Finding and Solving Java Deadlocks

Dr Heinz M. Kabutz

heinz@kabutz.net

@heinzkabutz

Last Updated 20 April 2016

Javaspecialists.eu
java training

Heinz Kabutz

- **Author of The Java Specialists' Newsletter**
  - **Articles about advanced core Java programming**
- **http://www.javaspecialists.eu**

Introduction

Structure Of Hands-On Lab

- **Three short lectures, each followed by a short lab**
  - **https://github.com/kabutz/DeadlockLabECESCON9**
    - **(or http://tinyurl.com/deadlocks2016)**
- **Fourth lab if we have time**

Questions

- **Please please please ask questions!**
- **Interrupt us at any time**
  - **This lab is on deadlocks, we need to keep focused in available time**
- **The only stupid questions are those you do not ask**
  - **Once you've asked them, they are not stupid anymore**
- **The more you ask, the more we all learn**

Avoiding Liveness Hazards

Avoiding Liveness Hazards

- **Fixing safety problems can cause liveness problems**
  - **Don't indiscriminately sprinkle "synchronized" into your code**

Deadly Embrace

- **Lock-ordering deadlocks**
  - **Typically when you lock two locks in different orders**
  - **Requires global analysis to make sure your order is consistent**
- **Lesson: only ever hold a single lock per thread!**

Thread Deadlocks in BLOCKED

- **A deadly embrace amongst synchronized leaves no way of recovery**
  - **We have to restart the JVM**

Resource Deadlocks

- **This can happen with bounded queues or similar mechanisms meant to bound resource consumption**

Lab 1: Deadlock Resolution by Global Ordering

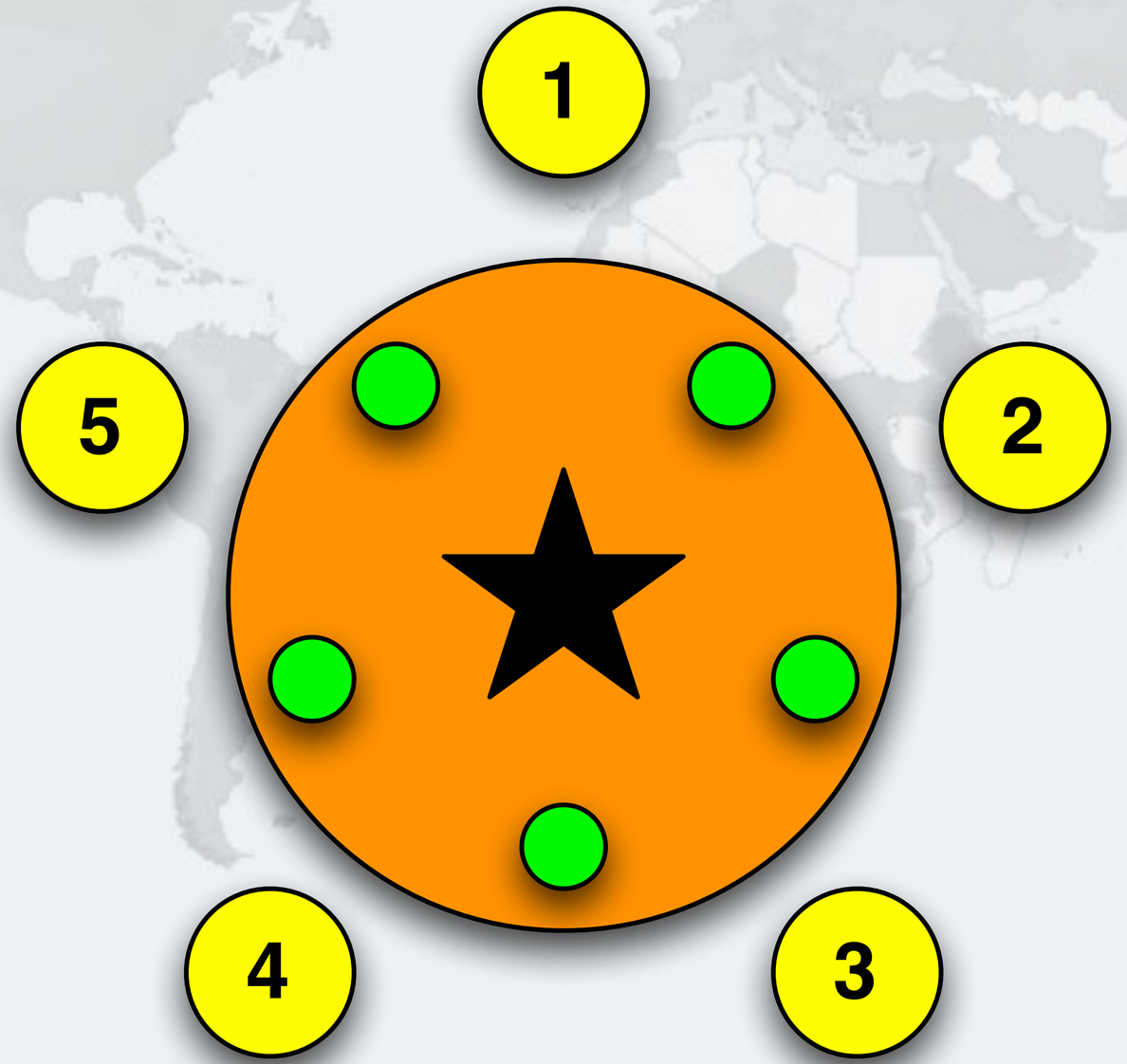
Lab 1: Deadlock Resolution By Global Ordering

- **Classic problem is that of the "dining philosophers"**
  - **We changed that to the "drinking philosophers"**
    - **That is where the word "symposium" comes from**
      - **sym - together, such as "symphony"**
      - **poto - drink**
    - **Ancient Greek philosophers used to get together to drink & think**
- **In our example, a philosopher needs two glasses to drink**
  - **First he takes the right one, then the left one**
  - **When he finishes drinking, he returns them and carries on thinking**

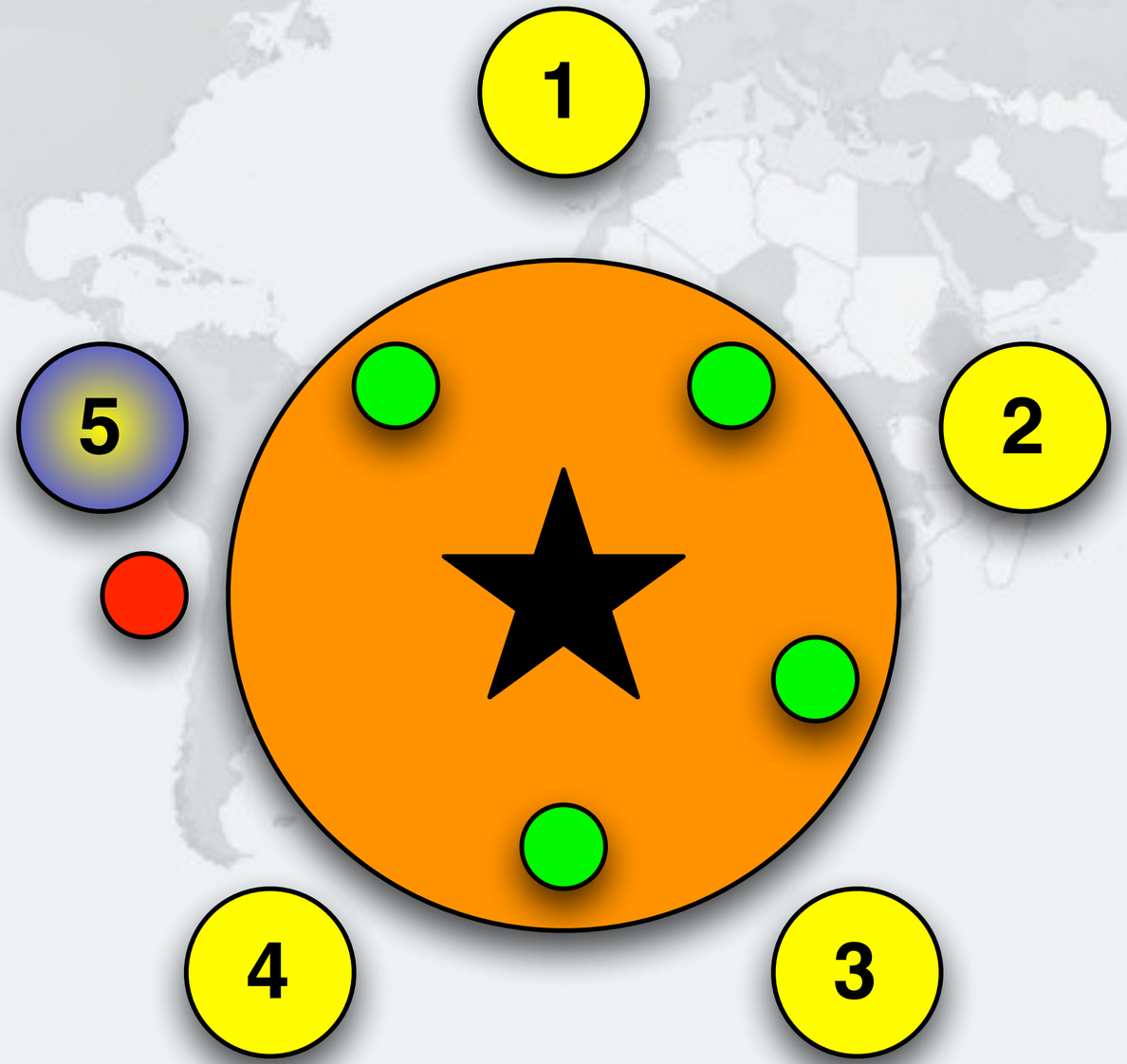
Our Drinking Philosophers

- **Our philosopher needs two glasses to drink**
  - **First he takes the right one, then the left one**
  - **When he's done, he returns the left and then the right**
  - **returns them and carries on thinking**

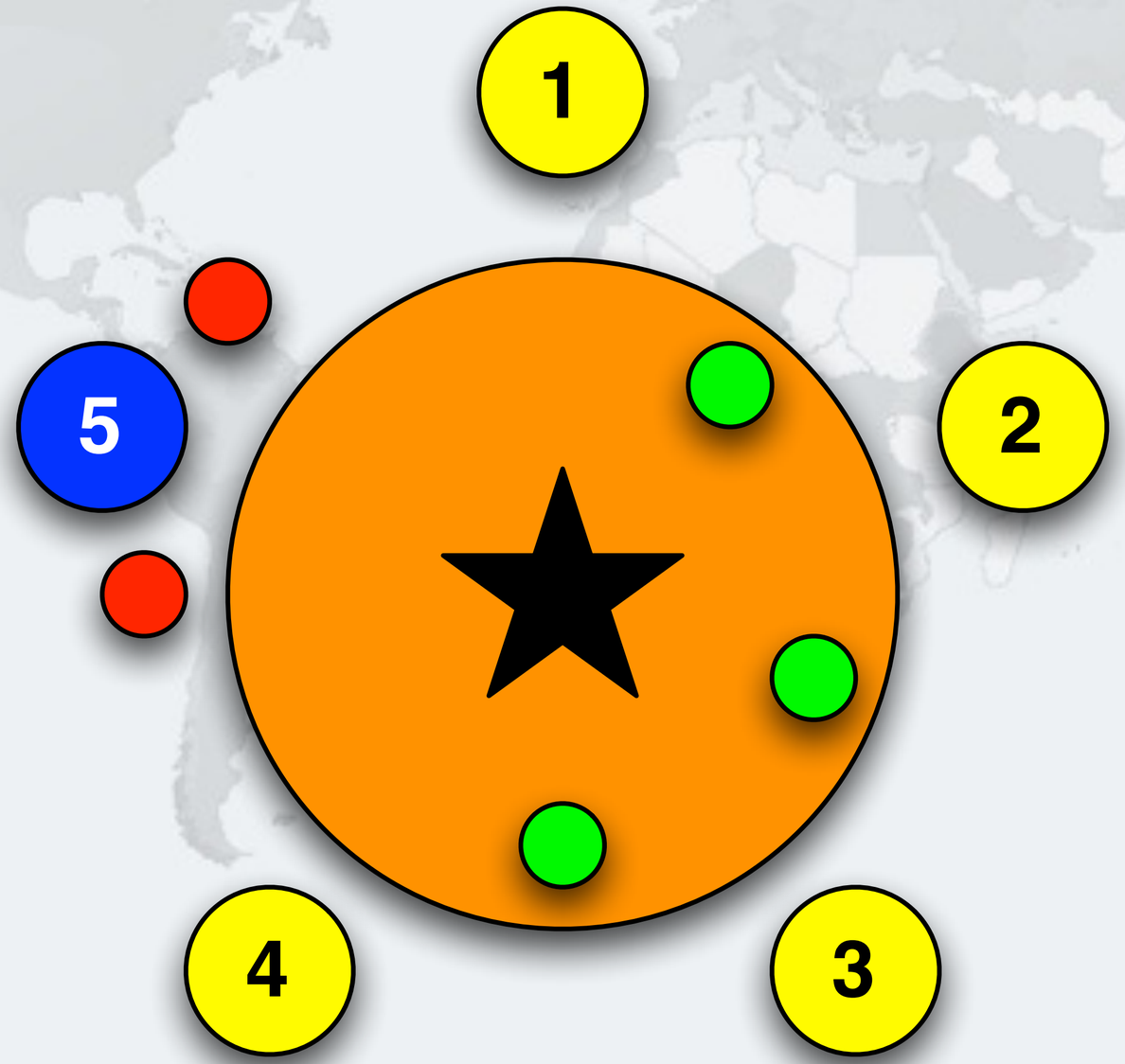
Table Is Ready, All Philosophers Are Thinking



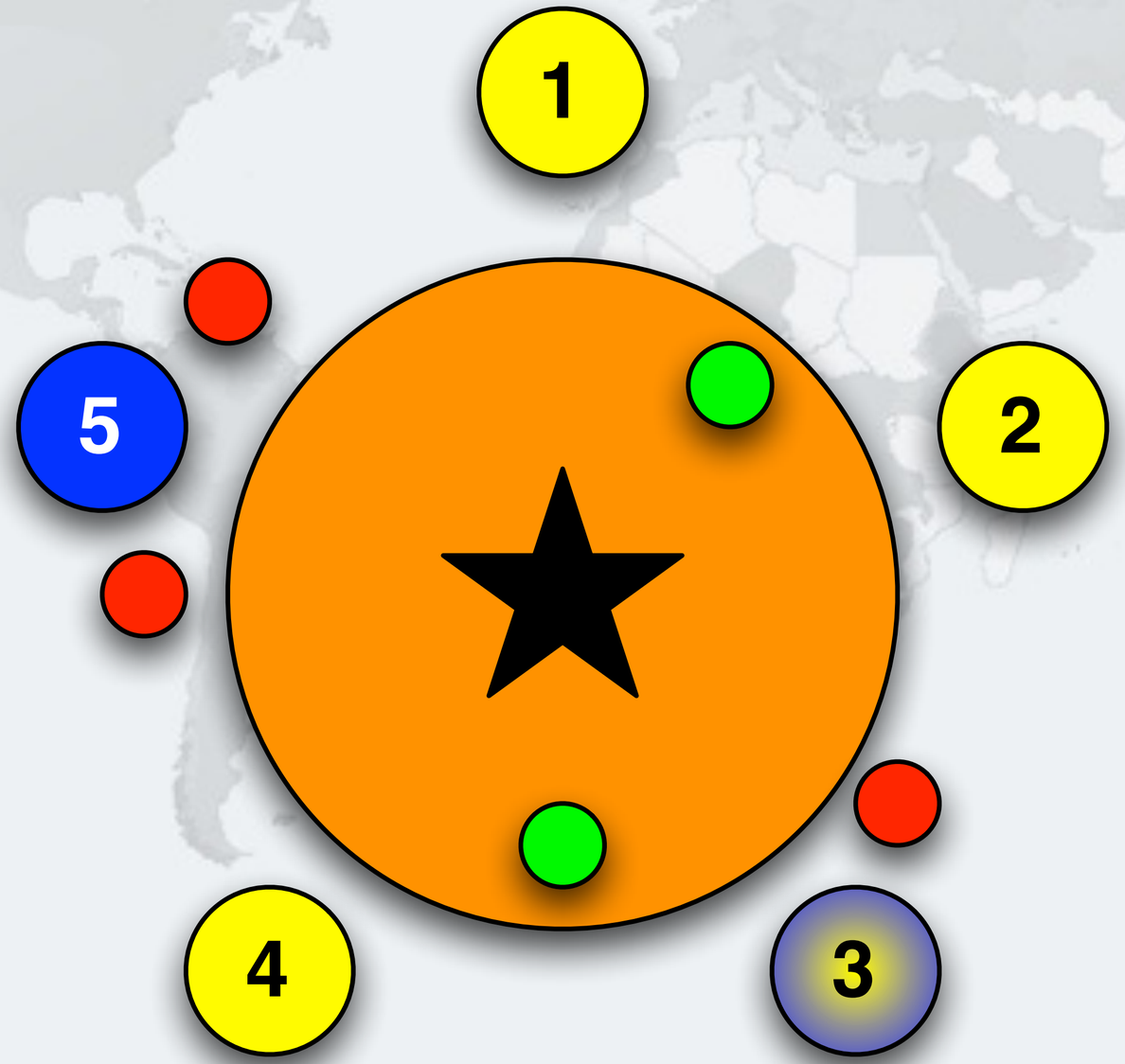
Philosopher 5 Wants To Drink, Takes Right Cup



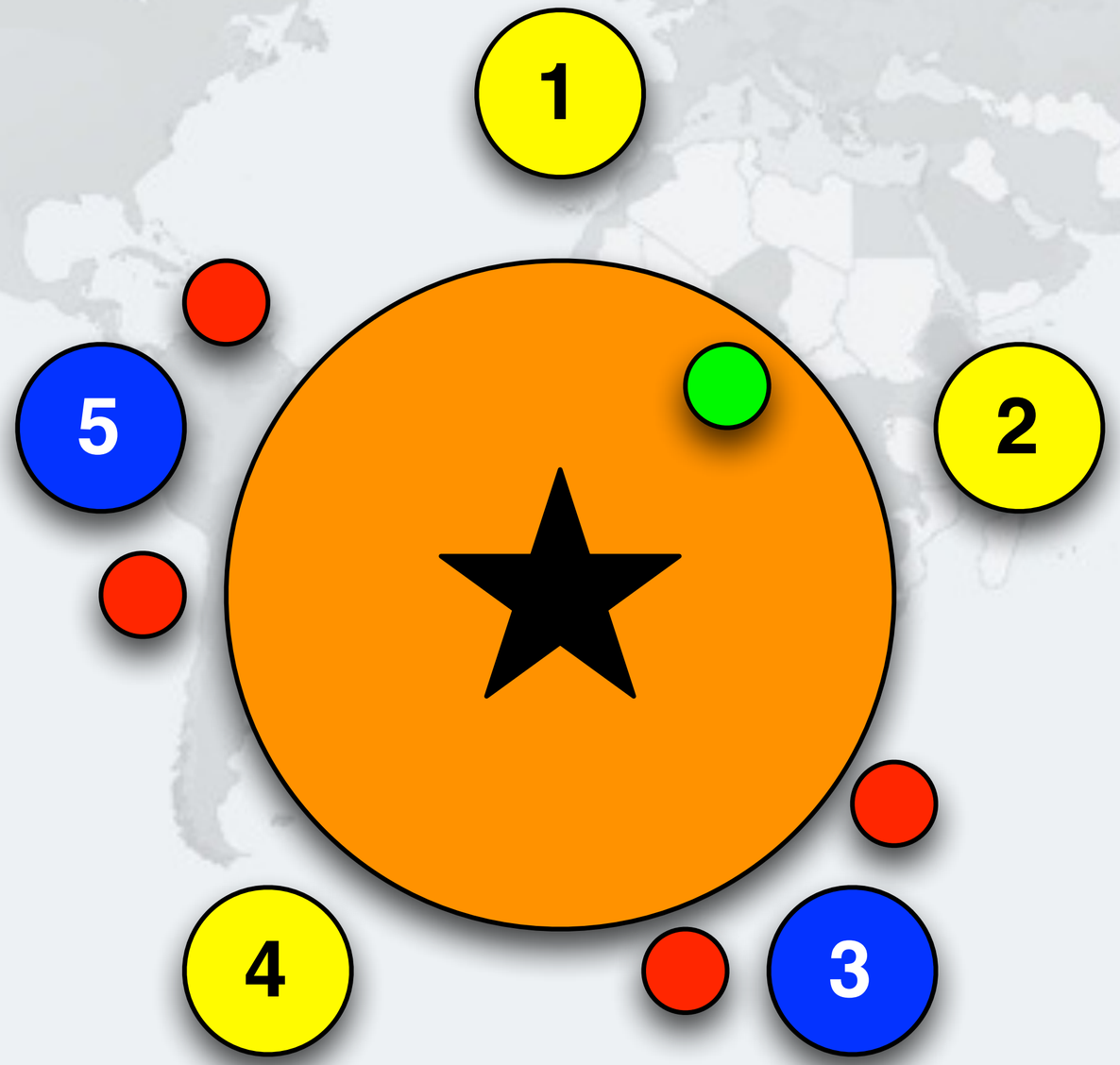
Philosopher 5 Is Now Drinking With Both Cups



Philosopher 3 Wants To Drink, Takes Right Cup

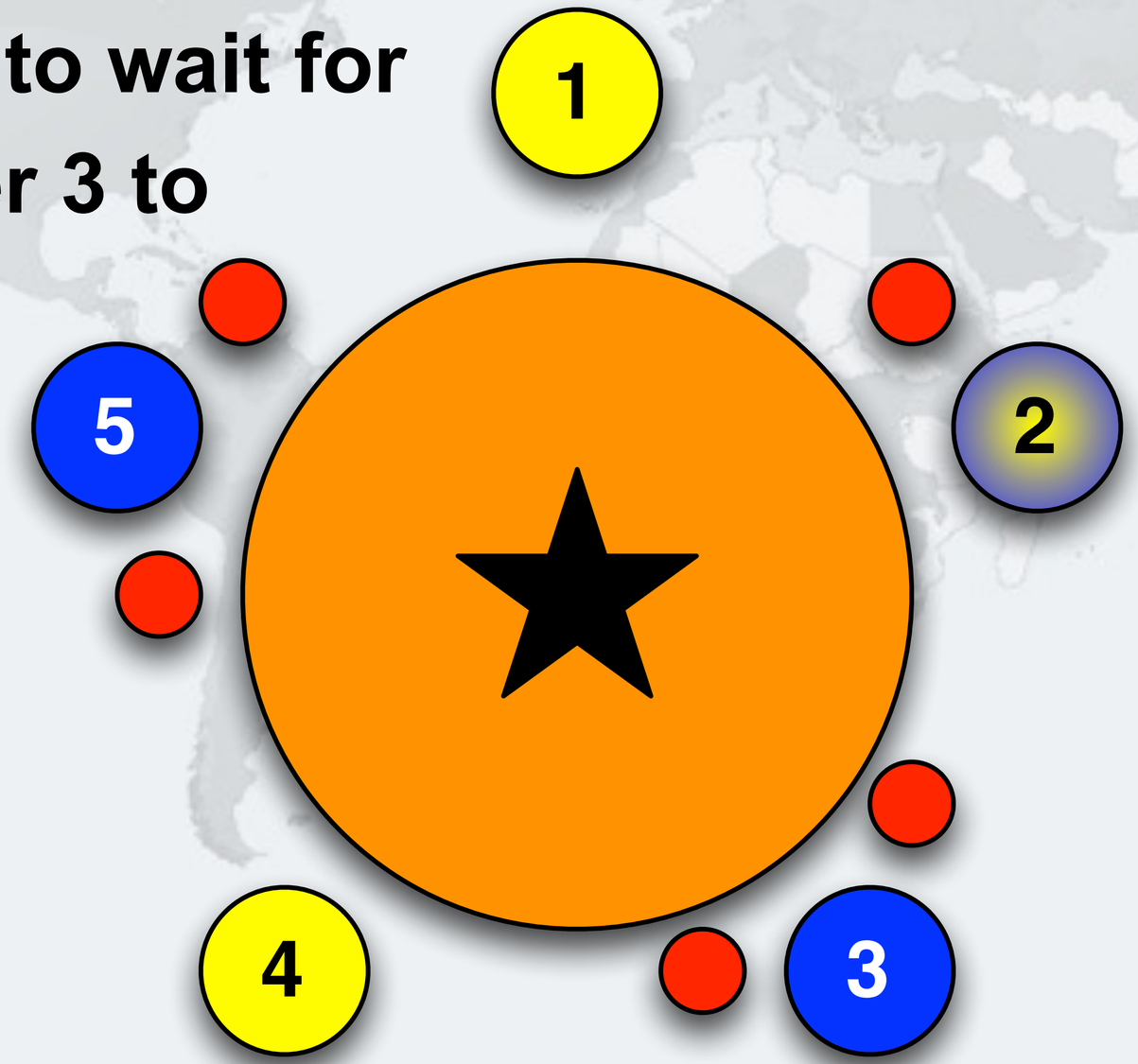


Philosopher 3 Is Now Drinking With Both Cups

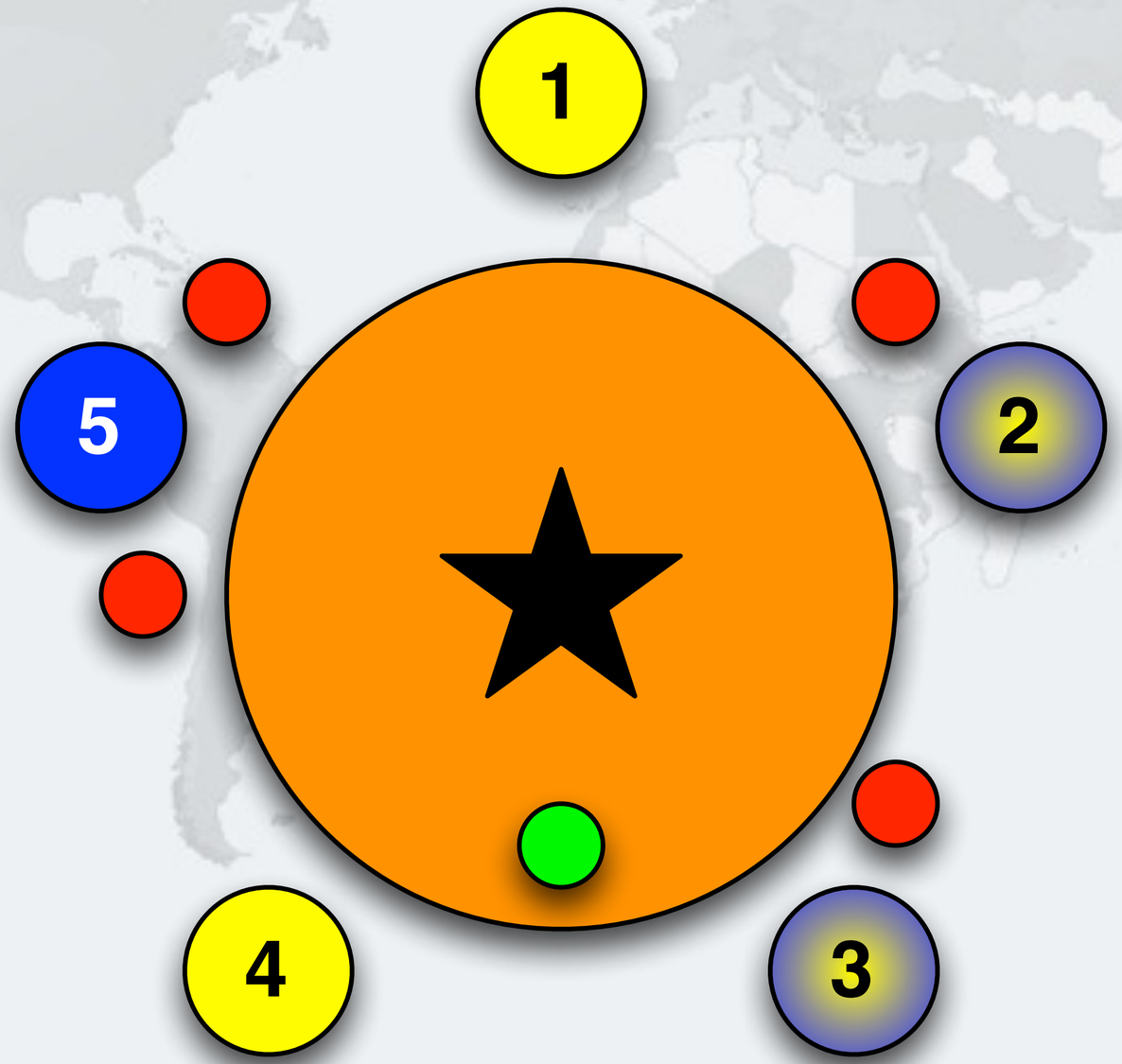


Philosopher 2 Wants To Drink, Takes Right Cup

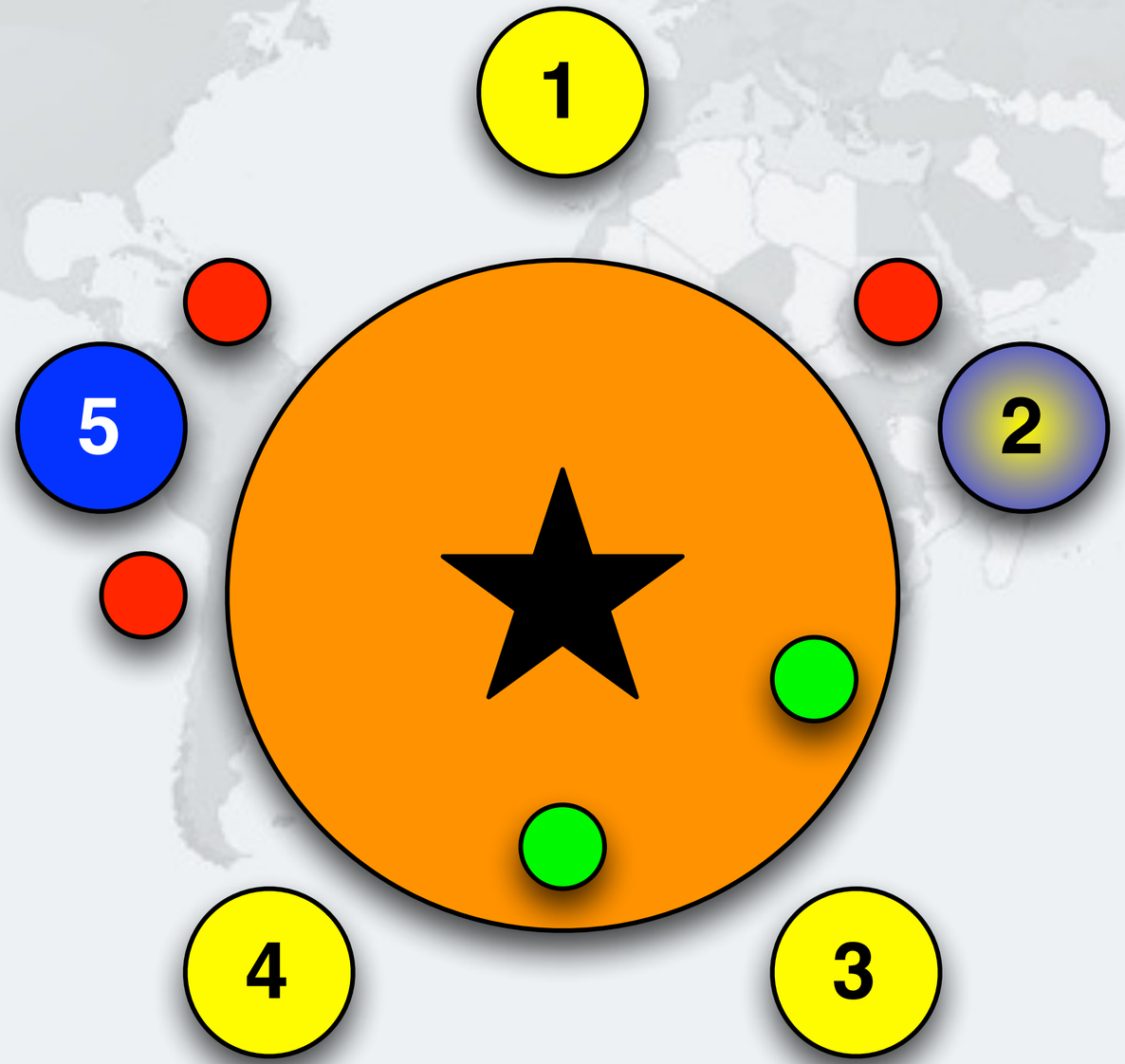
- **But he has to wait for Philosopher 3 to finish his drinking session**



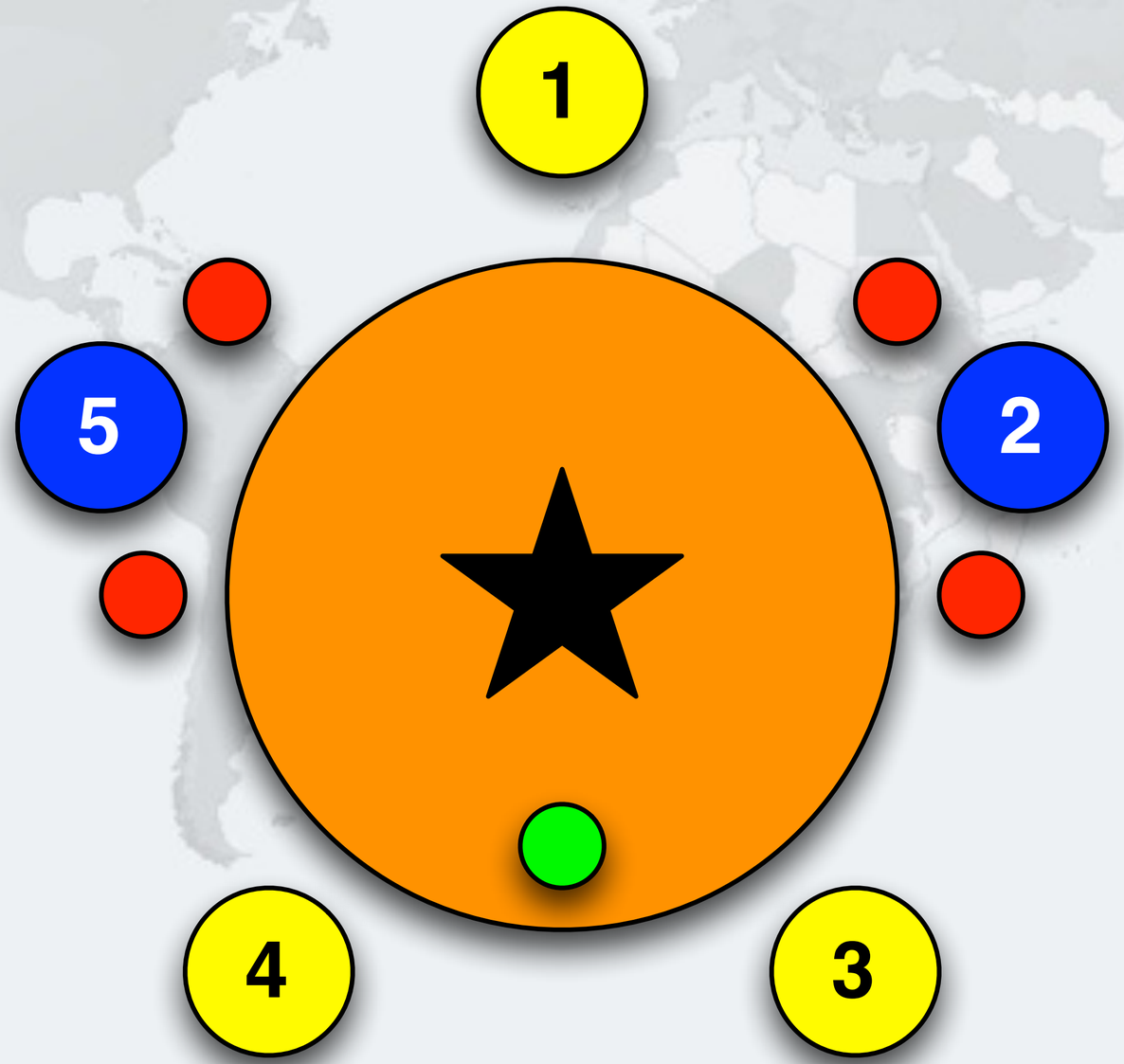
Philosopher 3 Finished Drinking, Returns Left Cup



Philosopher 3 Returns Right Cup



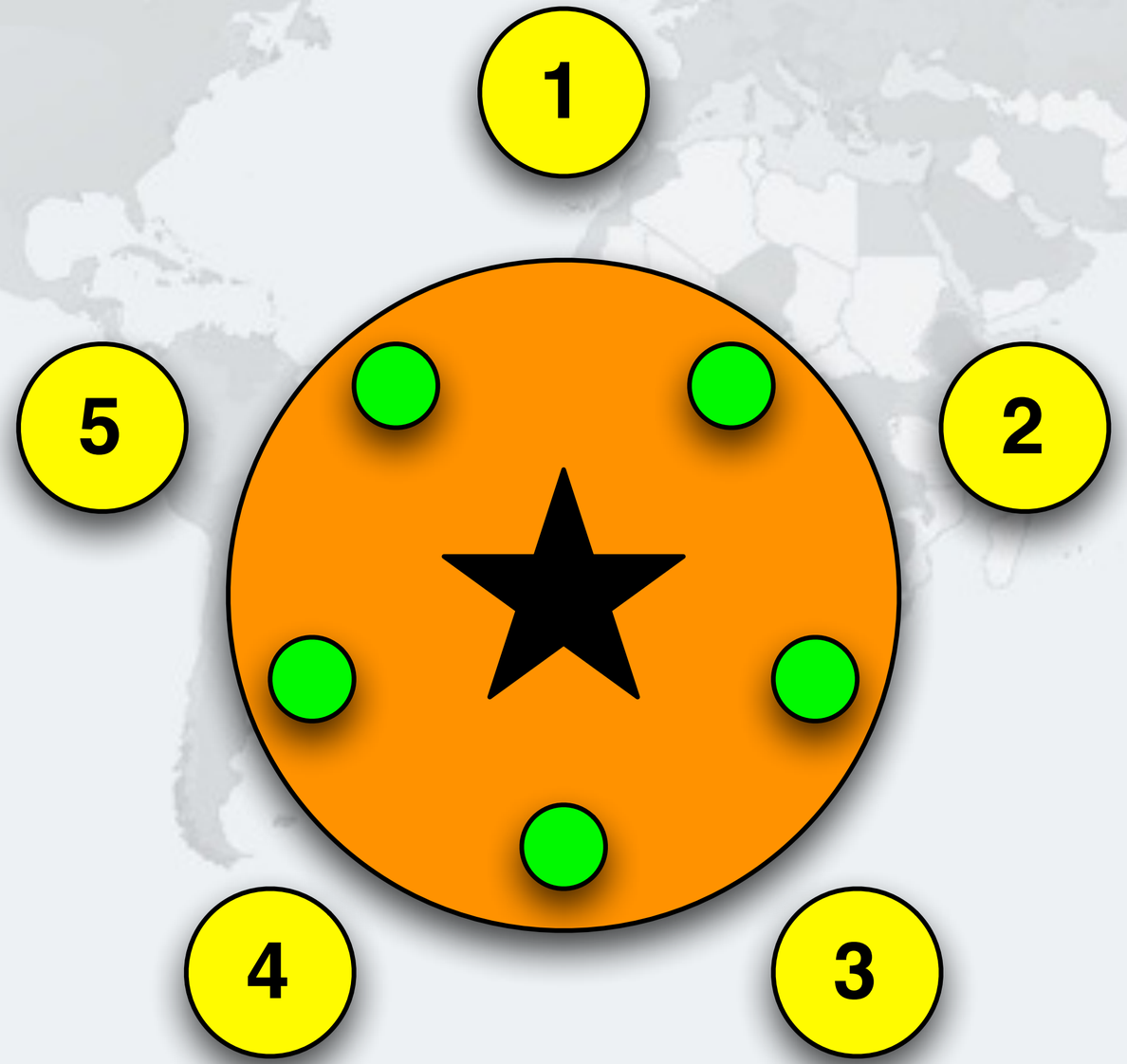
Philosopher 2 Is Now Drinking With Both Cups



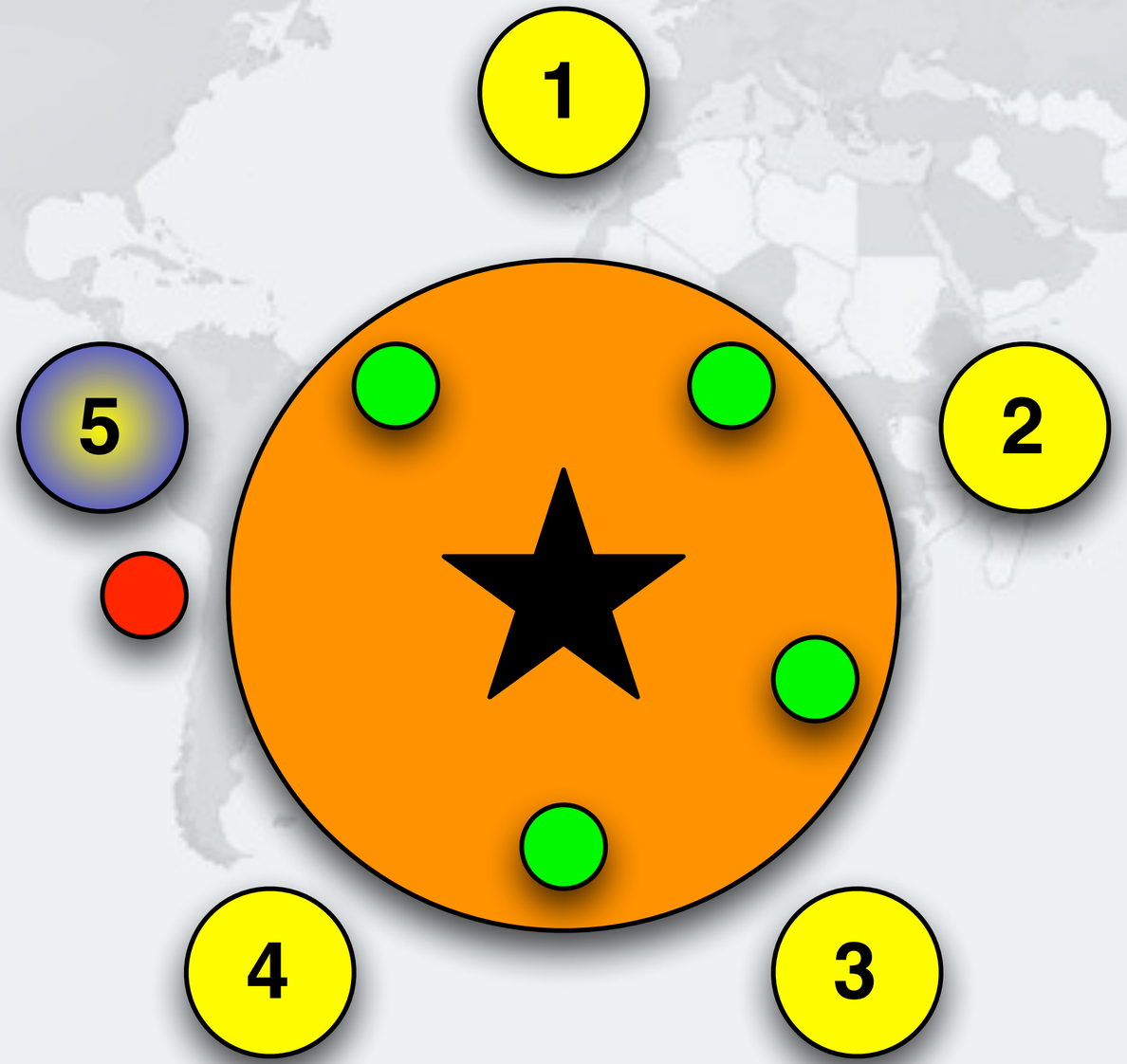
Drinking Philosophers In Limbo

- **The standard rule is that every philosopher first picks up the right cup, then the left**
  - **If all of the philosophers want to drink and they all pick up the right cup, then they all are holding one cup but cannot get the left cup**

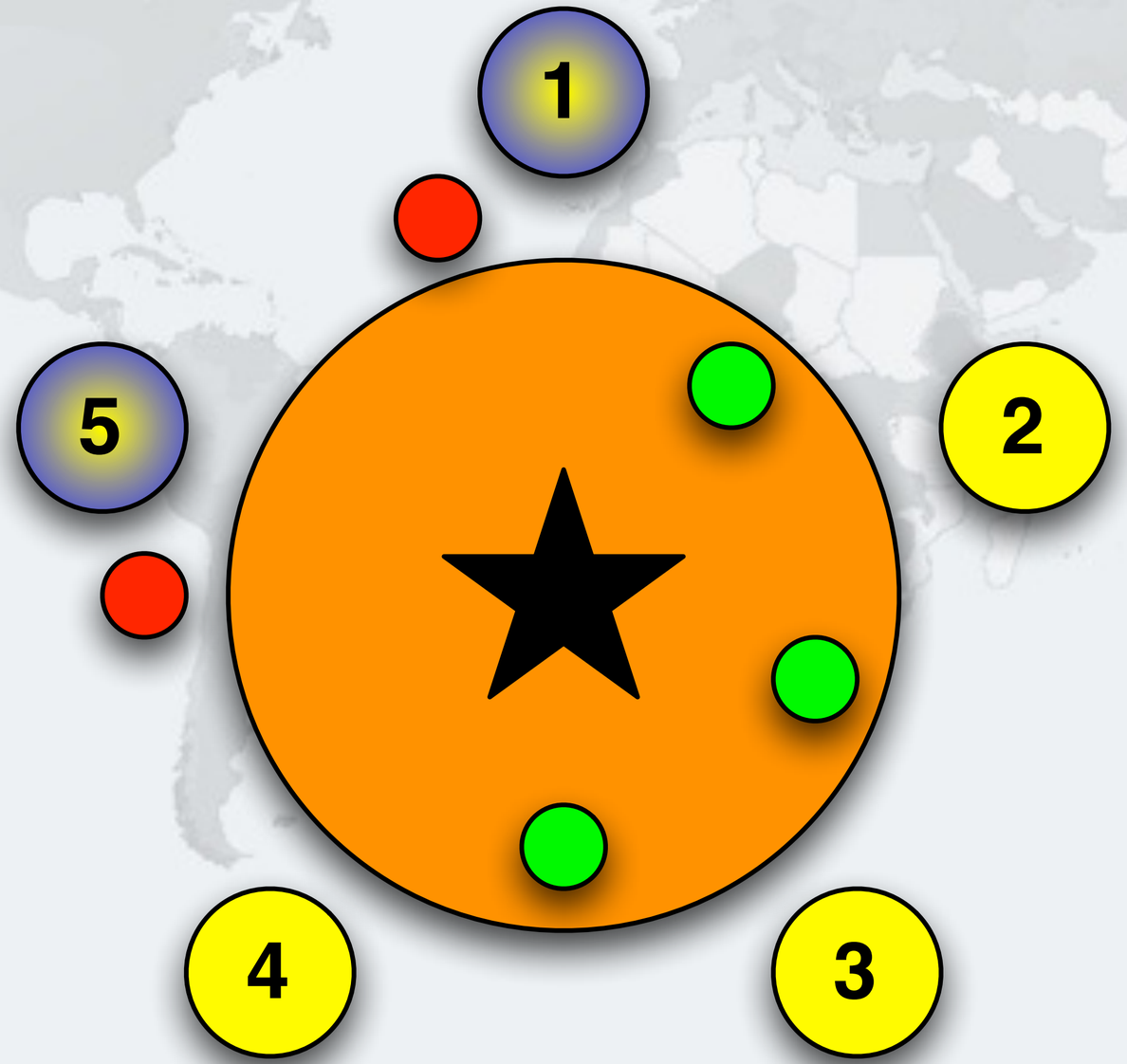
A Deadlock Can Easily Happen With This Design



Philosopher 5 Wants To Drink, Takes Right Cup

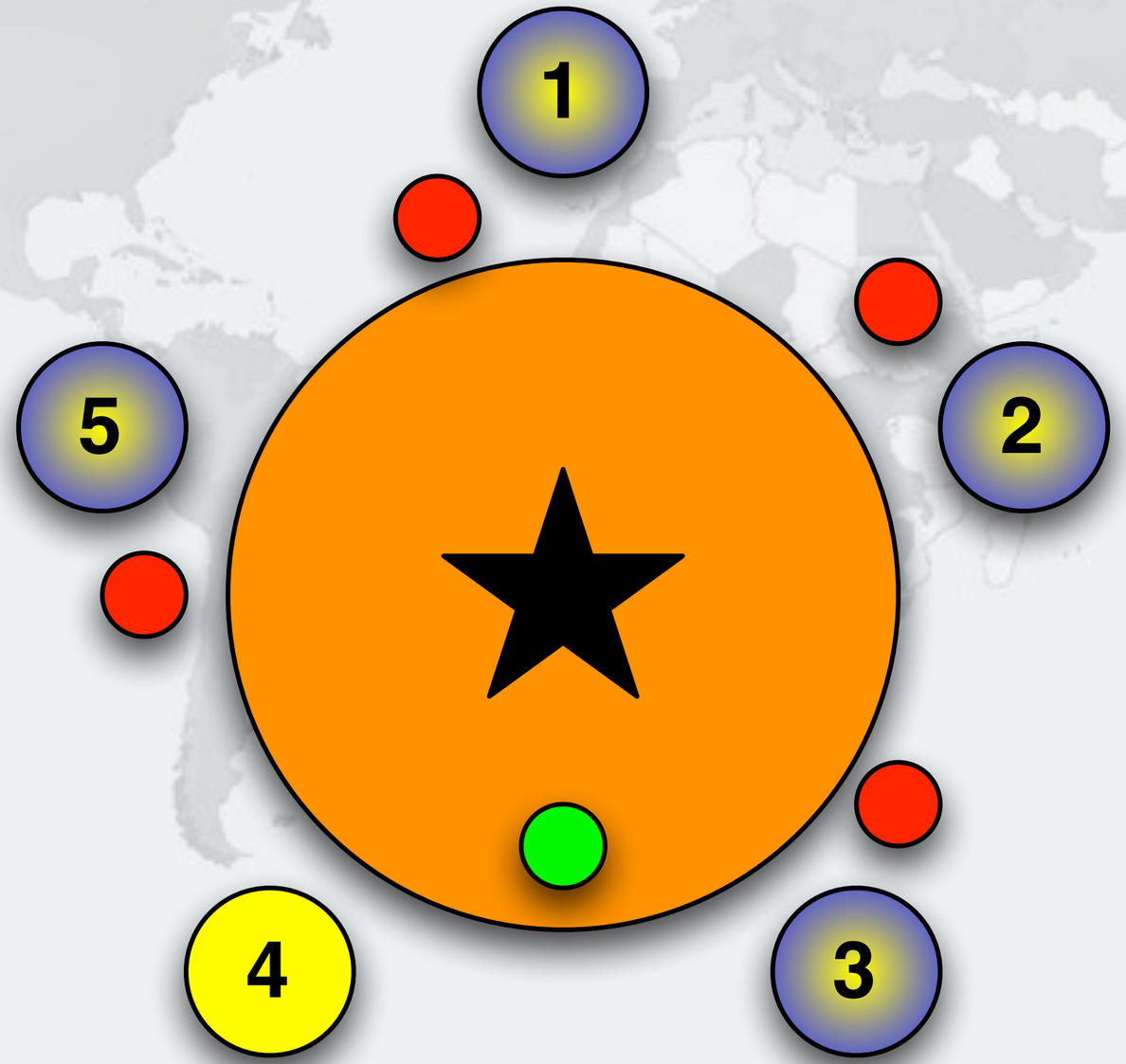


Philosopher 1 Wants To Drink, Takes Right Cup

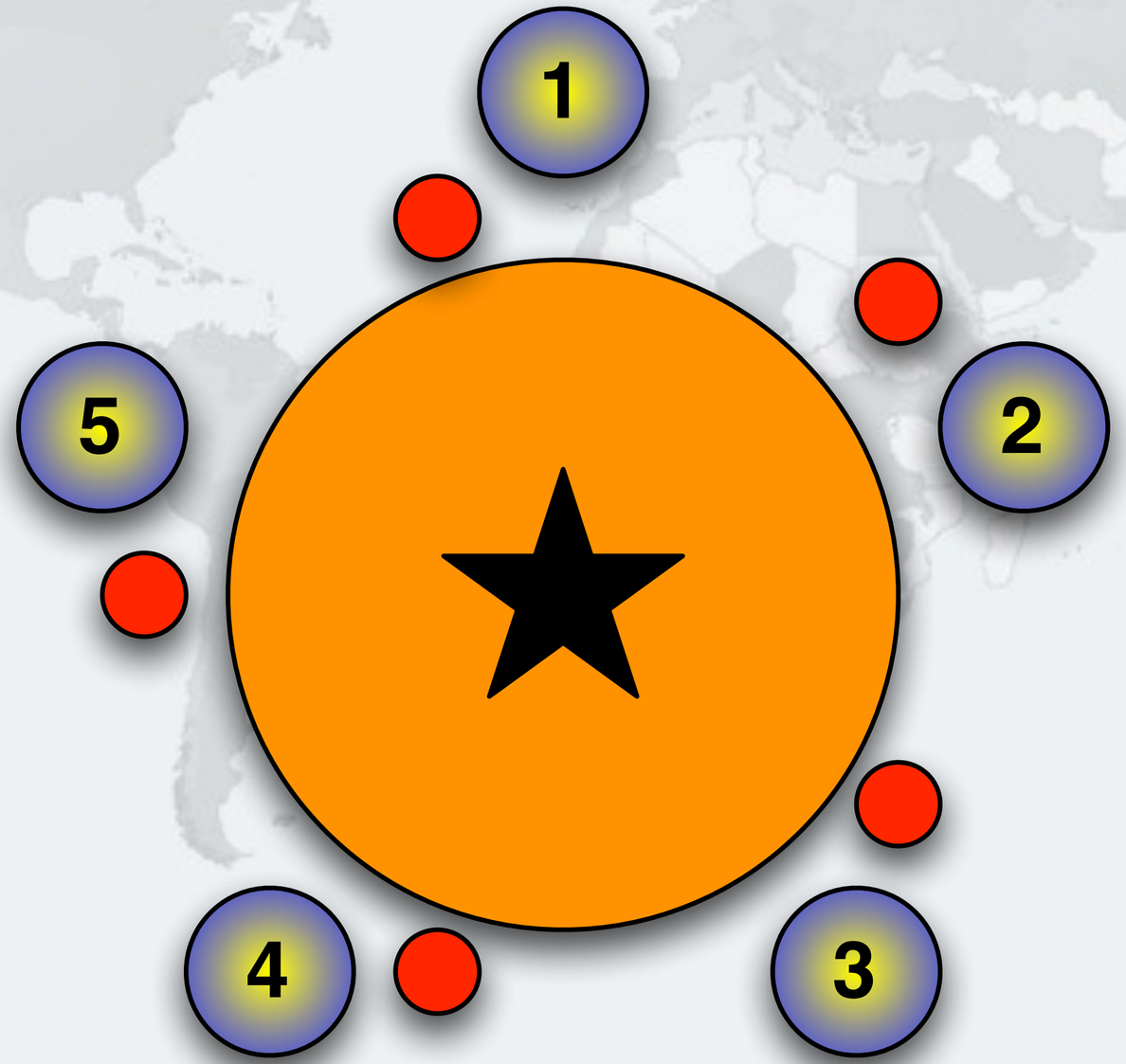


Philosopher 2 Wants To Drink, Takes Right Cup

Philosopher 3 Wants To Drink, Takes Right Cup

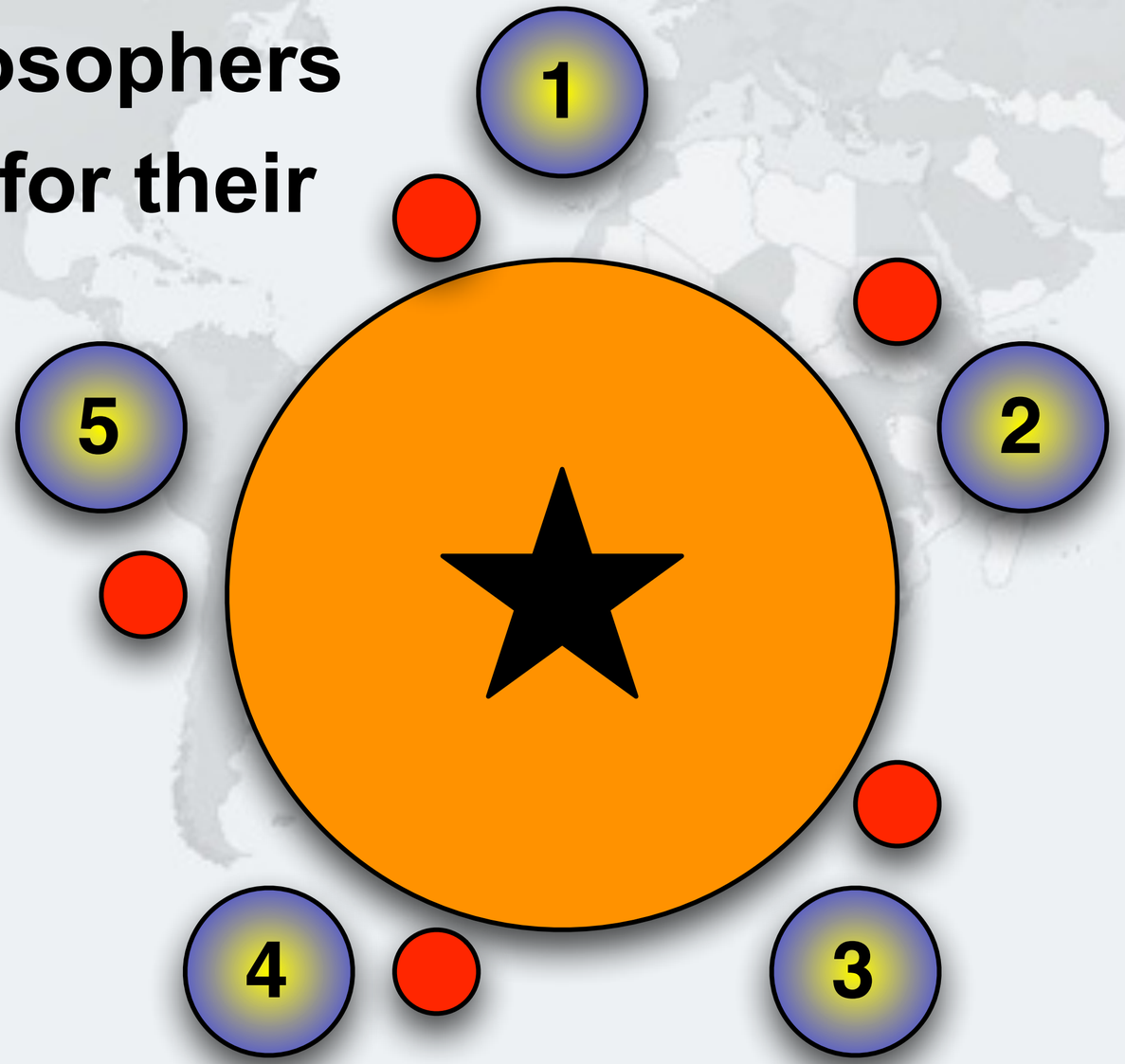


Philosopher 4 Wants To Drink, Takes Right Cup



Deadlock!

- **All the philosophers are waiting for their left cups, but they will never become available**



Global Order With Boozing Philosophers

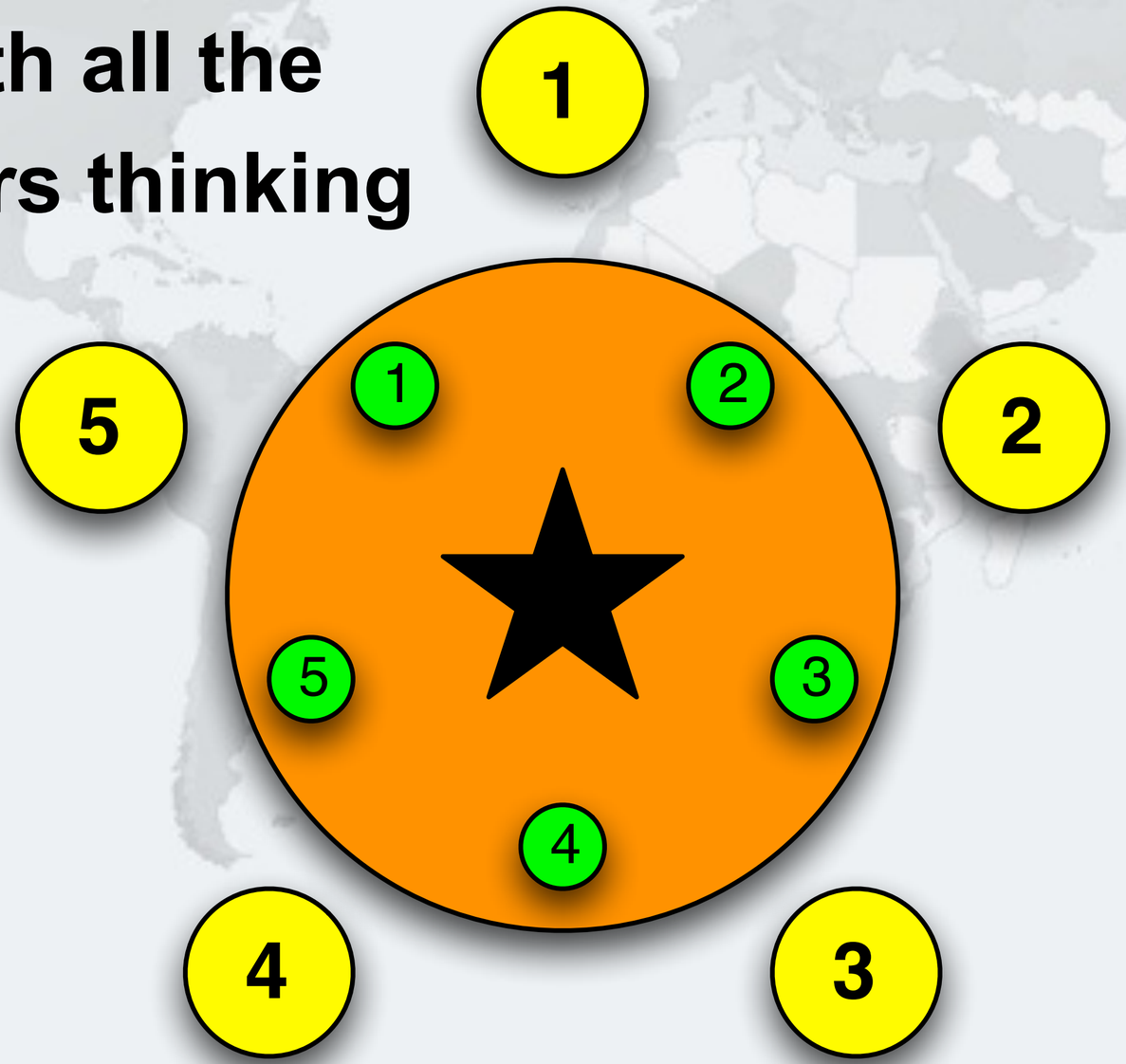
- **If all philosophers hold one cup, we deadlock**
  - **In our solution, we have to prevent that from happening**

Fixed Order Of Lock Acquisition

- **We can solve the deadlock with the "dining philosophers" by requiring that locks are always acquired in a set order**
  - **For example, we can make a rule that philosophers always first take the cup with the largest number**
    - **If it is not available, we block until it becomes available**
  - **And return the cup with the lowest number first**

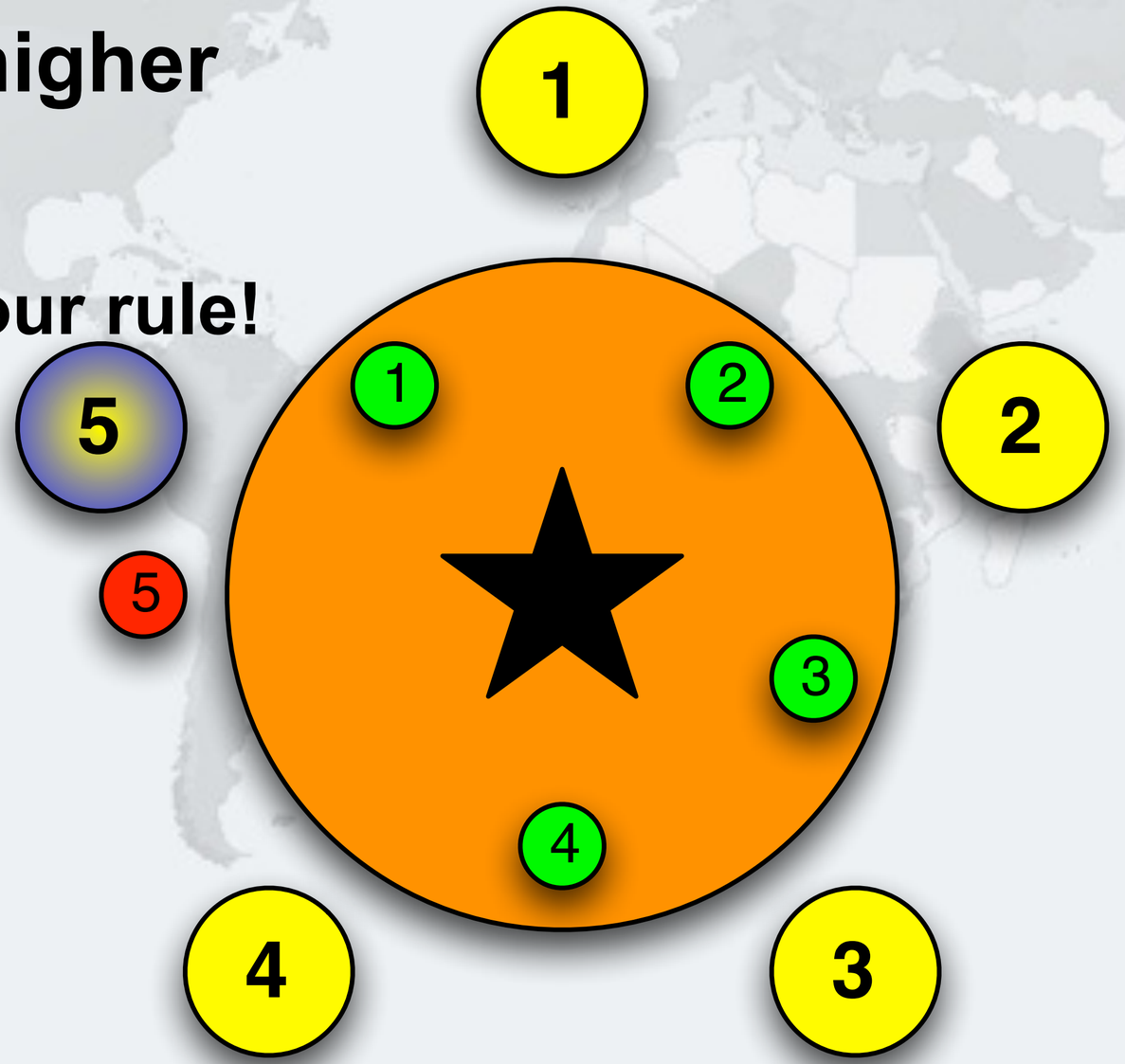
Global Lock Ordering

- **We start with all the philosophers thinking**



Philosopher 5 Takes Cup 5

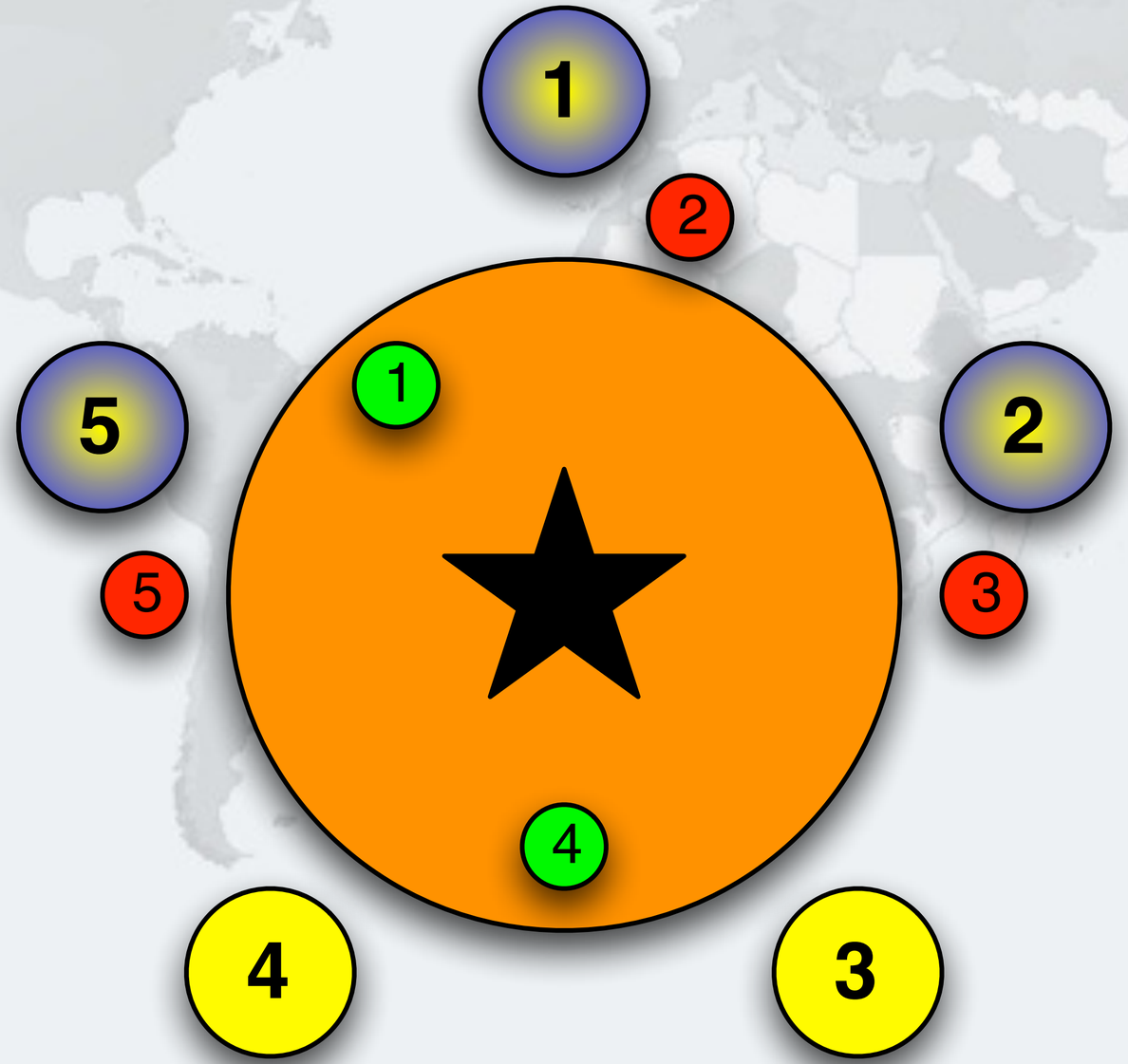
- **Cup 5 has higher number**
  - **Remember our rule!**



Philosopher 1 Takes Cup 2

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

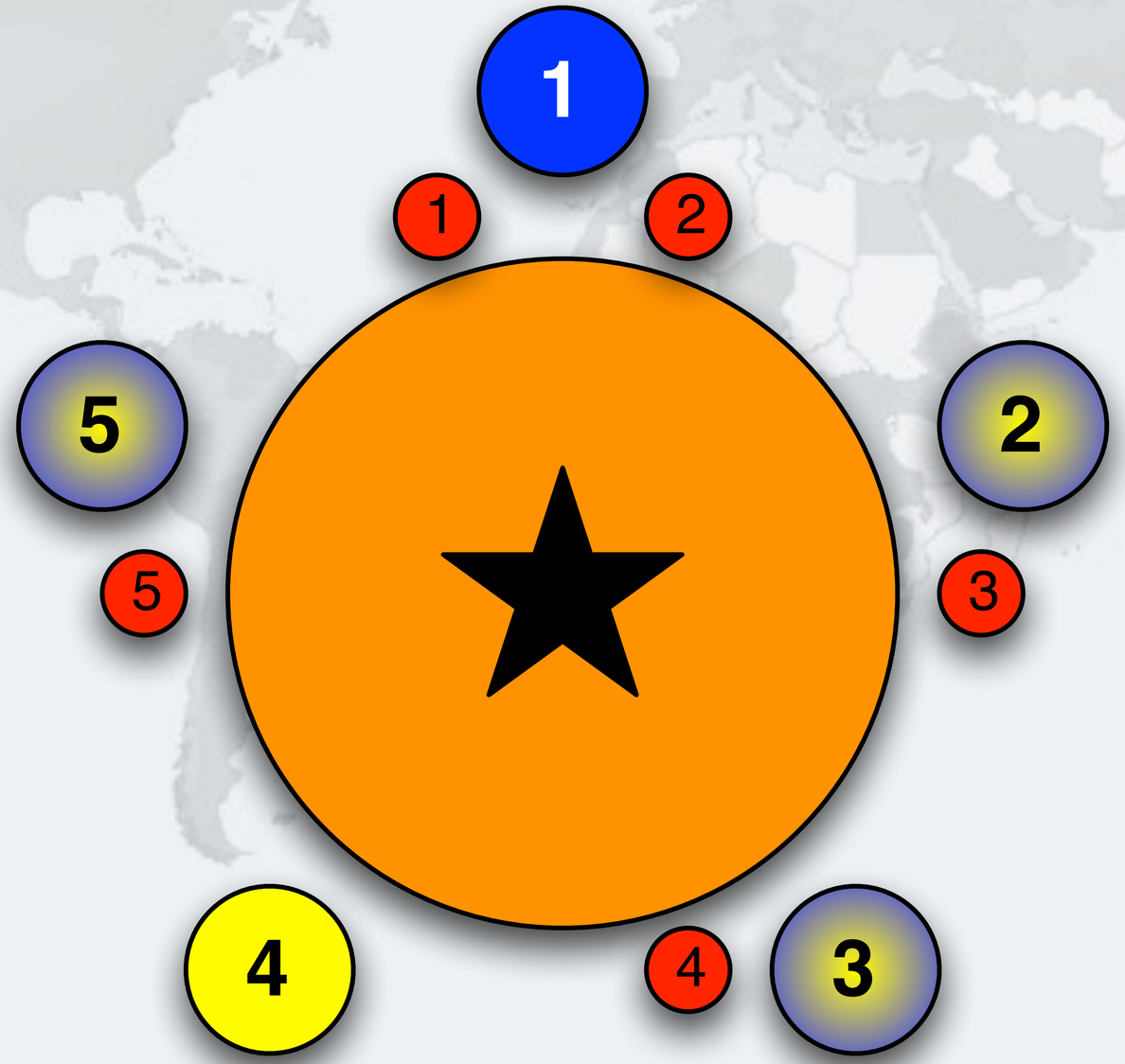
Philosopher 2 Takes Cup 3



Philosopher 3 Takes Cup 4

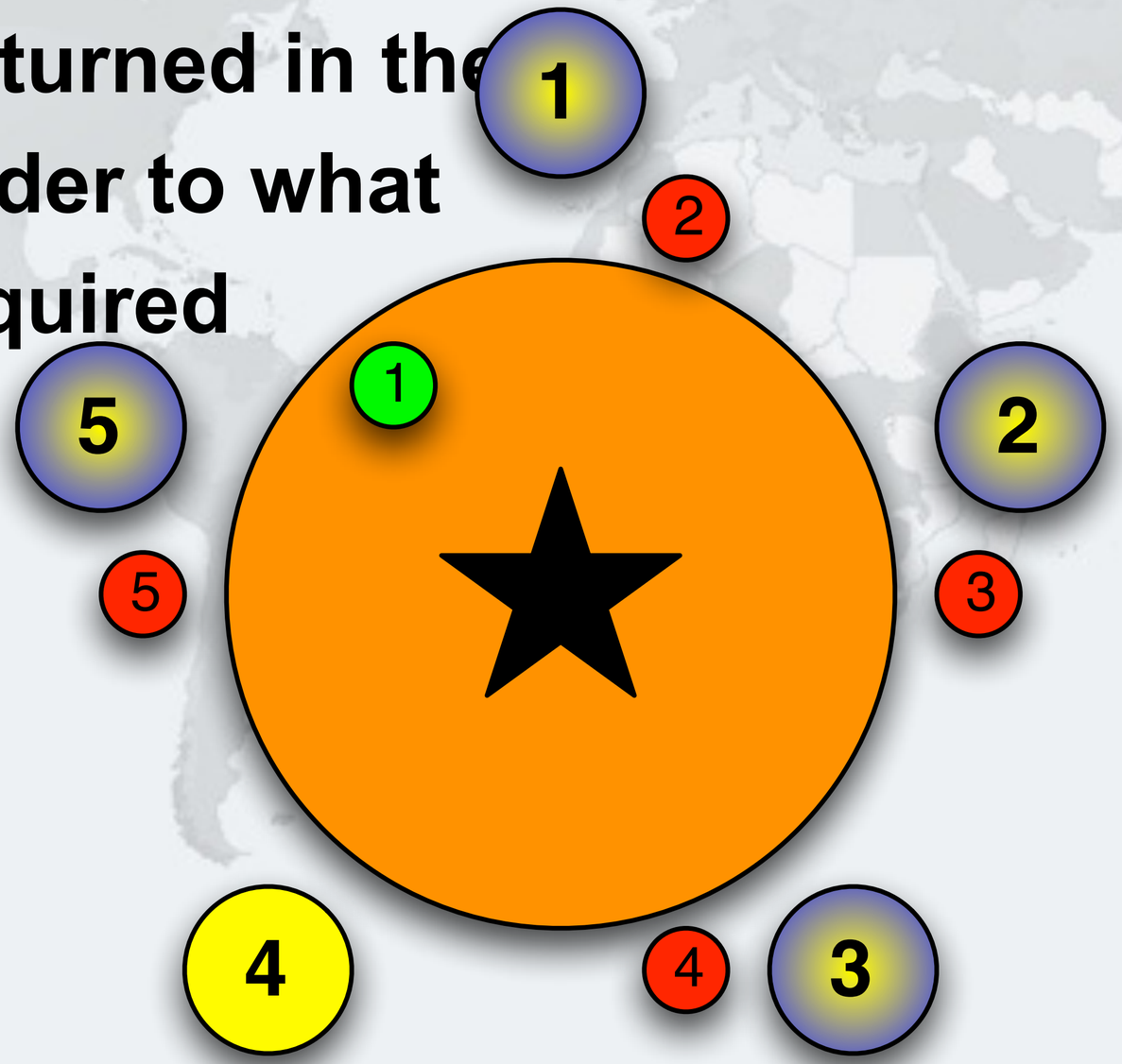
- **Note that philosopher 4 is prevented from holding one cup**

Philosopher 1 Takes Cup 1 - Drinking

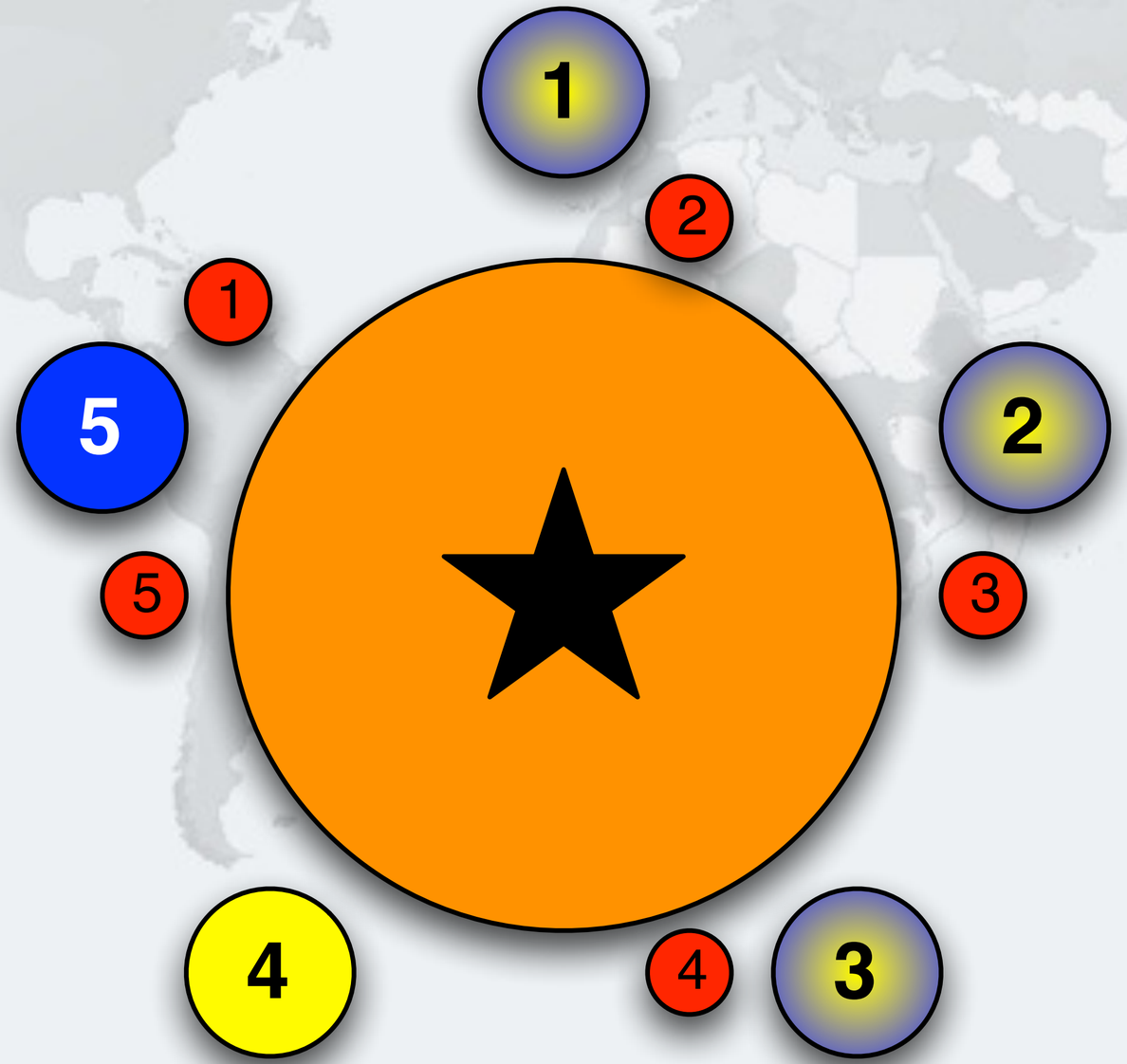


Philosopher 1 Returns Cup 1

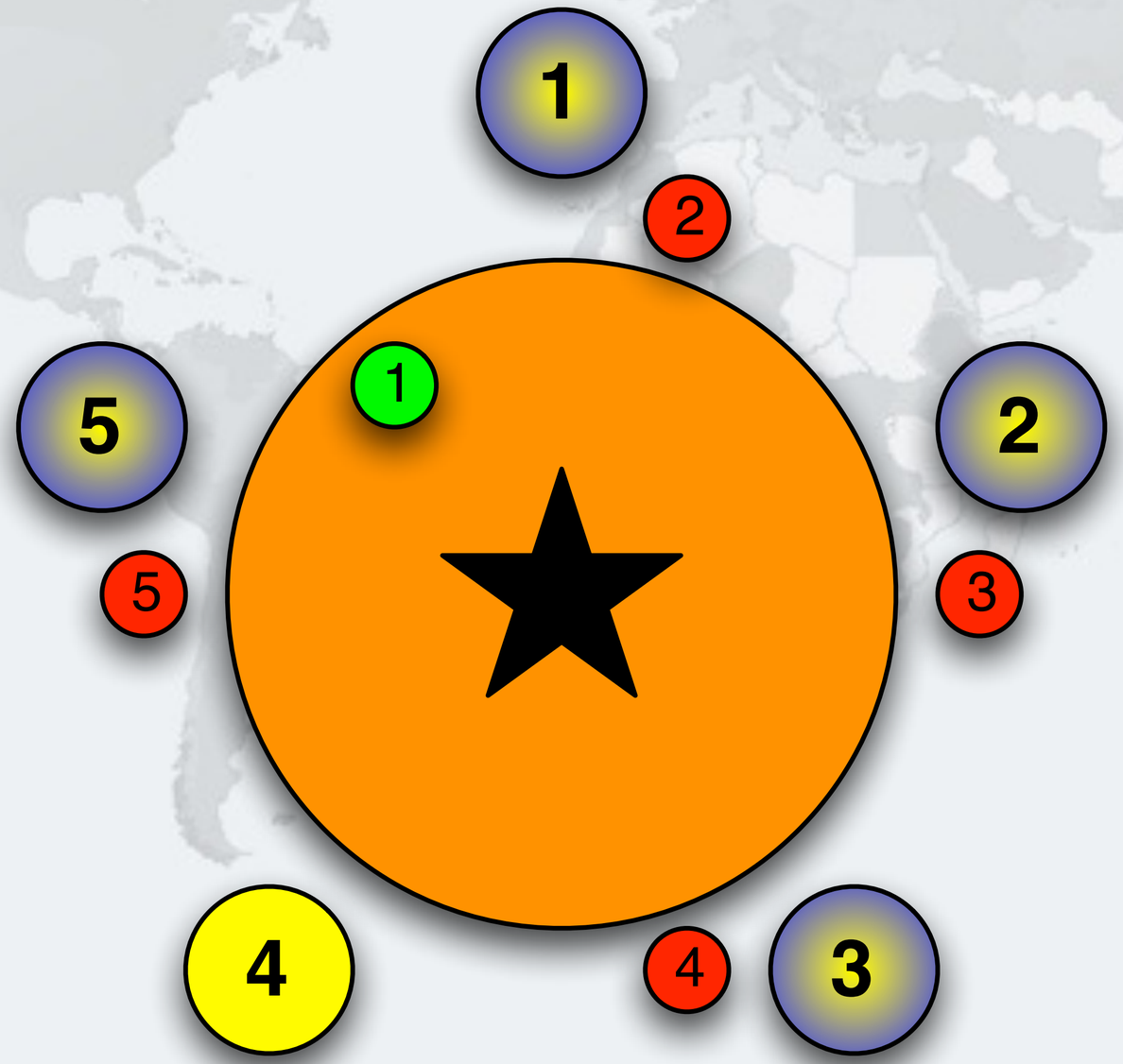
- Cups are returned in the opposite order to what they are acquired



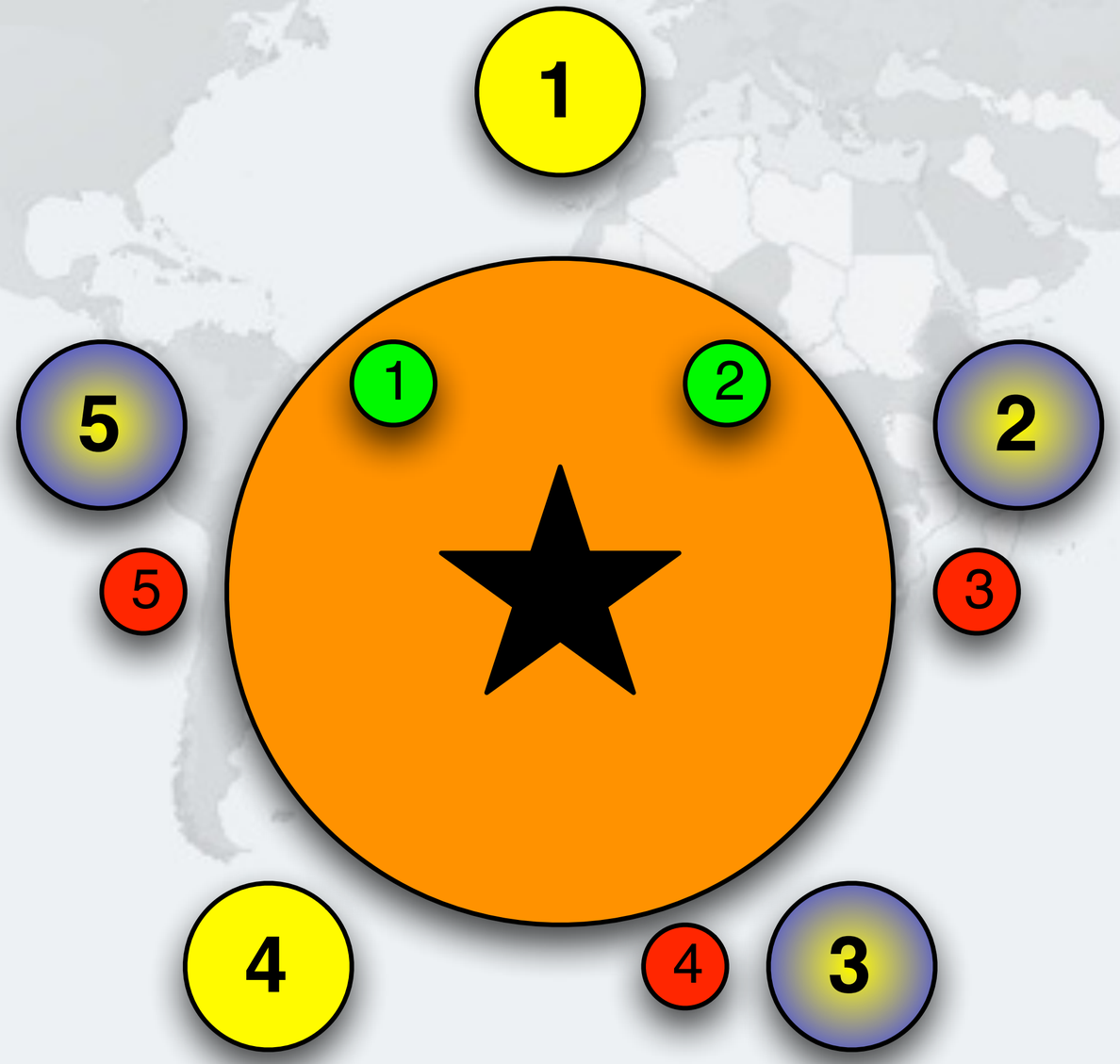
Philosopher 5 Takes Cup 1 - Drinking



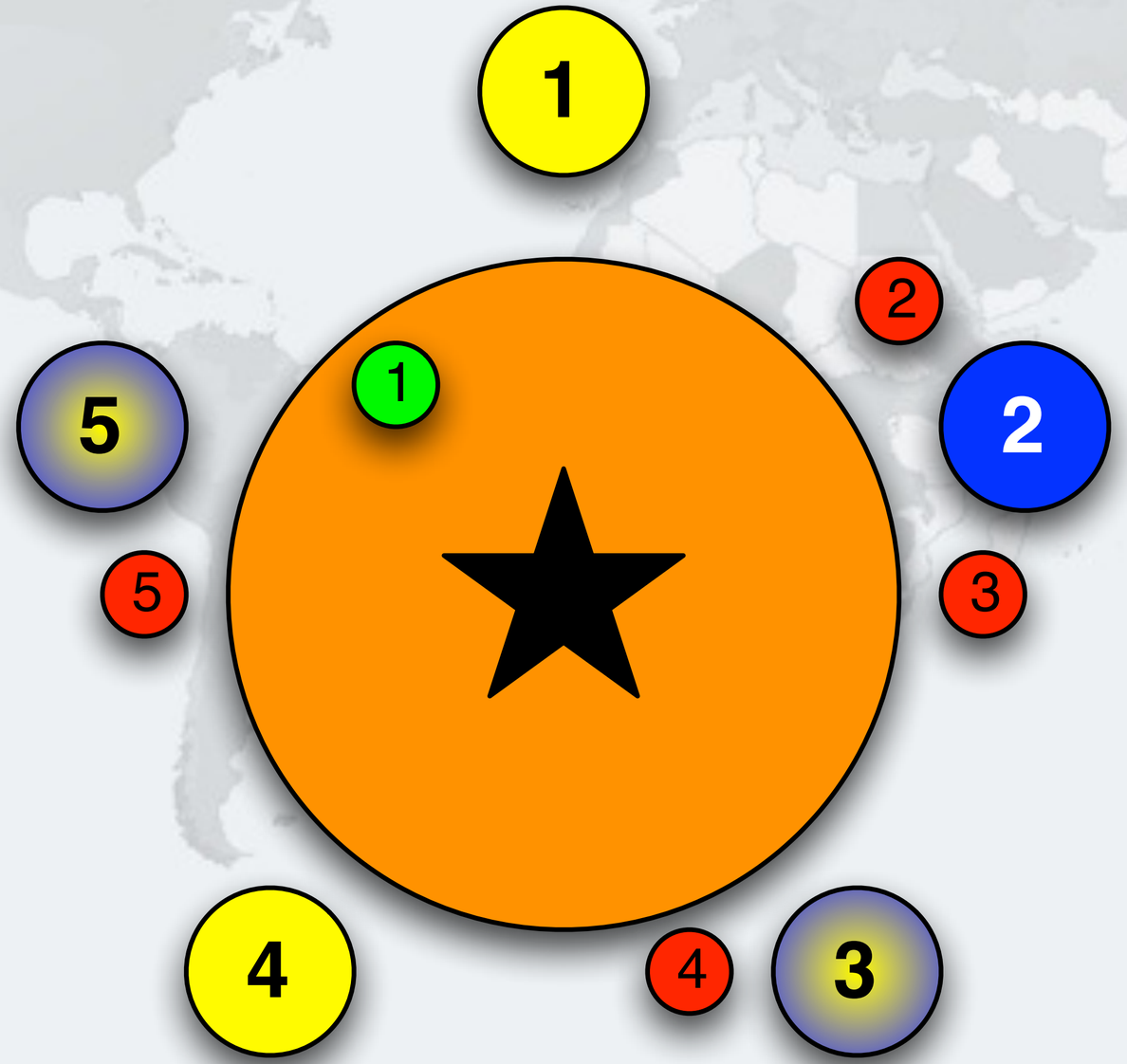
Philosopher 5 Returns Cup 1



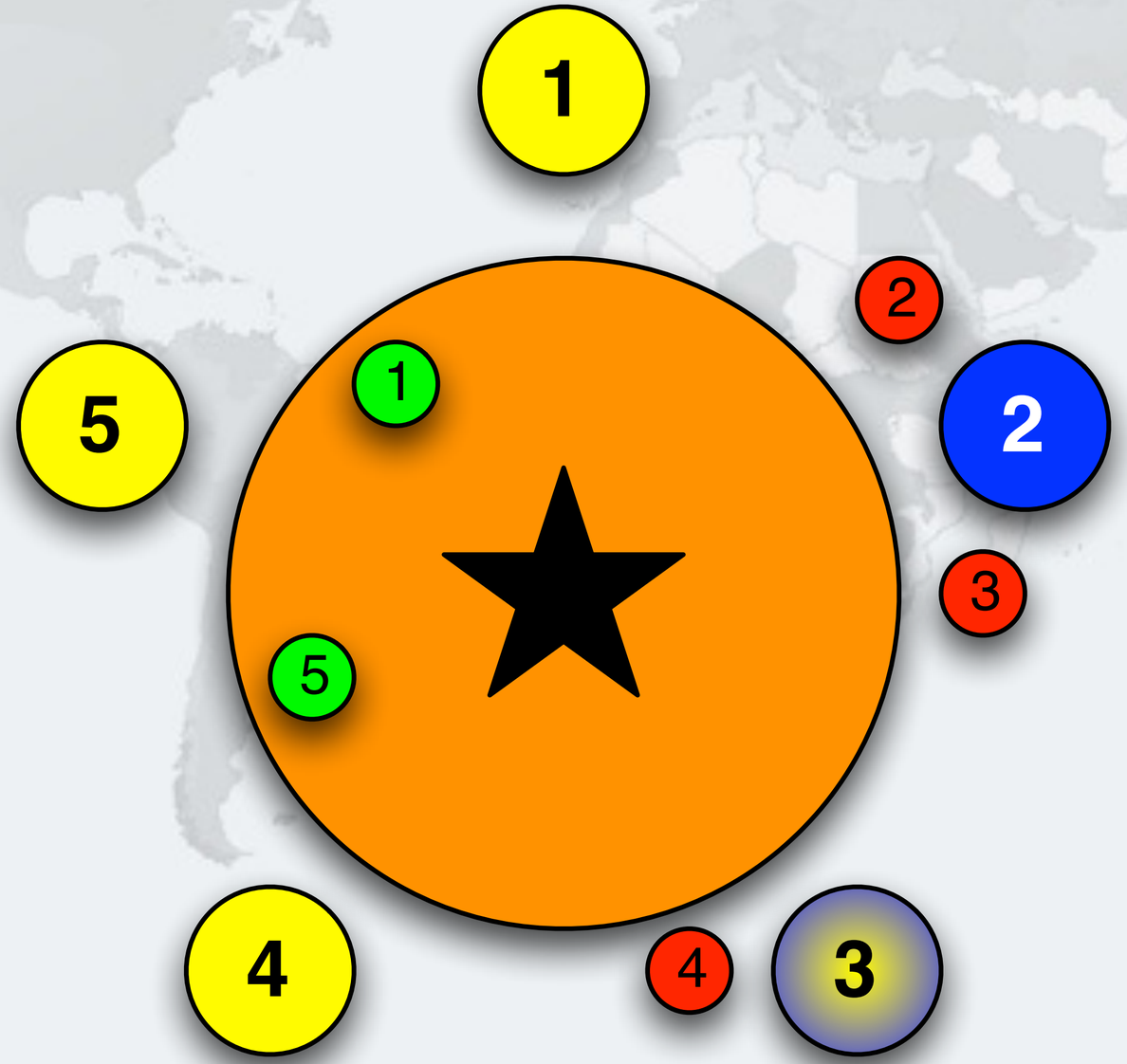
Philosopher 1 Returns Cup 2



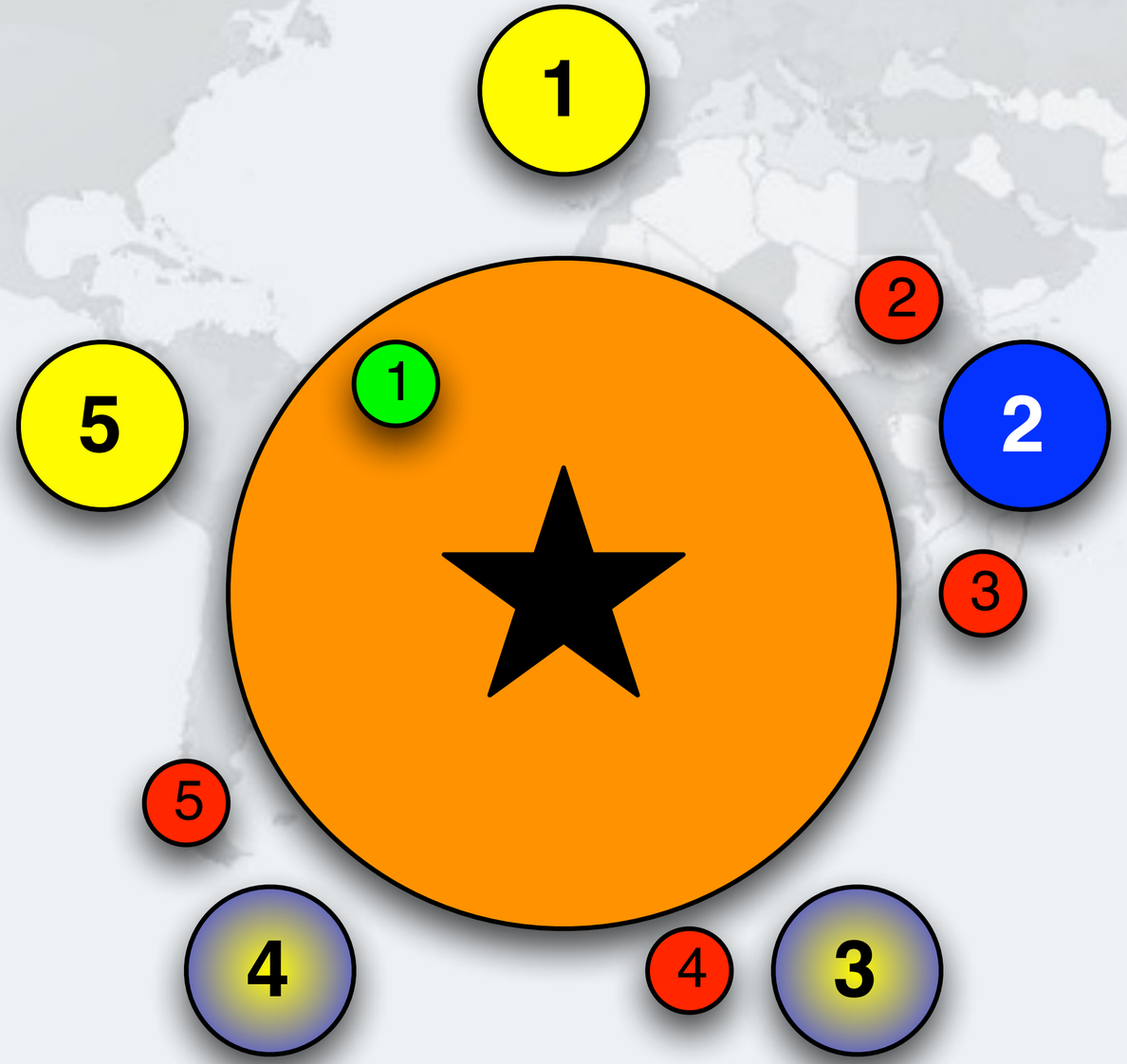
Philosopher 2 Takes Cup 2 - Drinking



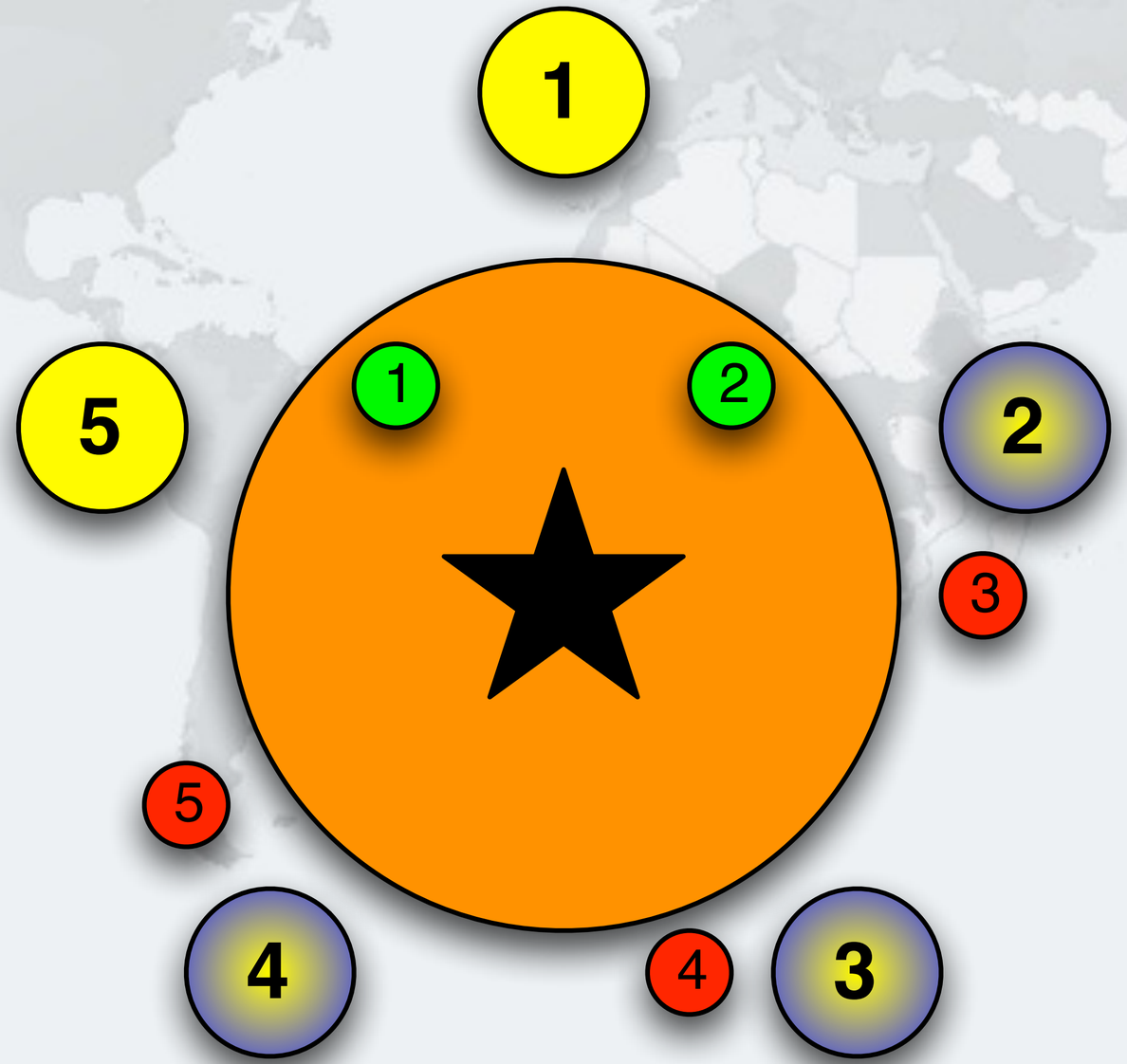
Philosopher 5 Returns Cup 5



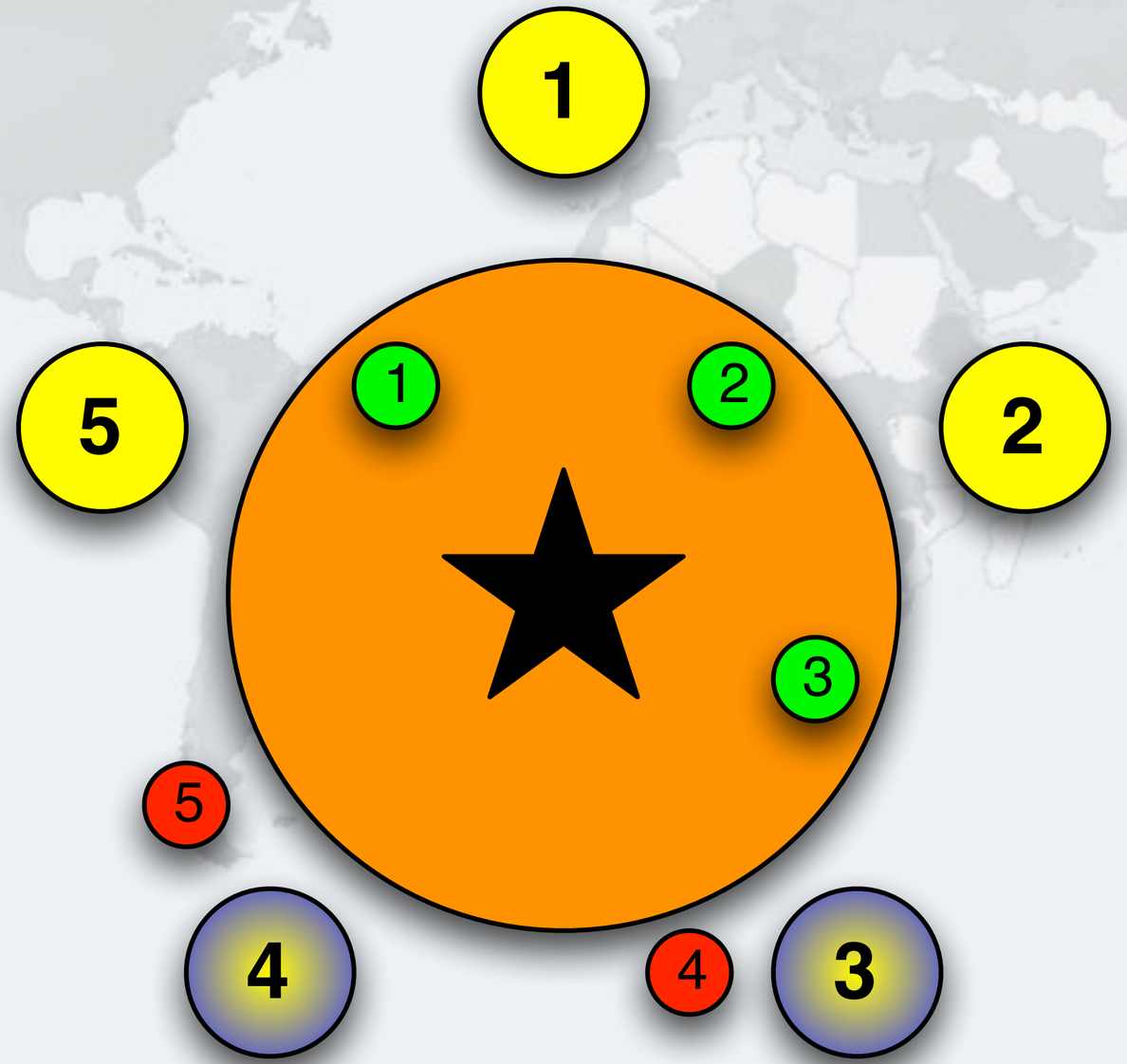
Philosopher 4 Takes Cup 5



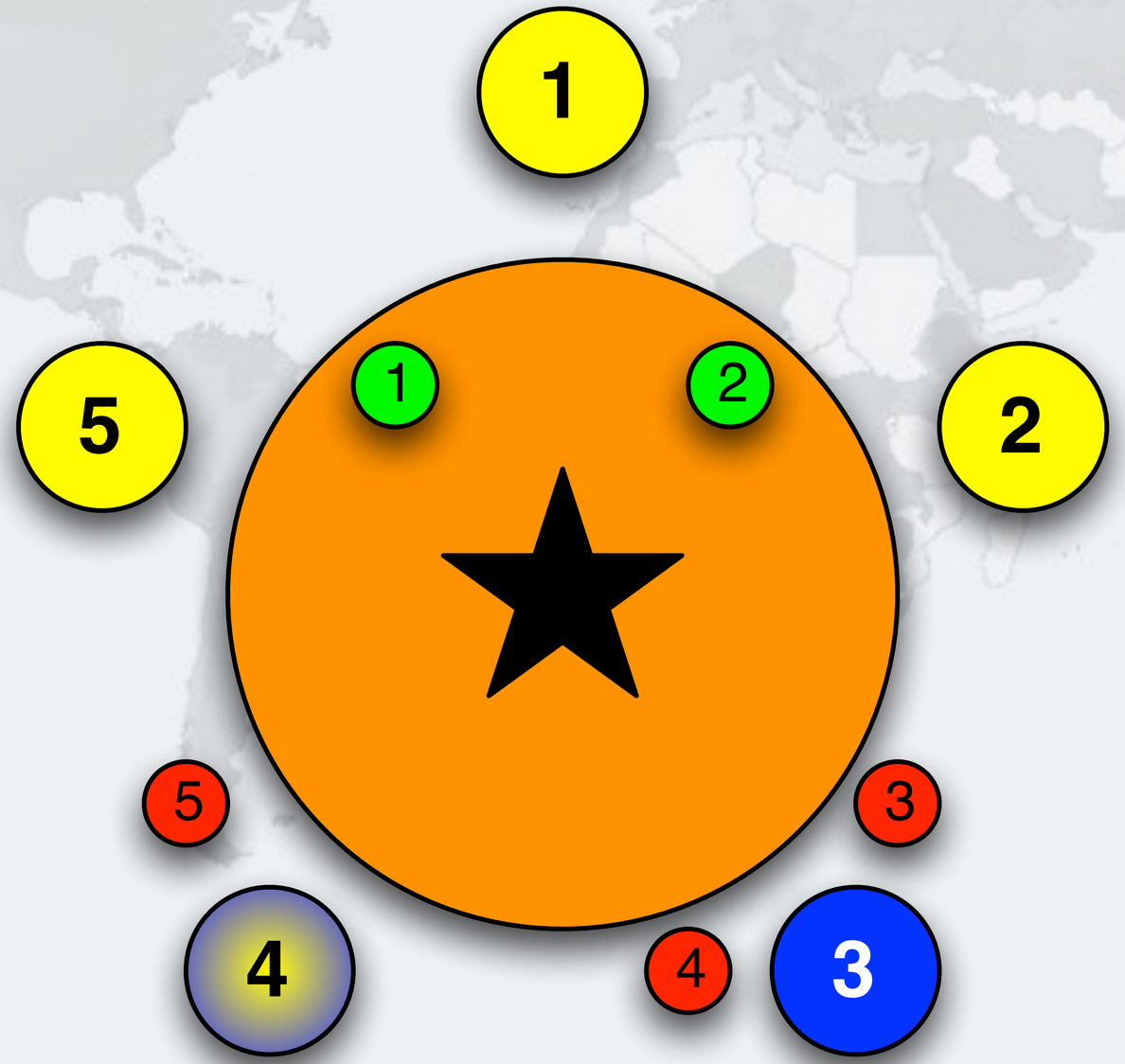
Philosopher 2 Returns Cup 2



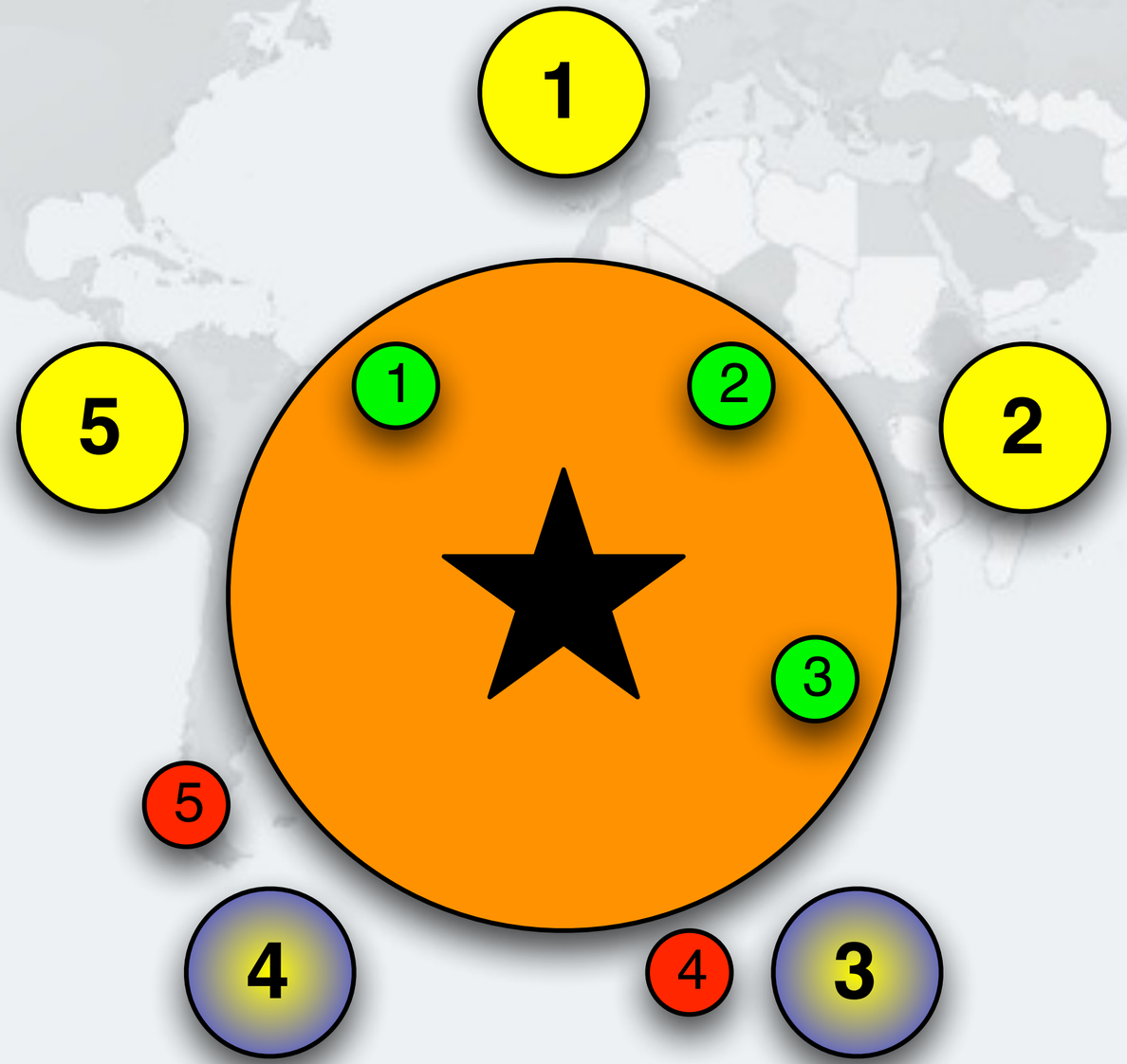
Philosopher 2 Returns Cup 3



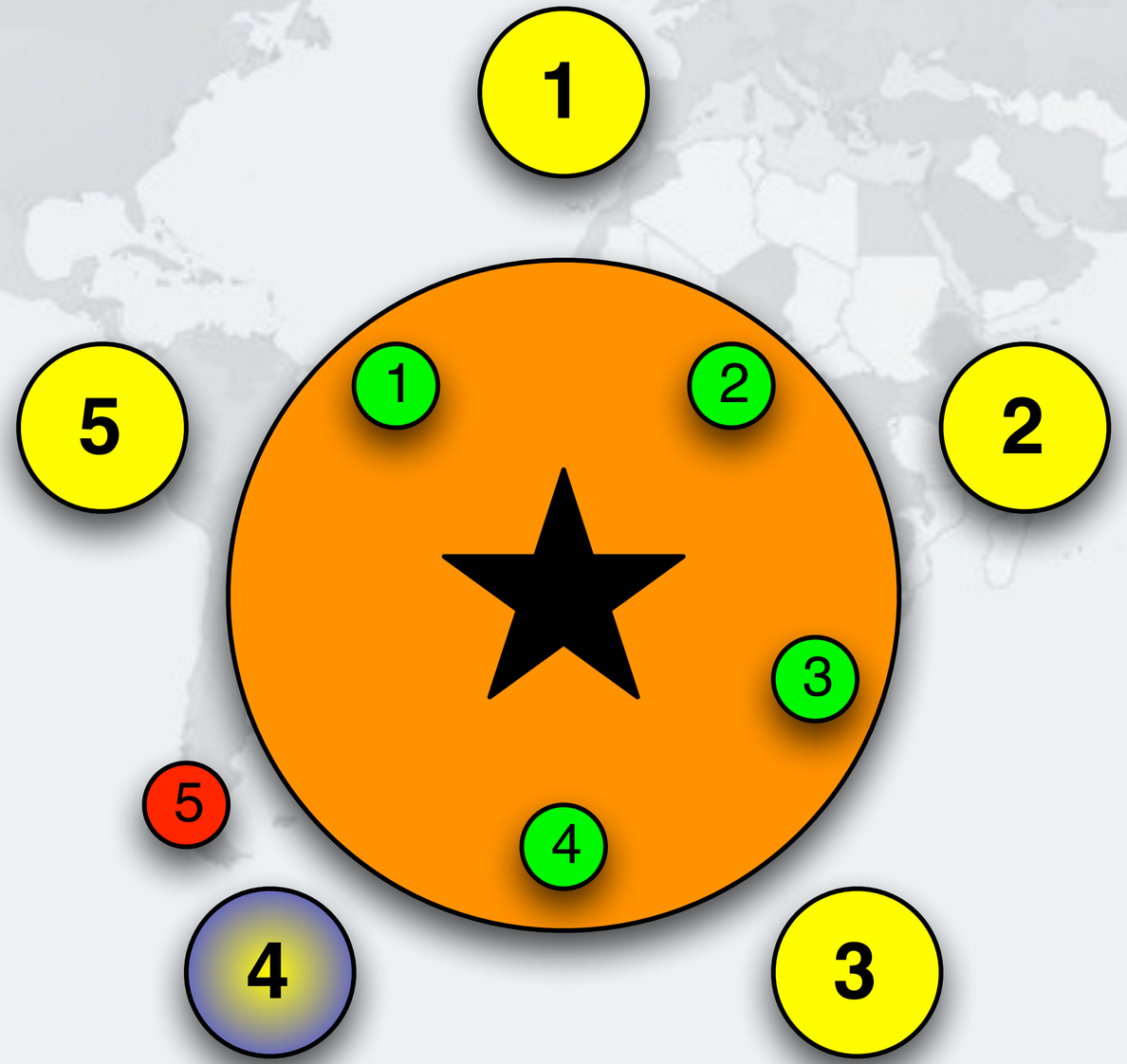
Philosopher 3 Takes Cup 3 - Drinking



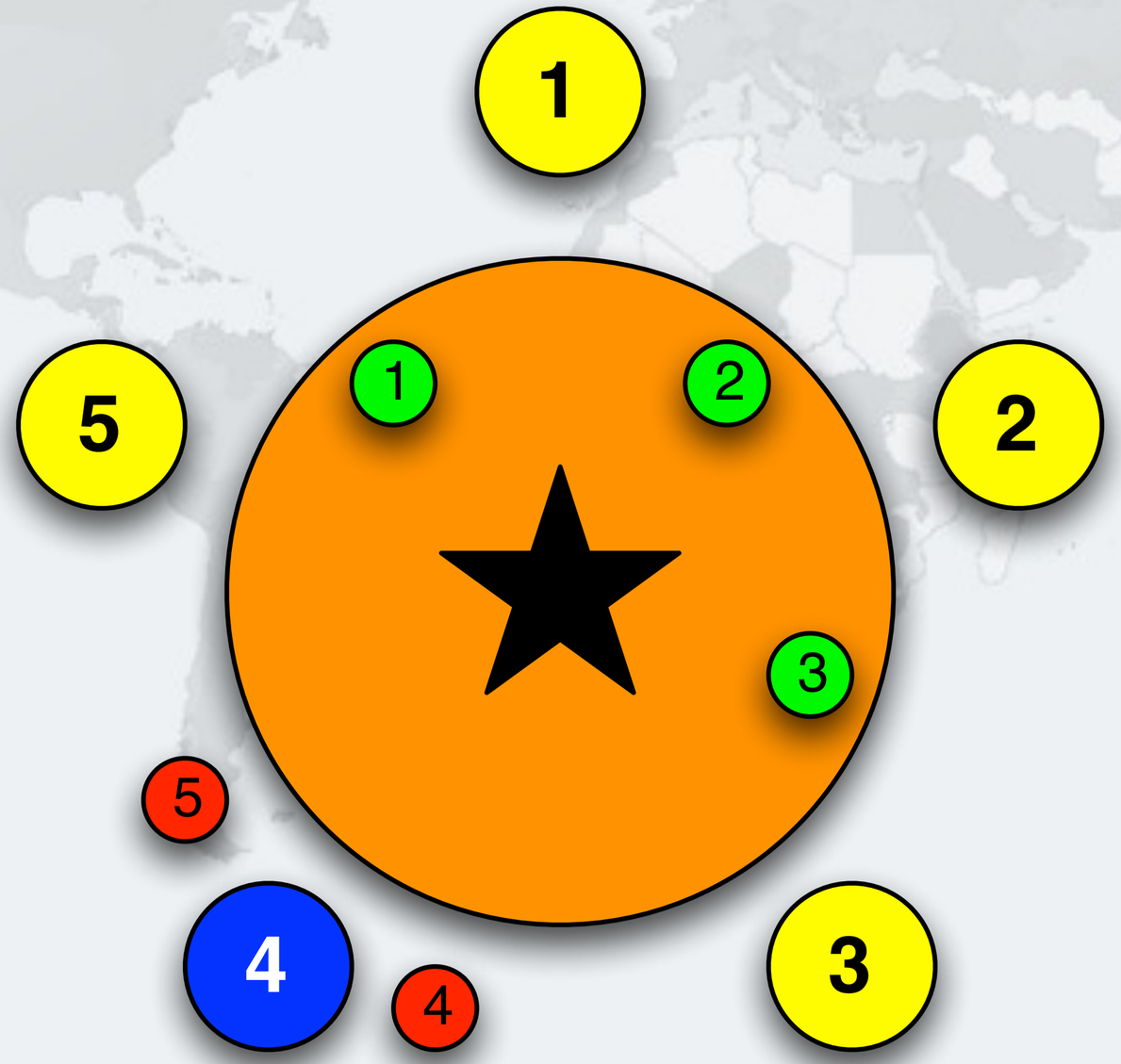
Philosopher 3 Returns Cup 3



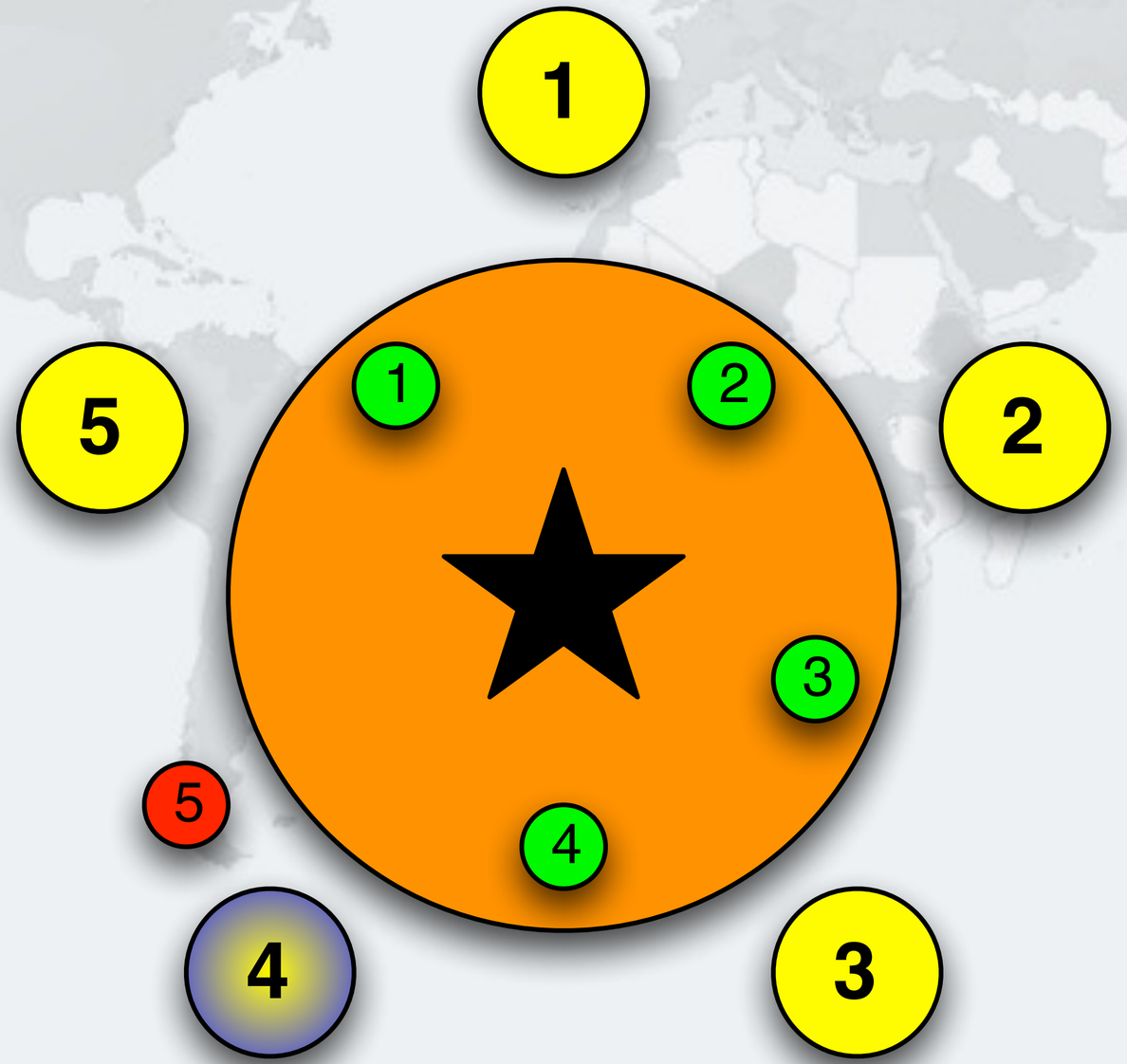
Philosopher 3 Returns Cup 4



Philosopher 4 Takes Cup 4 - Drinking

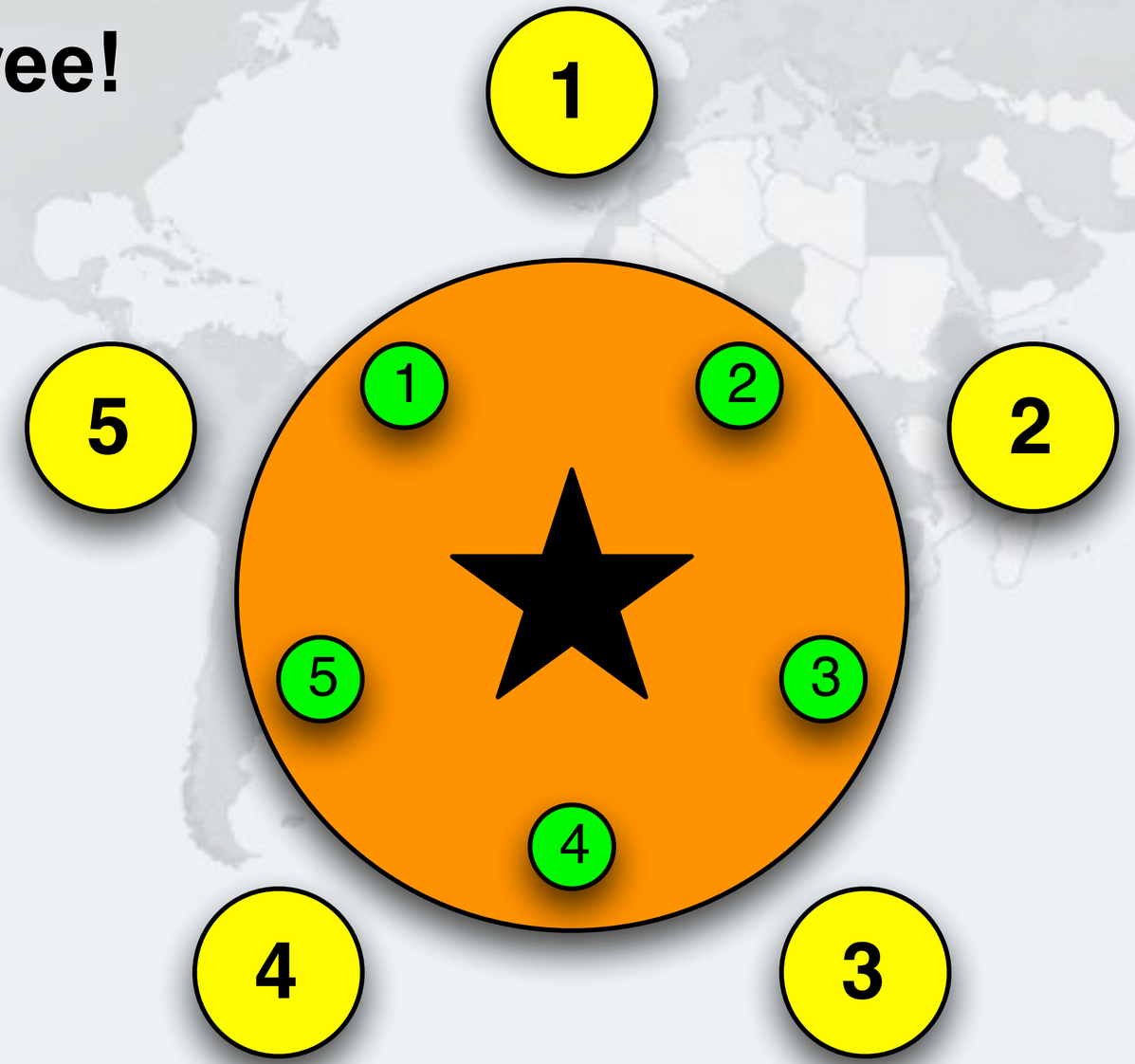


Philosopher 4 Returns Cup 4



Philosopher 4 Returns Cup 5

- **Deadlock free!**

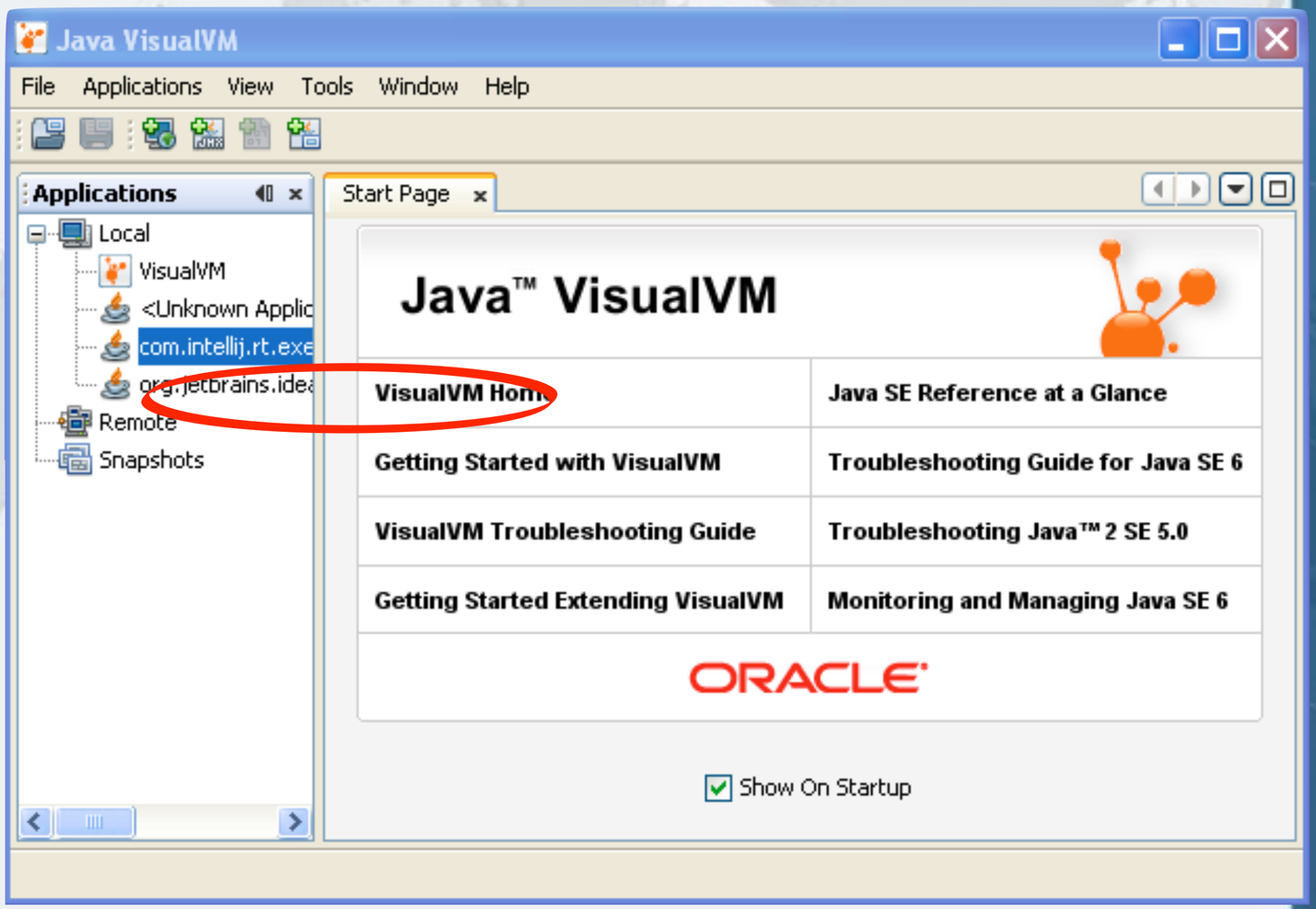


Deadlock Is Avoided

- **Impossible for all philosophers to hold one cup**

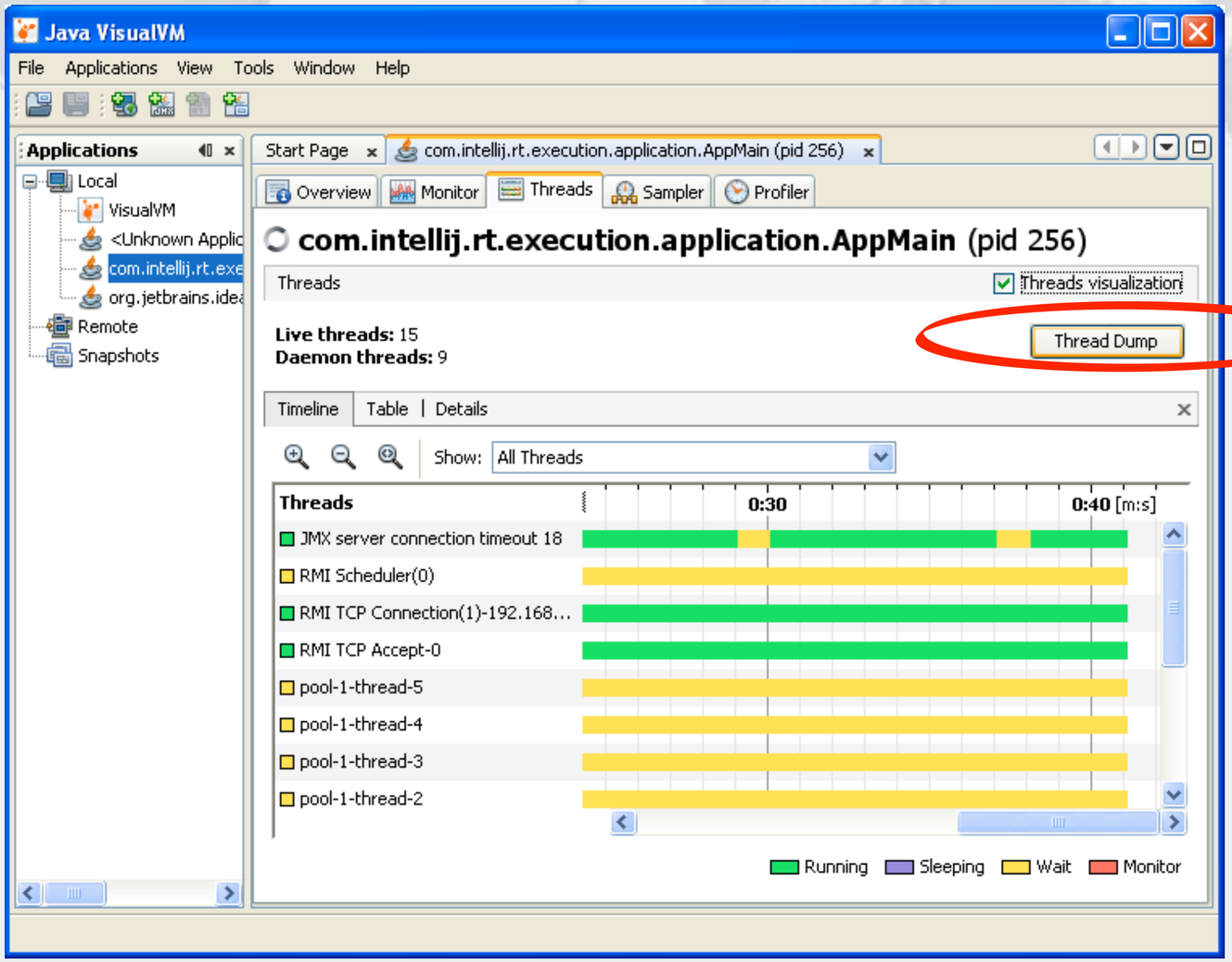
Capturing A Stack Trace

- **JVisualVM is a tool for monitoring what the JVM is doing**
  - **Found in the JDK/bin directory**
  - **Double-click on application**

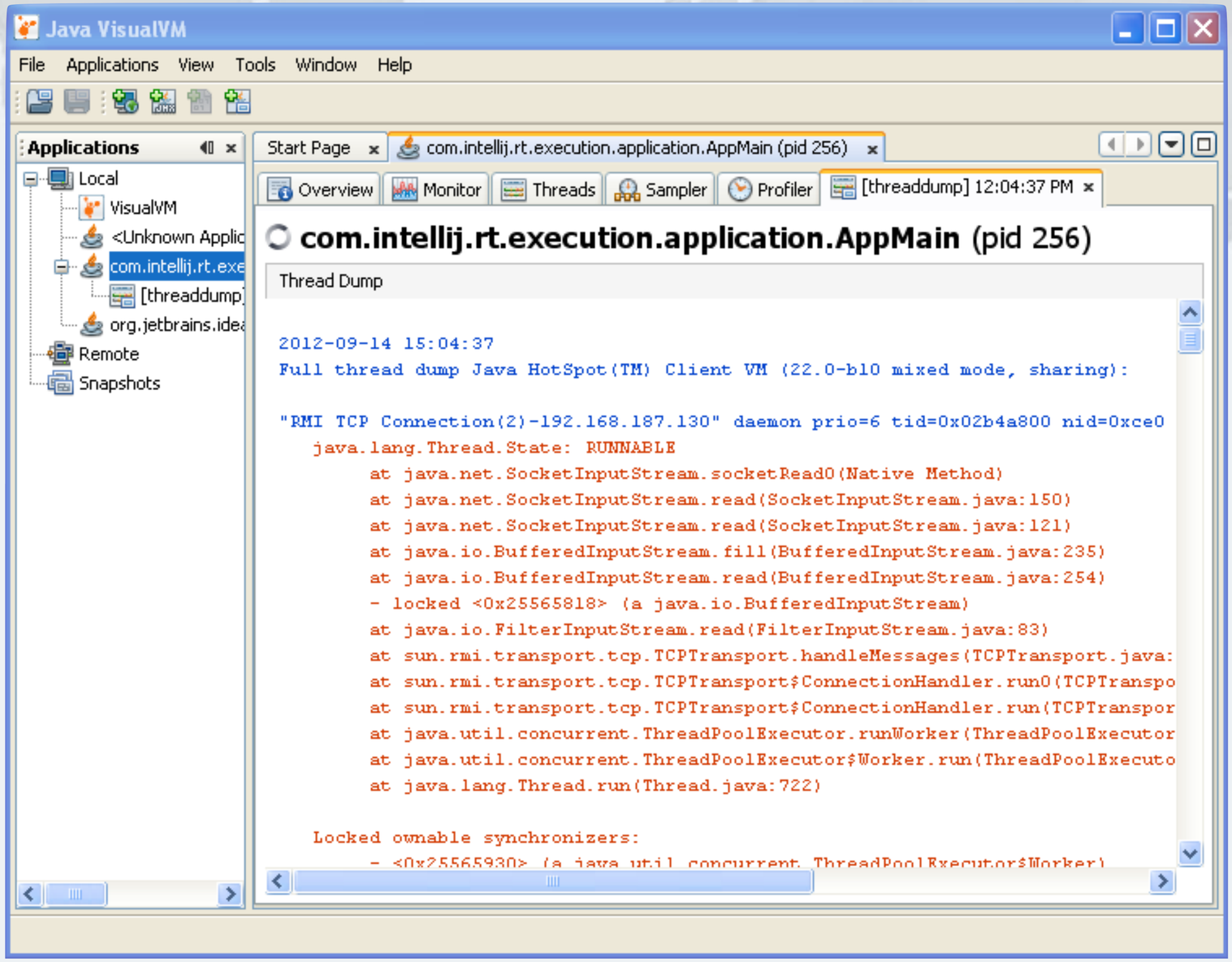


Click On "Threads" Tab

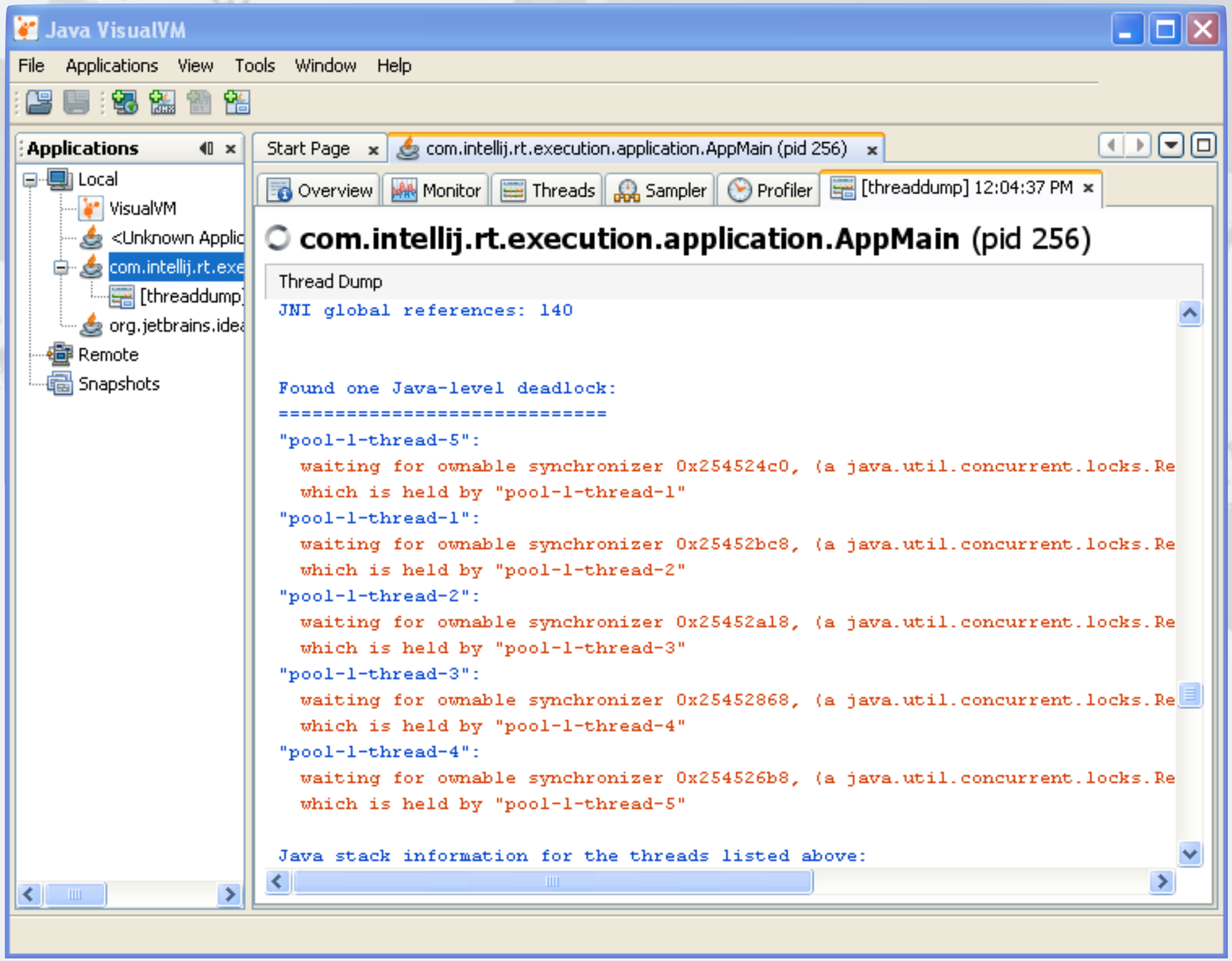
- **Click on "Thread Dump" button**



Stack Trace Shows What Threads Are Doing



It Can Even Detect A Java-Level Deadlock



Tools jstack and jps

- **For the hardcode geek, we have command line tools**
  - **jps**
    - **shows your Java process ids**
  - **jstack pid**
    - **shows what your JVM is currently doing**
  - **Tools are in your jdk/bin directory**

Lab 1 Exercise

Lab1 Exercise lab1/readme.txt http://tinyurl.com/deadlocks2016

- **In our first lab, a bunch of philosophers (Thinker) are sitting around a table at their symposium and are using two cups of wine (Krasi) to quench their thirst. Each of them first grabs the left and then the right cup. If they all grab the right cup at the same time, we will have some unhappy philosophers caught in limbo.**
  1. **To run the code you can either use the run.bat file or mvn -Prun. To compile you can use mvn install.**
  2. **Run the code and verify that you see a deadlock by capturing a stack trace. Depending on your machine, you might need a few runs to see the issue.**
  3. **Once you have discovered the deadlock, verify that it involves the left and right locks.**
  4. **Now define a global ordering for the locks. For example, you can either let Krasi implement Comparable and give it a number to sort by, or you can use System.identityHashCode() to be able to compare the cups. (Warning: Sadly, the identity hash code is not guaranteed to be unique. Thus you have to plan for this. It is easier to make Krasi comparable.)**
  5. **Verify that the deadlock has now disappeared.**
- **Good luck! You have 20 minutes to solve this lab.**

Lab1 Exercise Solution Explanation

- **Goal: Prevent all philosophers from holding a single cup**

Lab1 Exercise Solution Explanation

- **Goal: Prevent all philosophers from holding a single cup**

Thinker	Cup 1 right	Cup 2 left
1	1	2
2	2	3
3	3	4
4	4	5
5	5	1

Thinker	Cup 1 big	Cup 2 small
1	2	1
2	3	2
3	4	3
4	5	4
5	5	1

- **The set of first cups is 2,3,4,5**
  - **This means that at most four philosophers can hold a single cup!**

Lab 2: Deadlock resolution by tryLock

Avoiding Liveness Hazards

Lab 2: Deadlock Resolution By Trylock

- **Same problem as in Lab 1**
- **But our solution will be different**
- **Instead of a global order on the locks**
  - **We lock the first lock**
  - **We then try to lock the second lock**
    - **If we can lock it, we start drinking**
    - **If we cannot, we back out completely and try again**
      - **What about starvation or livelock?**

Lock And Reentrantlock

- **The Lock interface offers different ways of locking:**
  - **Unconditional, polled, timed and interruptible**

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)
    throws InterruptedException;
  void unlock();
  Condition newCondition();
}
```

- **Lock implementations must have same memory-visibility semantics as intrinsic locks (synchronized)**

Reentrantlock Implementation

- **Like synchronized, it offers reentrant locking semantics**
- **Also, we can interrupt threads that are waiting for locks**
  - **Actually, the ReentrantLock never causes the thread to be BLOCKED, but always WAITING**
  - **If we try to acquire a lock unconditionally, interrupting the thread will simply go back into the WAITING state**
    - **Once the lock has been granted, the thread interrupts itself**

Using The Explicit Lock

- **We have to call unlock() in a finally block**
  - **Every time, without exception**
  - **There are FindBugs detectors that will look for forgotten "unlocks"**

```
private final Lock lock = new ReentrantLock();

public void update() {
  lock.lock(); // this should be before try
  try {
    // update object state
    // catch exceptions and restore
    // invariants if necessary
  } finally {
    lock.unlock();
  }
}
```

Polled Lock Acquisition

- **Instead of unconditional lock, we can tryLock()**

```
if (lock.tryLock()) {
  try {
    balance = balance + amount;
  } finally {
    lock.unlock();
  }
} else {
  // alternative path
}
```

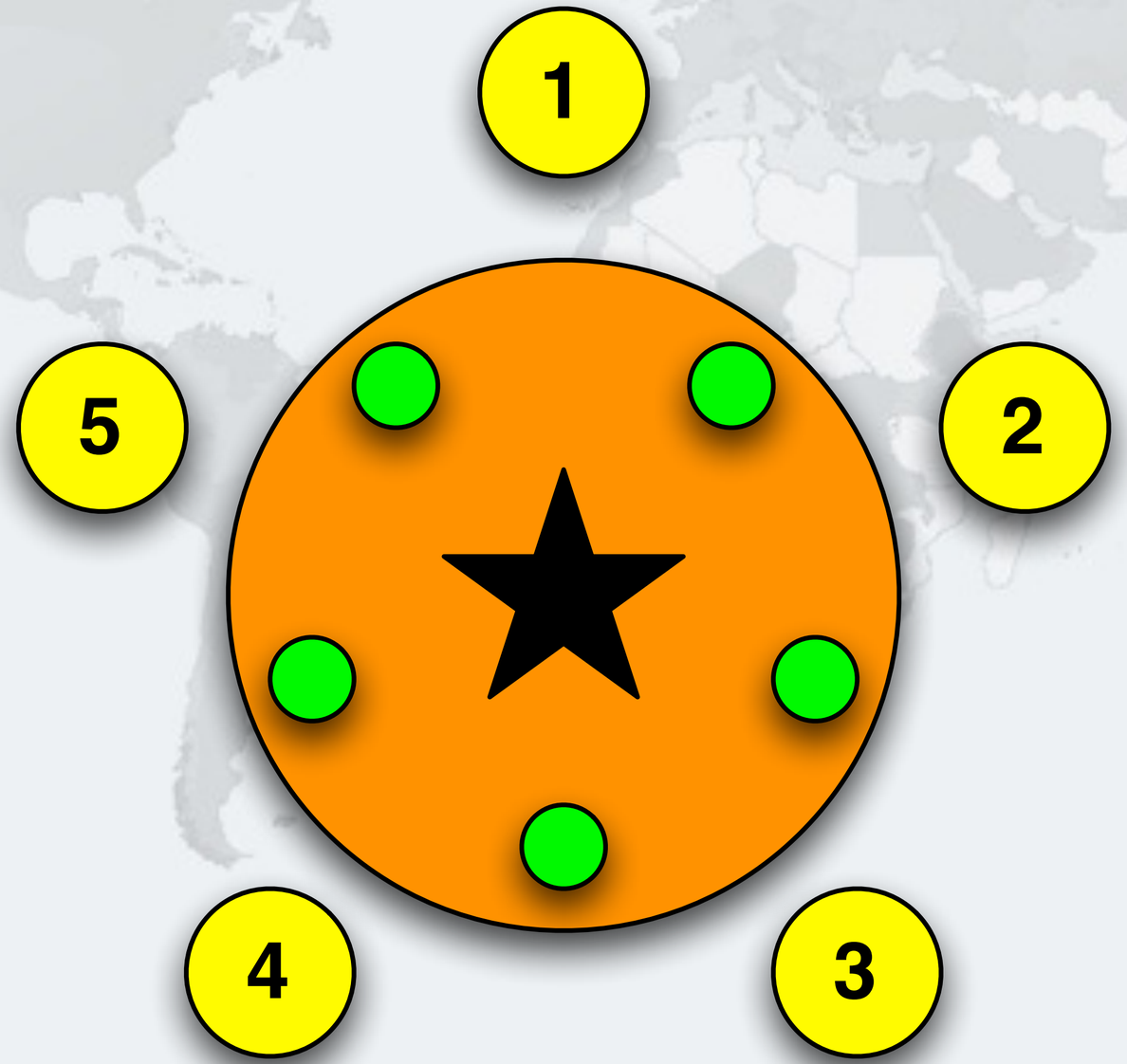
Using Try-Lock To Avoid Deadlocks

- **Deadlocks happen when we lock multiple locks in different orders**
- **We can avoid this by using tryLock()**
  - **If we do not get lock, sleep for a random time and then try again**
  - **Must release *all* held locks, or our deadlocks become livelocks**
- **This is possible with synchronized, see my newsletter**
  - **http://www.javaspecialists.eu/archive/Issue194.html**

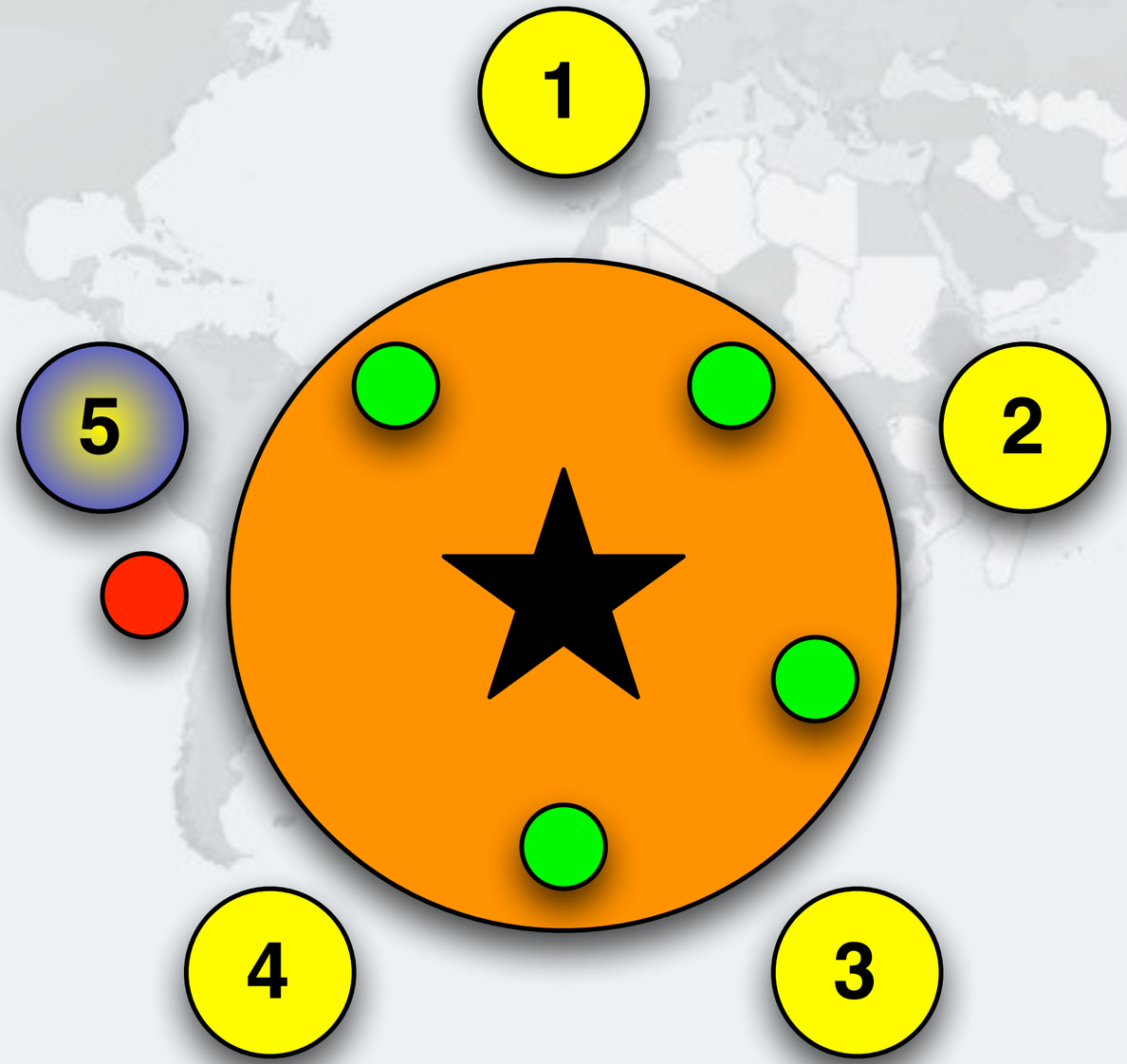
Using Trylock() To Avoid Deadlocks

```
public void drink() {
  while (true) {
    right.lock();
    try {
      if (left.tryLock()) {
        try {
          // now we can finally drink and then return
          return;
        } finally {
          left.unlock();
        }
      }
    } finally {
      right.unlock();
    }
    LockSupport.parkNanos(System.nanoTime() & 0xffff);
  }
}
```

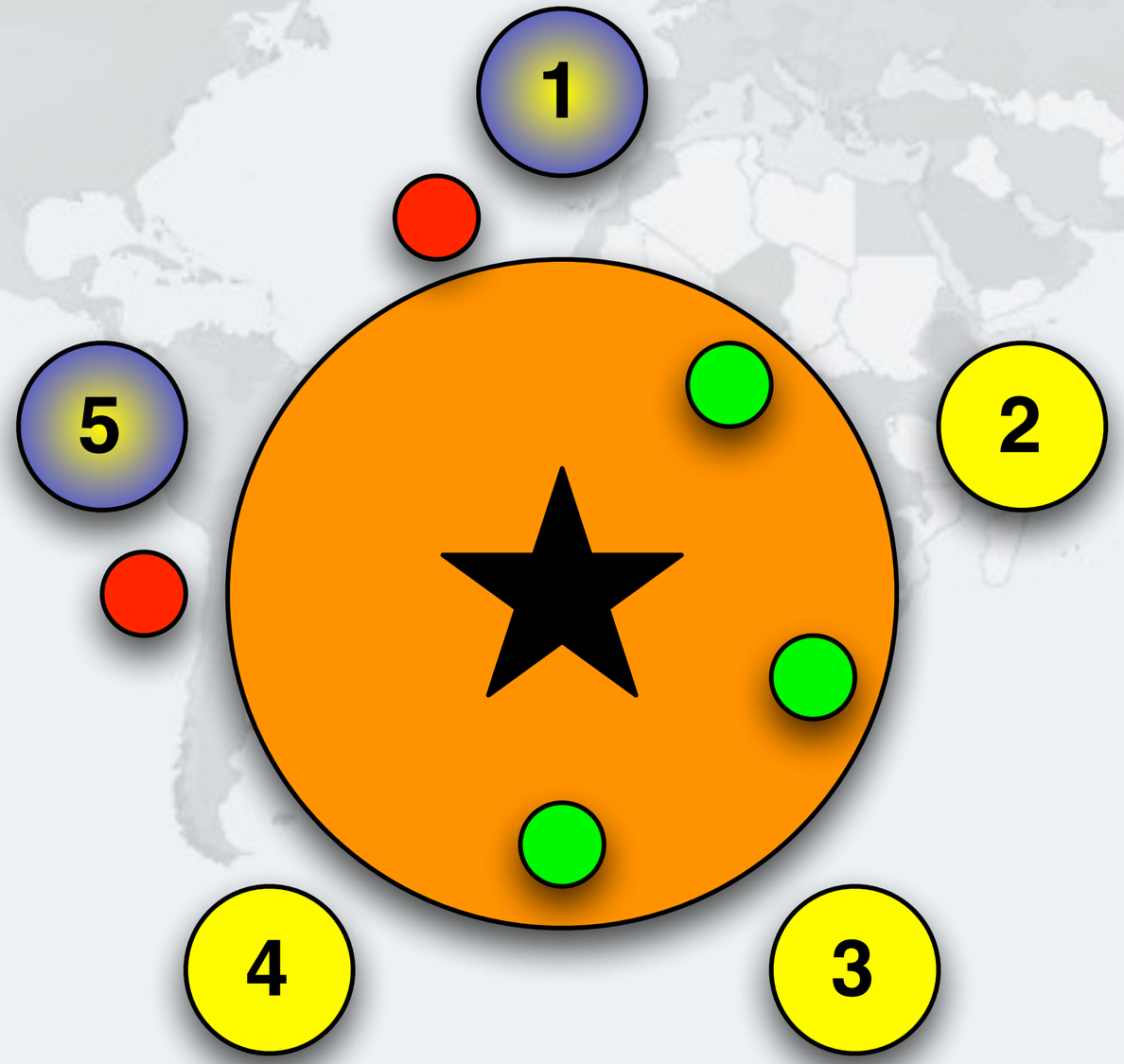
Deadlock Is Prevented In This Design



Philosopher 5 Wants To Drink, Takes Right Cup

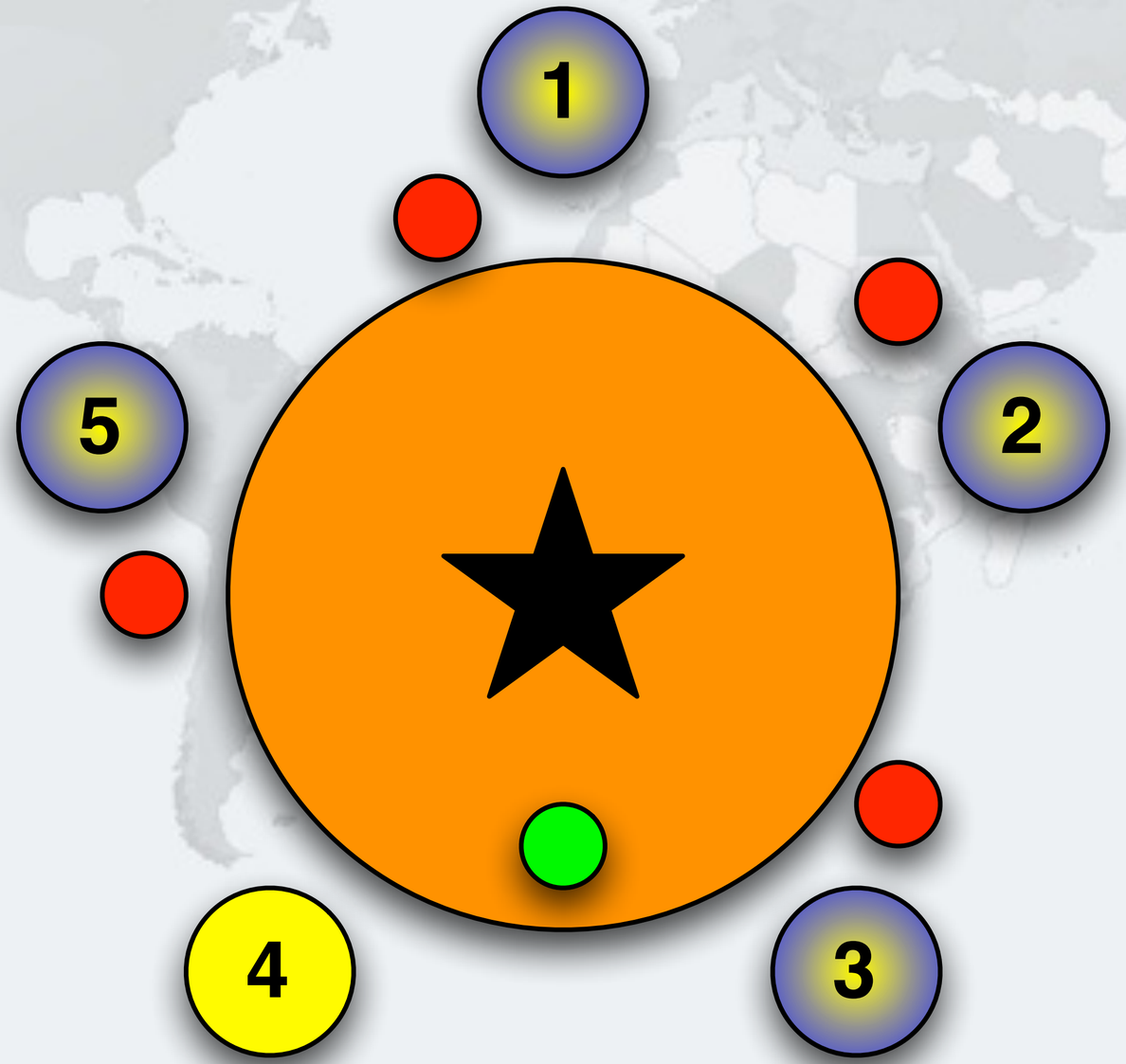


Philosopher 1 Wants To Drink, Takes Right Cup

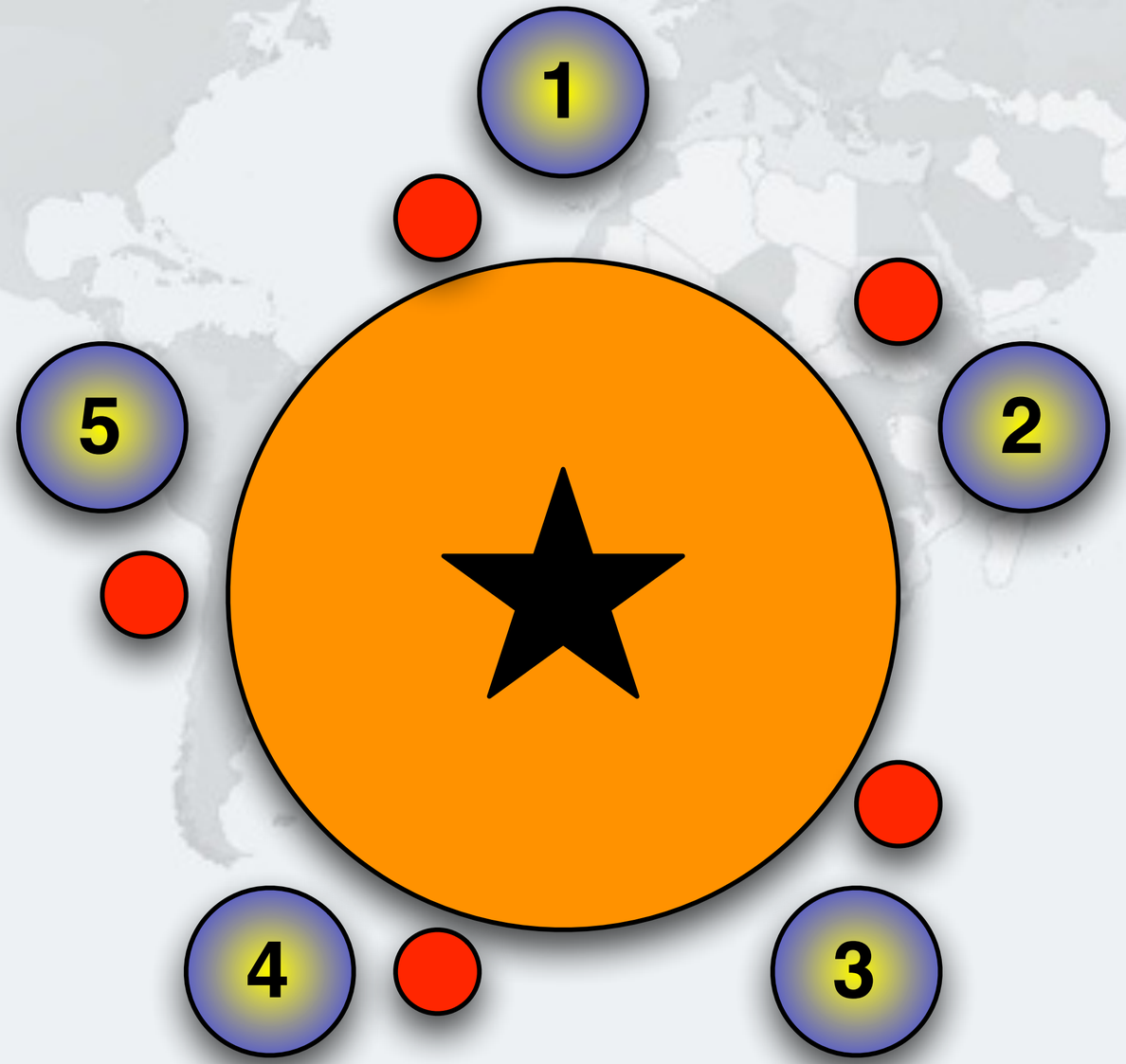


Philosopher 2 Wants To Drink, Takes Right Cup

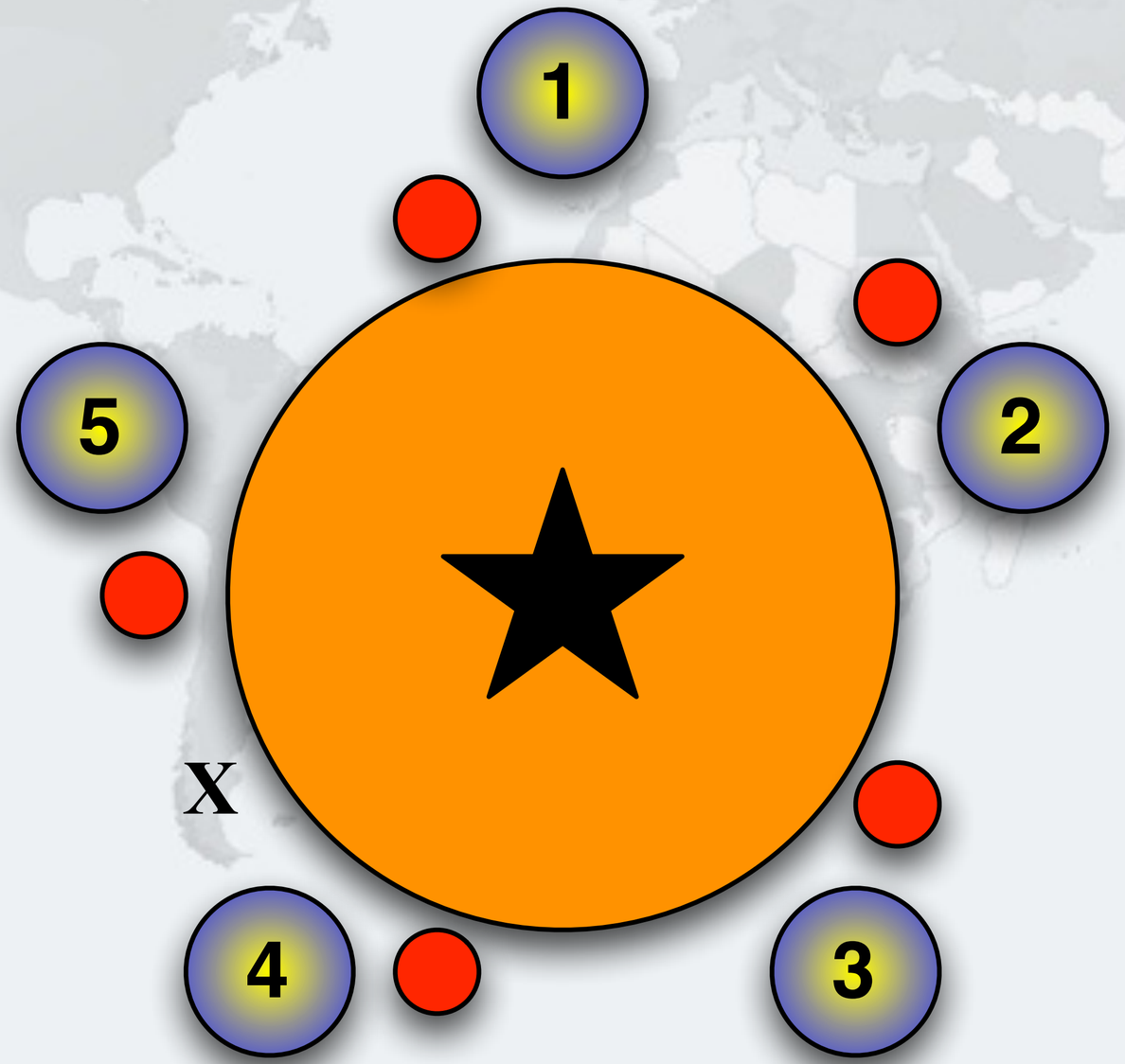
Philosopher 3 Wants To Drink, Takes Right Cup



Philosopher 4 Wants To Drink, Takes Right Cup

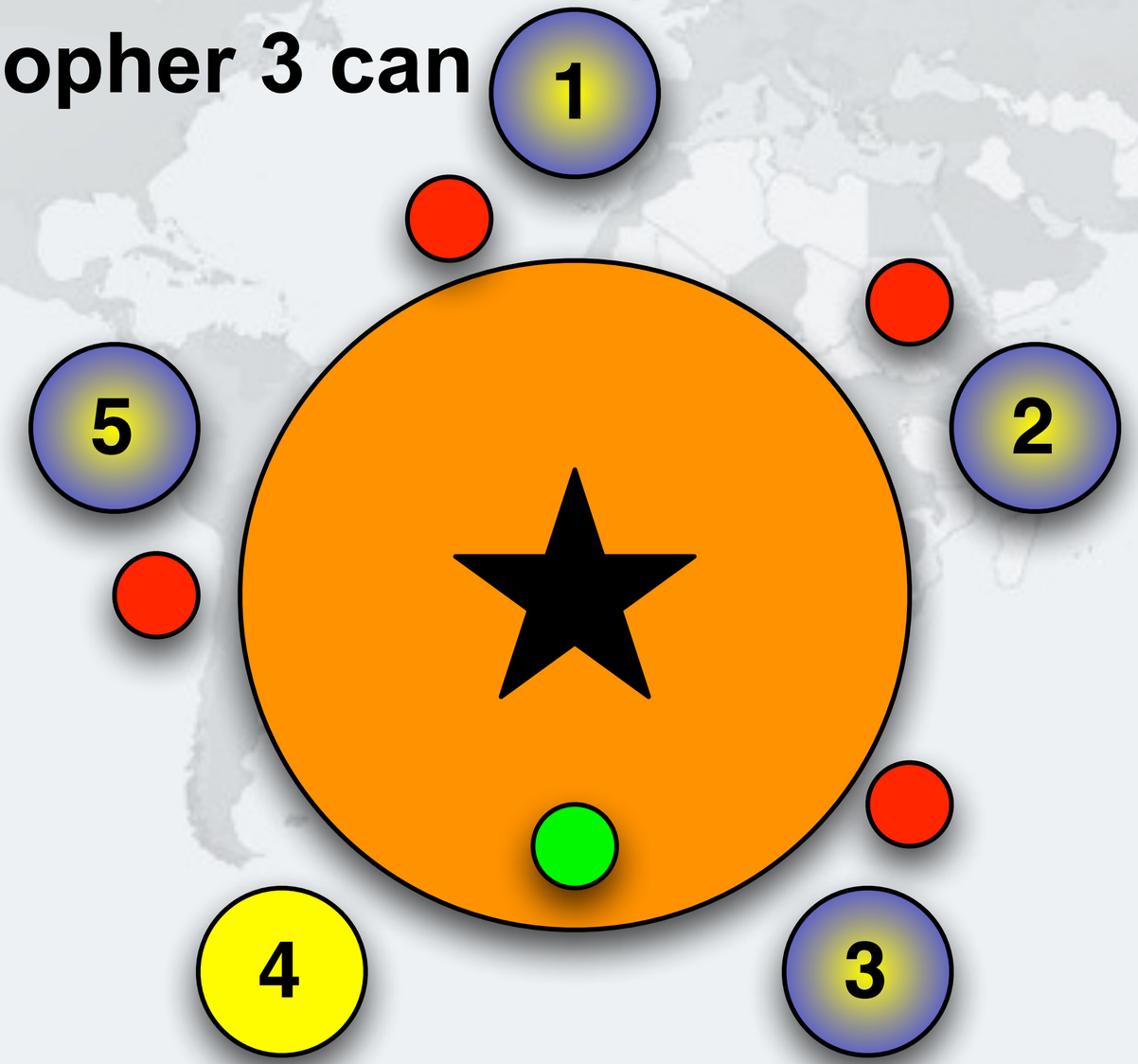


Philosopher 4 Tries To Lock Left, Not Available



Philosopher 4 Unlocks Right Again

- **Now Philosopher 3 can drink**



Lab 2 Exercise

Deadlock resolution by tryLock

Javaspecialists.eu
java training

Lab2 Exercise lab2/readme.txt http://tinyurl.com/deadlocks2016

- **Run Main class to trigger deadlock**
  - **You might need a few runs**
- **Capture a stack trace with JVisualVM**
- **Verify the deadlock involves the left and right locks**
- **Use Lock.tryLock() to avoid blocking on the inner lock (forever)**
  - **lock the right**
  - **tryLock the left**
    - **if success, then drink and unlock both**
    - **otherwise, unlock right and retry**
  - **Change the Thinker.java file**
- **Verify that the deadlock has now disappeared**

Lab2 Exercise Solution Explanation

- **Goal: Prevent all philosophers from forever blocking on the second cup**
  - **A philosopher should not die of thirst**
    - **We need to avoid livelocks**
    - **lock/tryLock vs. tryLock/tryLock**

Lab 3: Resource Deadlock

Avoiding Liveness Hazards

Javaspecialists.eu
java training

Lab 3: Resource Deadlock

- **Problem: threads are blocked waiting for a finite resource that never becomes available**
- **Examples:**
  - **Resources not being released after use**
    - **Running out of threads**
    - **Java Semaphores not being released**
  - **JDBC transactions getting stuck**
  - **Bounded queues or thread pools getting jammed up**

Challenge

- **Does not show up as a Java thread deadlock**
- **Problem thread could be in any state: RUNNABLE, WAITING, BLOCKED, TIMED_WAITING**

How To Solve Resource Deadlocks

- **Approach: If you can reproduce the resource deadlock**
  - **Take a thread snapshot shortly before the deadlock**
  - **Take another snapshot after the deadlock**
  - **Compare the two snapshots**
- **Approach: If you are already deadlocked**
  - **Take a few thread snapshots and look for threads that do not move**
- **It is useful to identify the resource that is being exhausted**
  - **A good trick is via phantom references (beyond scope of this lab)**

Resource Deadlocks

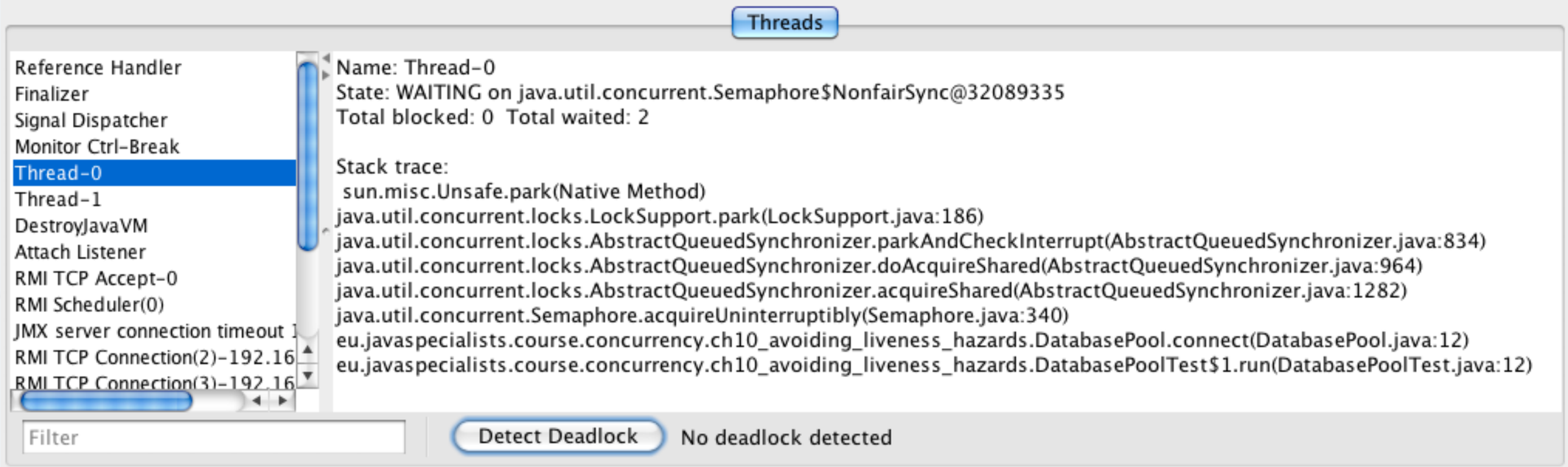
- **We can also cause deadlocks waiting for resources**
- **For example, say you have two DB connection pools**
  - **Some tasks might require connections to both databases**
  - **Thus thread A might hold semaphore for D1 and wait for D2, whereas thread B might hold semaphore for D2 and be waiting for D1**
- **Thread dump and ThreadMXBean does not show this as a deadlock!**

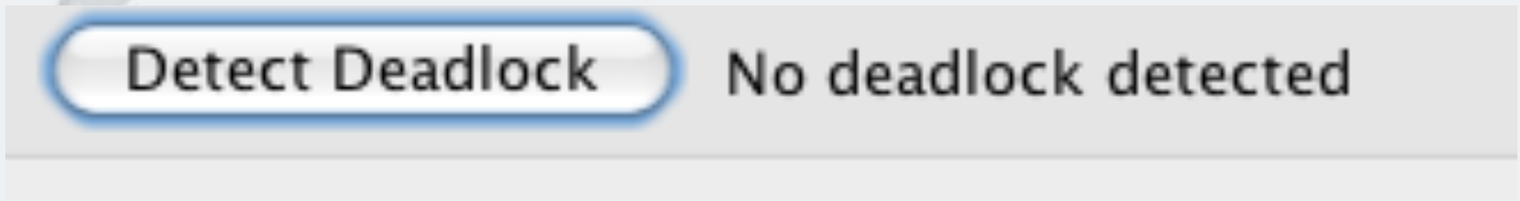
Our Databasepool - Connect() And Disconnect()

```
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 Detect This Deadlock

```
DatabasePool.connect
DatabasePool.connect
```





Stack Trace Gives A Vector Into The Code

```
locks.AbstractQueuedSynchronizer.doAcquireShared(AbstractQueuedSynchronizer.java:964)
locks.AbstractQueuedSynchronizer.acquireShared(AbstractQueuedSynchronizer.java:1282)
Semaphore.acquireUninterruptibly(Semaphore.java:340)
ourse.concurrency.ch10_avoiding_liveness_hazards.DatabasePool.connect(DatabasePool.java:12)
```

```
public class DatabasePool {
  // ...

  public void connect() {
    connections.acquireUninterruptibly(); // line 12
    System.out.println("DatabasePool.connect");
  }
}
```

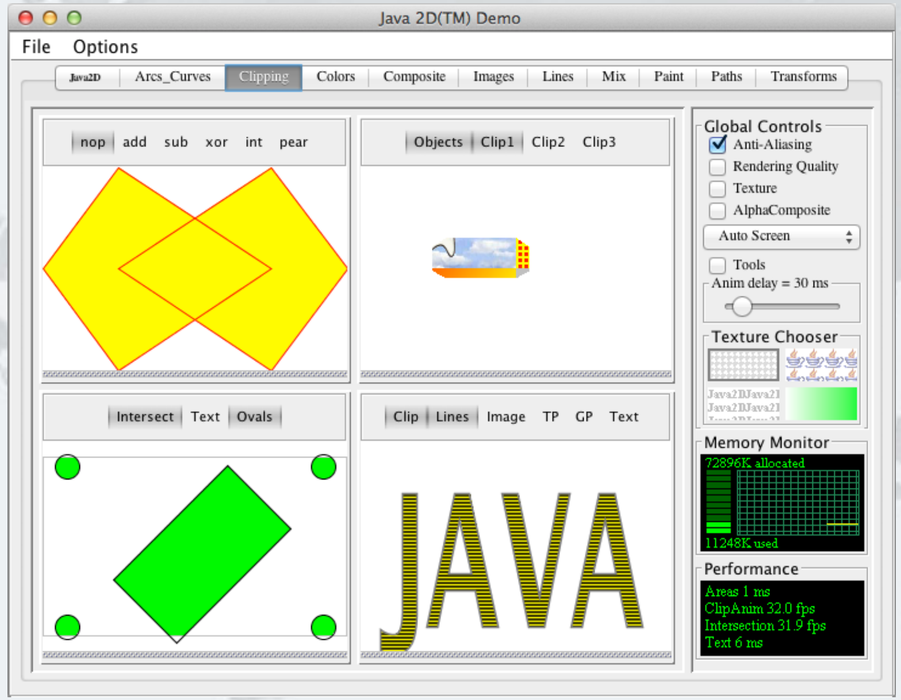
Lab 3 Exercise

Resource Deadlock

Javaspecialists.eu
java training

Lab3 Exercise lab3/readme.txt

http://tinyurl.com/deadlocks2016



- **Start our modified Java2Demo**
- **Connect JVisualVM and dump all threads**
- **Use Java2Demo for a while until it deadlocks**
- **Get another thread dump and compare to the first one**
  - **This should show you where the problem is inside your code**
- **Fix the problem and verify that it has been solved**
  - **Hint: Your colleagues probably write code like this, but you shouldn't**

Lab3 Exercise Solution Explanation

- **Goal: Ensure that resources are released after use**
- **Diff between the two thread dumps using jps and jstack**

```
< at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:2043)
< at java.awt.EventQueue.getNextEvent(EventQueue.java:531)
< at java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThread.java:213)
---
> at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(AbstractQueuedSynchronizer.java:834)
> at java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedInterruptibly(AbstractQueuedSynchronizer.java:994)
> at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(AbstractQueuedSynchronizer.java:1303)
> at java.util.concurrent.Semaphore.acquire(Semaphore.java:317)
> at eu.javaspecialists.deadlock.lab3.java2d.MemoryManager.gc(MemoryManager.java:56)
> at eu.javaspecialists.deadlock.lab3.java2d.MemoryMonitor$Surface.paint(MemoryMonitor.java:153)
```

– **Fault is probably in our classes, rather than JDK**

What Is Wrong With This Code?

```
/**
 * Only allow a maximum of 30 threads to call System.gc() at a time.
 */
public class MemoryManager extends Semaphore {
  private static final int MAXIMUM_NUMBER_OF_CONCURRENT_GC_CALLS = 30;

  public MemoryManager() {
    super(MAXIMUM_NUMBER_OF_CONCURRENT_GC_CALLS);
  }

  public void gc() {
    try {
      acquire();
      try {
        System.gc();
      } finally {
        System.out.println("System.gc() called");
        release();
      }
    } catch (Exception ex) {
      // ignore the InterruptedException
    }
  }
}
```

Calling System.gc() is baddd (but not **the** problem)

Empty catch block hides problem

Lab 4: Combining Your Skills

Avoiding Liveness Hazards

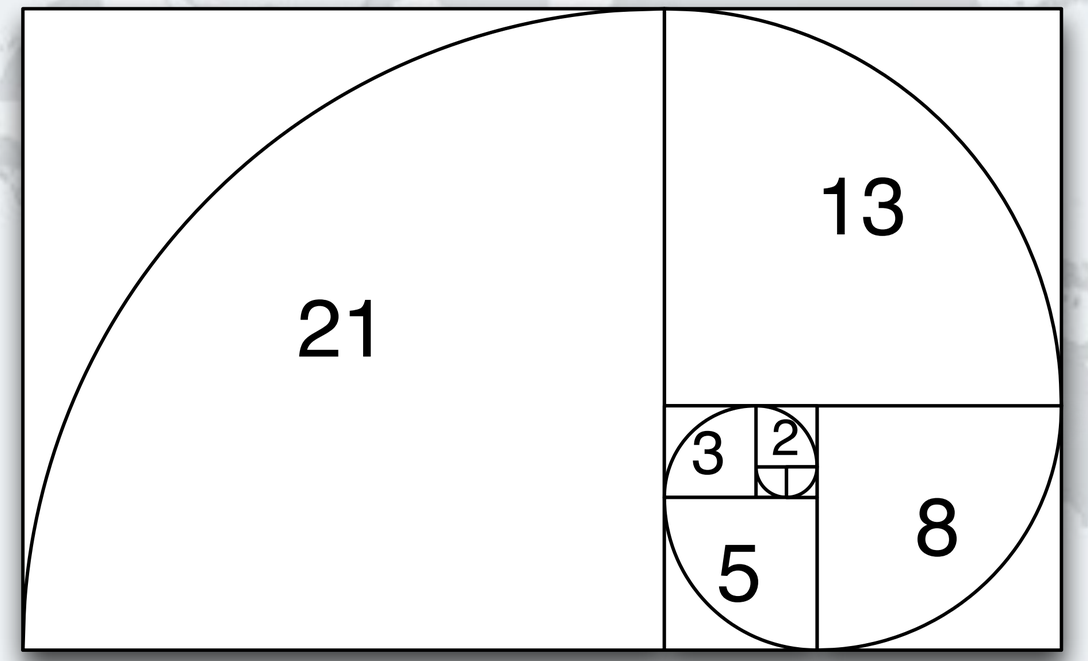
Lab 4: Combining Your Skills

- **Problem: try to solve lab 4 using the skills learned**
- **Be careful - it is not as easy as it looks :-)**
- **http://tinyurl.com/deadlocks2016**

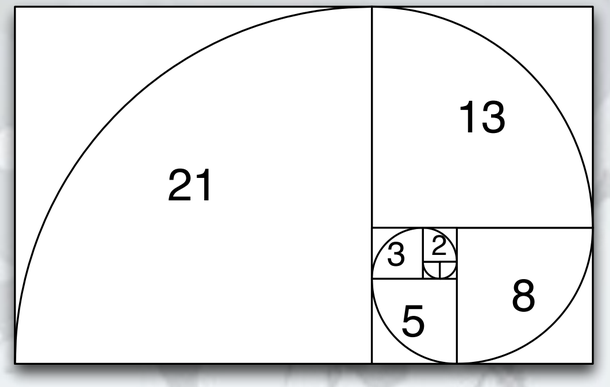
Lab 5: Speeding Up Fibonacci

Lab 5: Speeding Up Fibonacci

- **Number sequence named after Leonardo of Pisa**
  - **F0 = 0**
  - **F1 = 1**
  - **Fn = Fn-1 + Fn-2**
- **Thus the next number is equal to the sum of the two previous numbers**
  - **e.g. 0, 1, 1, 2, 3, 5, 8, 13, 21, …**
- **The numbers get large very quickly**



First Attempt At Writing A Fibonacci Method



- **Taking our recursive definition**
  - $F_0 = 0, F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$
- **Our first attempt writes a basic recursive function**

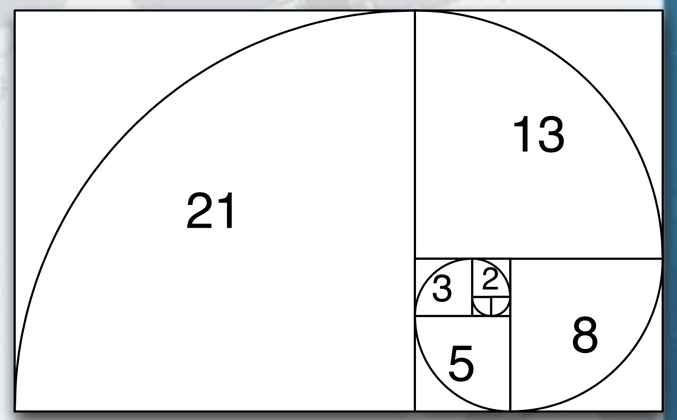
```
public long f(int n) {
  if (n <= 1) return n;
  return f(n-1) + f(n-2);
}
```

- **But this has exponential time complexity**
  - f(n+10) is 1000 slower than f(n)

2nd Attempt at Coding Fibonacci

- **Instead of a recursive method, we use iteration:**

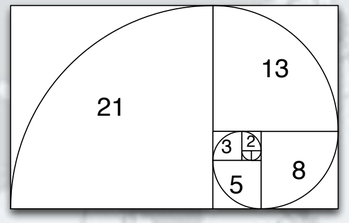
```
public static long f(int n) {
  long n0 = 0, n1 = 1;
  for (int i = 0; i < n; i++) {
    long temp = n1;
    n1 = n1 + n0;
    n0 = temp;
  }
  return n0;
}
```



- **This algorithm has linear time complexity**
  - **Solved f(1_000_000_000) in 1.7 seconds**
    - **However, the numbers overflow so the result is incorrect**
    - **We can use BigInteger, but its add() is also linear, so time is quadratic**
    - **We need a better algorithm**

3rd Attempt Dijkstra's Sum of Squares

- **Dijkstra noted the following formula for Fibonacci**
  - $F_{2n-1} = F_{n-1}^2 + F_n^2$
  - $F_{2n} = (2 \times F_{n-1} + F_n) \times F_n$



- **Logarithmic time complexity and can be parallelized**
  - **Java 8 uses better BigInteger multiply() algorithms**
    - **Karatsuba complexity is $O(n^{1.585})$**
    - **3-way Toom Cook complexity is $O(n^{1.465})$**
    - **Previous versions of Java had complexity $O(n^2)$**
    - **Unfortunately multiply() in BigInteger is only available single-threaded - we'll fix that in Lab 5.3**

Lab 5.1: Dijkstra's Sum Of Squares

- **Implement this algorithm using BigInteger**
  - $F_{2n-1} = F_{n-1}^2 + F_n^2$
  - $F_{2n} = (2 \times F_{n-1} + F_n) \times F_n$
- **Run all tests in FibonacciTest and record the times**
- **Do it yourself - no cheating with Google!**

Lab 5.2: Parallelize Your Algorithm

- **We can parallelize by using common Fork/Join Pool**

```
private final class FibonacciTask extends RecursiveTask<BigInteger> {
  private final int n;
  private FibonacciTask(int n) {
    this.n = n;
  }
  protected BigInteger compute() {
    return f(n);
  }
}
```

- **Next we fork() the 1st task, do the 2nd and then join 1st**

```
FibonacciTask fn_1Task = new FibonacciTask(n - 1);
fn_1Task.fork();
BigInteger fn = f(n);
BigInteger fn_1 = fn_1Task.join();
```

Lab 5.3: Parallelize Biginteger

- **Using principles from lab 5.2, parallelize methods in eu.javaspecialists.performance.math.BigInteger**
  - **multiplyKaratsuba()**
  - **multiplyToomCook3()**
  - **squareKaratsuba()**
  - **squareToomCook3()**

Lab 5.4: Cache Results

- **Dijkstra's Sum of Squares needs to work out some values several times. Cache results to avoid this.**
- **Make sure you implement a "reserved caching scheme" where if one thread says he wants to calculate some value, others would wait**
  - **e.g. have a special BigInteger that signifies RESERVED**
    - **First thing a task would do is check if map contains that**
    - **If it doesn't, it puts it in and thus reserves it**
    - **If it does, it waits until the task is done and uses that value**

Lab 5.5: ManagedBlocker

- **ForkJoinPool is configured with *desired parallelism***
  - **Number of active threads**
  - **ForkJoinPool mostly used with CPU intensive tasks**
- **If one of the FJ Threads has to block, a new thread can be started to take its place**
  - **This is done with the ManagedBlocker**
- **Change your cache to use ManagedBlocker to keep parallelism high**

Wrap Up

Avoiding Liveness Hazards

Conclusion On Deadlocks

- **Concurrency is difficult, but there are tools and techniques that we can use to solve problems**
- **These are just a few that we use**
- **For more information, have a look at**
  - **The Java Specialists' Newsletter**
    - **http://www.javaspecialists.eu**
- **Made in Chania (mostly)**

Finding and Solving Java Deadlocks

Dr Heinz M. Kabutz
heinz@kabutz.net
@heinzkabutz

Javaspecialists.eu
java training