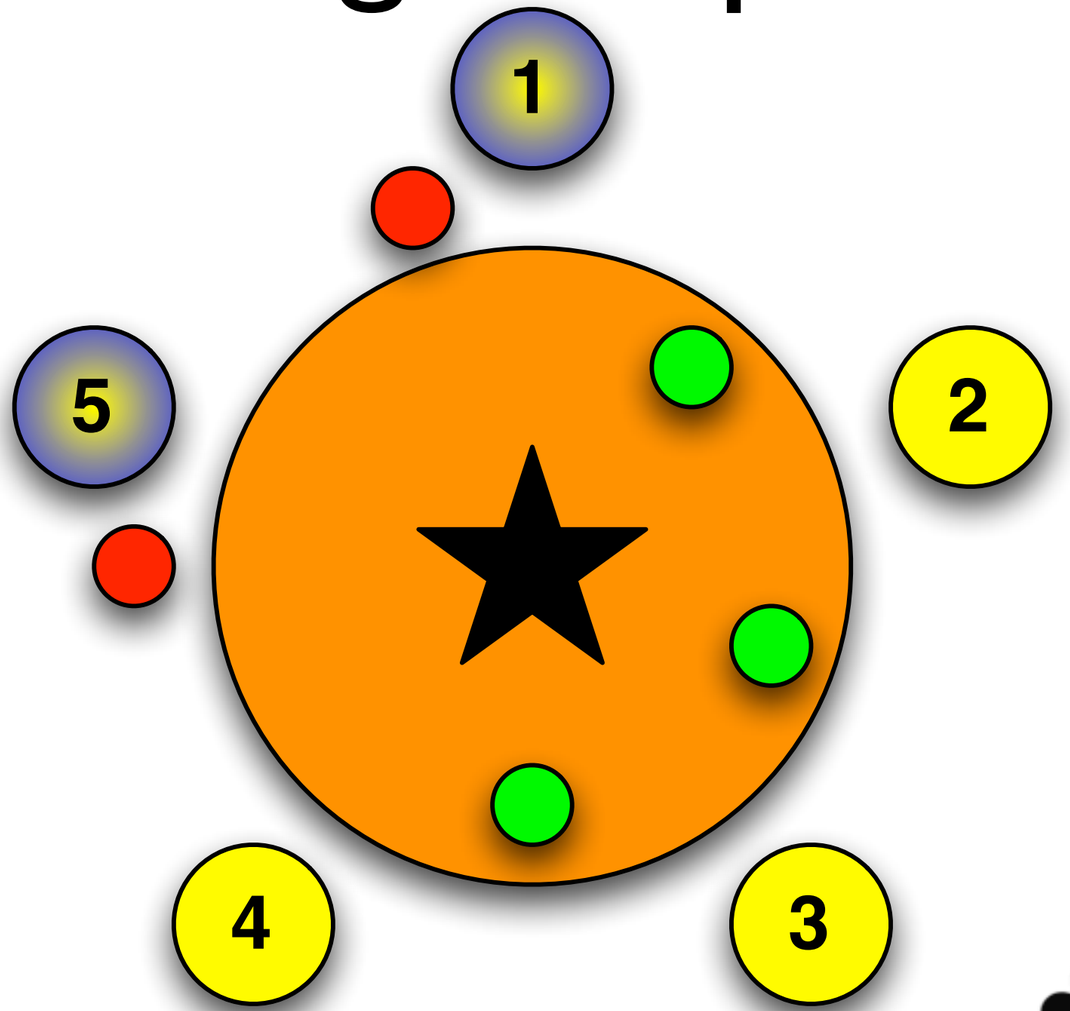
A deadlock can easily happen with this design



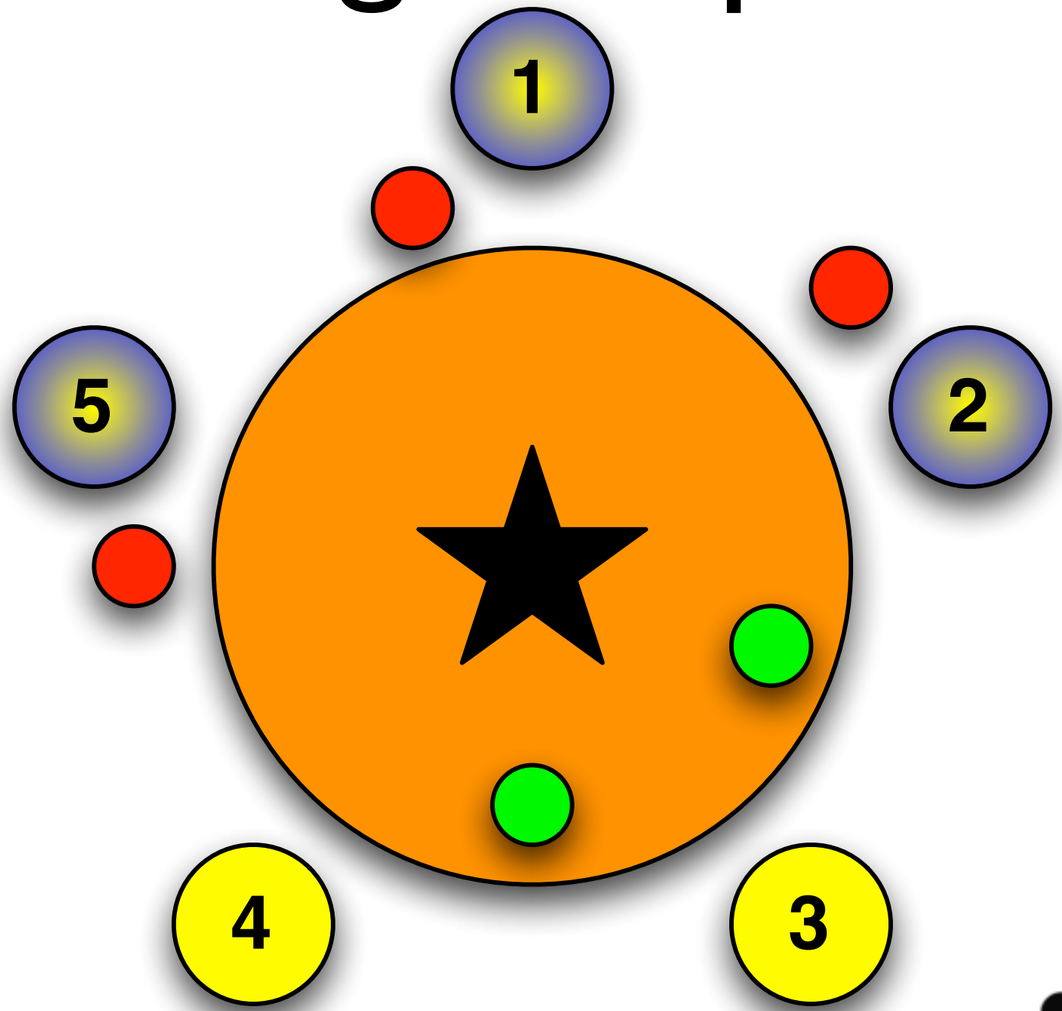
Philosopher 5 wants to drink, takes right cup



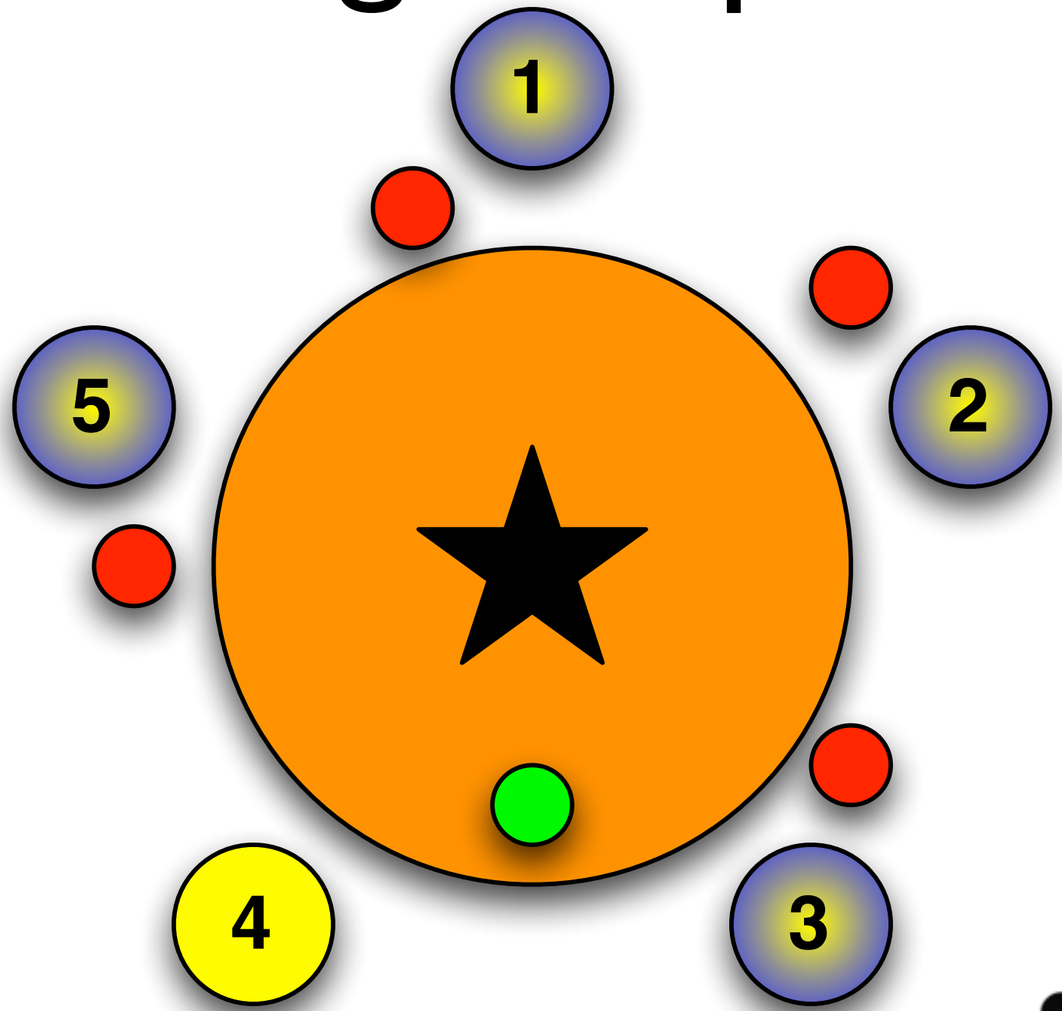
Philosopher 1 wants to drink, takes right cup



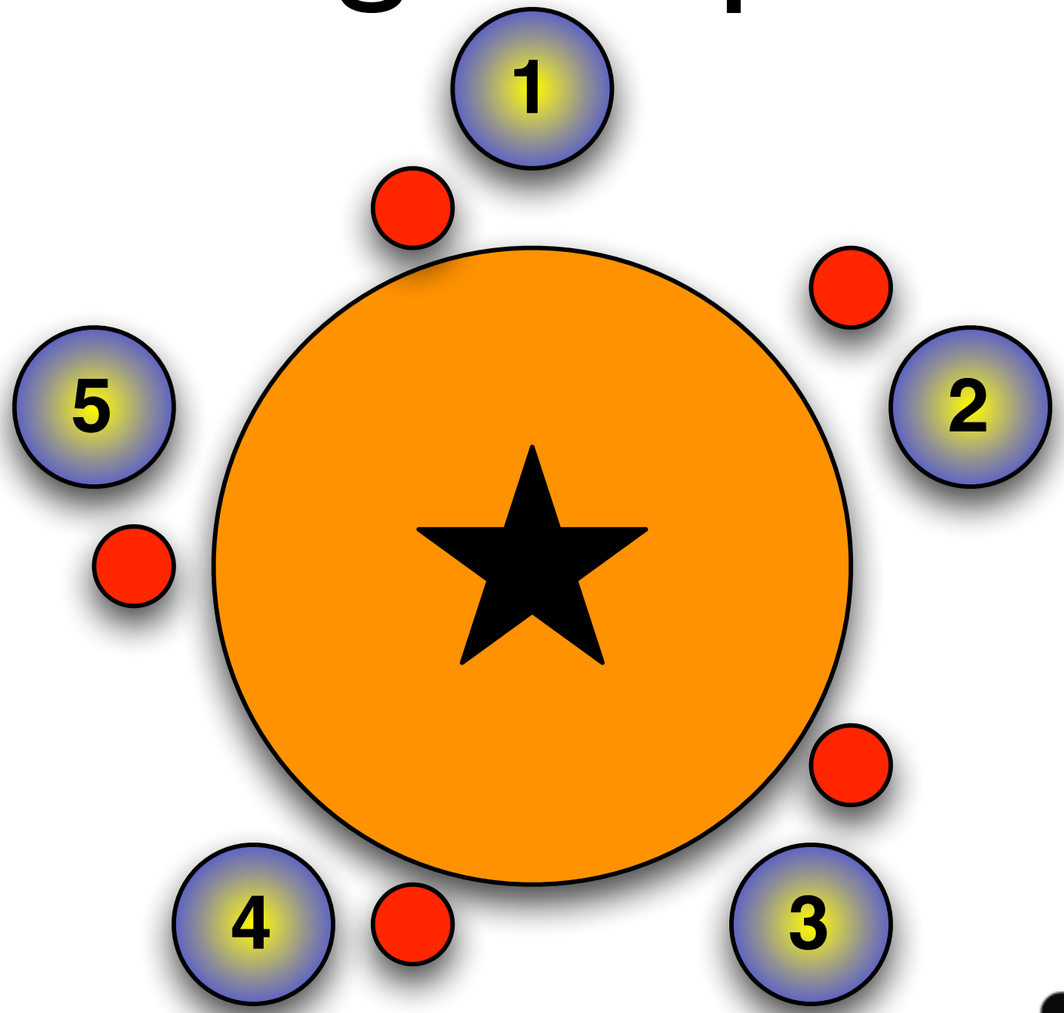
Philosopher 2 wants to drink, takes right cup



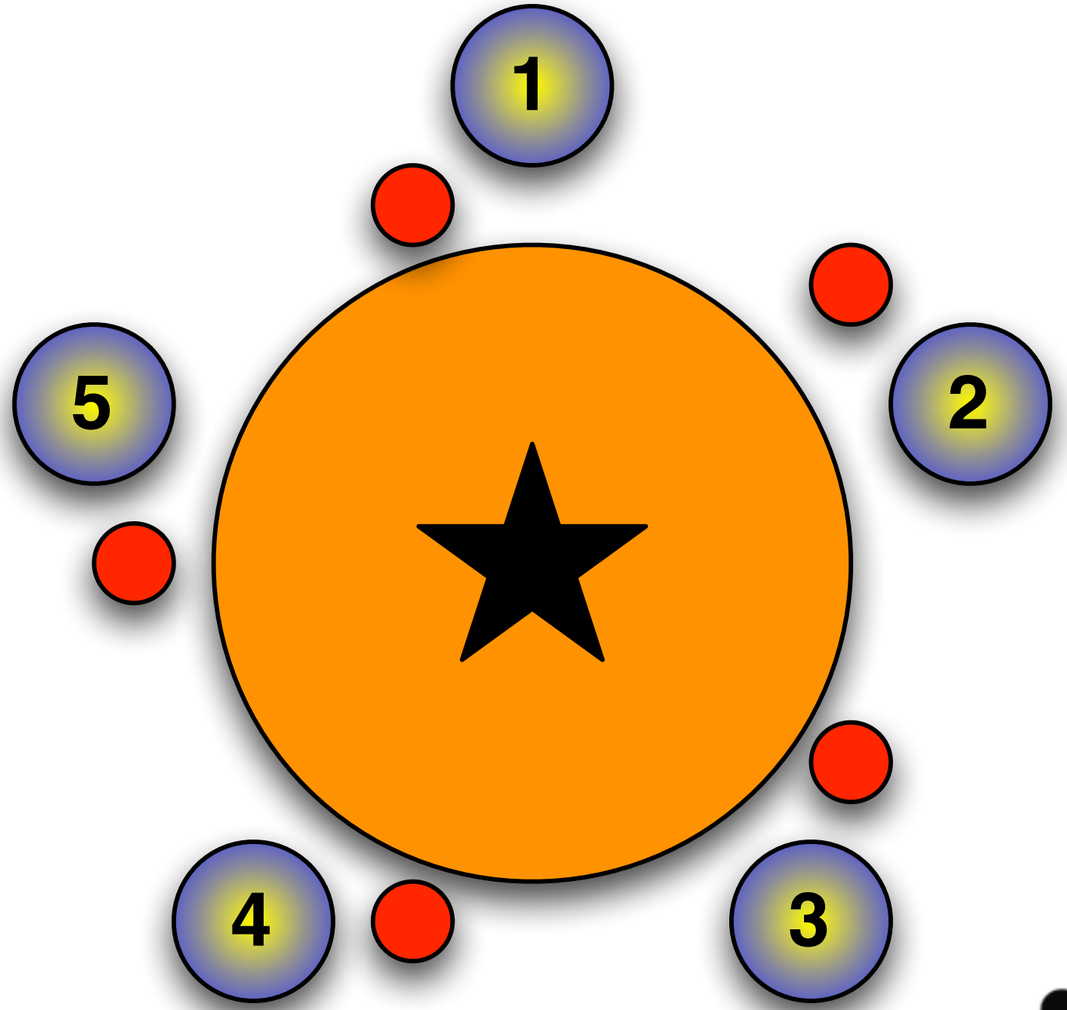
Philosopher 3 wants to drink, takes right cup



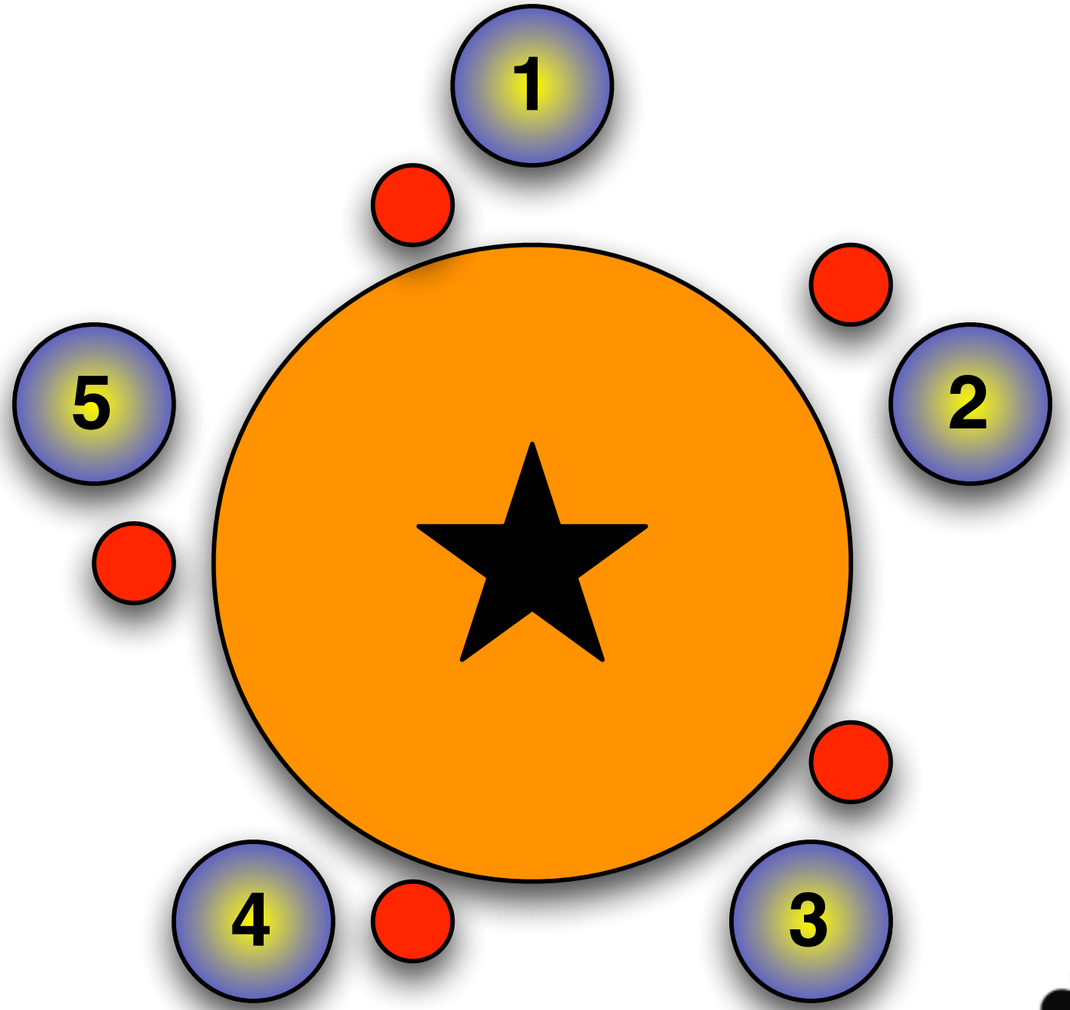
Philosopher 4 wants to drink, takes right cup



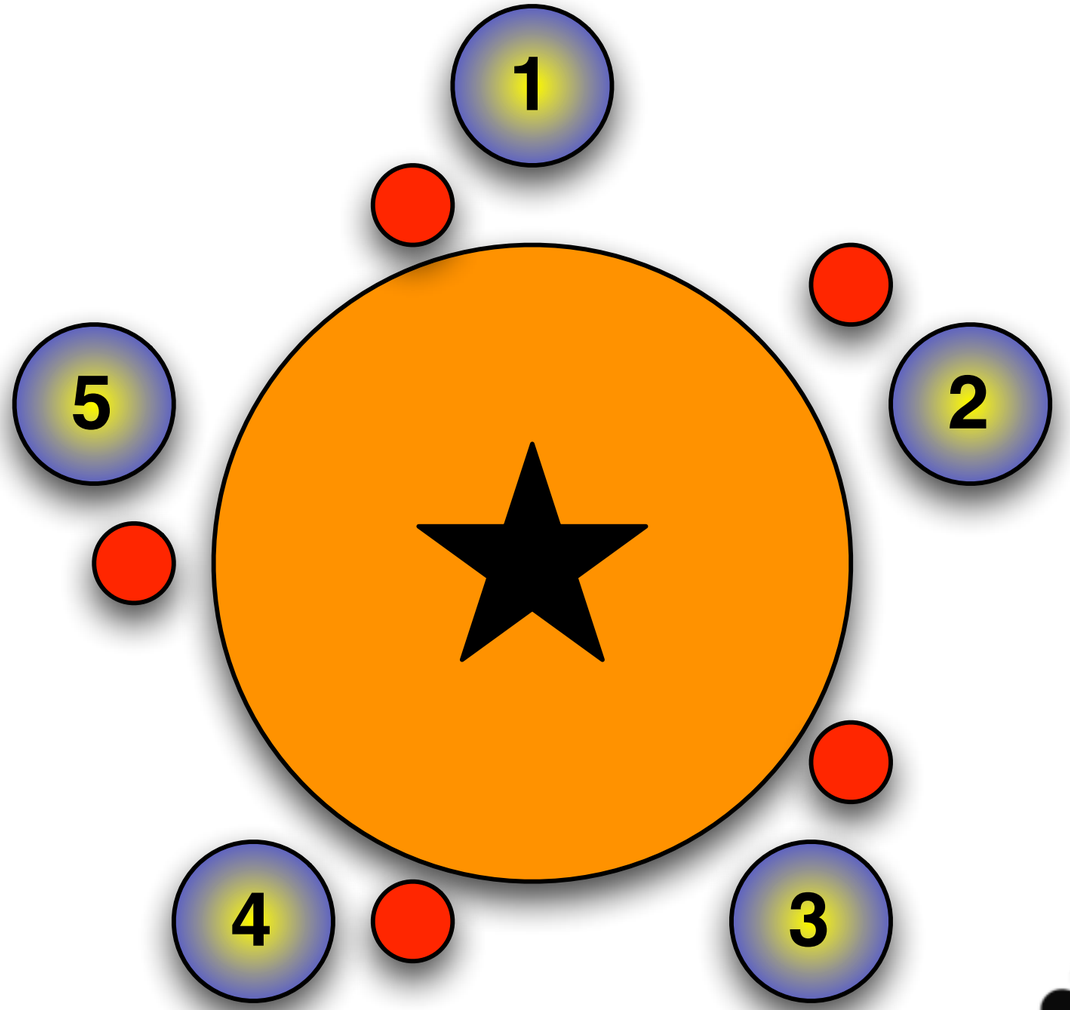
Deadlock!



they will all stay thirsty



this is a *deadly embrace*



this can be detected automatically

search the graph of call stacks

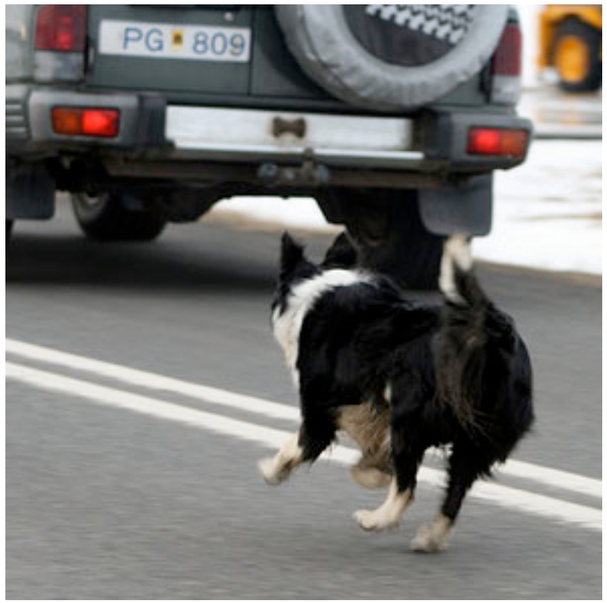
and look for circular dependencies

ThreadMXBean does that for us

simple deadlocks are quickly found

but what do you DO with them?

what will the dog do with the car if he ever catches it?



databases choose deadlock victim

but what does Java do?

it should probably throw an Error

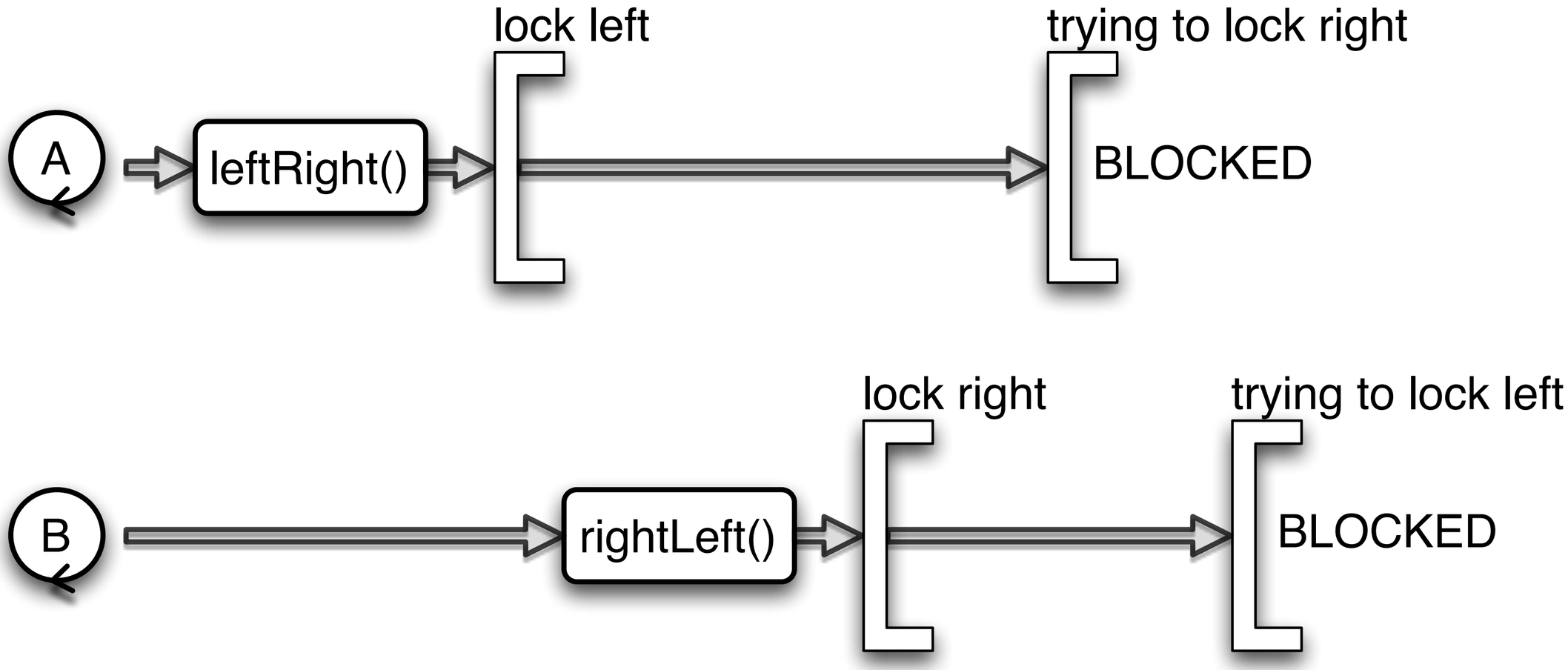
but the threads simply hang up

it is best to avoid causing them!

global order of locks in program

```
public class LeftRightDeadlock {
  private final Object left = new Object();
  private final Object right = new Object();
  public void leftRight() {
    synchronized (left) {
      synchronized (right) {
        doSomething();
      }
    }
  }
  public void rightLeft() {
    synchronized (right) {
      synchronized (left) {
        doSomethingElse();
      }
    }
  }
}
```

Interleaving causes deadlock



we define a fixed global order

e.g. left before right

deadlock is solved!

```
public void rightLeft() {
  synchronized (left) {
    synchronized (right) {
      doSomethingElse();
    }
  }
}
```

our drinking philosophers

each cup gets a unique number

lock the highest number cup first

then lock the lower number cup

drink

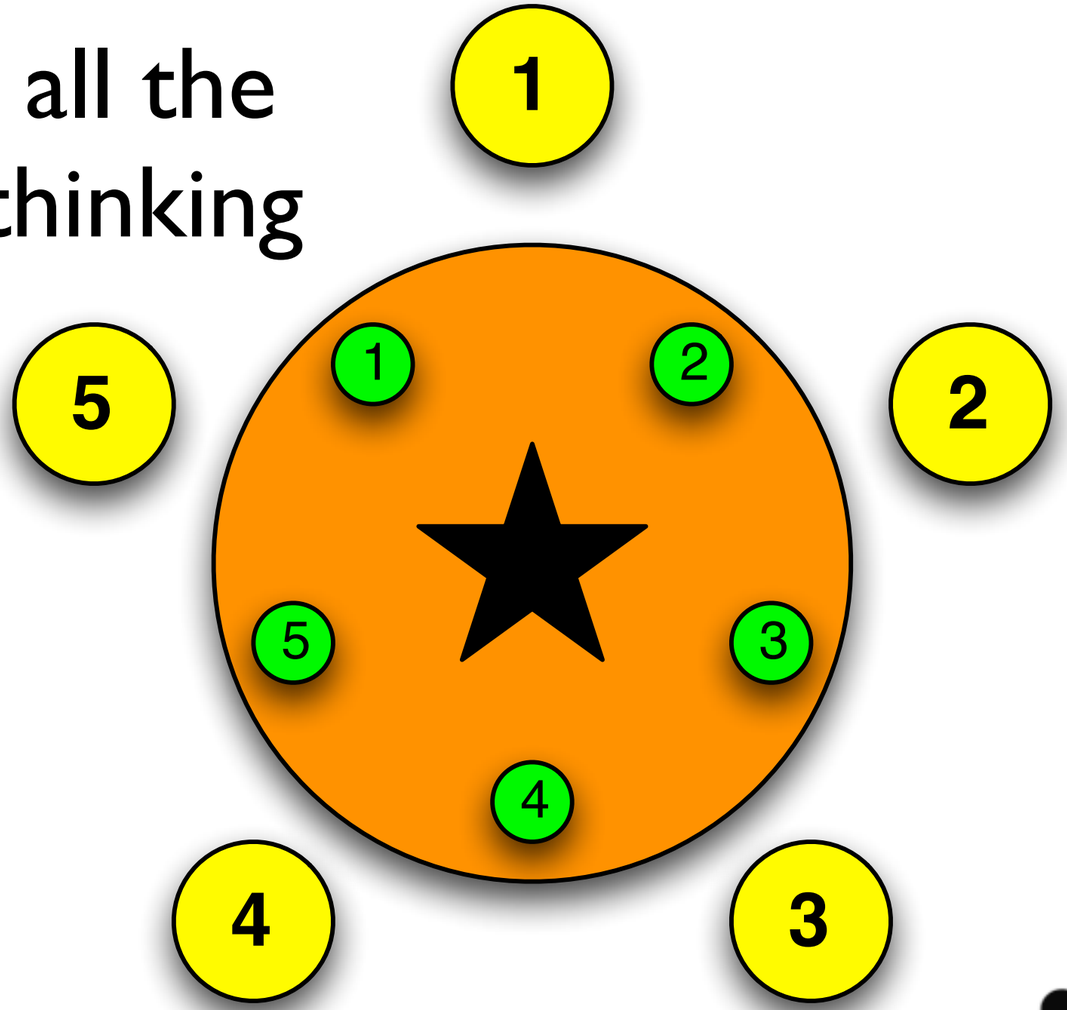
unlock lower order cup

unlock higher order cup

think

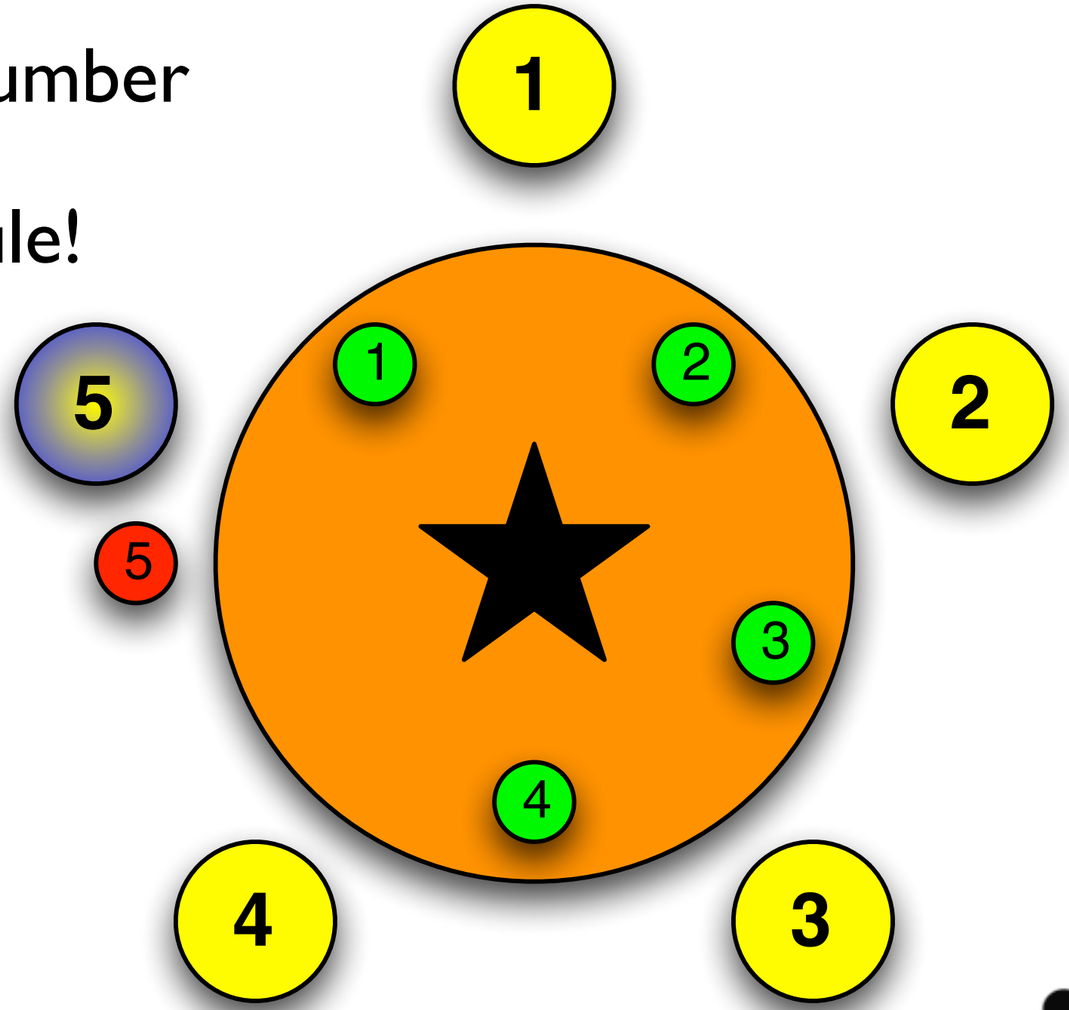
Global Lock ordering

- We start with all the philosophers thinking



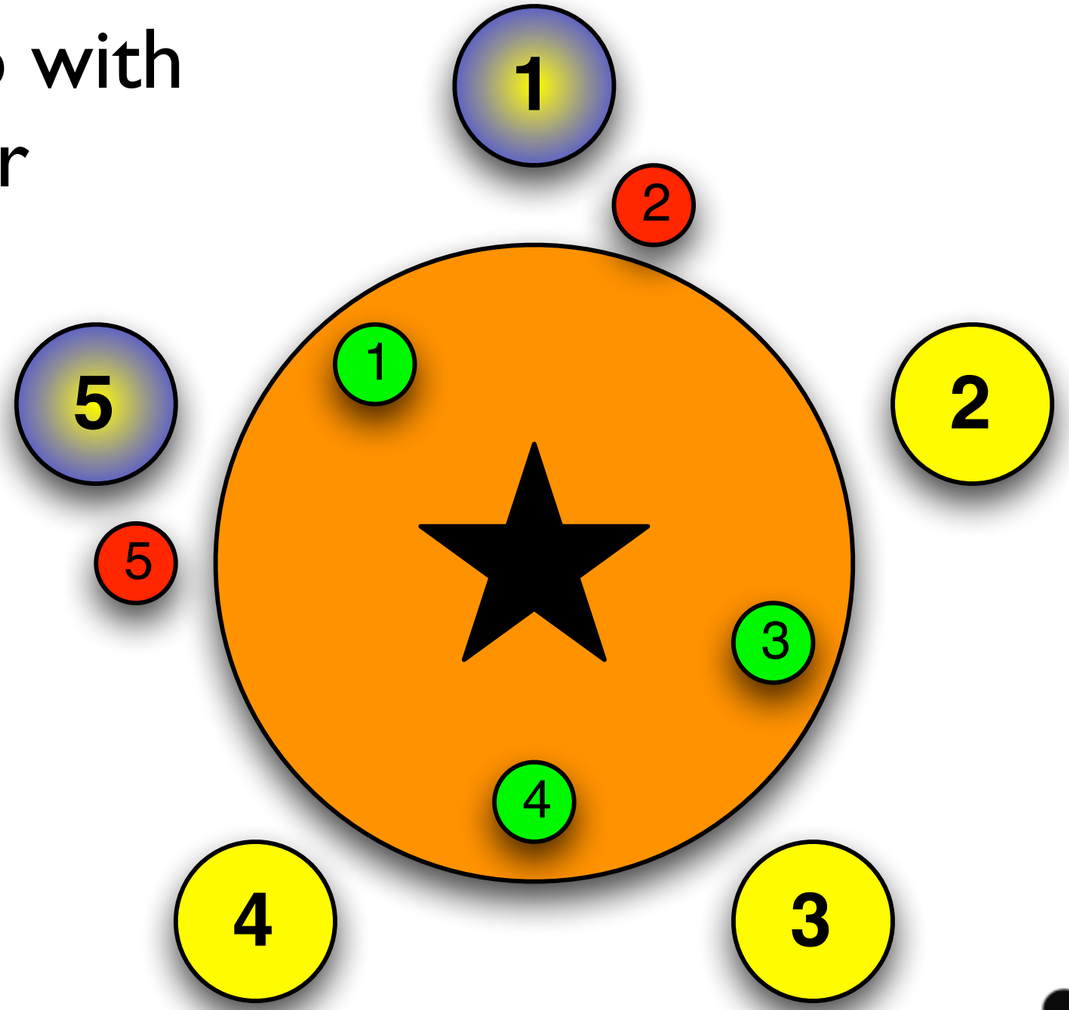
Philosopher 5 takes cup 5

- Cup 5 has higher number
  - Remember our rule!

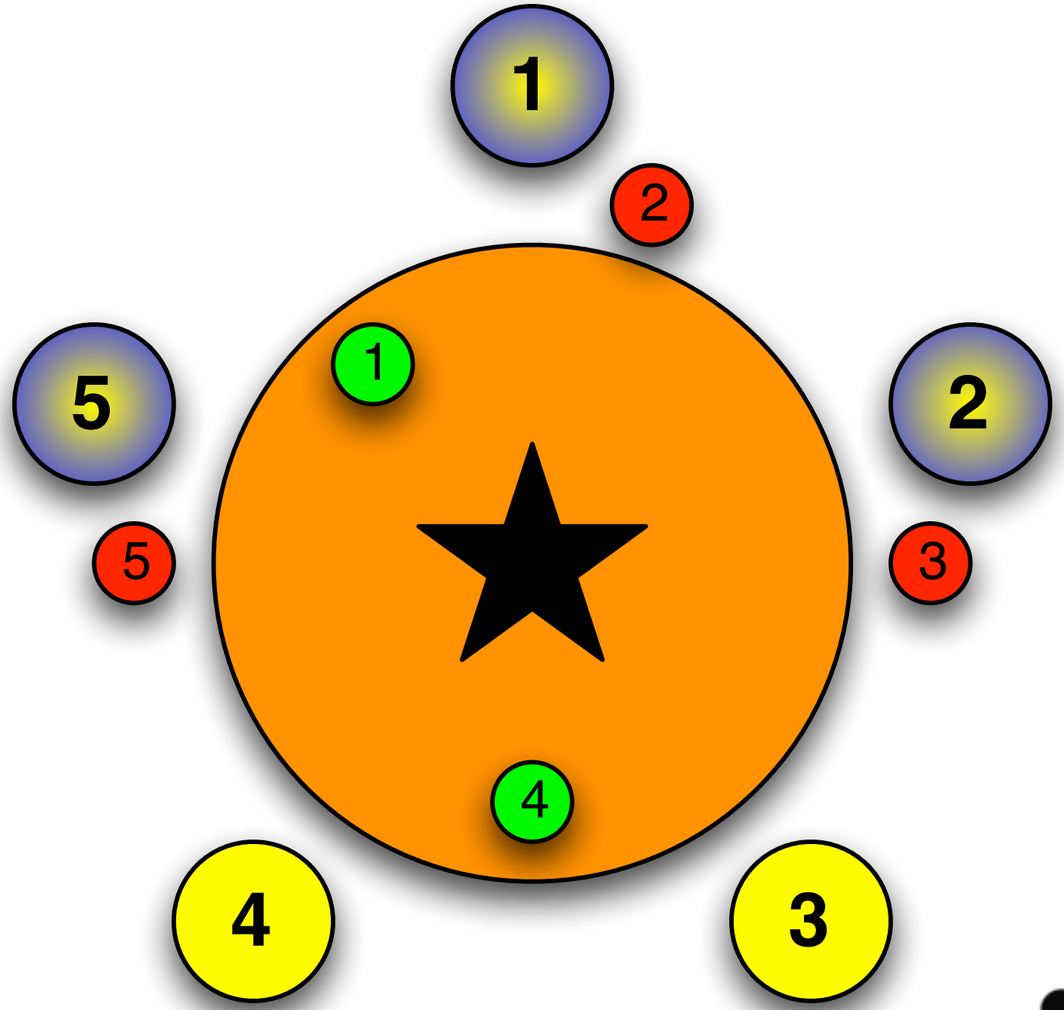


Philosopher 1 takes cup 2

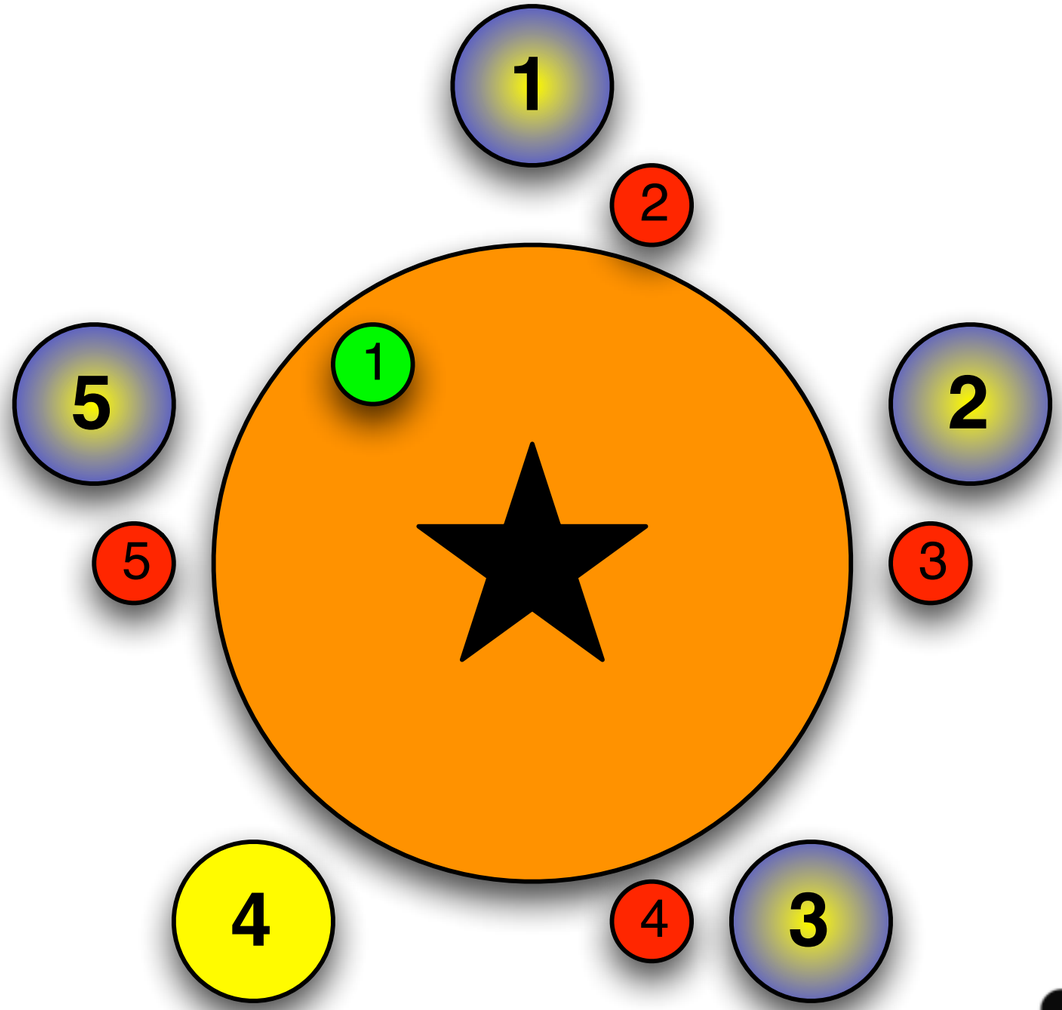
- Must take the cup with the higher number first
  - In this case cup 2



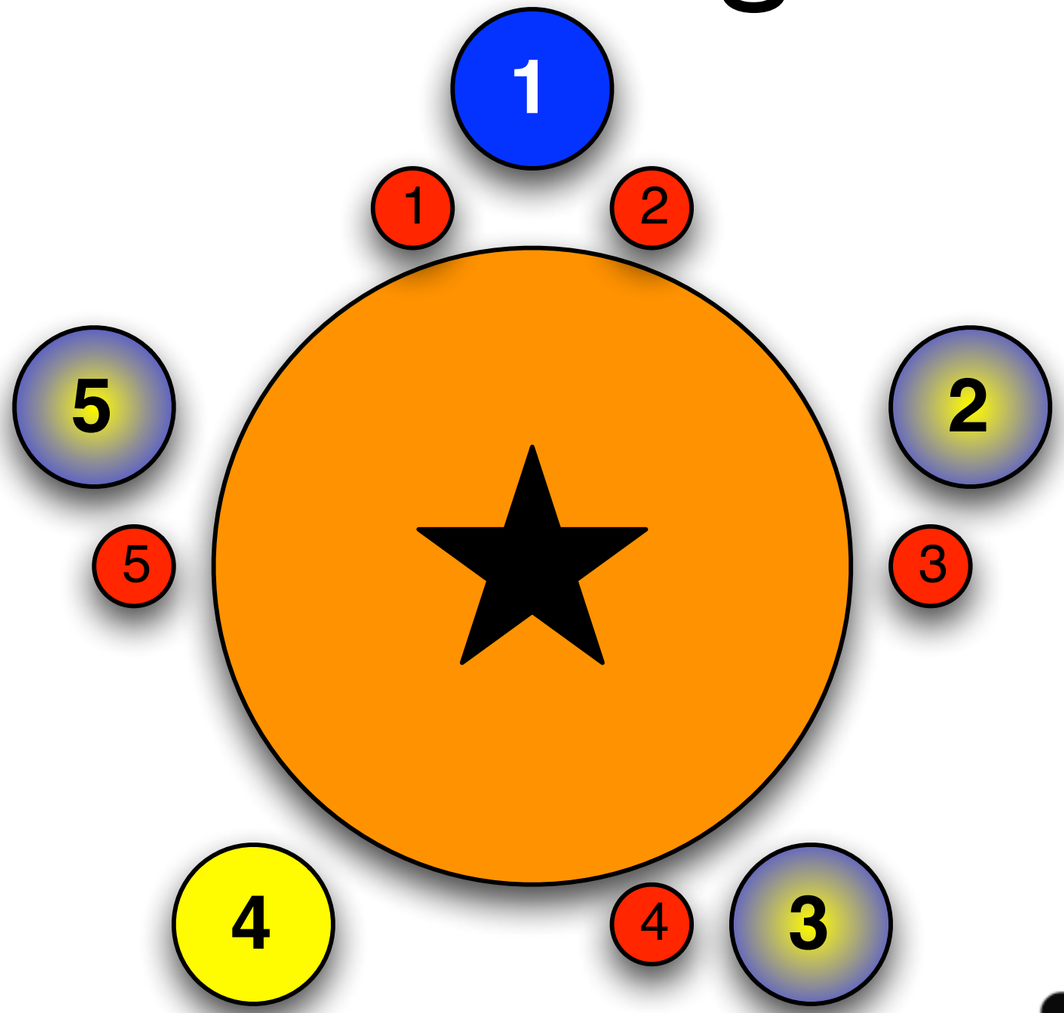
Philosopher 2 takes cup 3



Philosopher 3 takes cup 4

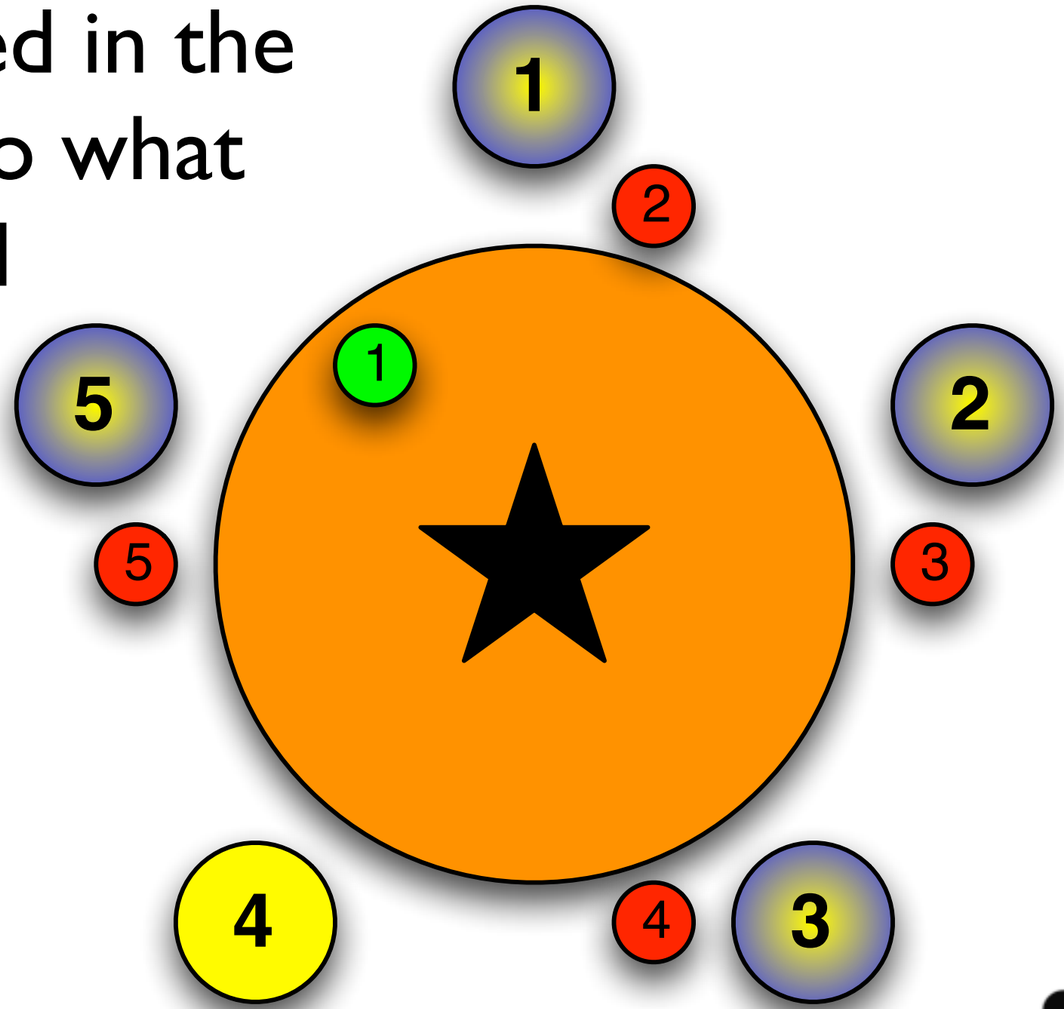


Philosopher 1 takes cup 1 - Drinking

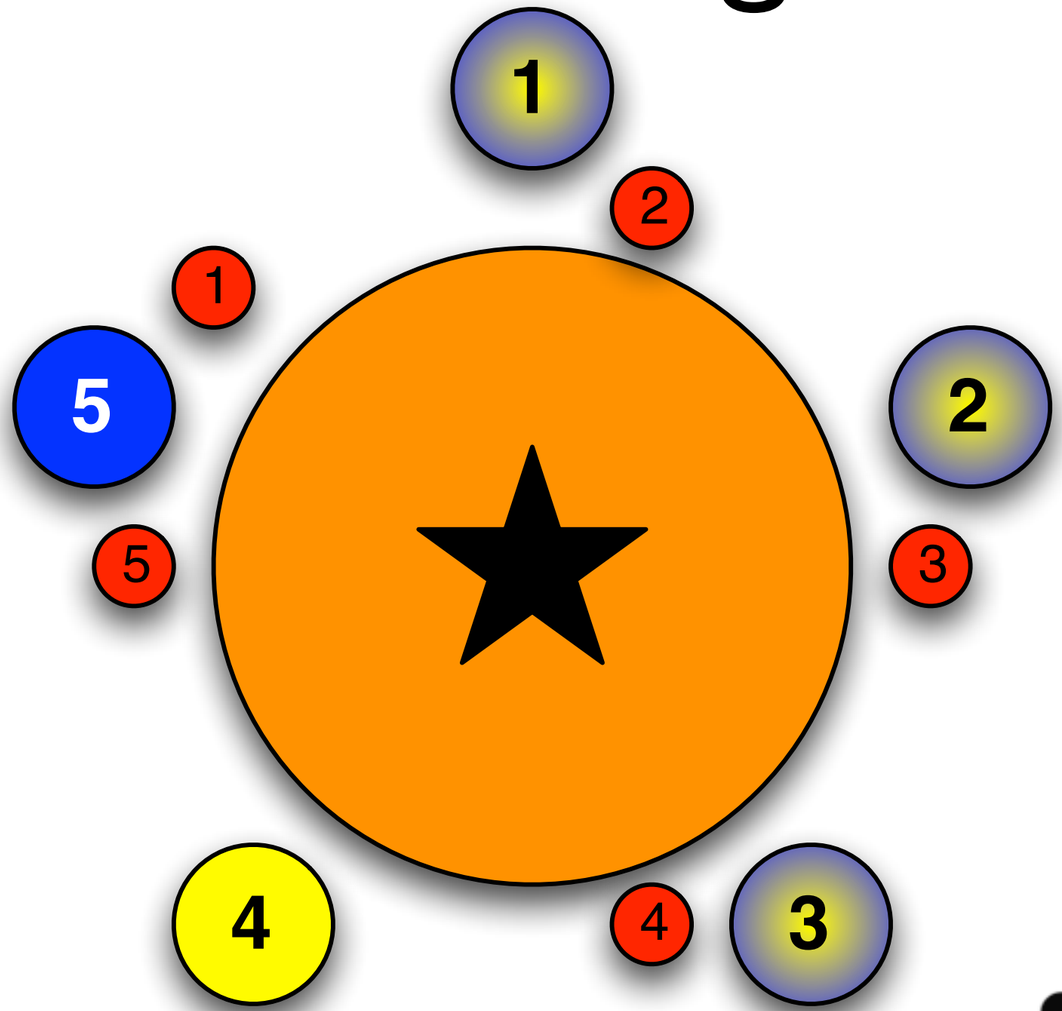


Philosopher 1 returns Cup 1

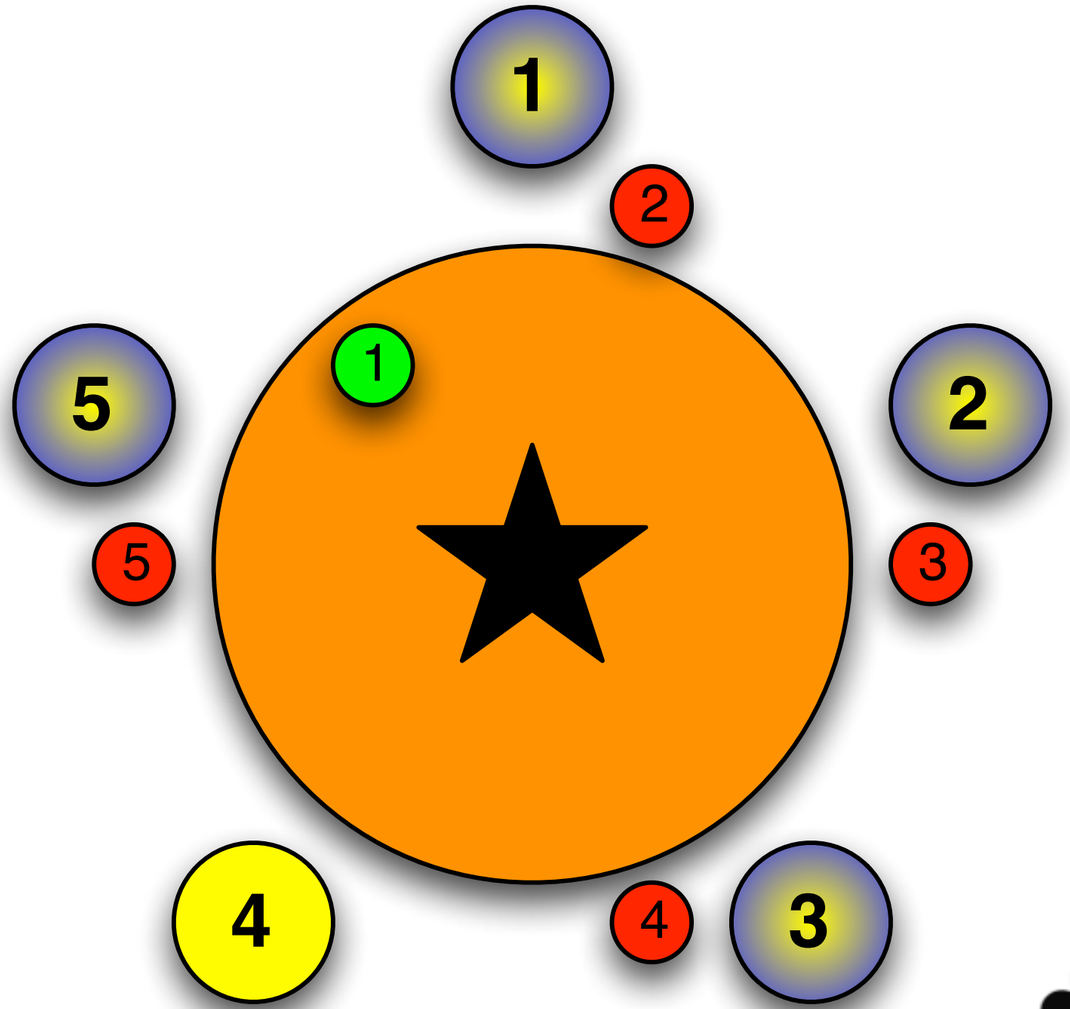
- Cups are returned in the opposite order to what they are acquired



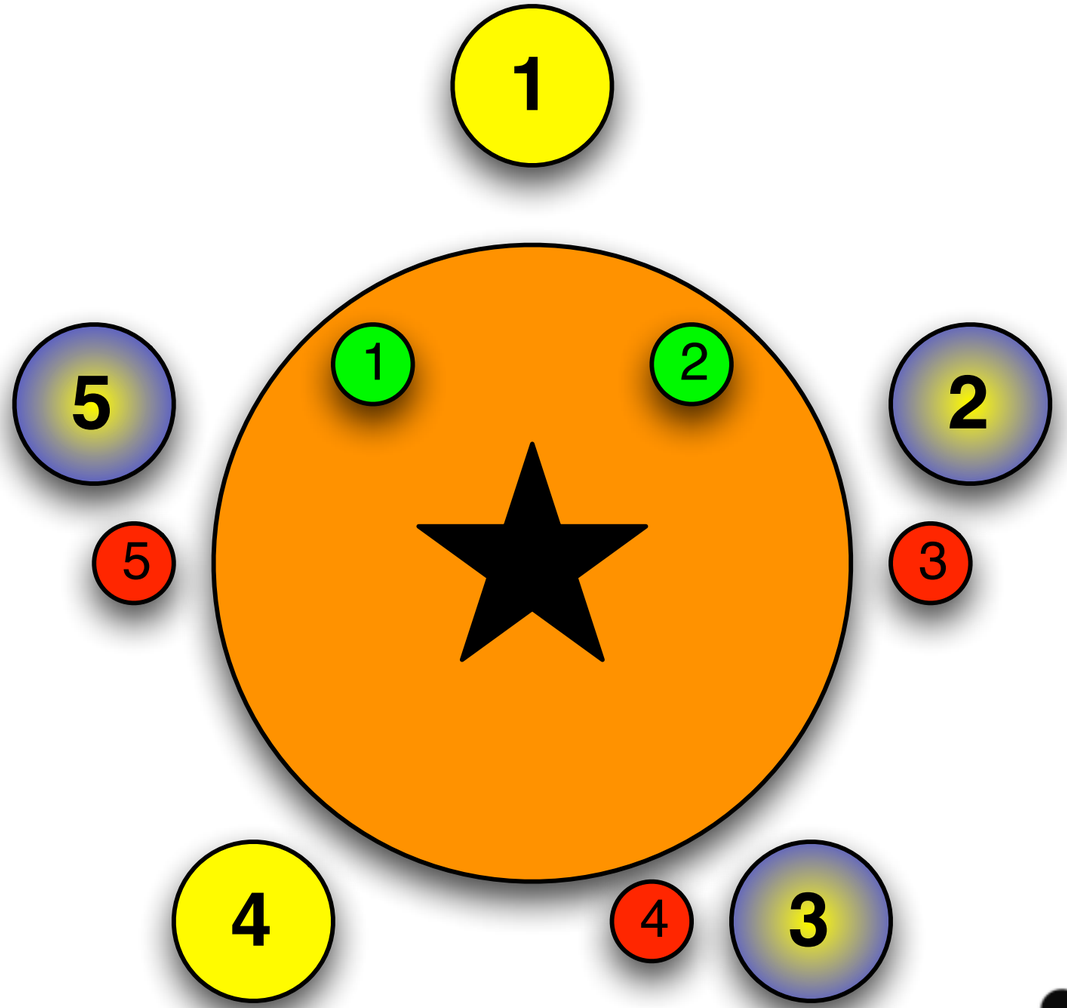
Philosopher 5 takes cup 1 - Drinking



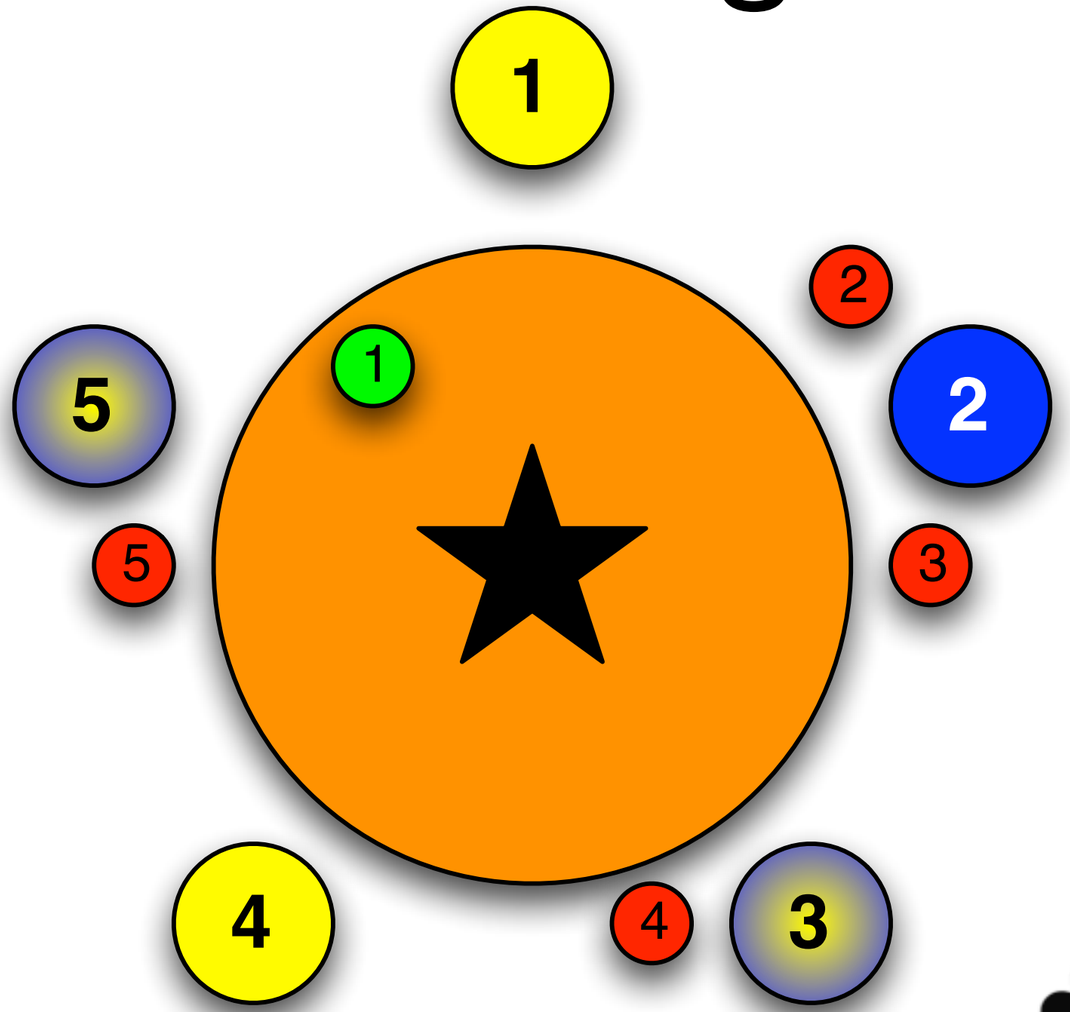
Philosopher 5 returns cup 1



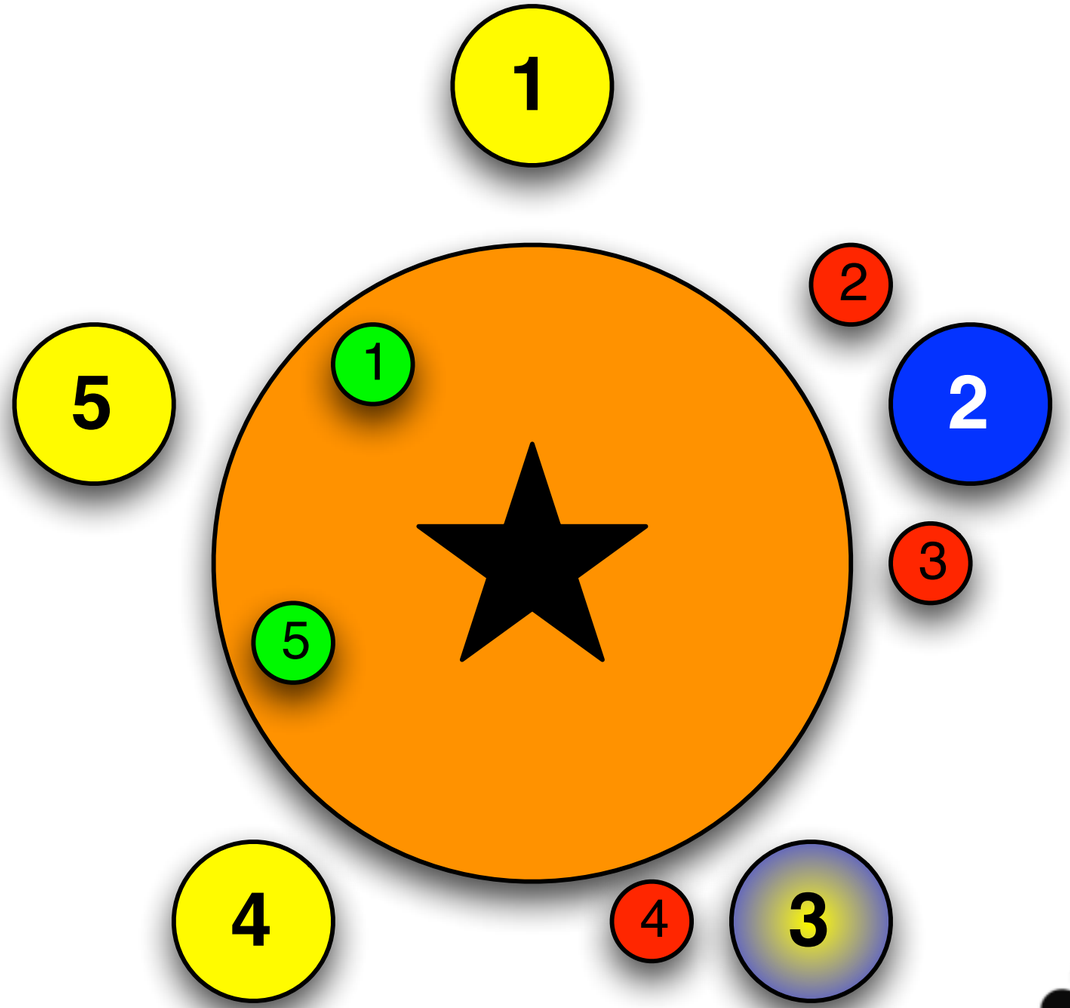
Philosopher 1 returns cup 2



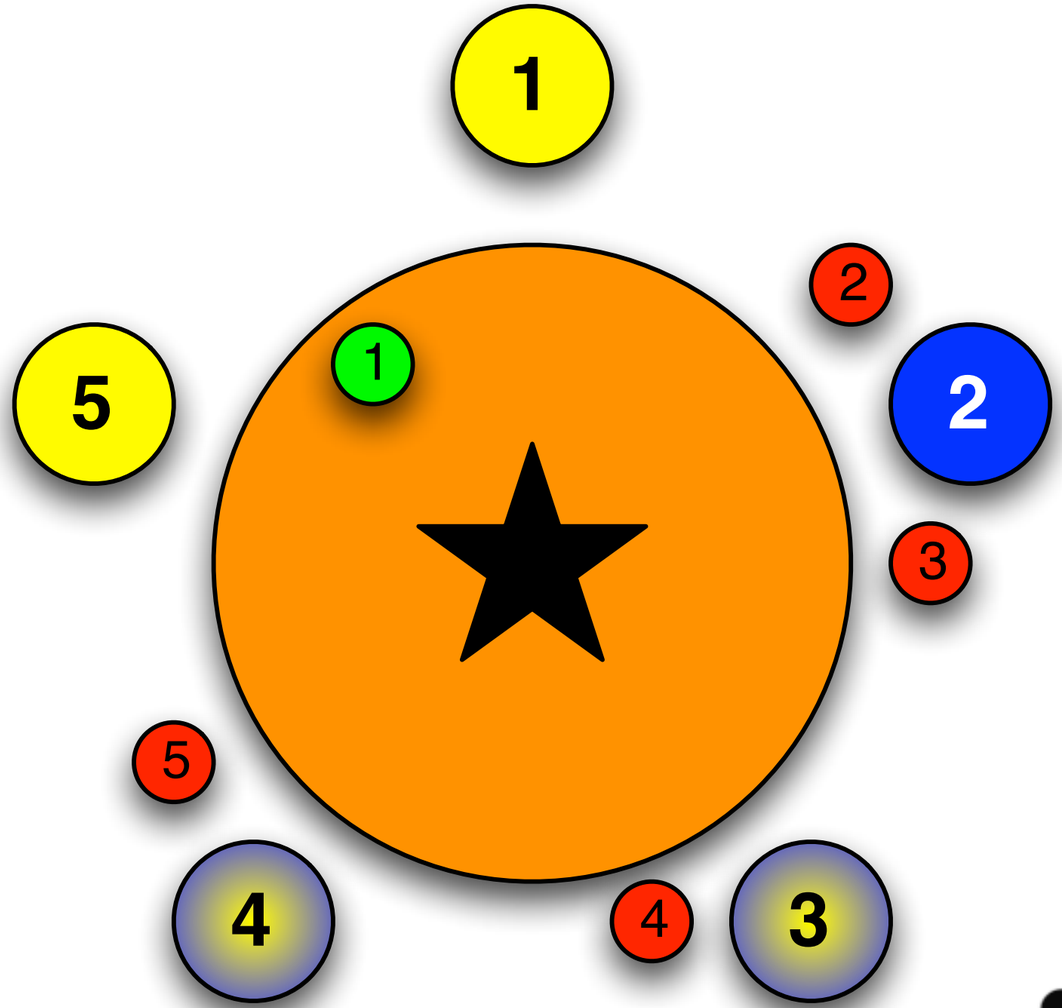
Philosopher 2 takes cup 2 - Drinking



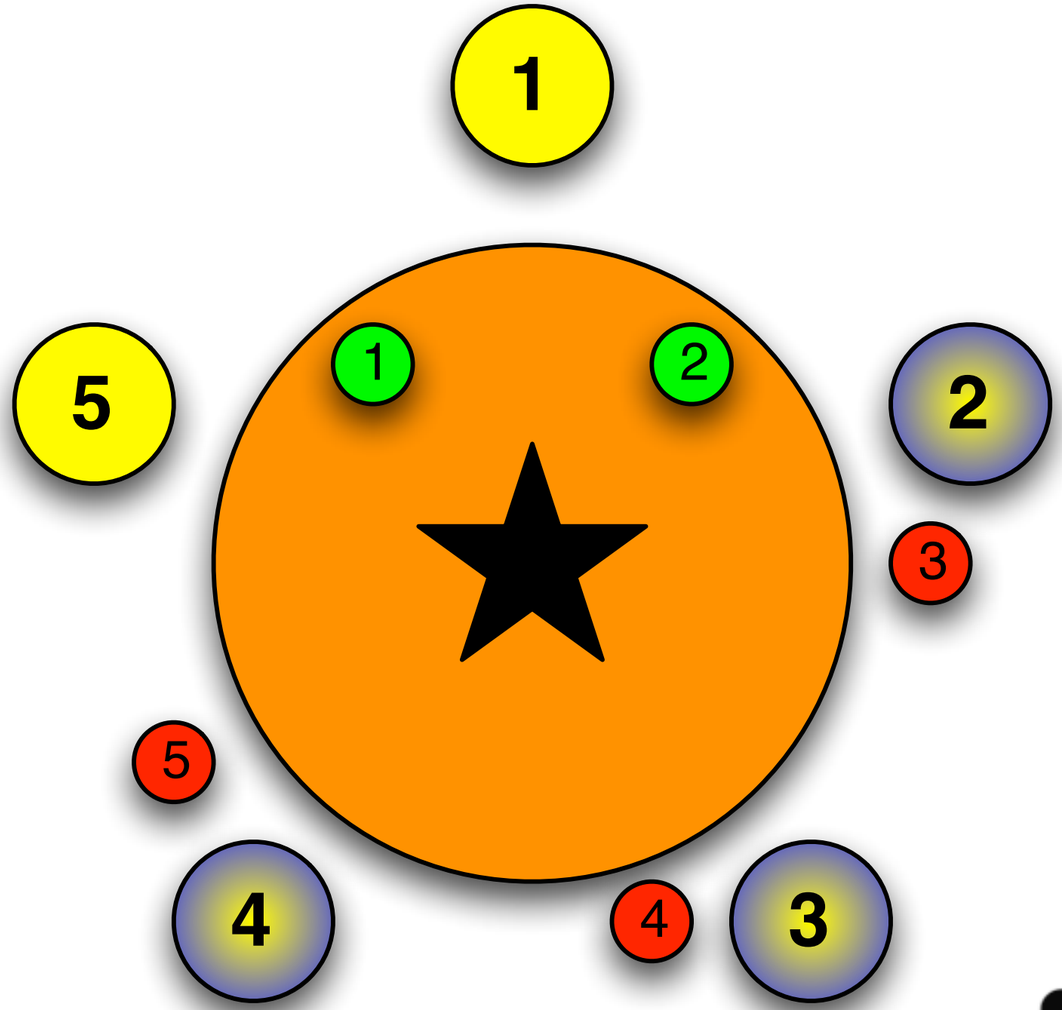
Philosopher 5 returns cup 5



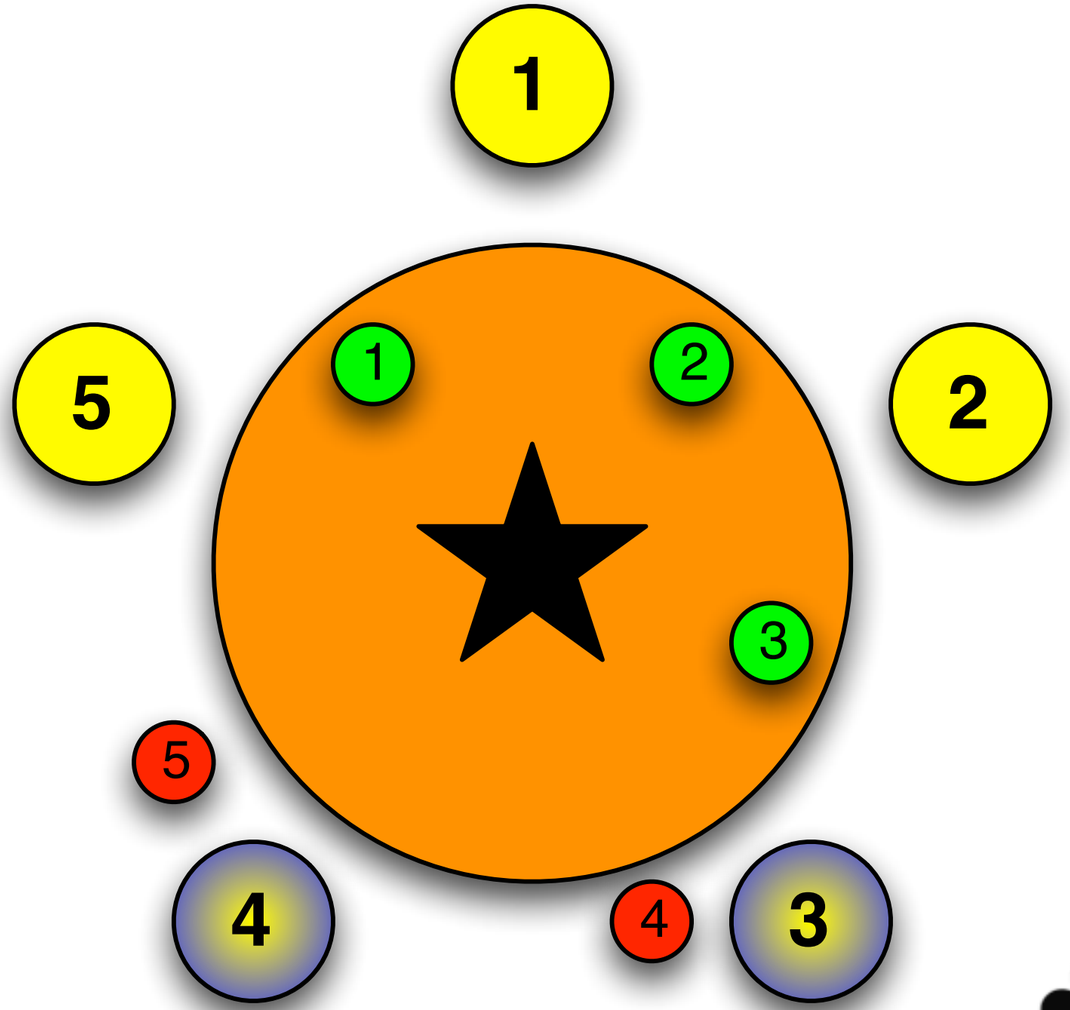
Philosopher 4 takes cup 5



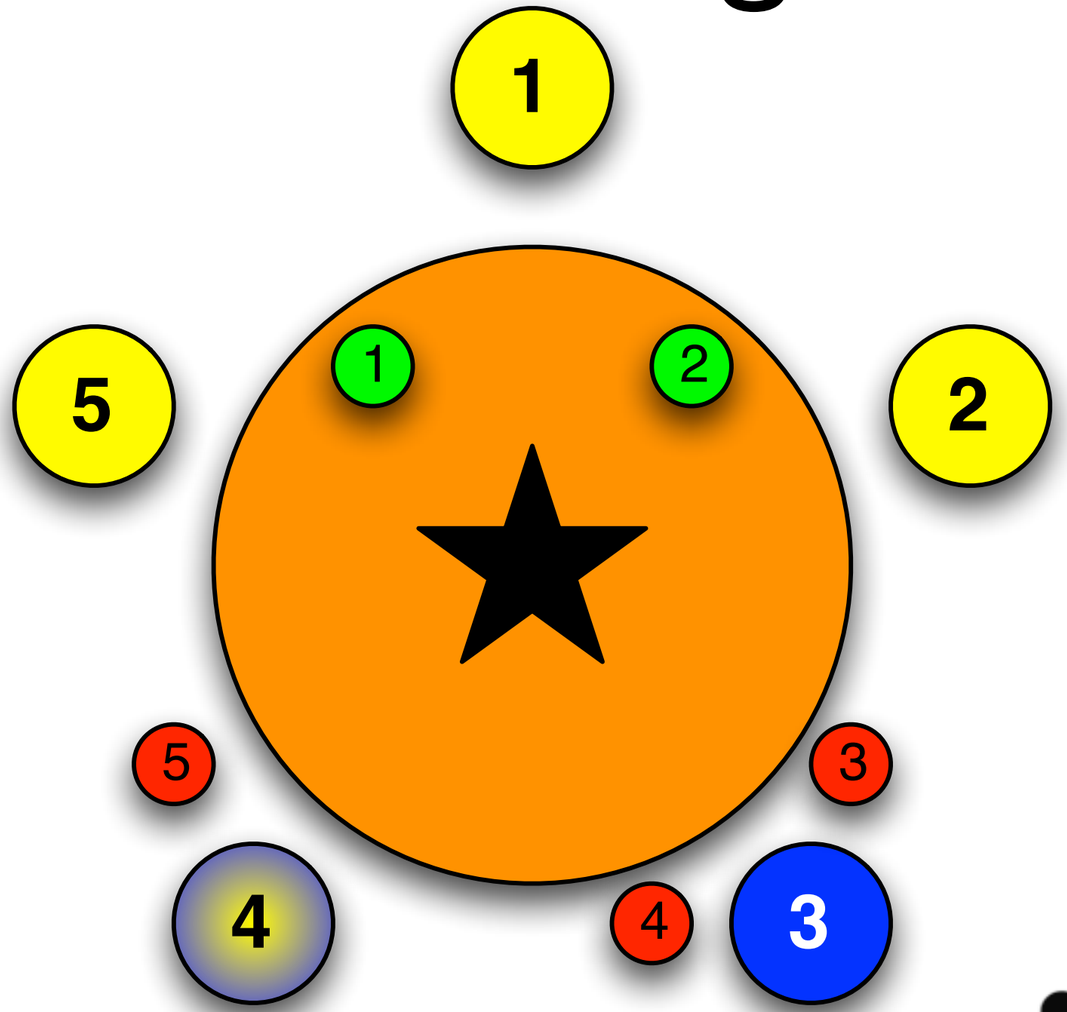
Philosopher 2 returns cup 2



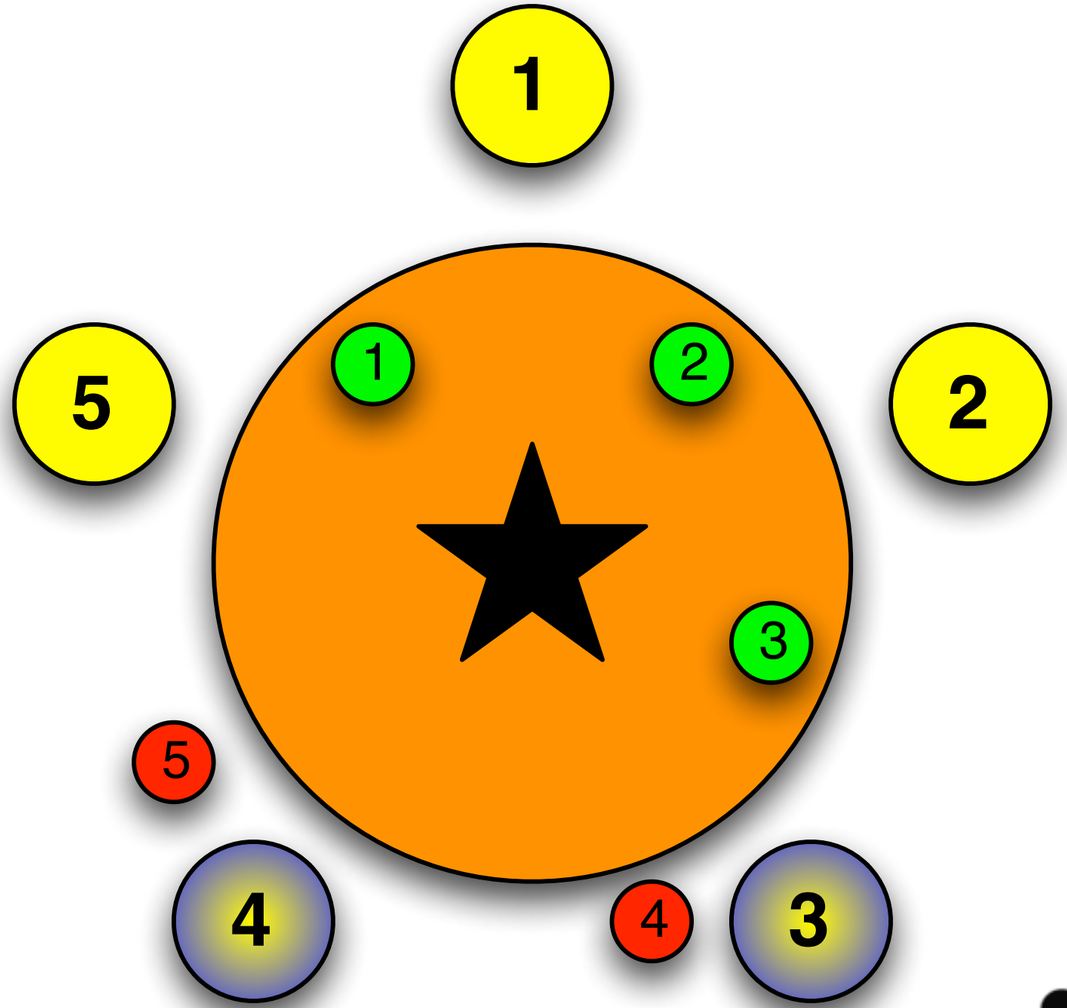
Philosopher 2 returns cup 3



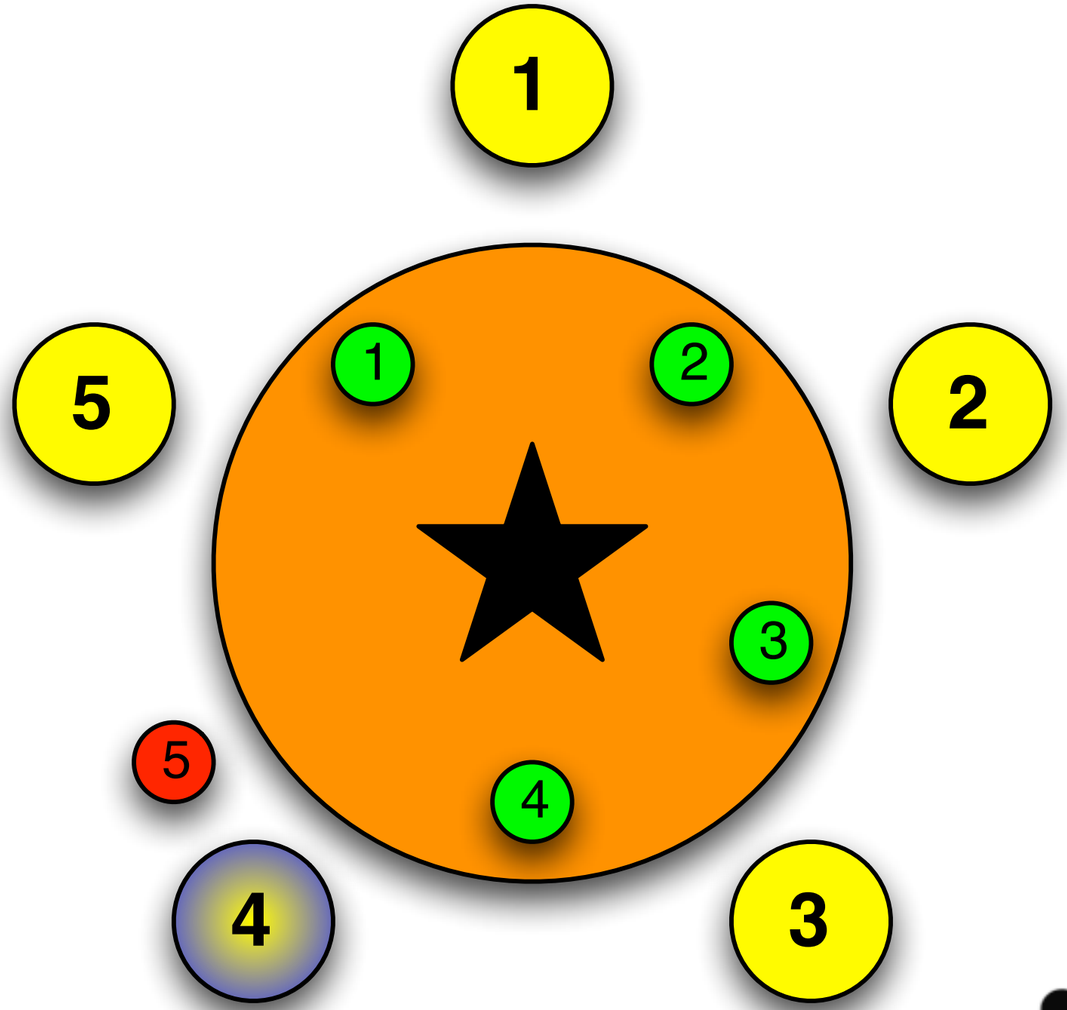
Philosopher 3 takes cup 3 - Drinking



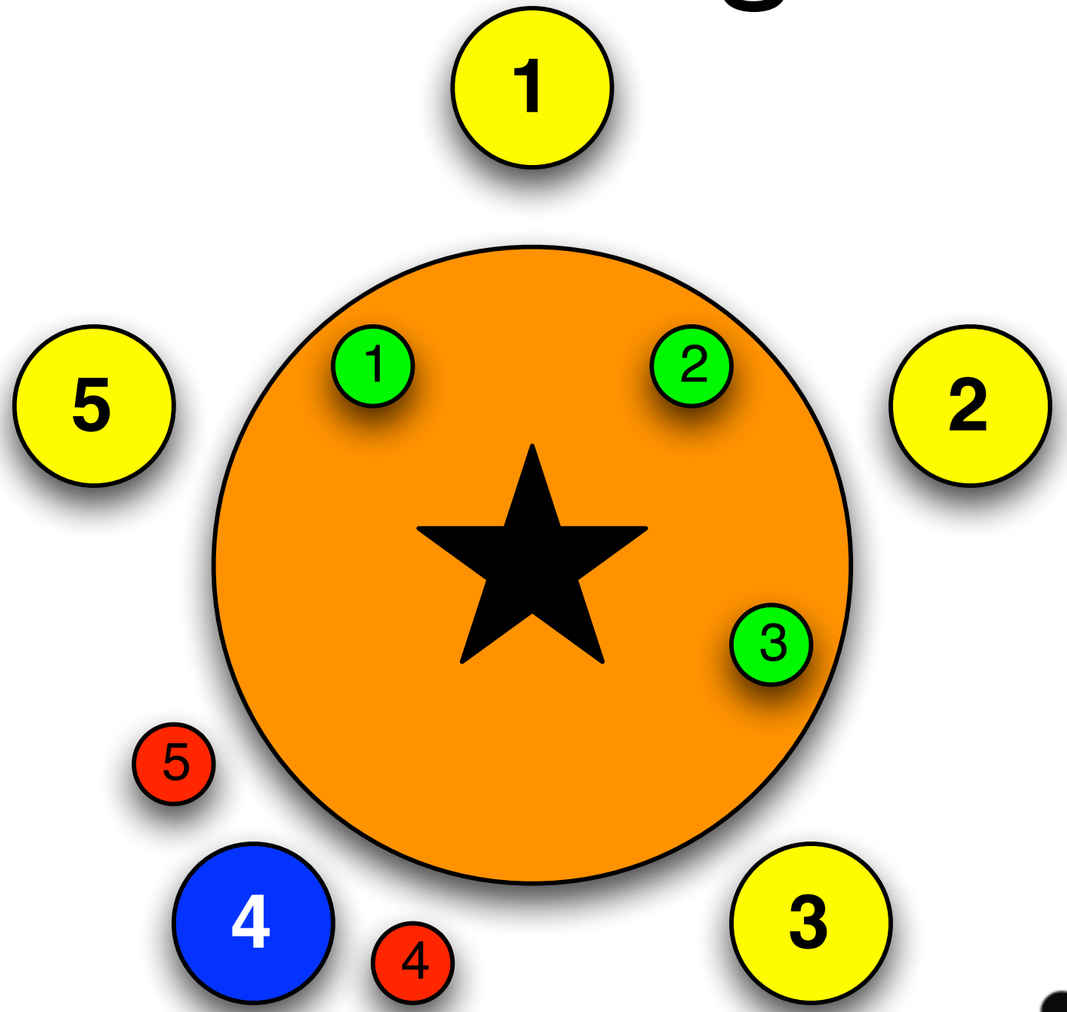
Philosopher 3 Returns cup 3



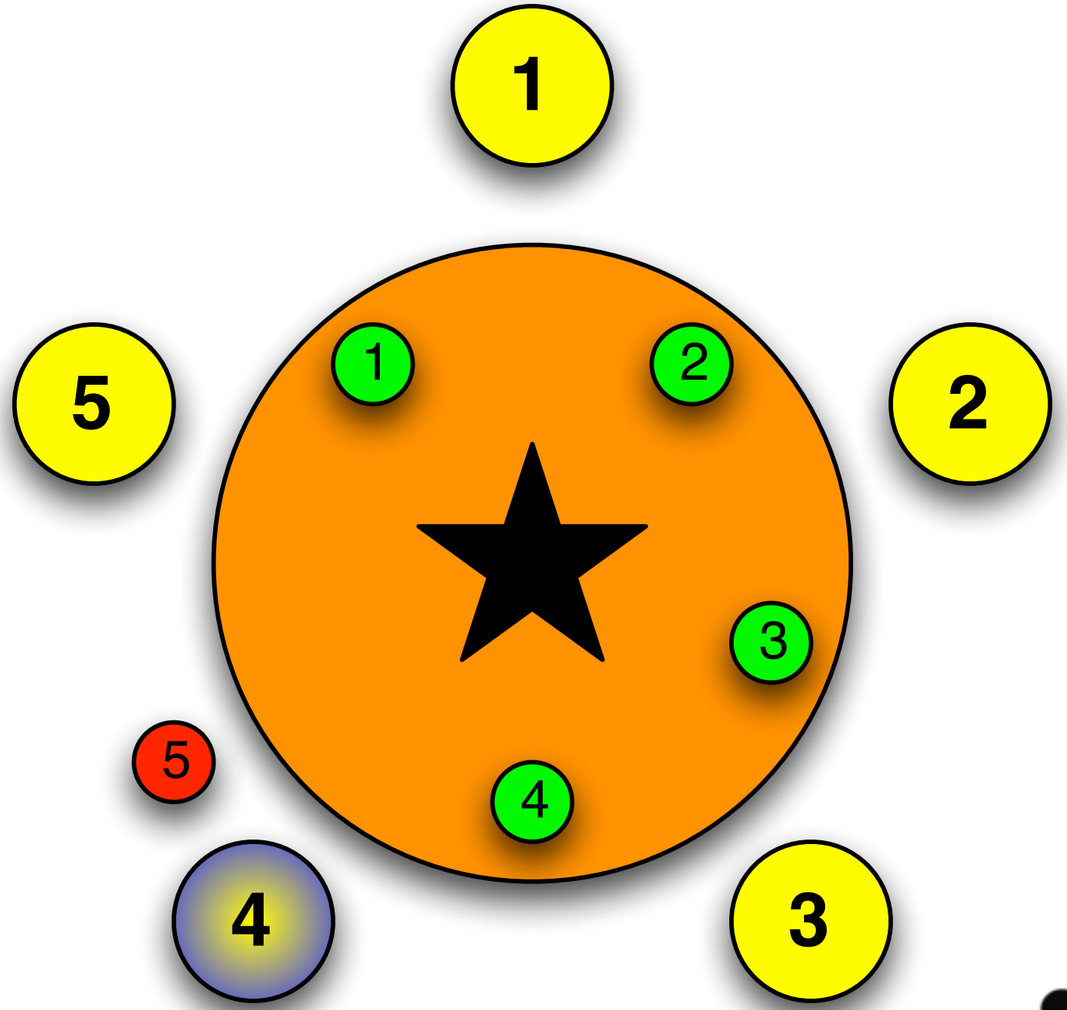
Philosopher 3 Returns cup 4



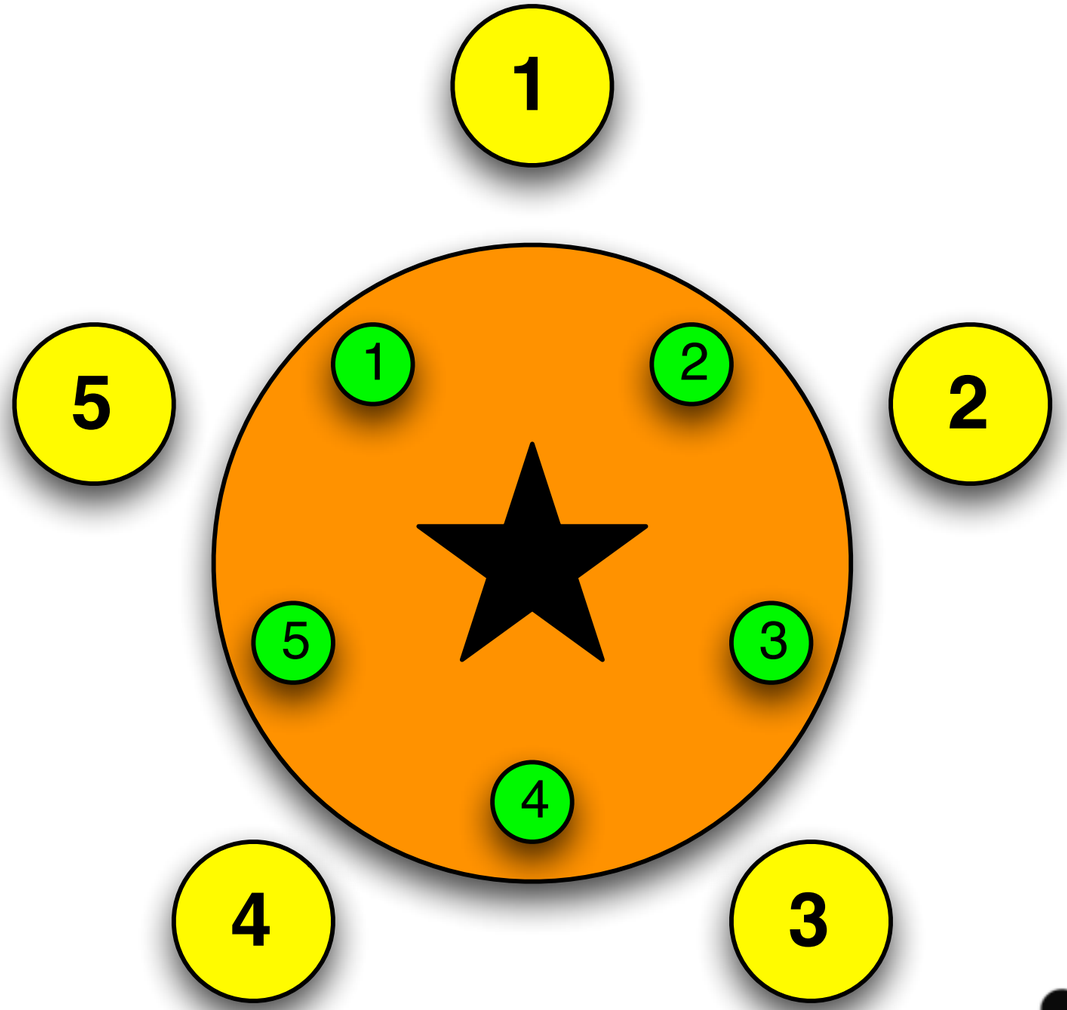
Philosopher 4 takes cup 4 - Drinking



Philosopher 4 Returns cup 4



Philosopher 4 Returns cup 5



sorting locks by property is easy

e.g. account IBAN number

or employee number

or some other characteristic

maybe even
System.identityHashCode()

use "open calls"

Vector synchronizes writeObject()

```
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
  s.defaultWriteObject();
}
```

synchronized writeObject() is invoking s.defaultWriteObject()

```
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
  s.defaultWriteObject();
}
```

this is *not* an "open call"

```
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
  s.defaultWriteObject();
}
```

but can it cause problems?

```
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
  s.defaultWriteObject();
}
```

yes it can!

```
private synchronized void writeObject(ObjectOutputStream s)
    throws IOException {
  s.defaultWriteObject();
}
```

serialize both vectors at same time

```
Vector v1 = new Vector();
Vector v2 = new Vector();
v1.add(v2);
v2.add(v1);
```

this could happen with nested data structures

it happened to an IBM client

```
private void writeObject(ObjectOutputStream stream)
    throws IOException {
  Vector<E> cloned = null;
  // this specially fix is for a special dead-lock in customer
  // program: two vectors refer each other may meet dead-lock in
  // synchronized serialization. Refer CMVC-103316.1
  synchronized (this) {
    try {
      cloned = (Vector<E>) super.clone();
      cloned.elementData = elementData.clone();
    } catch (CloneNotSupportedException e) {
      // no deep clone, ignore the exception
    }
  }
  cloned.writeObjectImpl(stream);
}

private void writeObjectImpl(ObjectOutputStream stream)
   throws IOException {
 stream.defaultWriteObject();
}
```

Java 7 uses an "open call"

```
private void writeObject(ObjectOutputStream s)
    throws IOException {
  final ObjectOutputStream.PutField fields = s.putFields();
  final Object[] data;
  synchronized (this) {
    fields.put("capacityIncrement", capacityIncrement);
    fields.put("elementCount", elementCount);
    data = elementData.clone();
  }
  fields.put("elementData", data);
  s.writeFields();
}
```

More synchronized writeObject()s

- File
- StringBuffer
- Throwable
- URL
- Hashtable
- and others

we have seen the *deadly embrace*

are there other deadlocks?

semaphore deadlocks

Bounded DB Connection Pools

- For example, say you have two DB connection pools
  - Some tasks might require connections to both databases
  - Thus thread A might hold semaphore for D1 and wait for D2, whereas thread B might hold semaphore for D2 and be waiting for D1

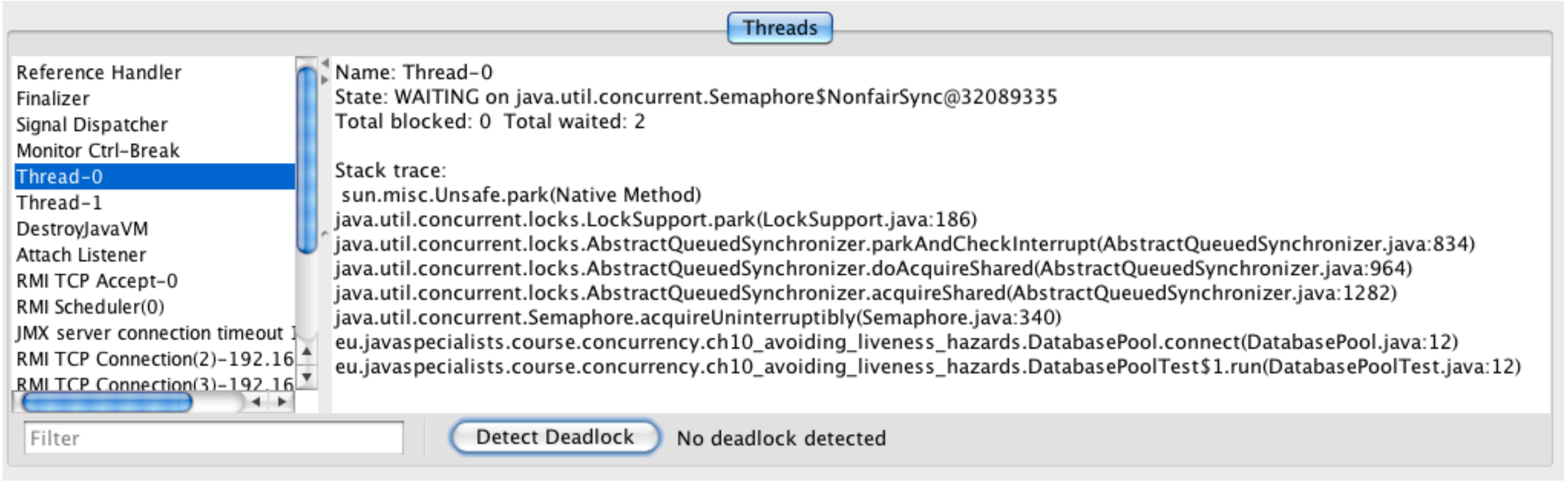
```
public class DatabasePool {
  private final Semaphore connections;
  public DatabasePool(int connections) {
    this.connections = new Semaphore(connections);
  }

  public void connect() {
    connections.acquireUninterruptibly();
    System.out.println("DatabasePool.connect");
  }

  public void disconnect() {
    System.out.println("DatabasePool.disconnect");
    connections.release();
  }
}
```

ThreadMXBean does not show this as a deadlock!

JVM does not detect this deadlock



JVM does not detect this deadlock

Avoiding and diagnosing deadlocks

Avoiding Liveness Hazards

Avoiding and diagnosing deadlocks

- You cannot get a lock-ordering deadlock with a single lock
  - Easiest way to avoid deadlocks
  - But not always practical

Avoiding and diagnosing deadlocks

- For multiple locks we need to define a lock order
  - specify and document possible lock sequences
  - Identify where multiple locks could be acquired
  - Do a global analysis to ensure that lock ordering is consistent
    - This can be extremely difficult in large programs

Use open calls whenever possible

Unit Testing for lock ordering deadlocks

- Code typically has to be called many times before a deadlock happens
- How many times do you need to call it to prove that there is no deadlock?
  - Nondeterministic unit tests are bad - they should either always pass or always fail

Testing the Bank

- In the transferMoney() method, a deadlock occurs if after the first lock is granted, the first thread is swapped out and another thread requests the second lock

```
public class Bank {
  public boolean transferMoney(Account from, Account to, Amount amount) {
    synchronized (from) {
      // if thread is swapped out here, we can end up with a deadlock
      synchronized (to) {
        return doActualTransfer(from, to, amount);
      }
    }
  }
}
```

We can force the context switch

simply add a sleep after first lock

Adding a sleep to cause deadlocks

```
public class Bank {
  public boolean transferMoney(Account from, Account to,
                               Amount amount) {
    synchronized (from) {
      sleepAWhileForTesting();
      synchronized (to) {
        return doActualTransfer(from, to, amount);
      }
    }
  }
  protected void sleepAWhileForTesting() {}
}
```

In our unit test we use a SlowBank

```
public class SlowBank extends Bank {
  private final long timeout;
  private final TimeUnit unit;
  public SlowBank(long timeout, TimeUnit unit) {
    this.timeout = timeout;
    this.unit = unit;
  }
  protected void sleepAWhileForTesting() {
    try {
      unit.sleep(timeout);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
  }
}
```

Won't this cause problems in production?

In production we only use Bank

HotSpot will optimize away empty sleepAWhileForTesting()

it is thus a zero cost

as long as SlowBank is never used

How can you verify the lock-ordering deadlock?

CTRL+Break, jstack, kill -3, jconsole, etc.

but that won't be a good unit test

Better via Java's ThreadMXBean

ThreadMXBean

- Get instance with ManagementFactory.getThreadMXBean()
- findMonitorDeadlockedThreads()
  - Includes only "monitor" locks, i.e. synchronized
  - Only way to find deadlocks in Java 5

ThreadMXBean

- Better is findDeadlockedThreads()
  - Includes "monitor" and "owned" (Java 5) locks
  - Preferred method to test for deadlocks
  - But, does not find deadlocks between semaphores
- See http://www.javaspecialists.eu/archive/Issue130.html

```
public class BankDeadlockTest {
  private final static ThreadMXBean tmb =
    ManagementFactory.getThreadMXBean();

  private boolean isThreadDeadlocked(long tid) {
    long[] ids = tmb.findDeadlockedThreads();
    if (ids == null) return false;
    for (long id : ids) {
      if (id == tid) return true;
    }
    return false;
  }
```

```
private void checkThatThreadTerminates(Thread thread)
    throws InterruptedException {
  for (int i = 0; i < 2000; i++) {
    thread.join(50);
    if (!thread.isAlive()) return;
    if (isThreadDeadlocked(thread.getId())) {
      fail("Deadlock detected!");
    }
  }
  fail(thread + " did not terminate in time");
}
```

```
  @Test
  public void testForTransferDeadlock() throws InterruptedException {
    final Account alpha = new Account(new Amount(1000));
    final Account ubs = new Account(new Amount(1000000));
    final Bank bank = new SlowBank(100, TimeUnit.MILLISECONDS);

    Thread alphaToUbs = new Thread("alphaToUbs") {
      public void run() {
        bank.transferMoney(alpha, ubs, new Amount(100));
      }
    };
    Thread ubsToAlpha = new Thread("ubsToAlpha") {
      public void run() {
        bank.transferMoney(ubs, alpha, new Amount(100));
      }
    };

    alphaToUbs.start();
    ubsToAlpha.start();

    checkThatThreadTerminates(alphaToUbs);
  }
}
```

Output with broken transferMoney() method

- We see the deadlock within about 100 milliseconds

```
junit.framework.AssertionFailedError: Deadlock detected!
  at BankDeadlockTest.checkThatThreadTerminates(BankDeadlockTest.java:20)
  at BankDeadlockTest.testForTransferDeadlock(BankDeadlockTest.java:55)
```

Output with broken transferMoney() method

- If we fix the transferMoney() method, it also completes within about 100 milliseconds
  - This is the time that we are sleeping for testing purposes
- Remember that the empty sleepAWhileForTesting() method will be optimized away by HotSpot

Timed lock attempts

- Another technique for solving deadlocks is to use the timed tryLock() method of explicit Java locks

However, if tryLock() fails it may be

a deadlock

or some thread taking a long time

or a thread in an infinite loop

But ThreadMXBean shows it as *deadlocked*

even if it is a temporary condition

The JVM has support for tryLock with synchronized

but it is hidden away

So here it is

Told you it was hidden!

It is hidden for a good reason

if you need tryLock functionality

rather use *explicit locks*
(ReentrantLock)

tryLock with synchronized

- Technically possible in Sun JVM using
  - Unsafe.getUnsafe().tryMonitorEnter(monitor)
  - Unsafe.getUnsafe().monitorExit(monitor)
- http://www.javaspecialists.eu/archive/Issue194.html

Deadlock analysis with thread dumps

Deadlock analysis with thread dumps

- We can also cause a thread dump in many ways:
  - Ctrl+Break on Windows or Ctrl-\ on Unix
  - Invoking "kill -3" on the process id
  - Calling jstack on the process id
    - Only shows deadlocks since Java 6
- Intrinsic locks show more information of where they were acquired

Can a deadlock be resolved?

NO with synchronized

YES with ReentrantLock

synchronized() goes into BLOCKED state

ReentrantLock.lock() goes into WAITING state

We can write a DealockArbitrator

This stops one of the threads with a DeadlockVictimError

Our special Error for deadlocks

```
public class DeadlockVictimError extends Error {
  private final Thread victim;
  public DeadlockVictimError(Thread victim) {
    super("Deadlock victim: " + victim);
    this.victim = victim;

  }
  public Thread getVictim() { return victim; }
}
```

```
public class DeadlockArbitrator {
  private static final ThreadMXBean tmb =
      ManagementFactory.getThreadMXBean();

  public boolean tryResolveDeadlock(
      int attempts, long timeout, TimeUnit unit)
      throws InterruptedException {
    for (int i = 0; i < attempts; i++) {
      long[] ids = tmb.findDeadlockedThreads();
      if (ids == null) return true;
      Thread t = findThread(ids[i % ids.length]);
      if (t == null)
        throw new IllegalStateException("Could not find thread");
      t.stop(new DeadlockVictimError(t));
      unit.sleep(timeout);
    }
    return false;
  }
```

finding threads by id is tricky

```
  public boolean tryResolveDeadlock() throws InterruptedException {
    return tryResolveDeadlock(3, 1, TimeUnit.SECONDS);
  }

  private Thread findThread(long id) {
    for (Thread thread : Thread.getAllStackTraces().keySet()) {
      if (thread.getId() == id) return thread;
    }
    return null;
  }
}
```

Applicability of DeadlockArbitrator

- Only use in extreme circumstances
  - Code that is outside your control and that deadlocks
  - Where you cannot prevent the deadlock
- Remember, it only works with ReentrantLock

Deadlock Victim Questions?

by Dr Heinz Kabutz && Olivier Croisier
The Java Specialists Newsletter && The Coder's Breakfast
heinz@javaspecialists.eu && olivier.croisier@zenika.com
twitter: @heinzkabutz && twitter: @oliviercroisier