Java Concurrency Live(li)ness Lab

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

Last Updated 2017-01-31

Javaspecialists.eu
Java training

© 2012-2017 Heinz Kabutz, All Rights Reserved

Live(li)ness

- Liveness (labs 1-4)
  - No deadlocks, no livelocks
- Liveliness (labs 5.1-5.5)
  - Full CPU utilization, no contention

tinyurl.com/jfokus2017

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Avoiding Liveness Hazards

© 2012-2017 Heinz Kabutz, All Rights Reserved

Avoiding Liveness Hazards

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

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

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!**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Thread Deadlocks in BLOCKED

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

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 1: Deadlock Resolution by Global Ordering

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu
java training

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**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

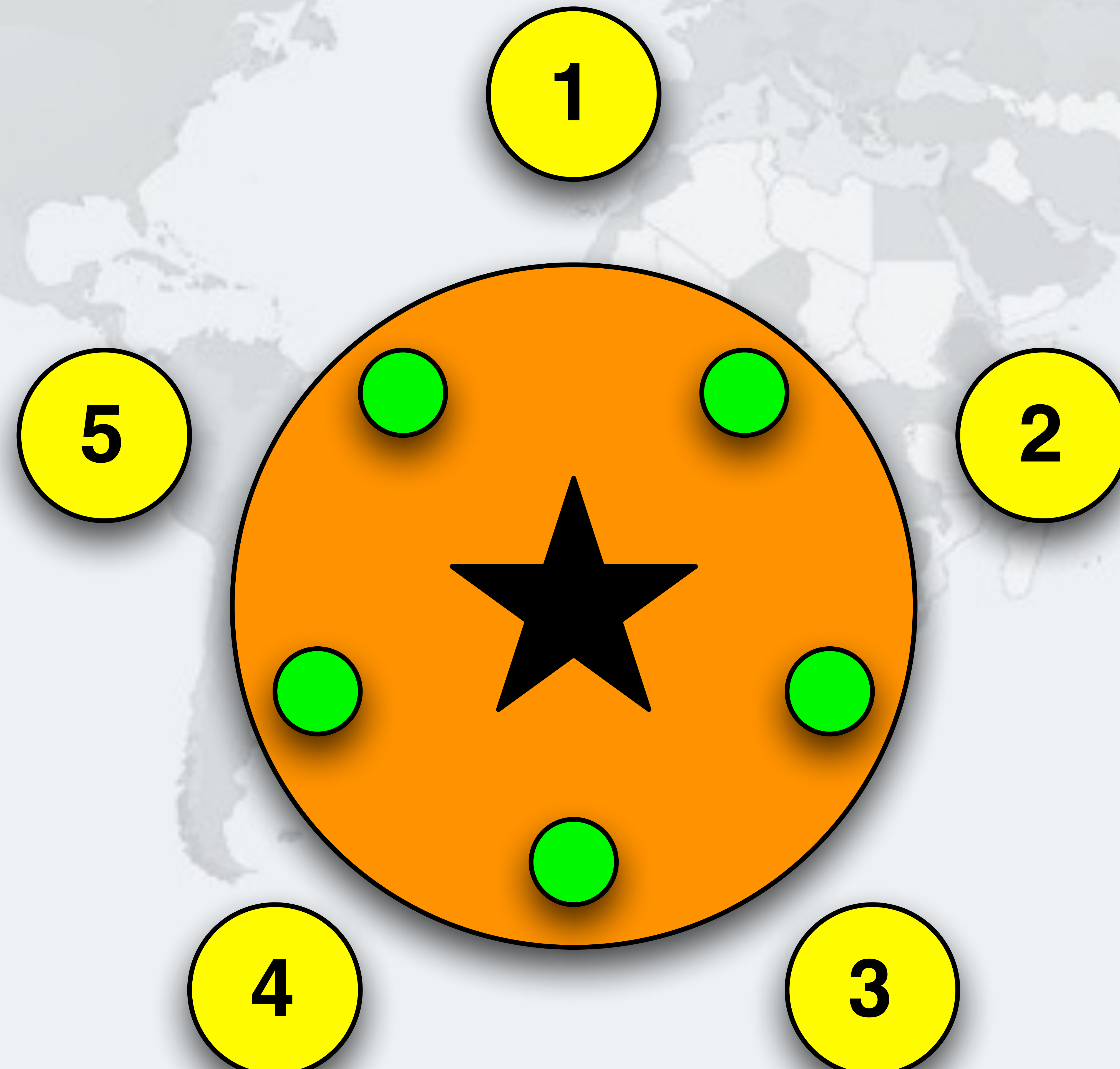
Our Drinking Philosophers

- **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**

Javaspecialists.eu

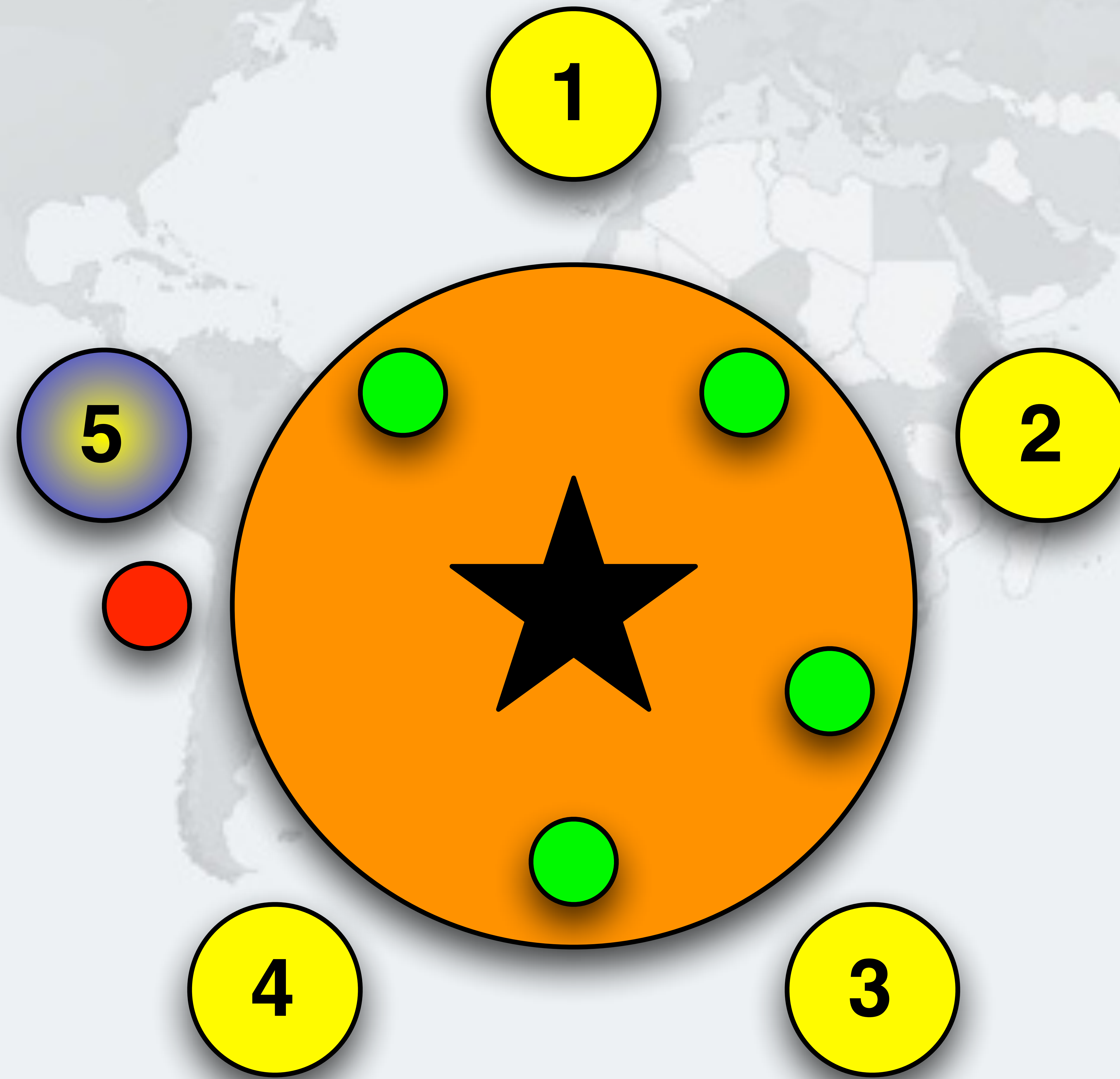
© 2012-2017 Heinz Kabutz, All Rights Reserved

Table is ready, all philosophers are thinking



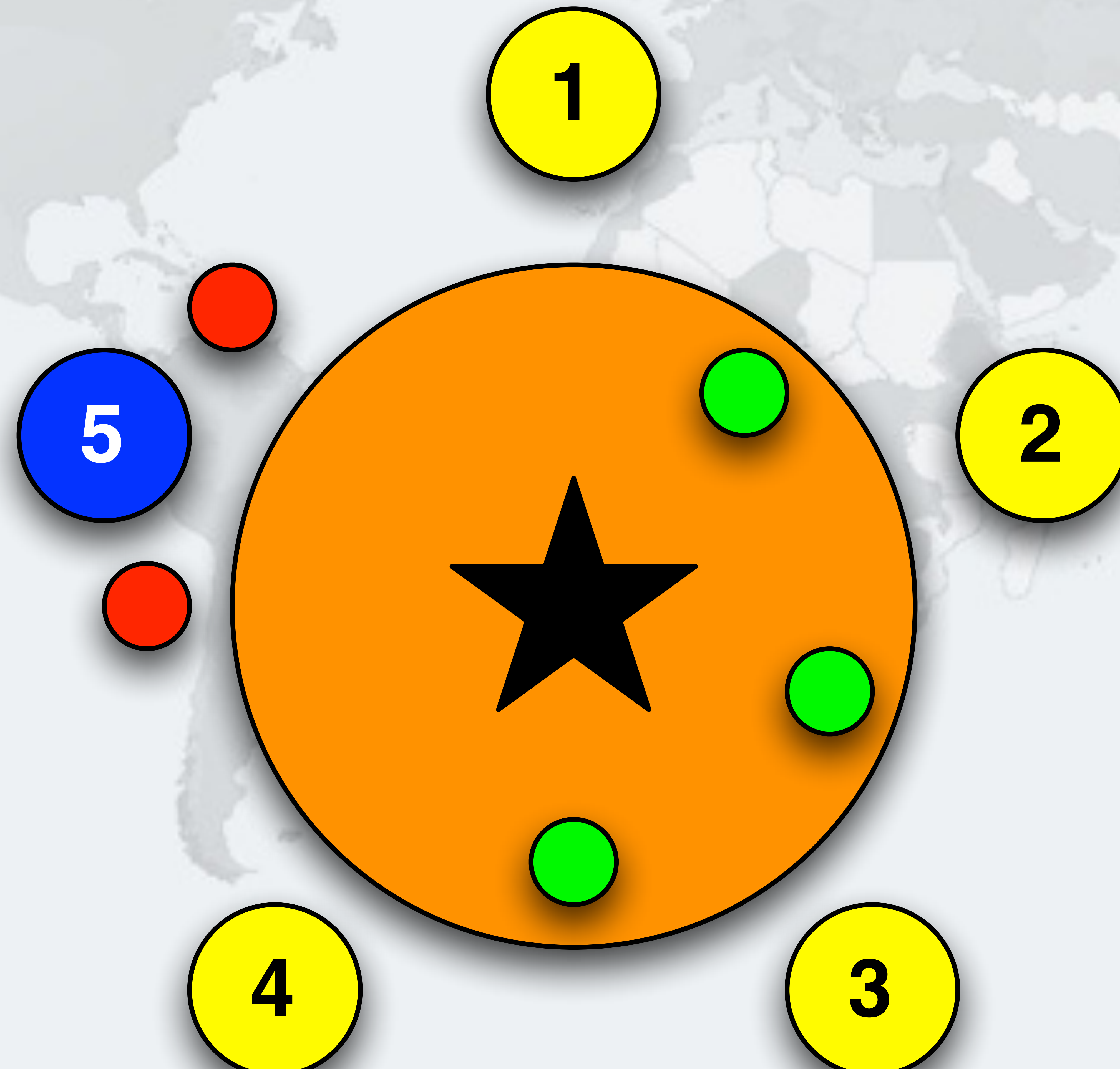
Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 5 wants to drink, takes right cup



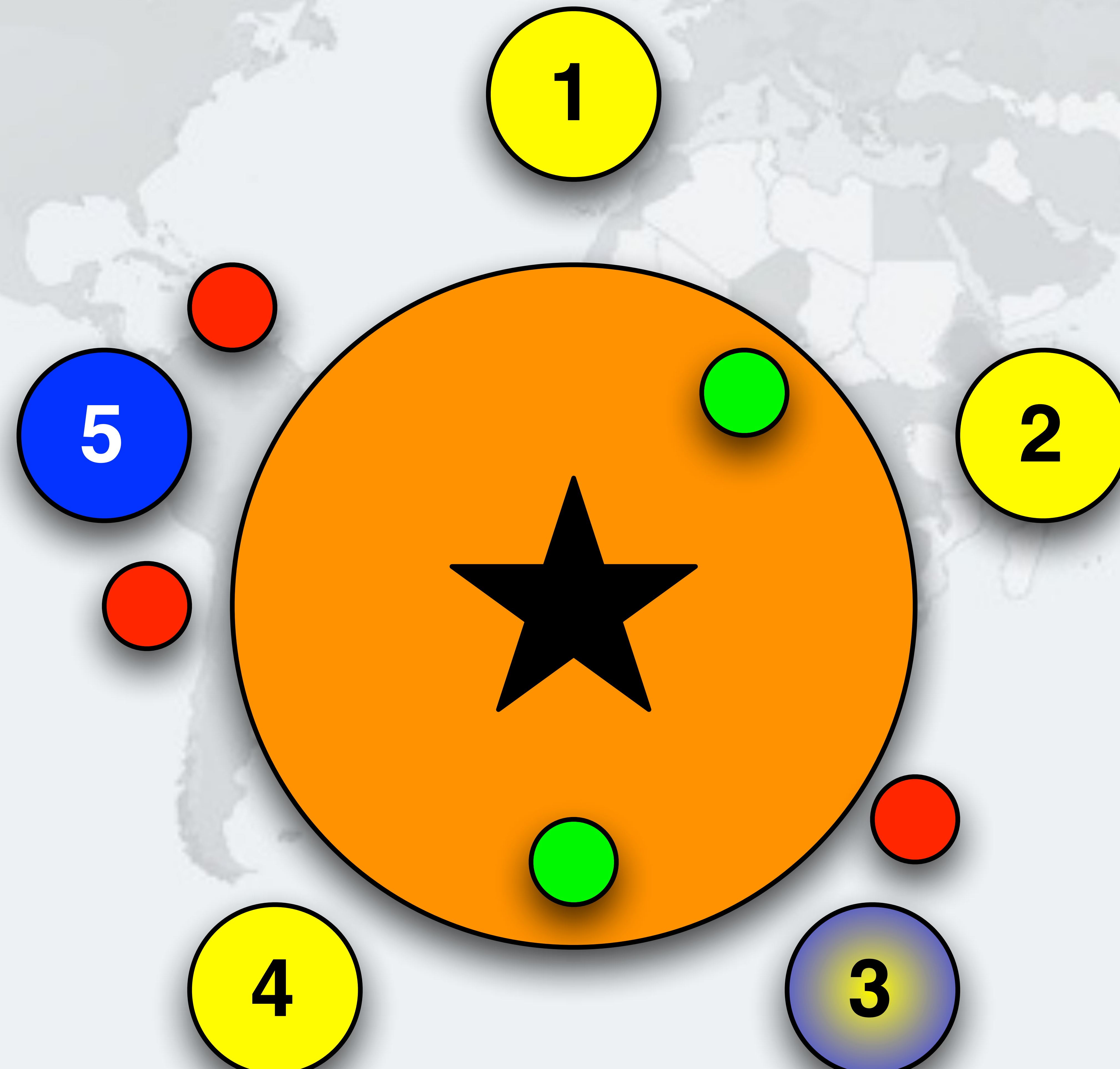
Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 5 is now drinking with both cups



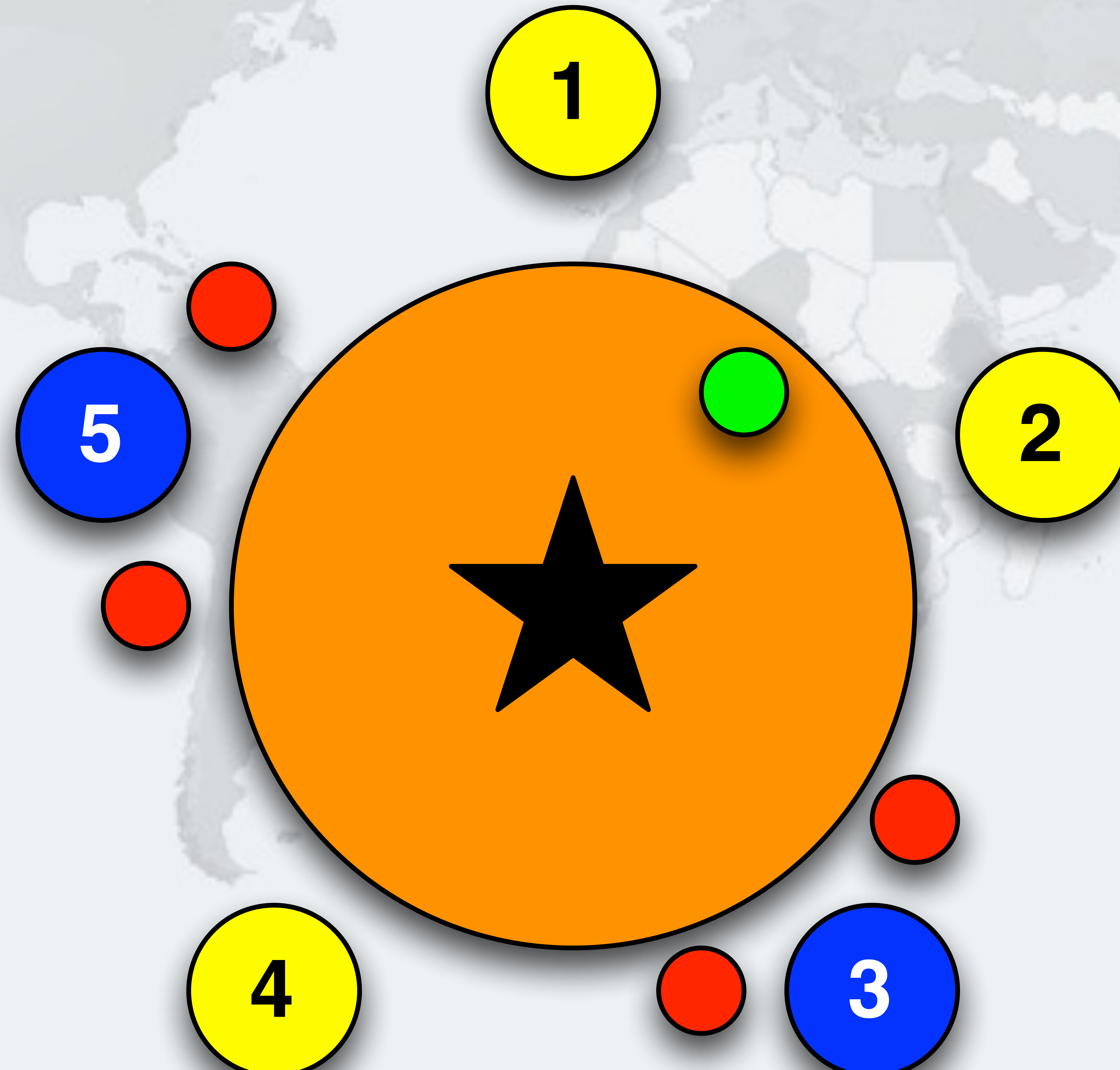
Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 3 wants to drink, takes right cup



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 3 is now drinking with both cups

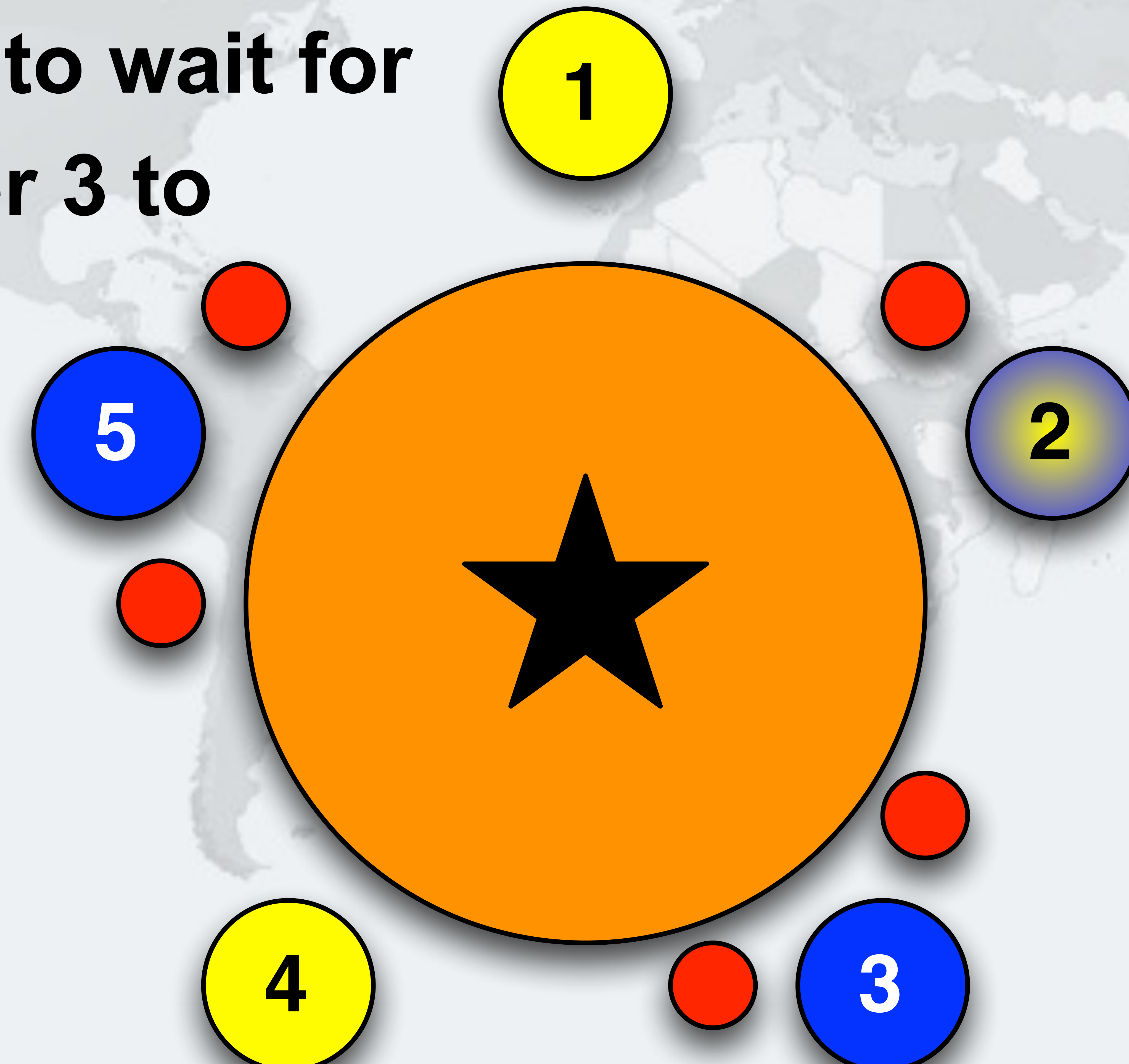


Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

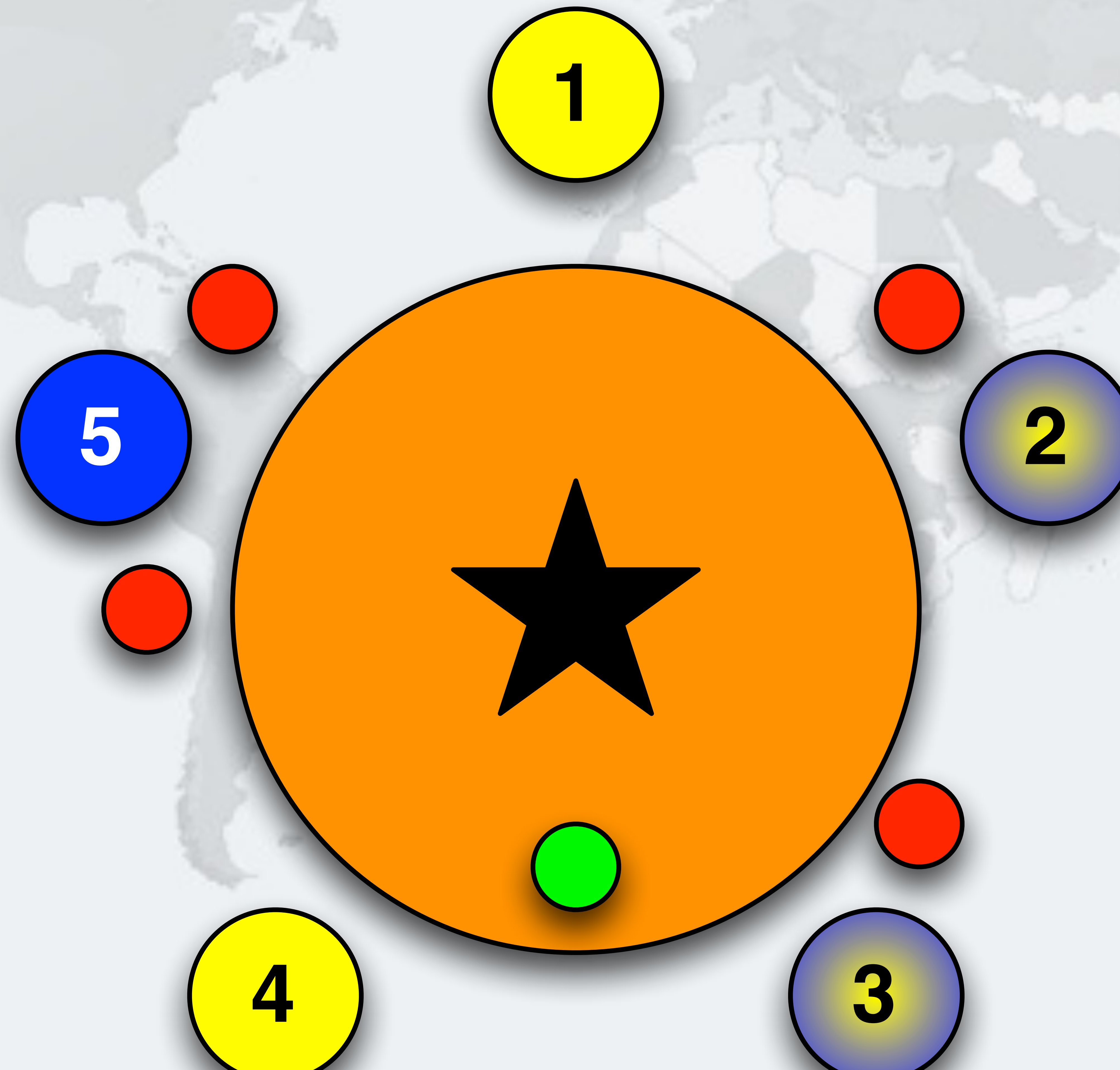
Philosopher 2 wants to drink, takes right cup

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



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 3 finished drinking, returns left cup

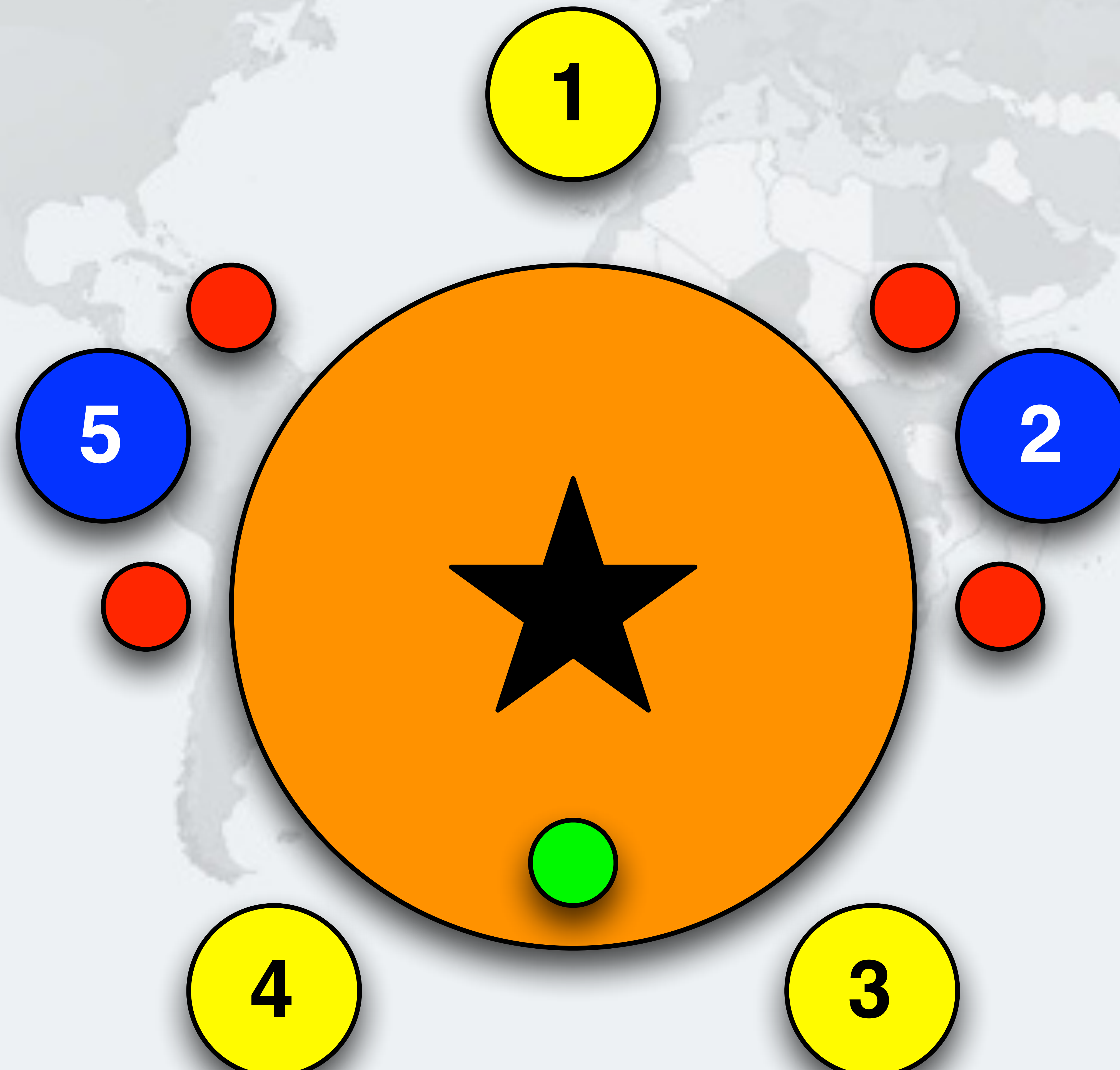


Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 3 returns right cup

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 2 is now drinking with both cups



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

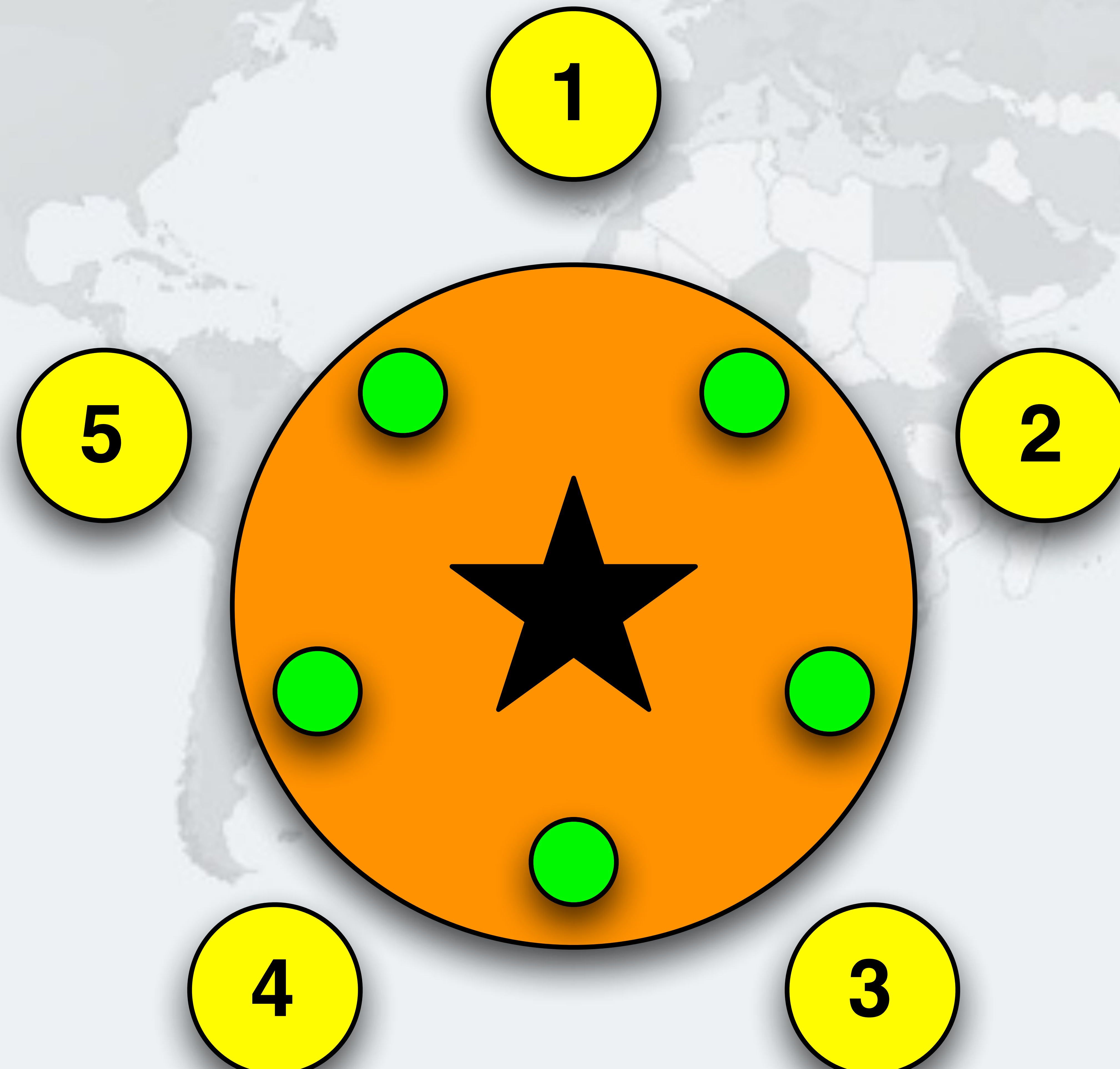
Drinking Philosophers in Limbo

- **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**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

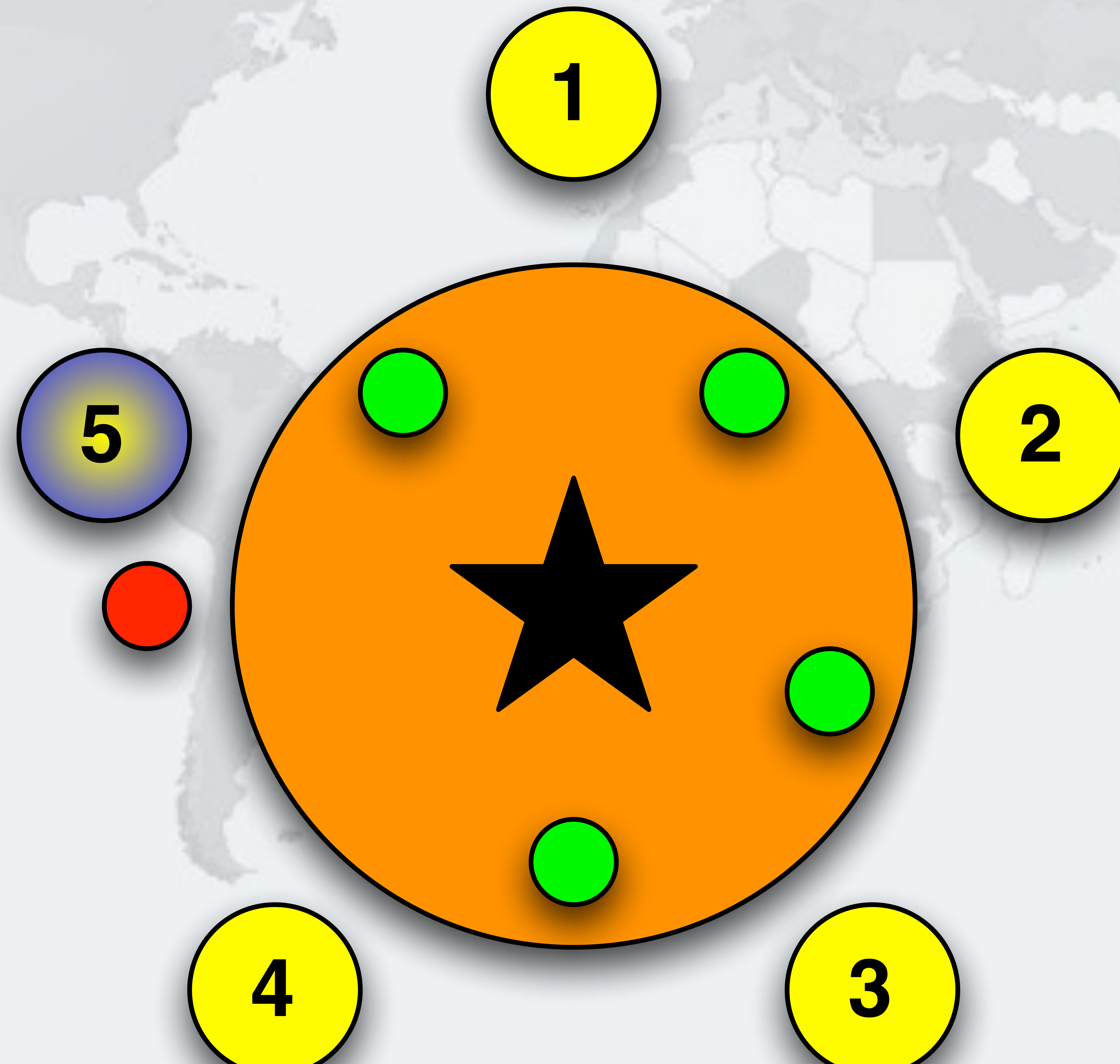
A deadlock can easily happen with this design



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

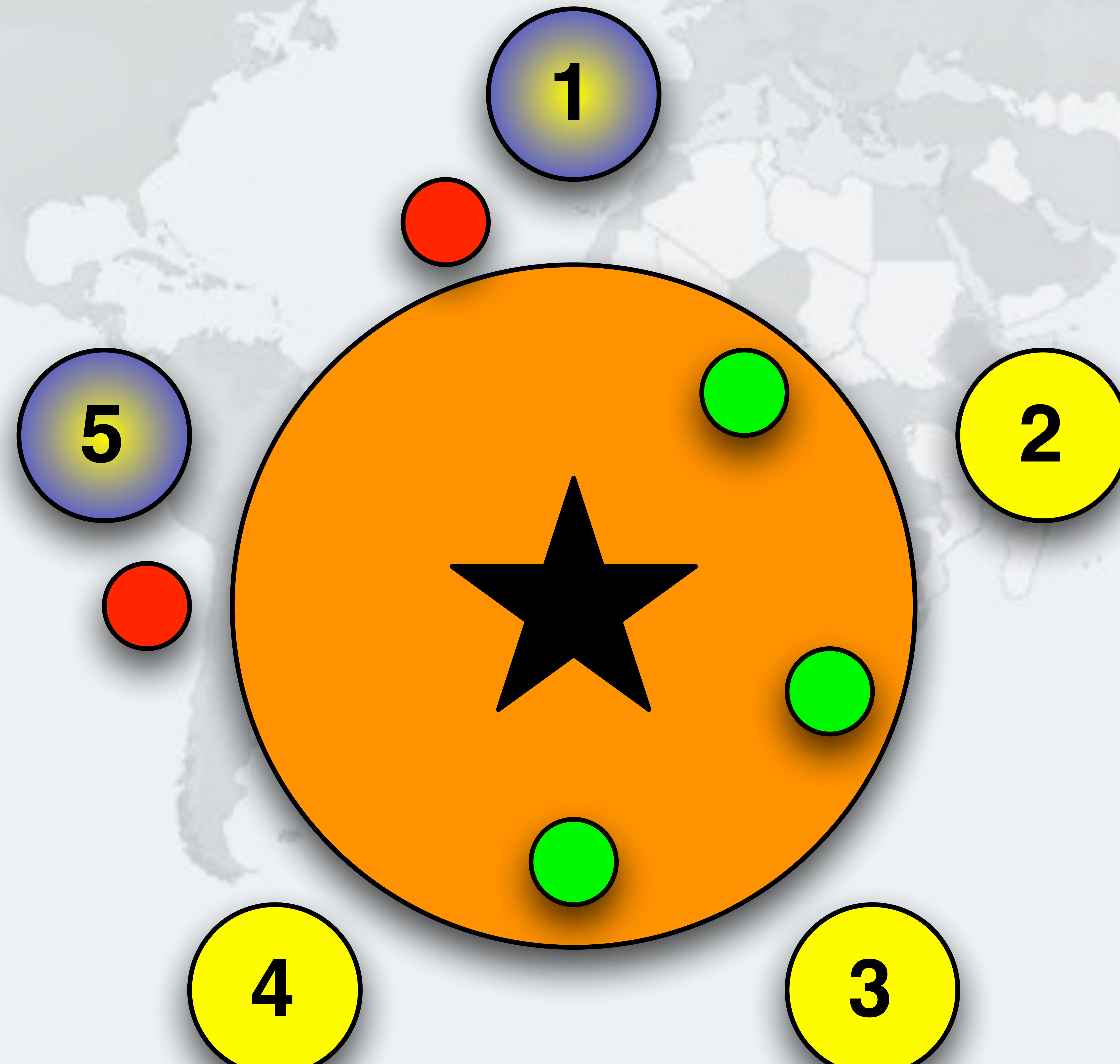
Philosopher 5 wants to drink, takes right cup



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 1 wants to drink, takes right cup

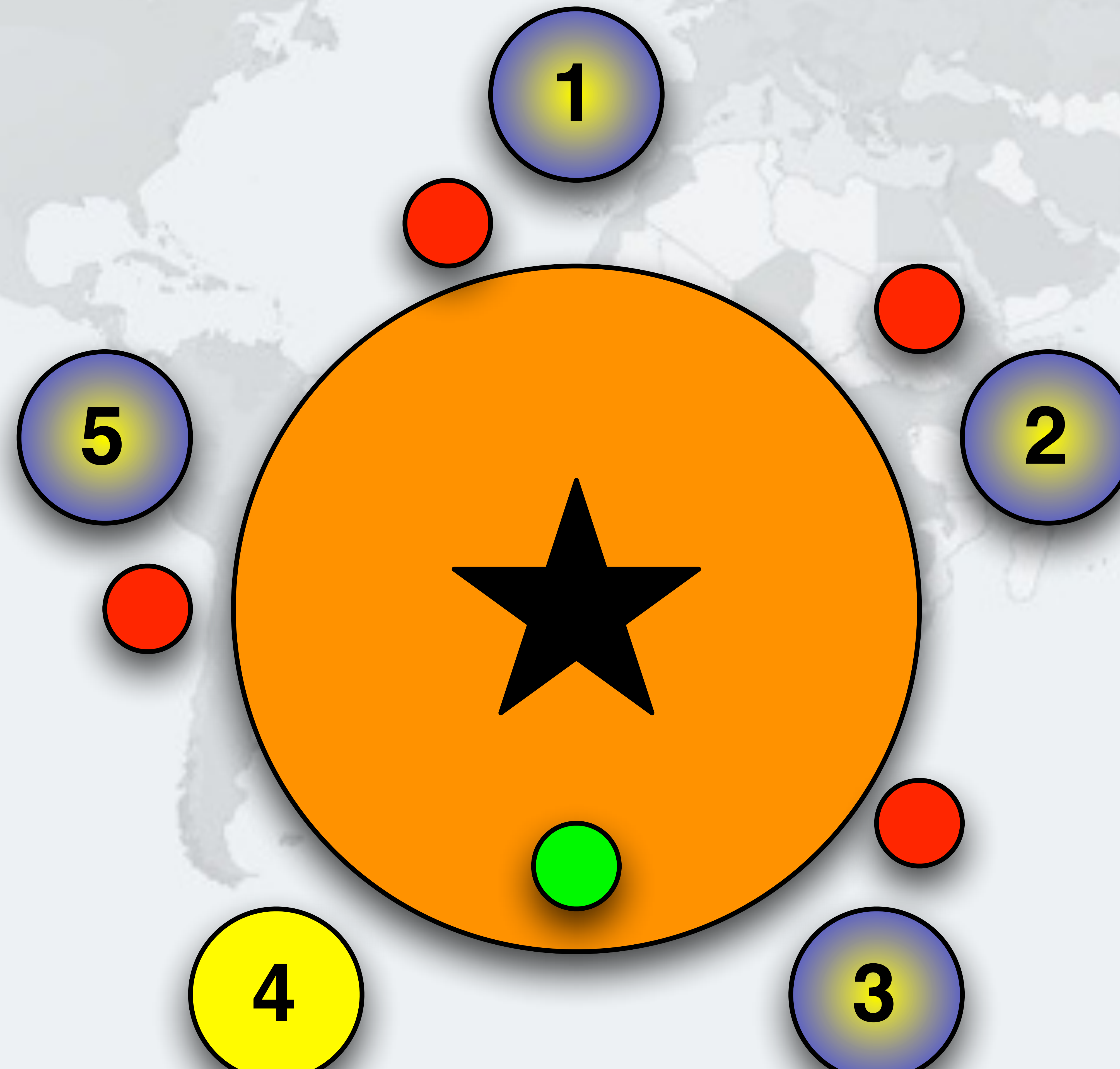


Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 2 wants to drink, takes right cup

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

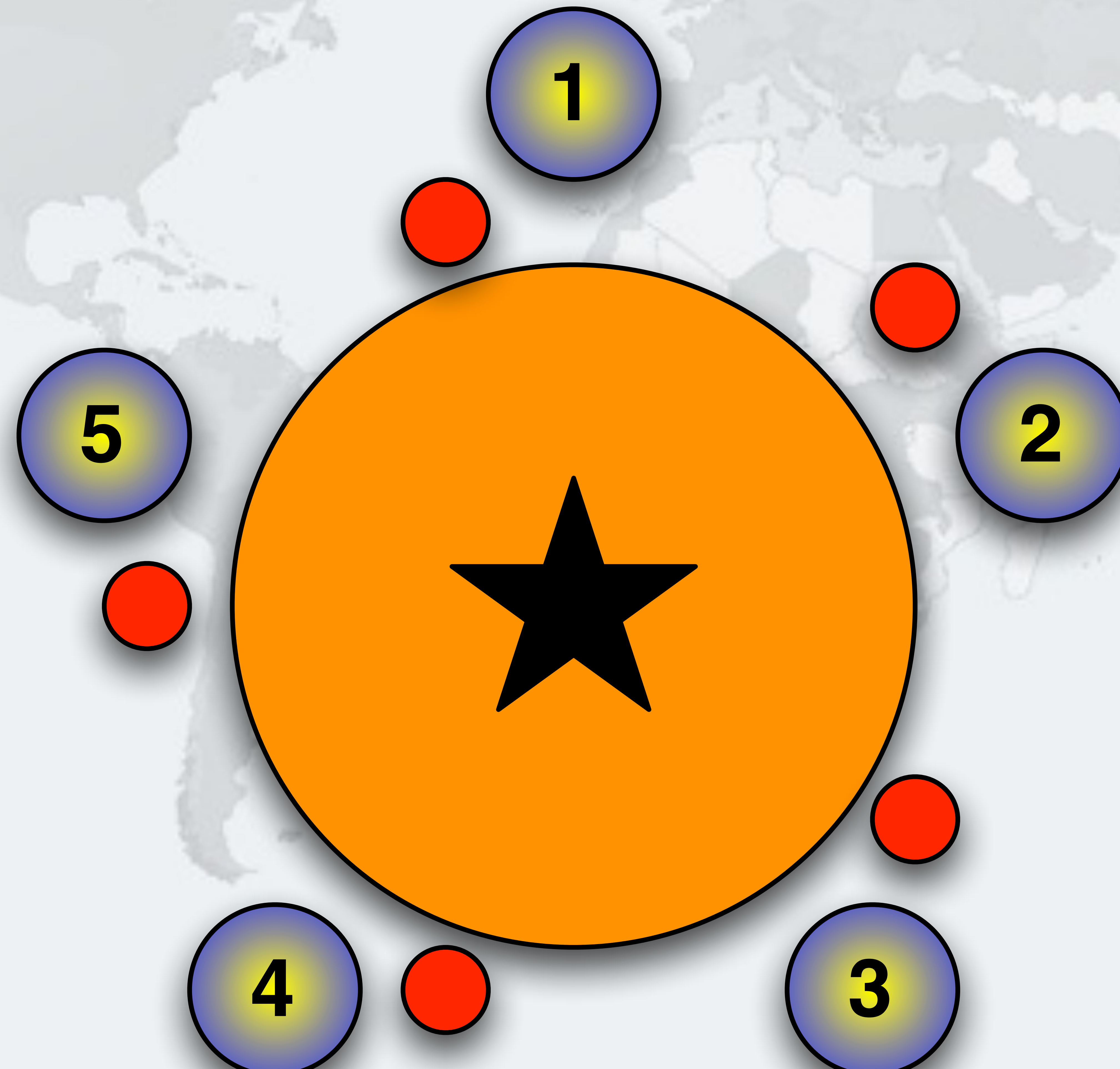
Philosopher 3 wants to drink, takes right cup



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

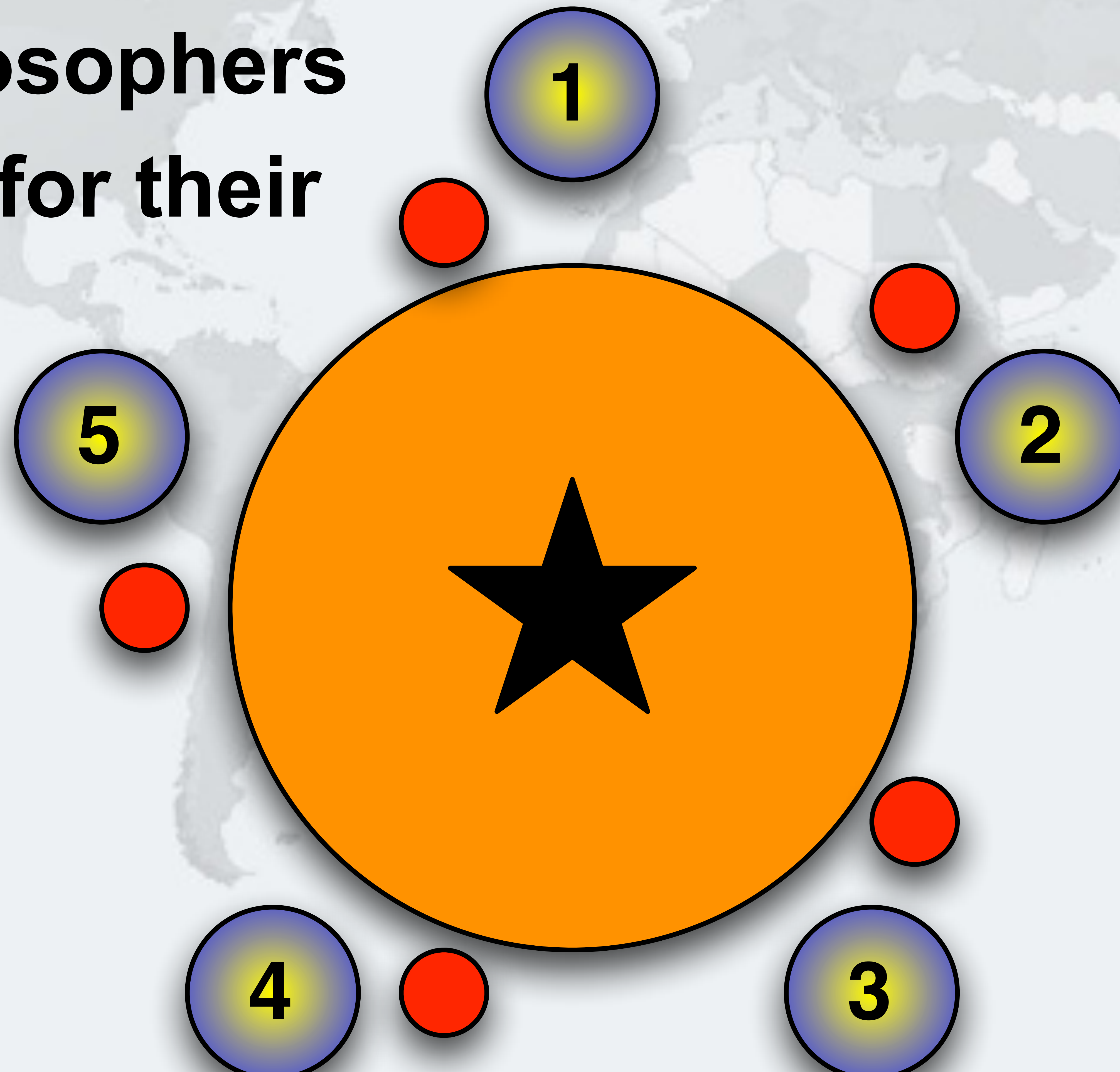
Philosopher 4 wants to drink, takes right cup



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Deadlock!

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



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Global order with boozing philosophers

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

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

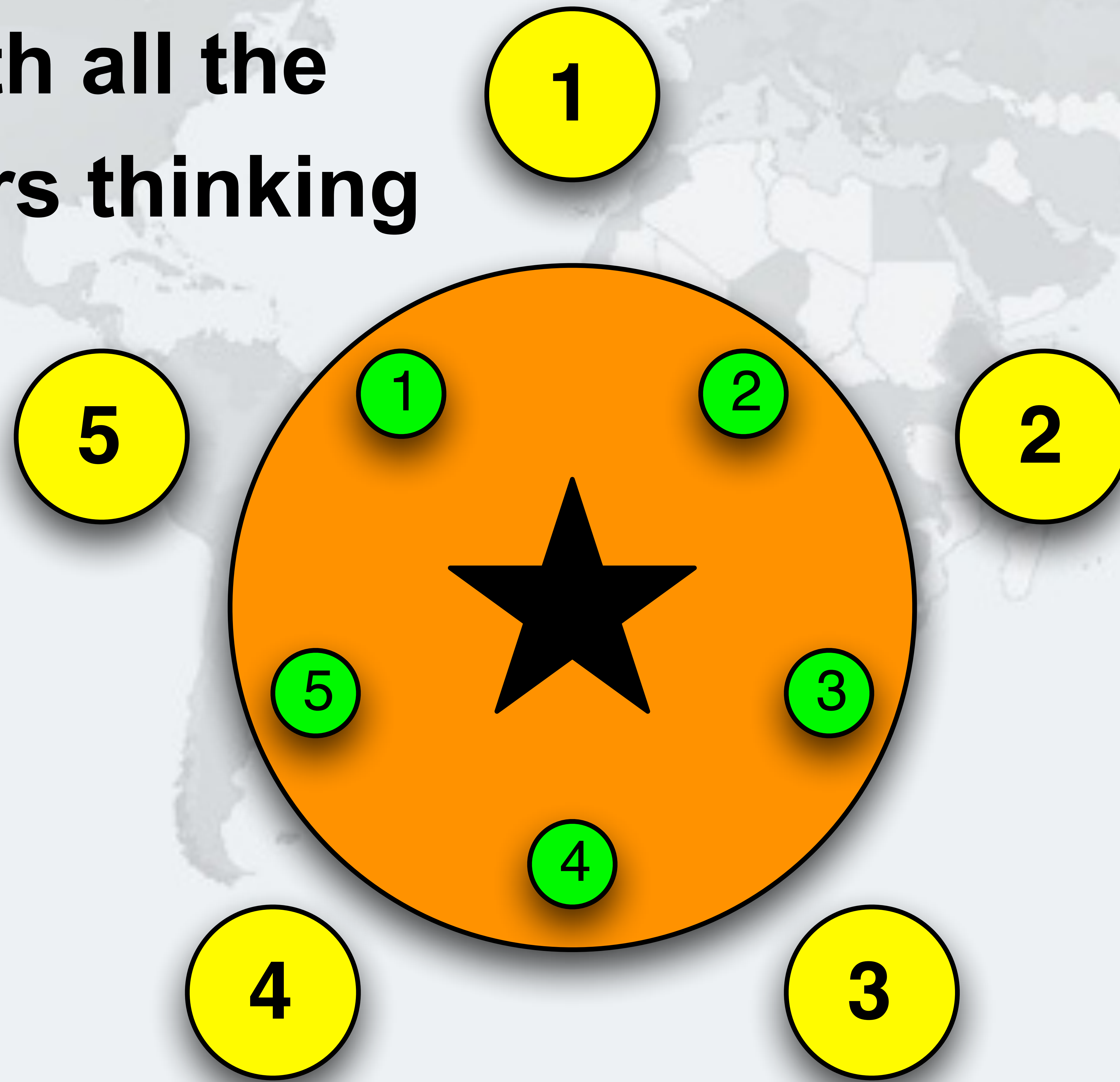
fixed order of lock acquisition

- **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**

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Global Lock ordering

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

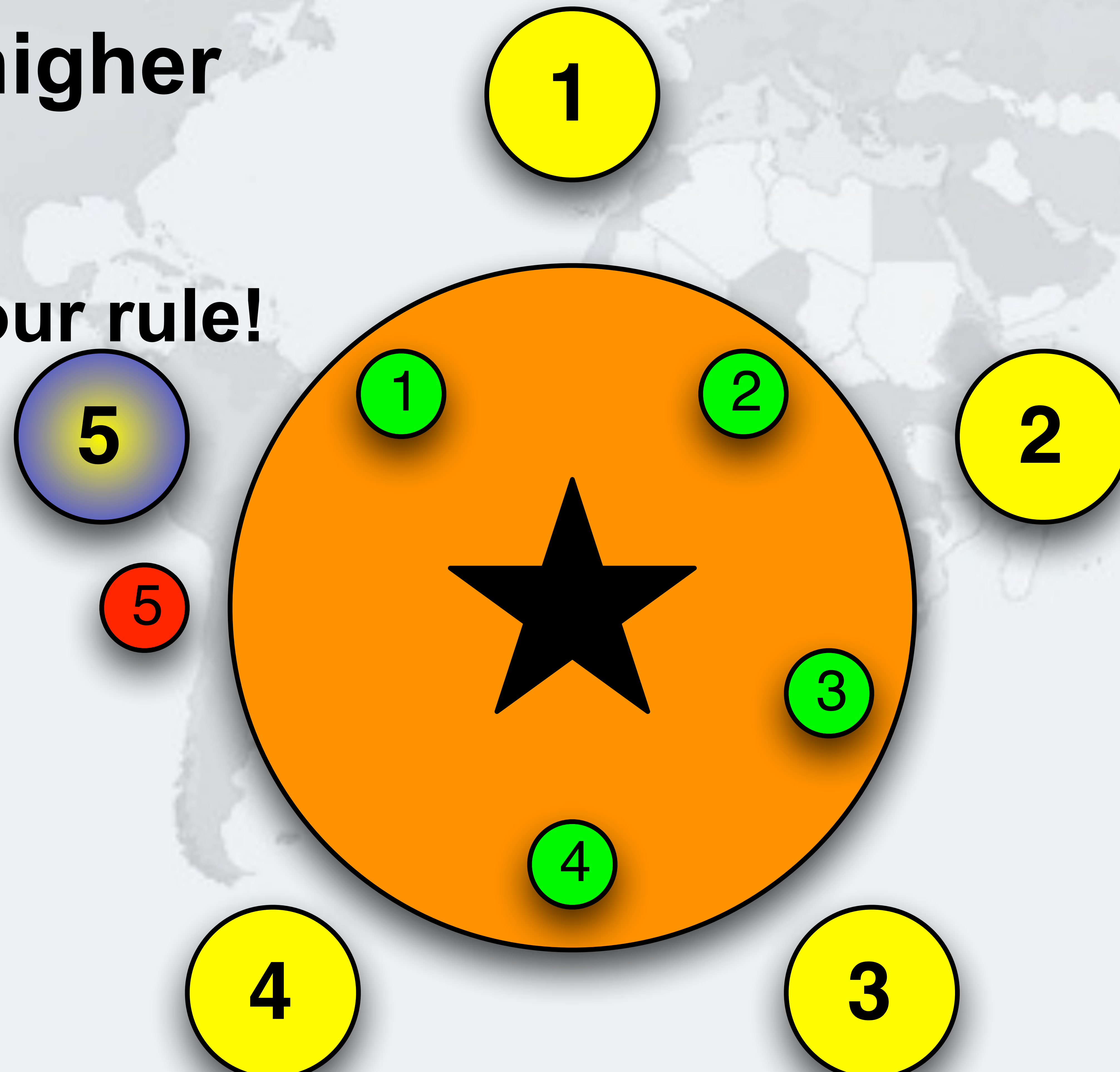


Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 5 takes cup 5

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



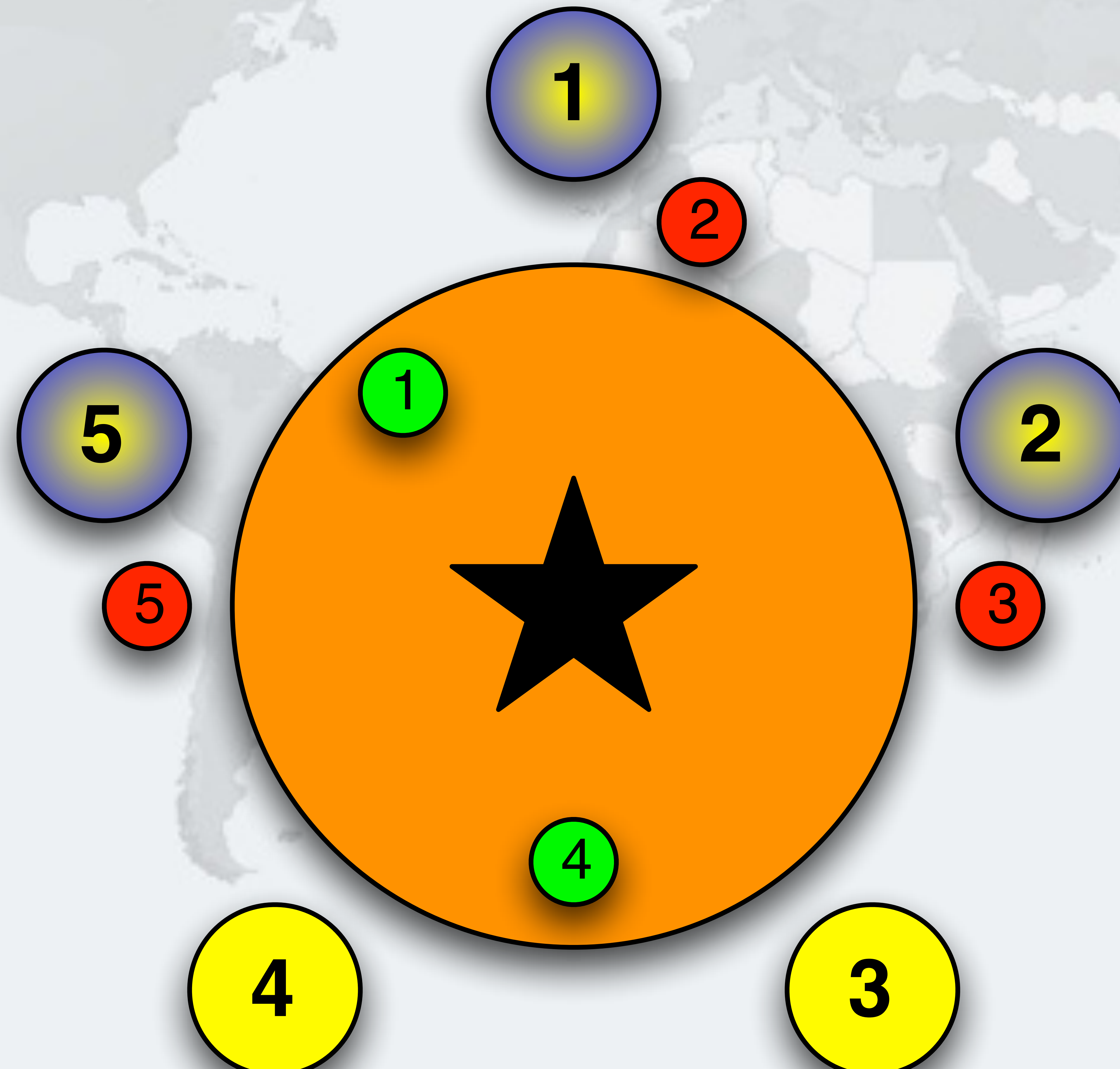
Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 1 takes cup 2

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

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 2 takes cup 3



Javaspecialists.eu

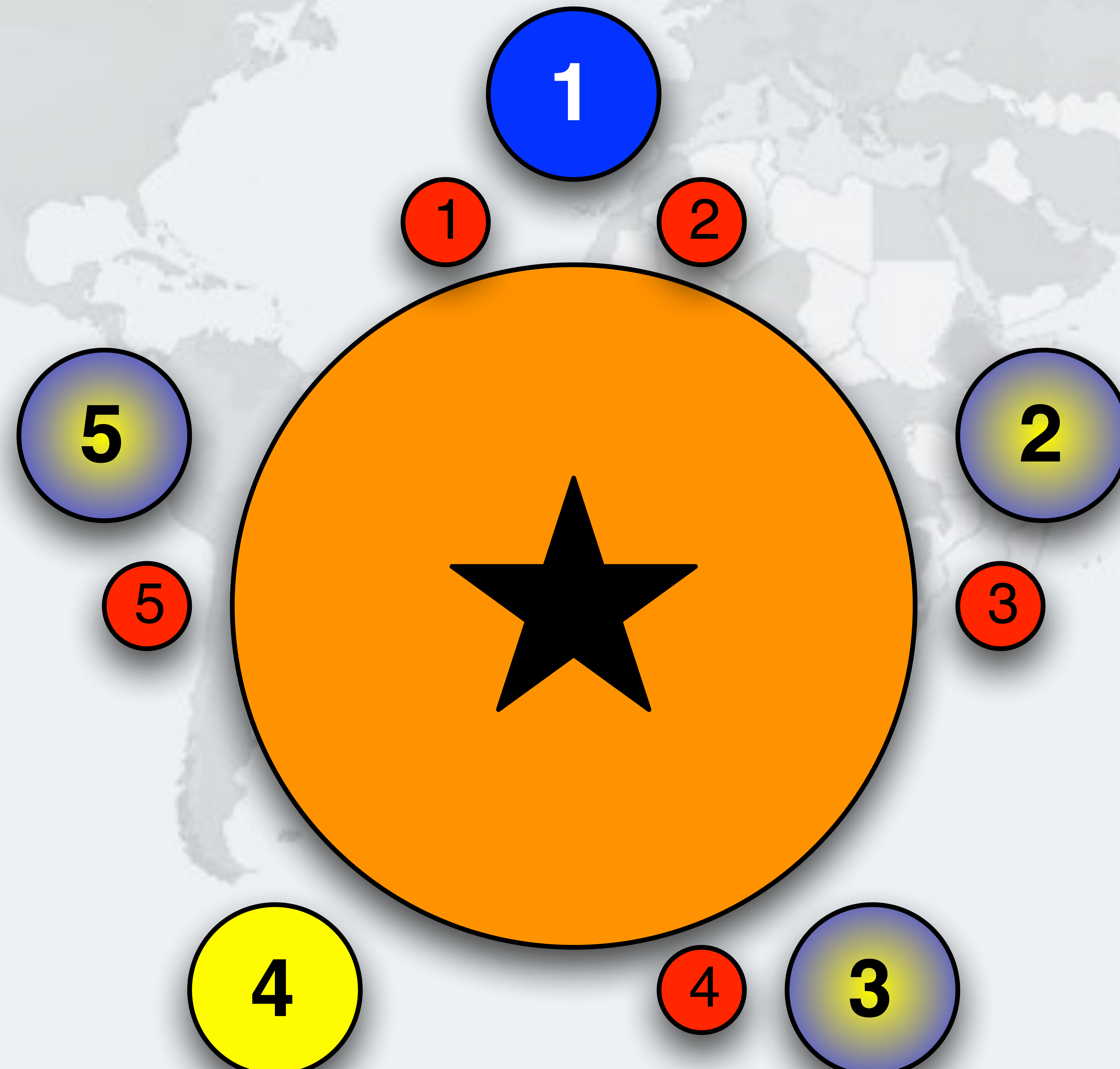
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 3 takes cup 4

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

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

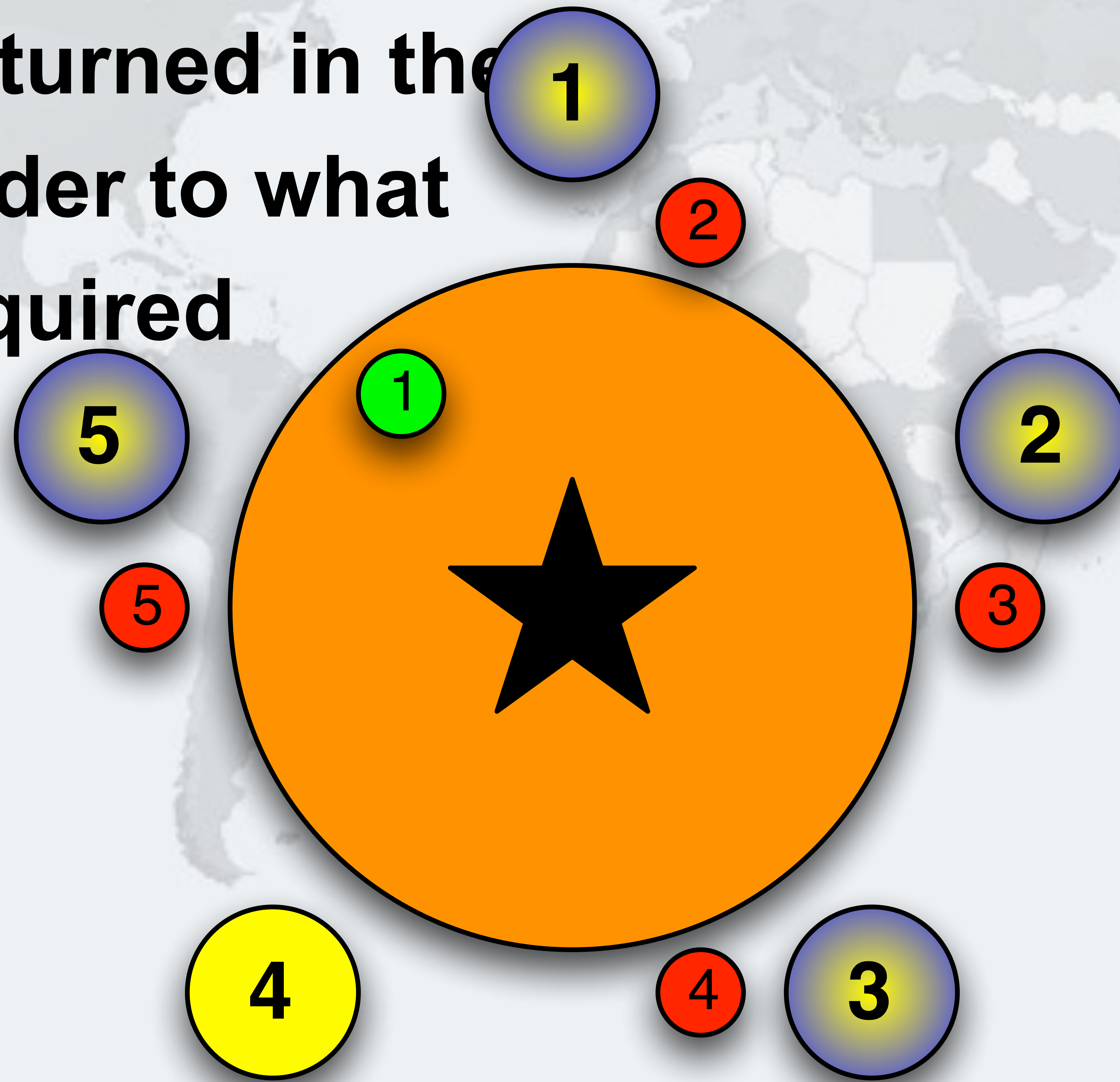
Philosopher 1 takes cup 1 - Drinking



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

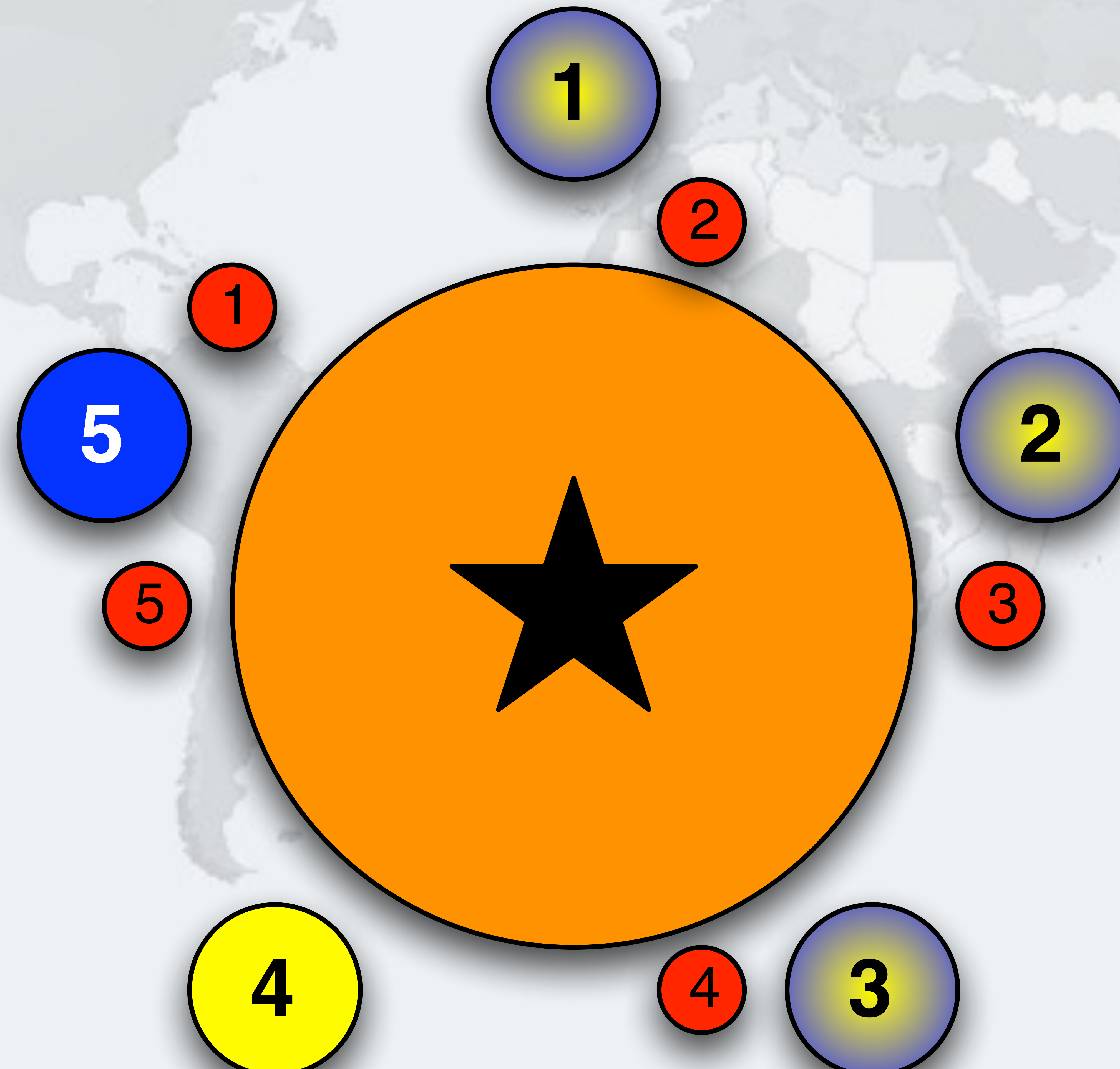
Philosopher 1 returns Cup 1

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



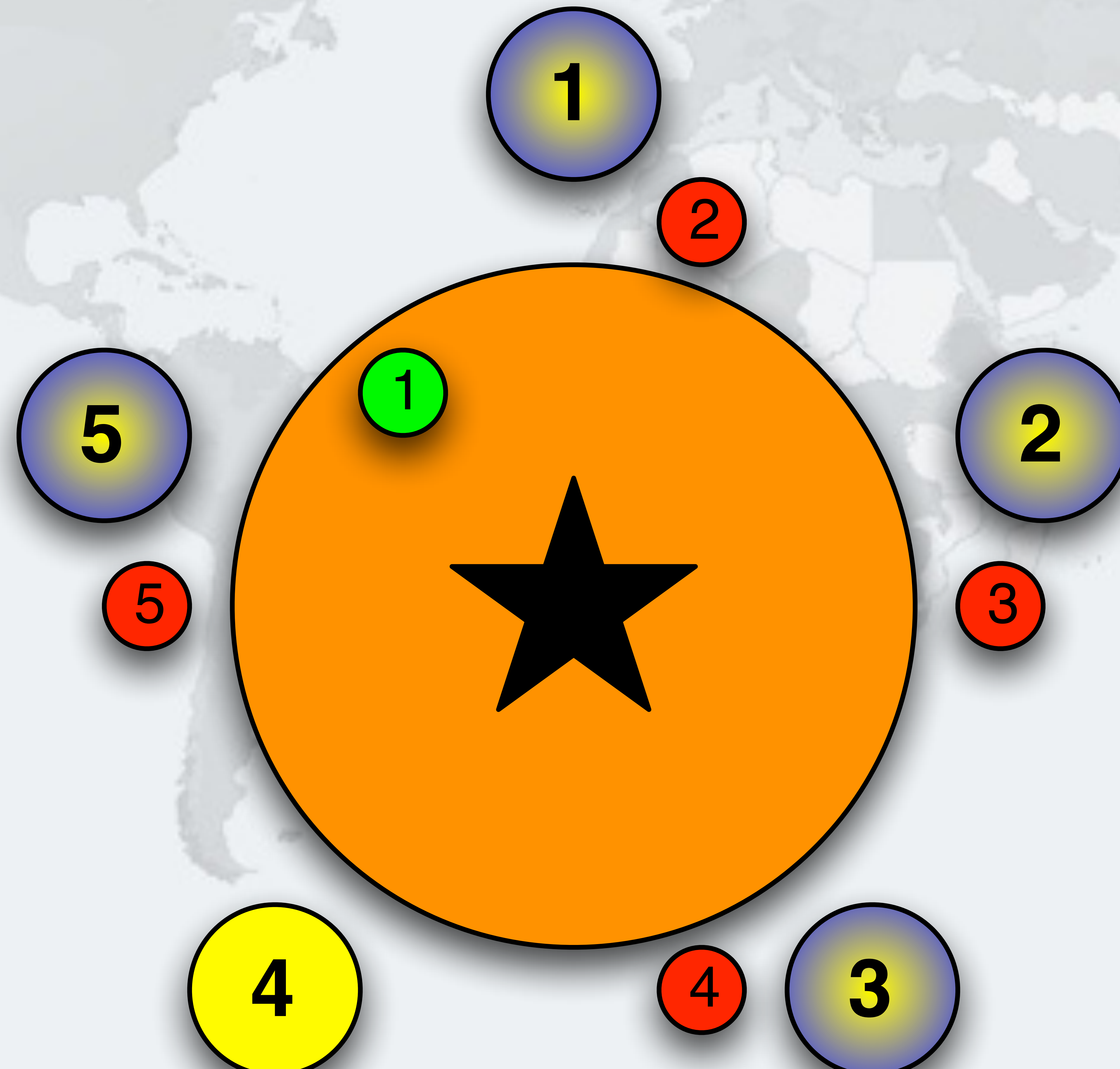
Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 5 takes cup 1 - Drinking



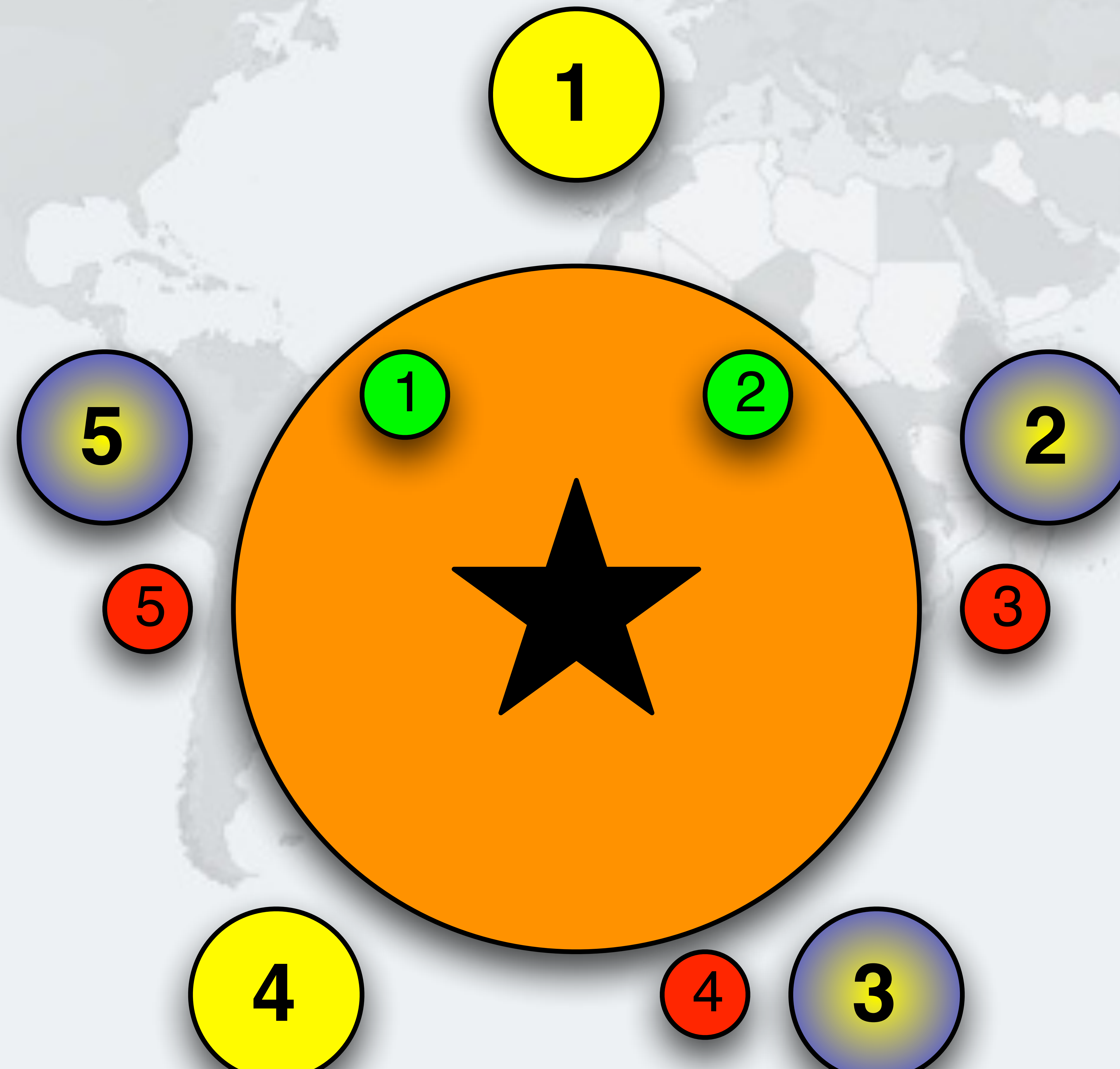
Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 5 returns cup 1



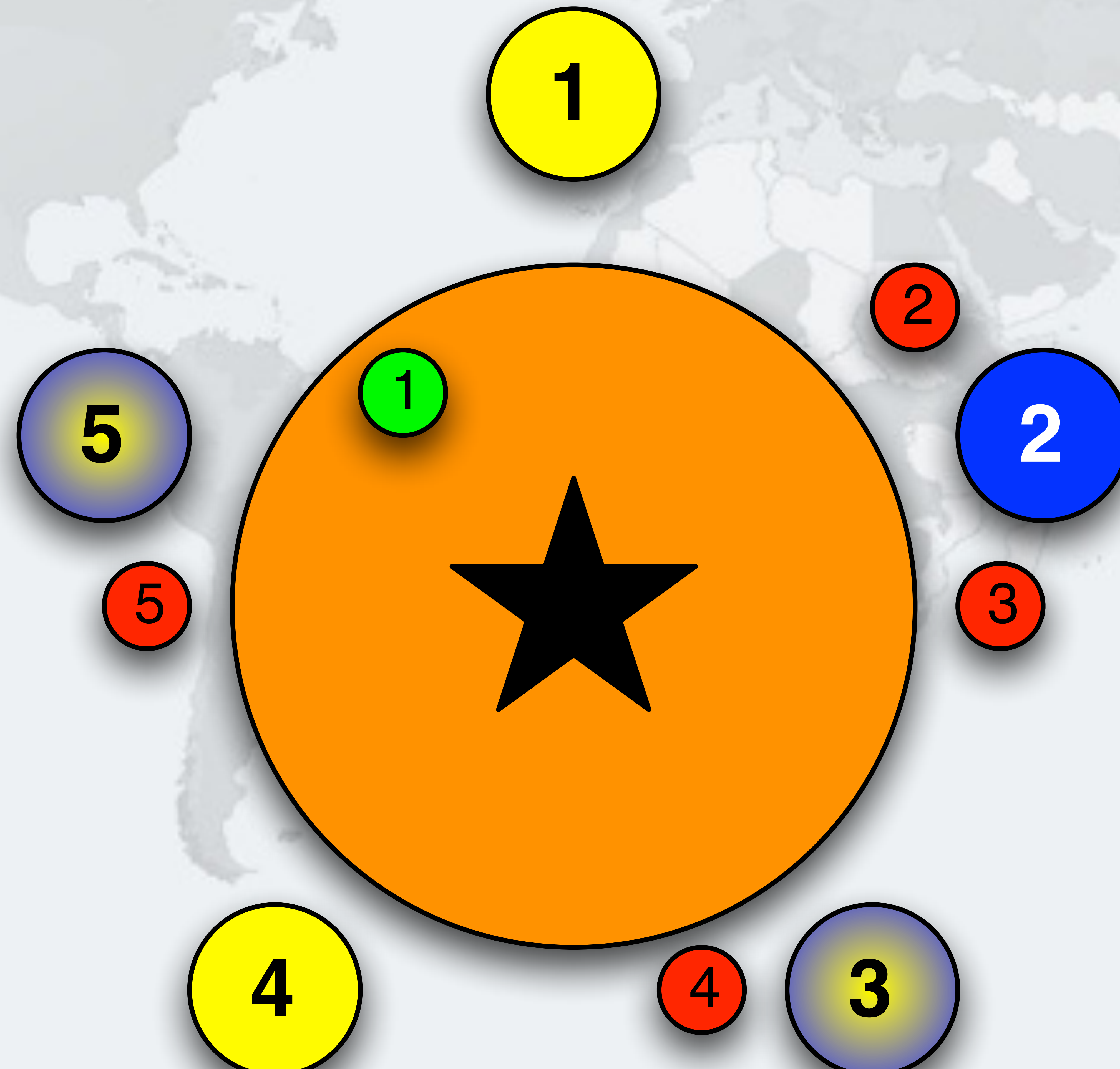
Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 1 returns cup 2



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

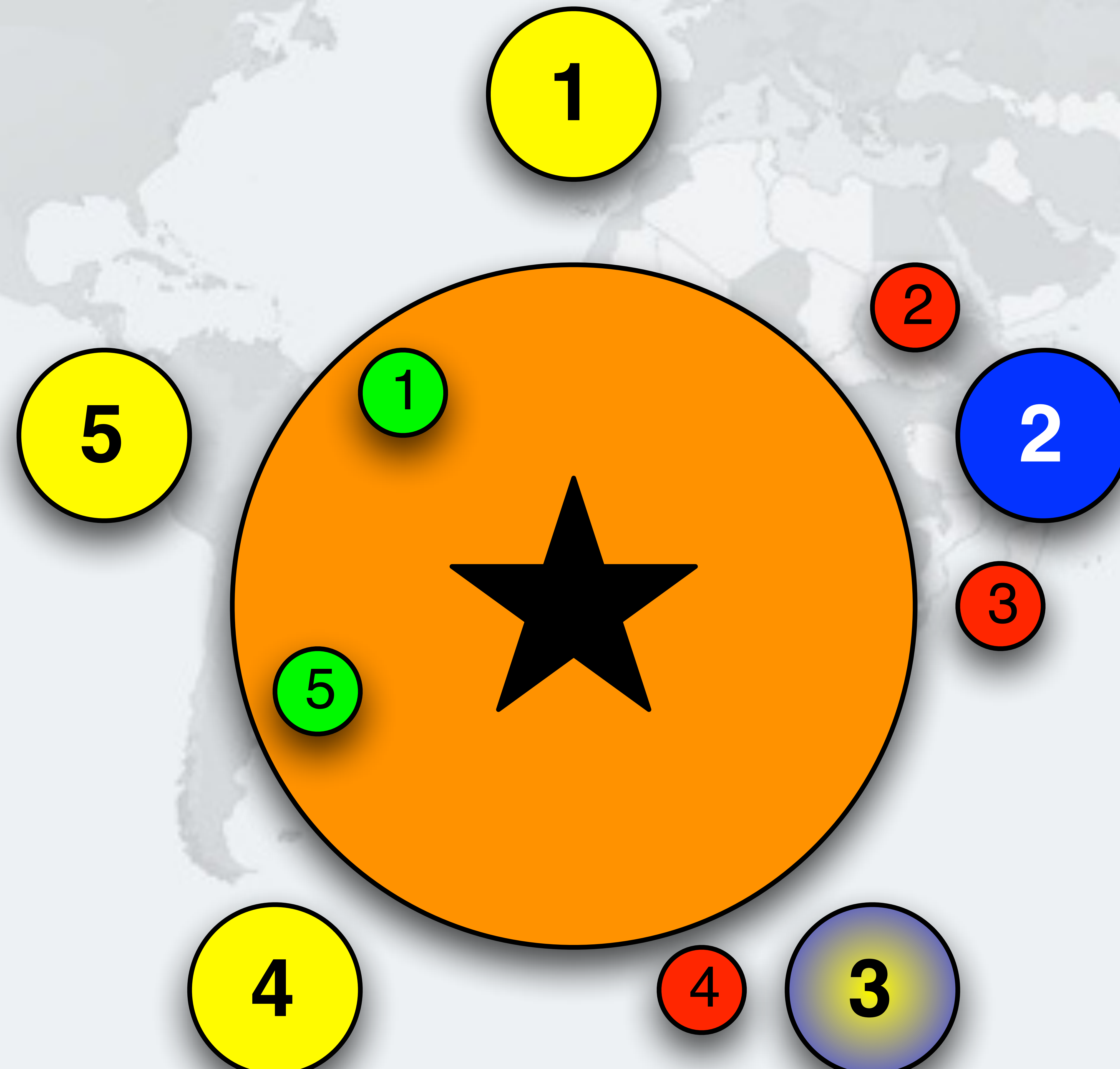
Philosopher 2 takes cup 2 - Drinking



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

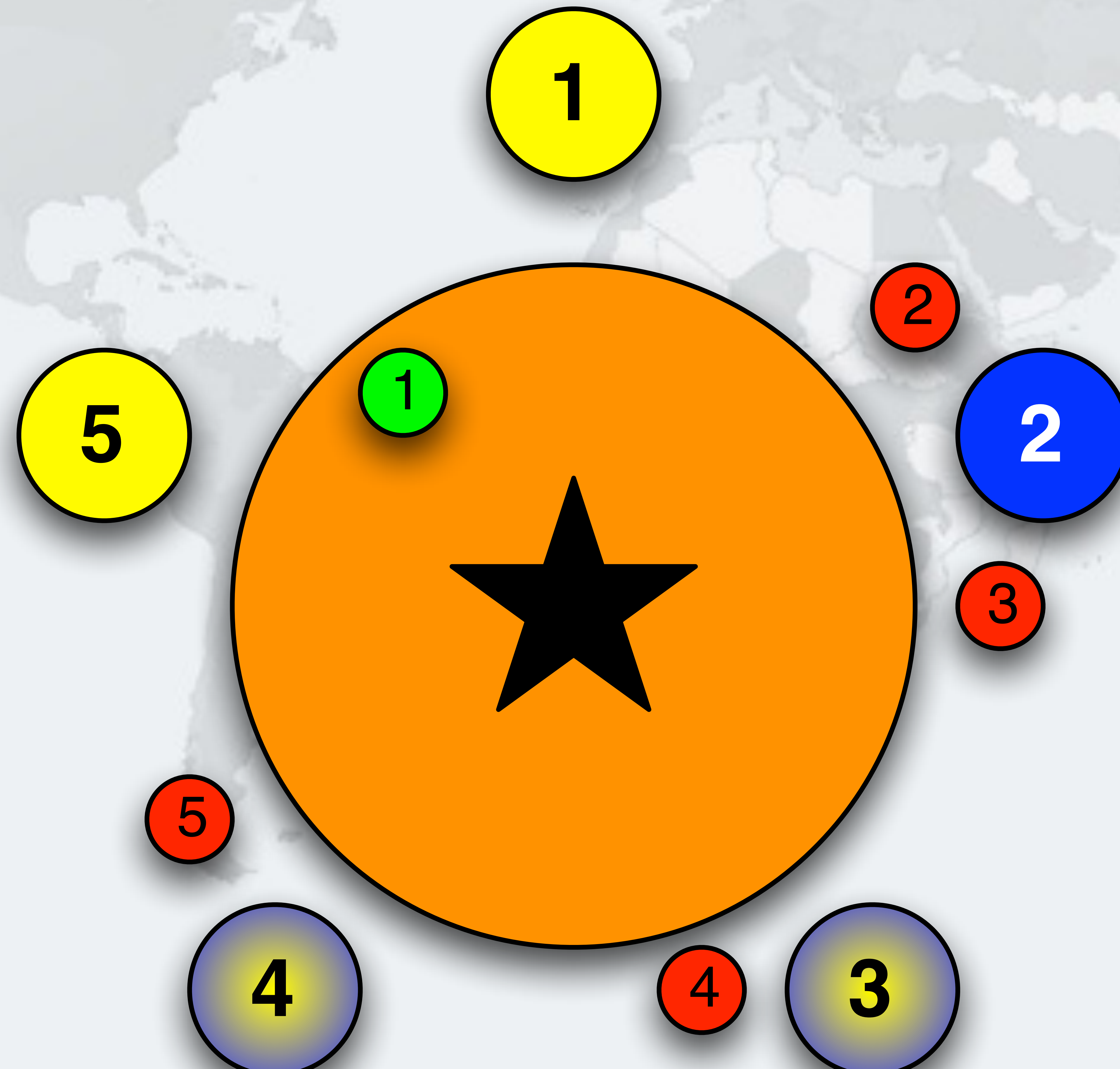
Philosopher 5 returns cup 5



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

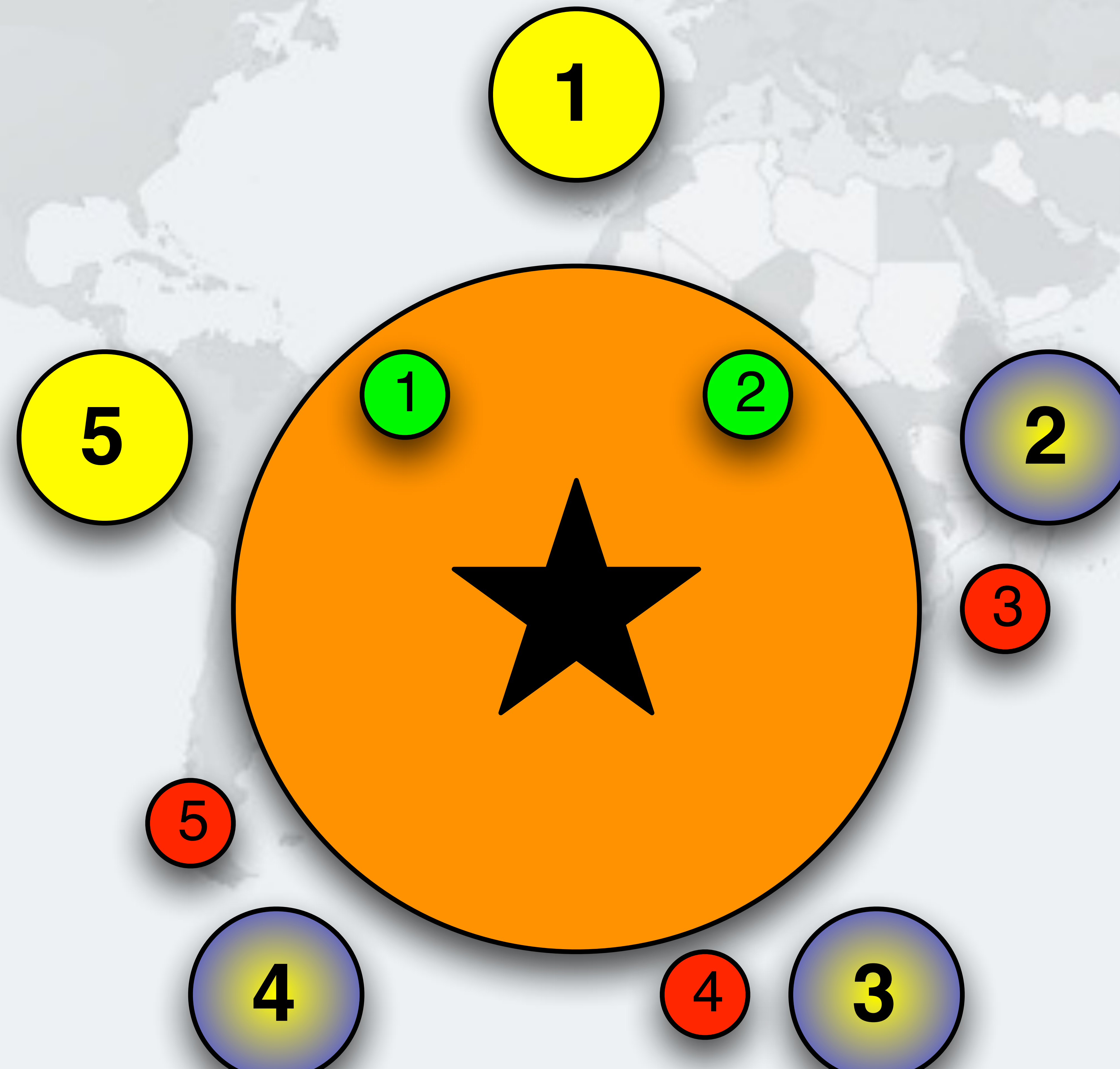
Philosopher 4 takes cup 5



Javaspecialists.eu

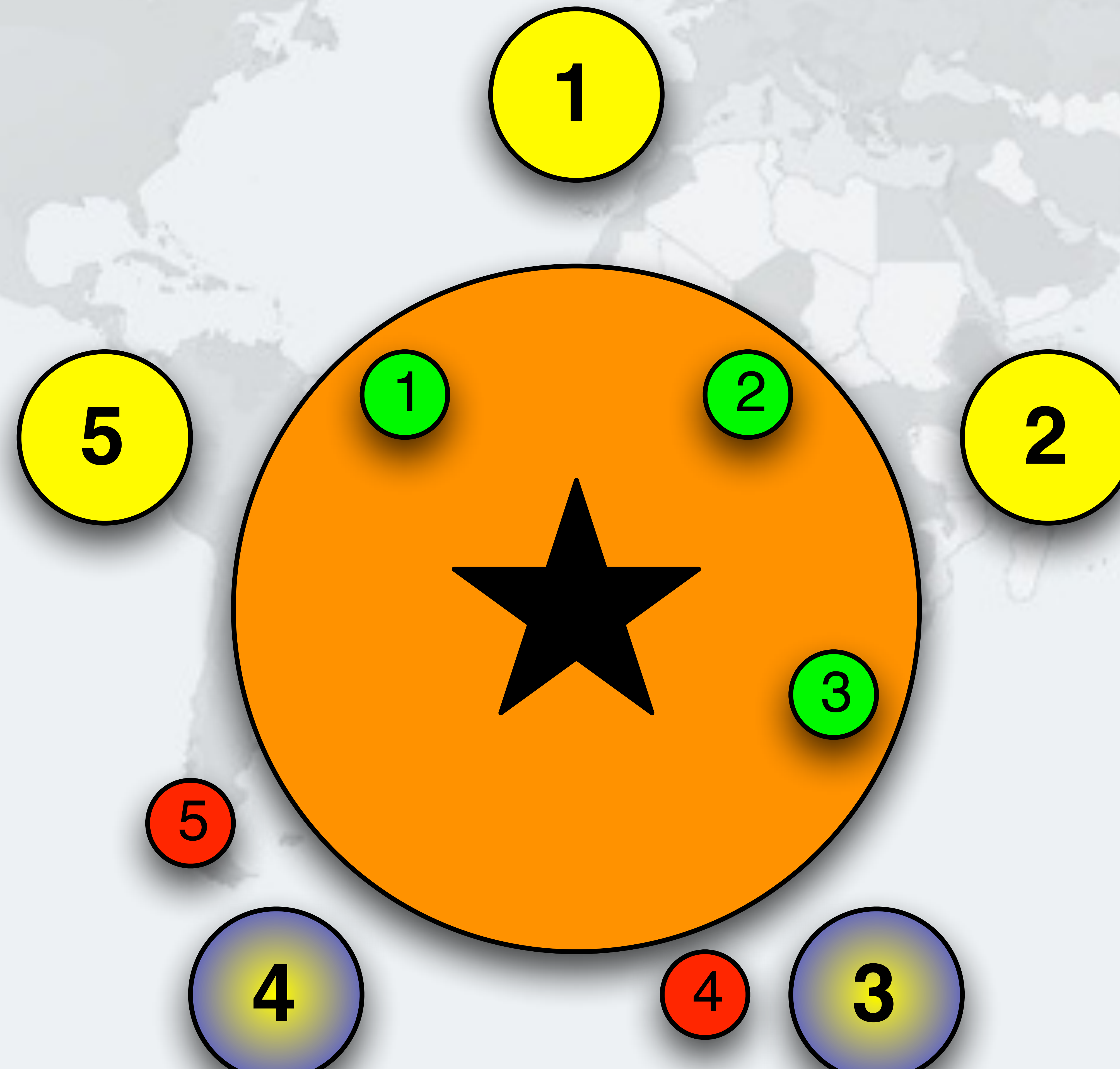
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 2 returns cup 2



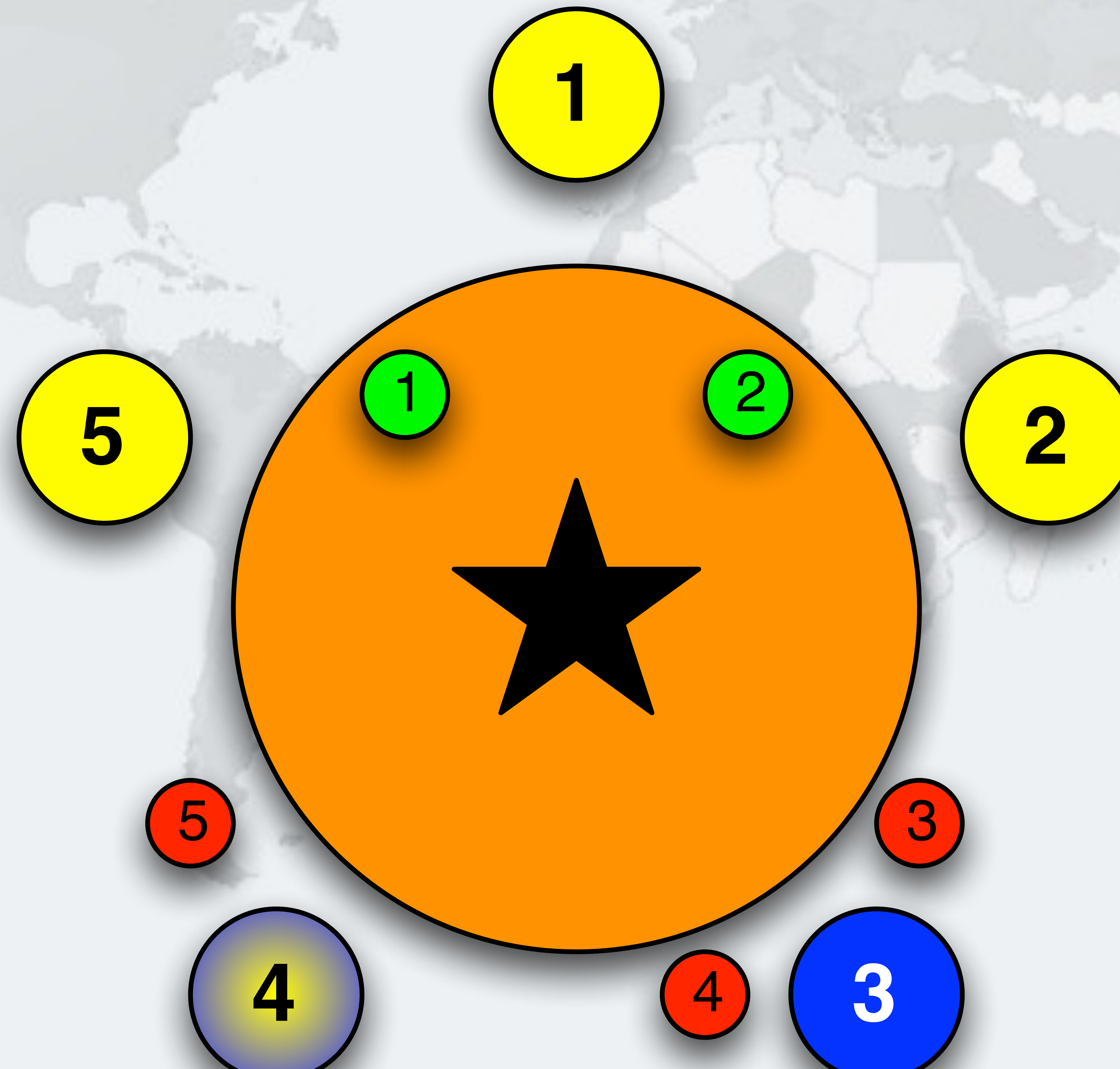
Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 2 returns cup 3



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

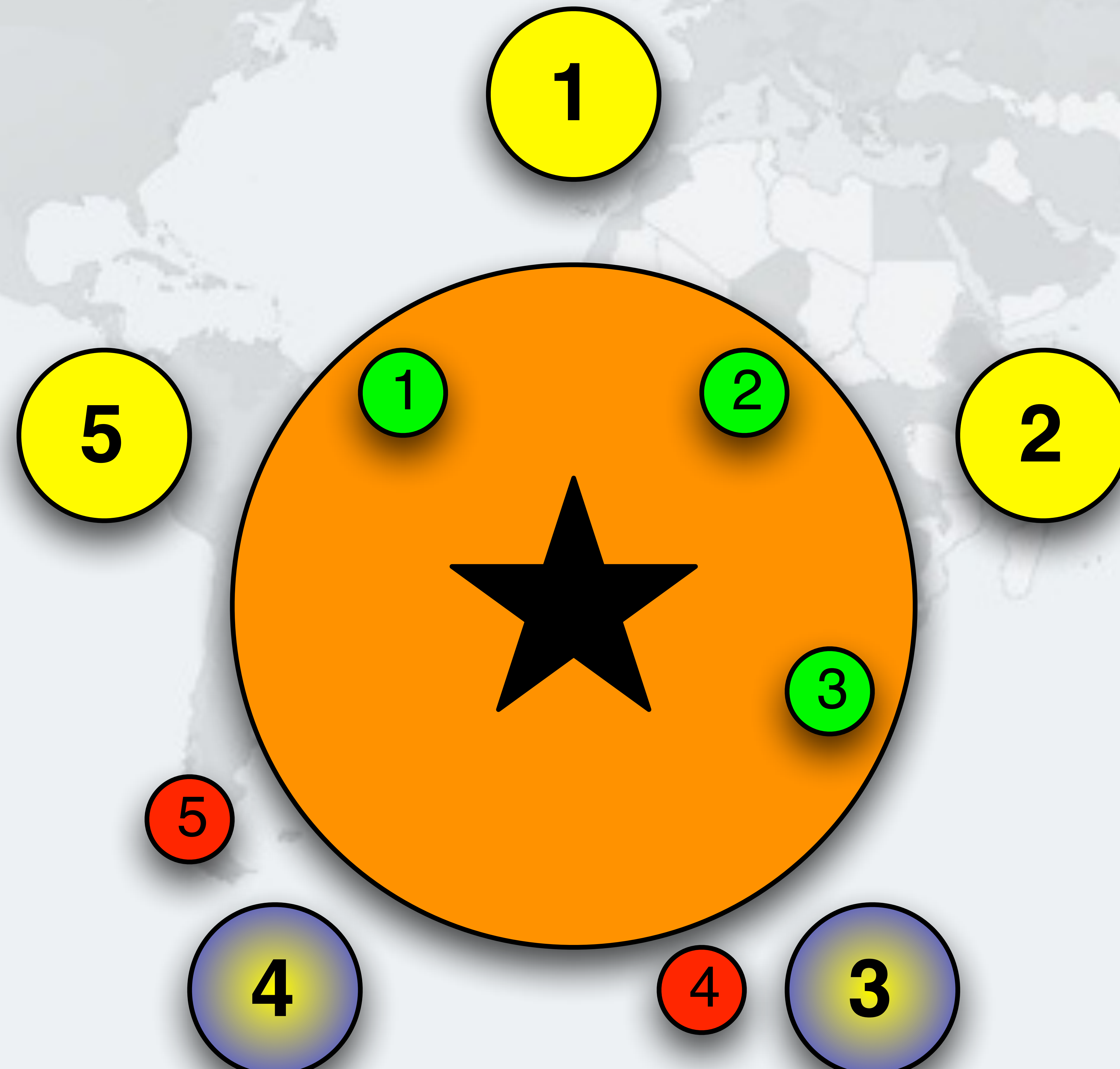
Philosopher 3 takes cup 3 - Drinking



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

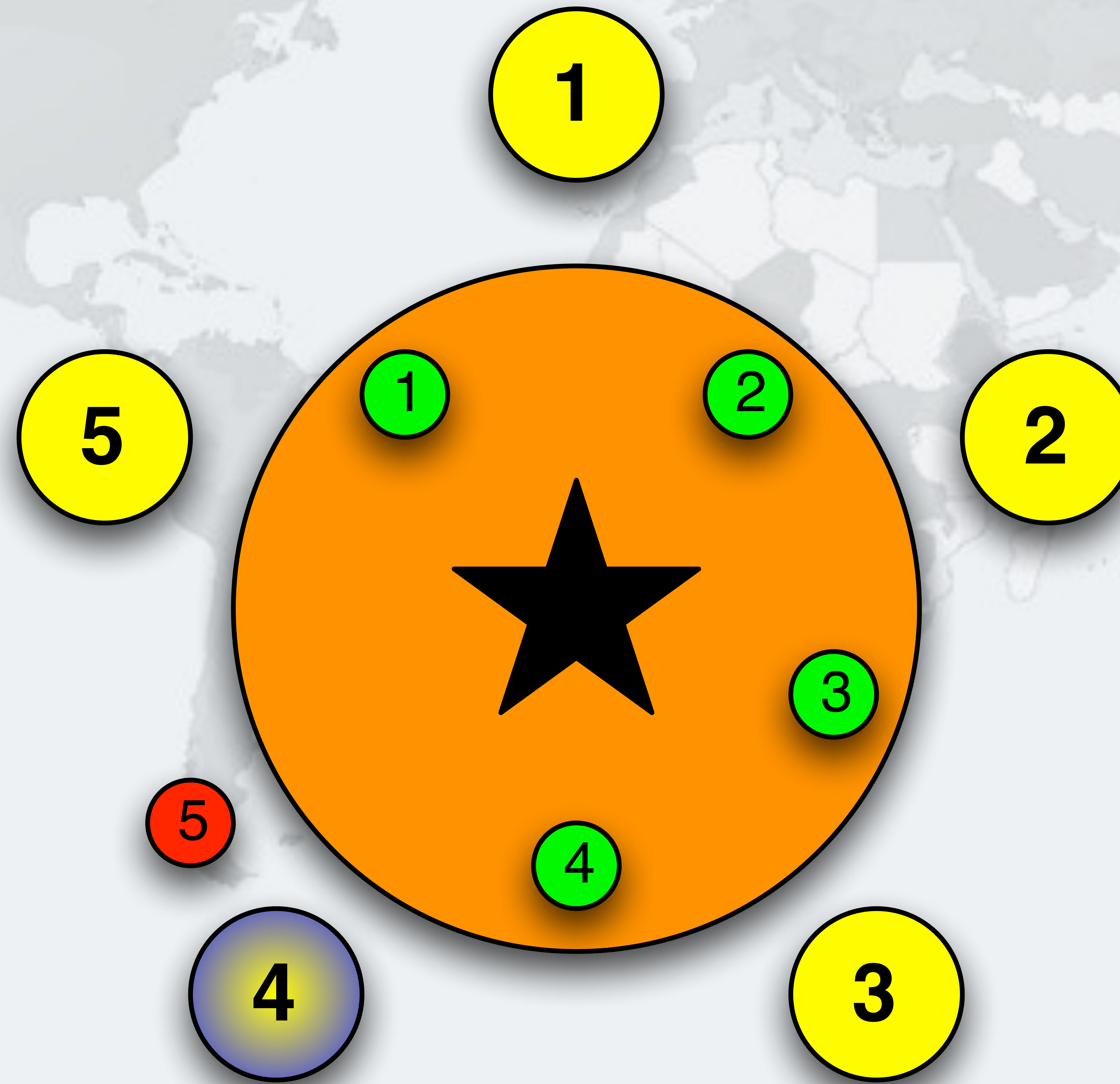
Philosopher 3 Returns cup 3



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

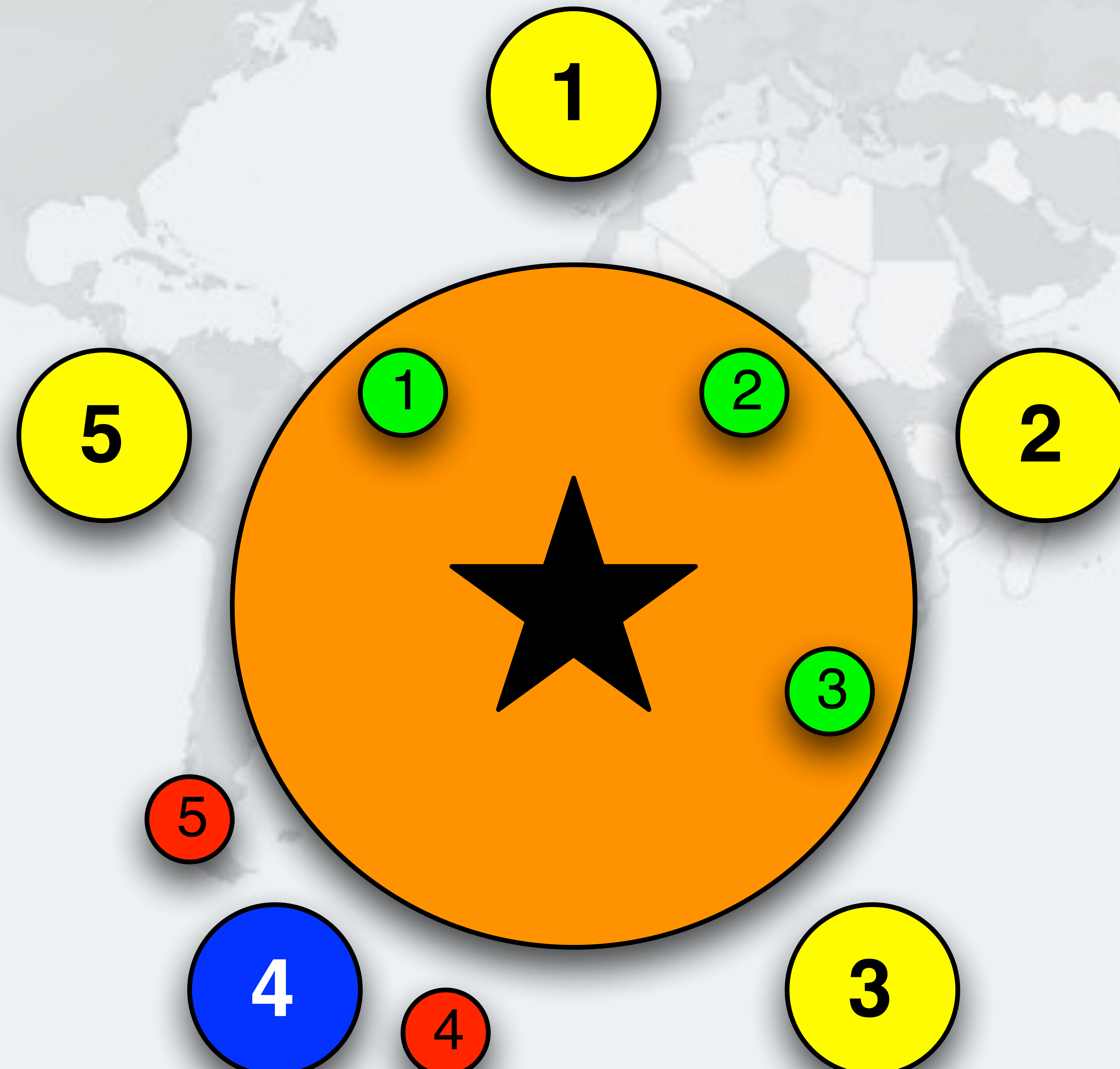
Philosopher 3 Returns cup 4



Javaspecialists.eu

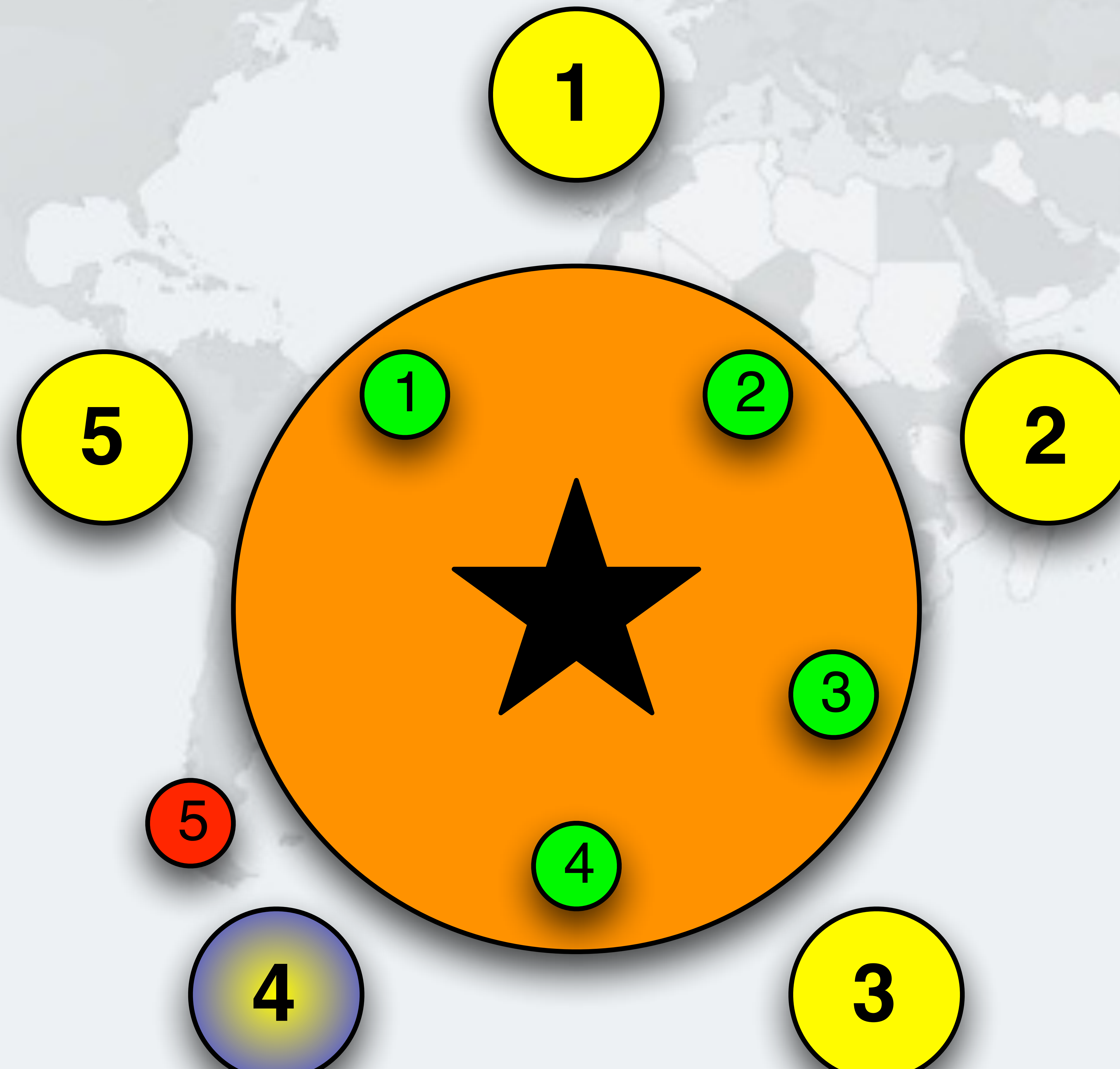
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 4 takes cup 4 - Drinking



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

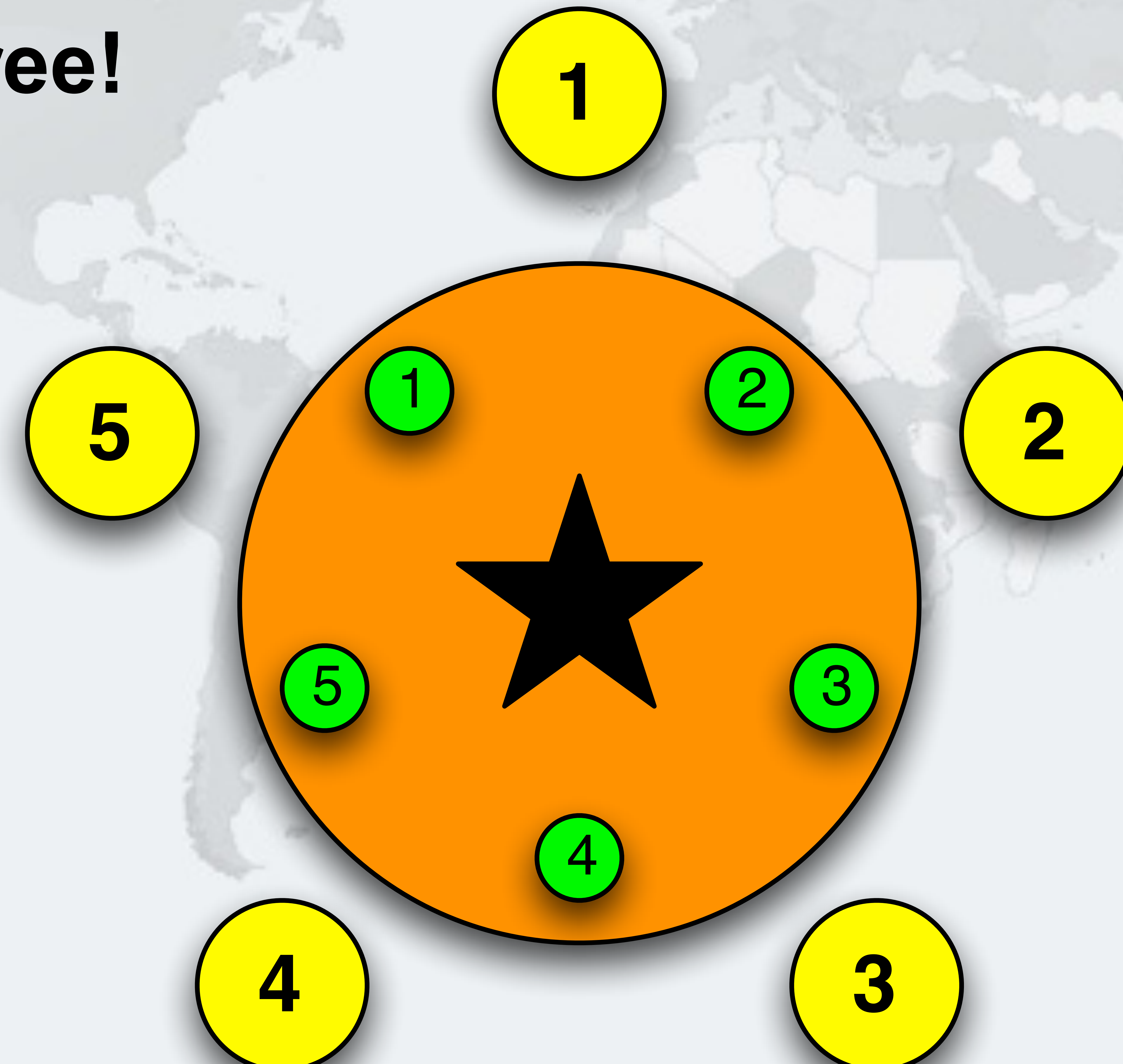
Philosopher 4 Returns cup 4



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 4 Returns cup 5

- **Deadlock free!**



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Deadlock is avoided

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

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Tools jstack and jps

- **We have command line tools**
  - `jps`
    - **shows your Java process ids**
  - `jstack -l pid`
    - **shows what your JVM is currently doing**
  - **Tools are in your jdk/bin directory**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 1 Exercise

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu
java training

Lab1 - Save our philosophers

- **Define a global order on the locks**
  1. **Run eu.javaspecialists.deadlock.lab1.Main**
  2. **Make the Krasi object implement Comparable**
  3. **Lock first on bigger, than on smaller Krasi**
  4. **Verify that the deadlock has now disappeared**

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 2: Deadlock resolution by tryLock

Avoiding Liveness Hazards

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu
java training

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?

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu

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)**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

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**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

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();
  }
}
```

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu

Polled lock acquisition

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

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

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

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**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

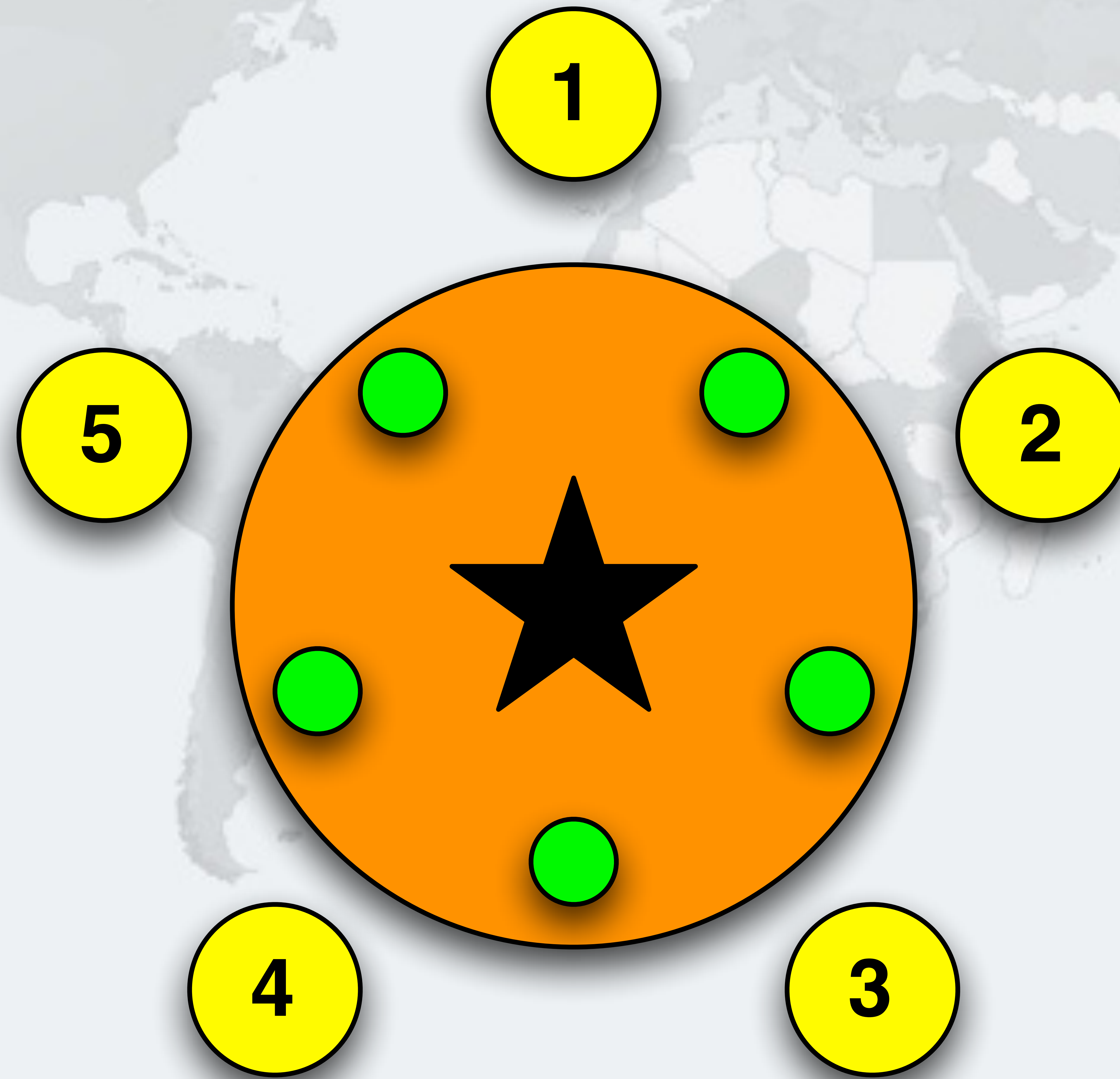
Using Trylock() To Avoid Deadlocks

```
public void drink() {
  while (true) {
    left.lock();
    try {
      if (right.tryLock()) {
        try {
          // now we can finally drink and then return
          return;
        } finally {
          right.unlock();
        }
      }
    } finally {
      left.unlock();
    }
  }
}
```

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

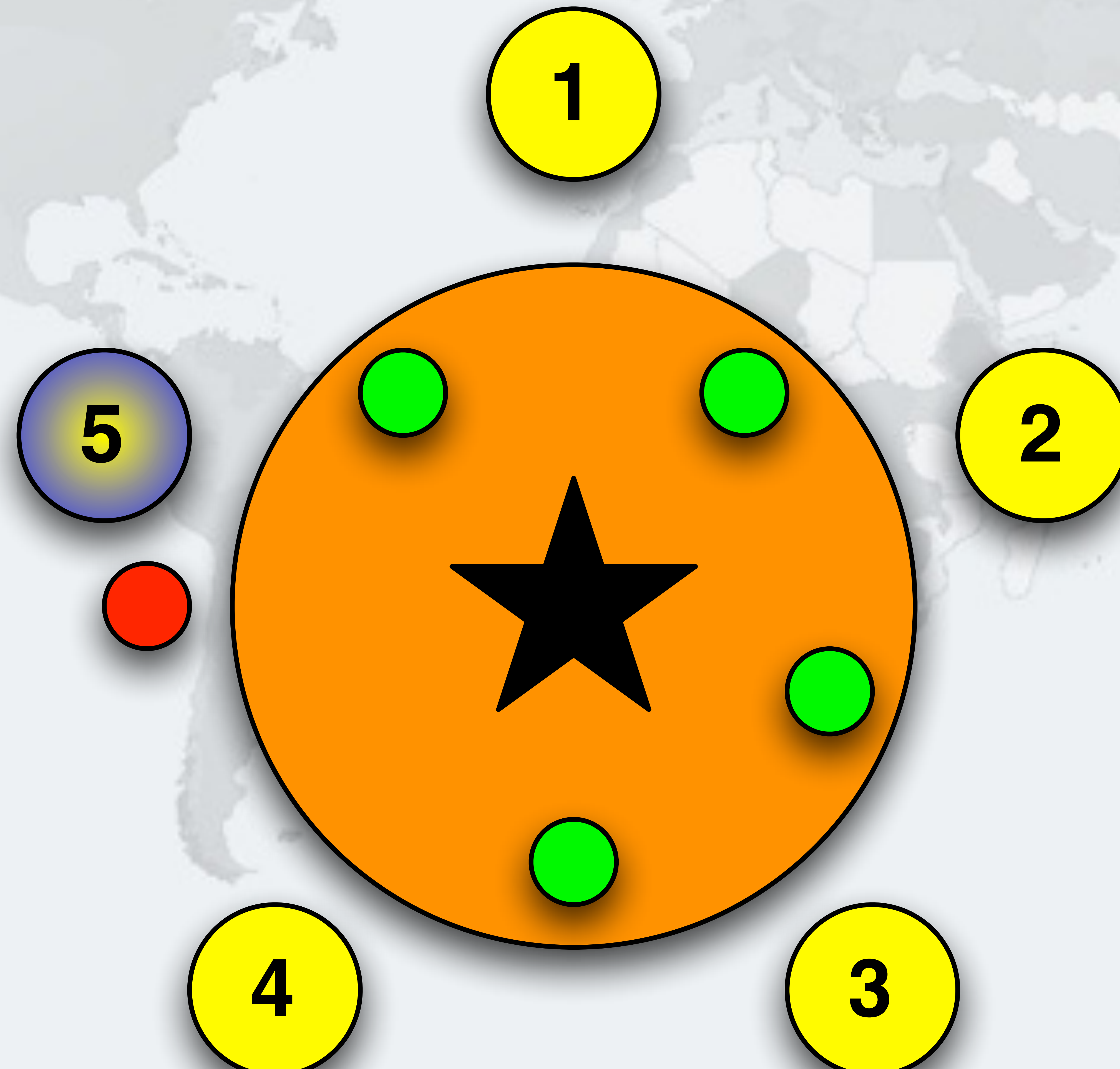
Deadlock is prevented in this design



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

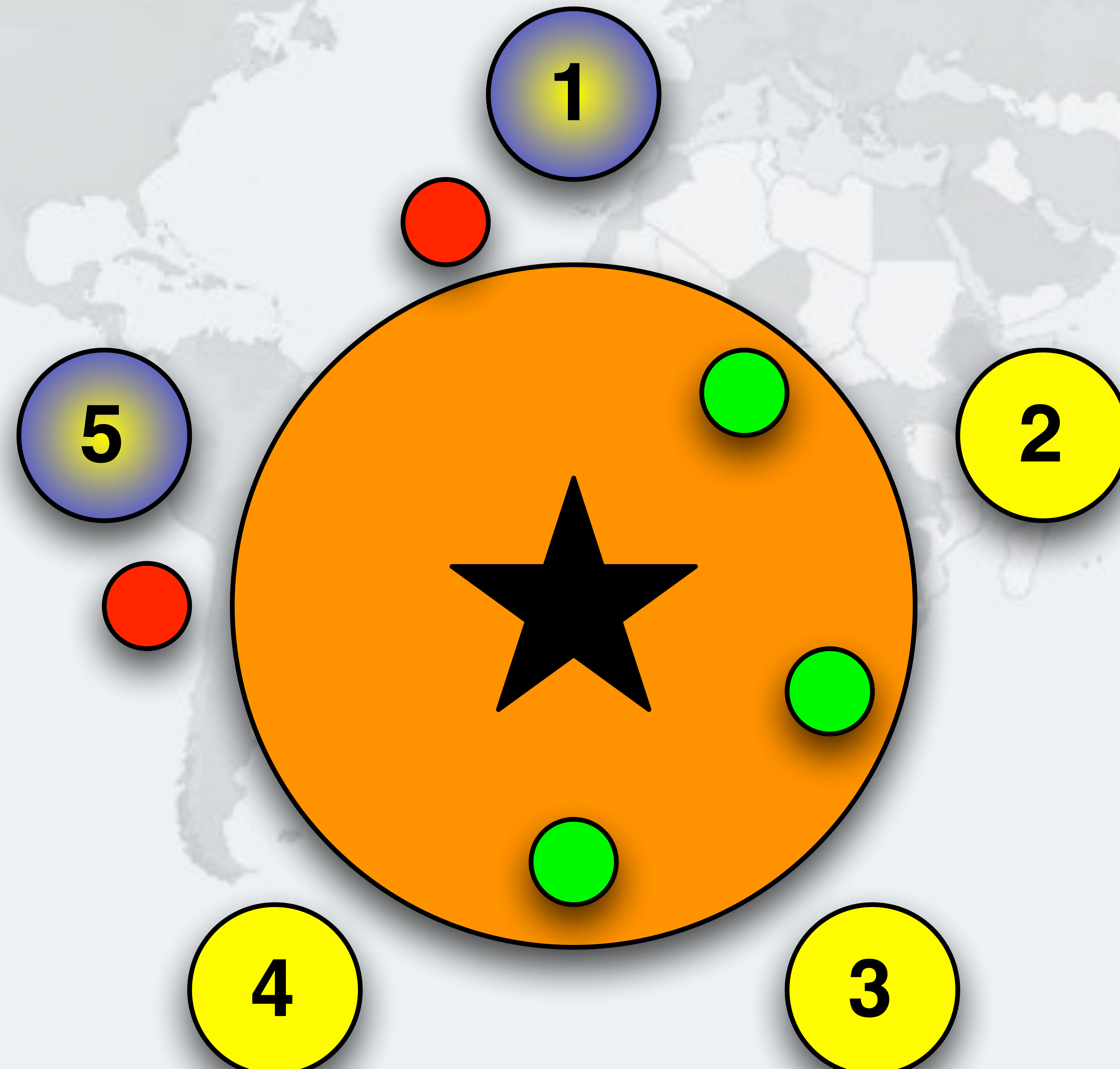
Philosopher 5 wants to drink, takes right cup



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 1 wants to drink, takes right cup



Javaspecialists.eu

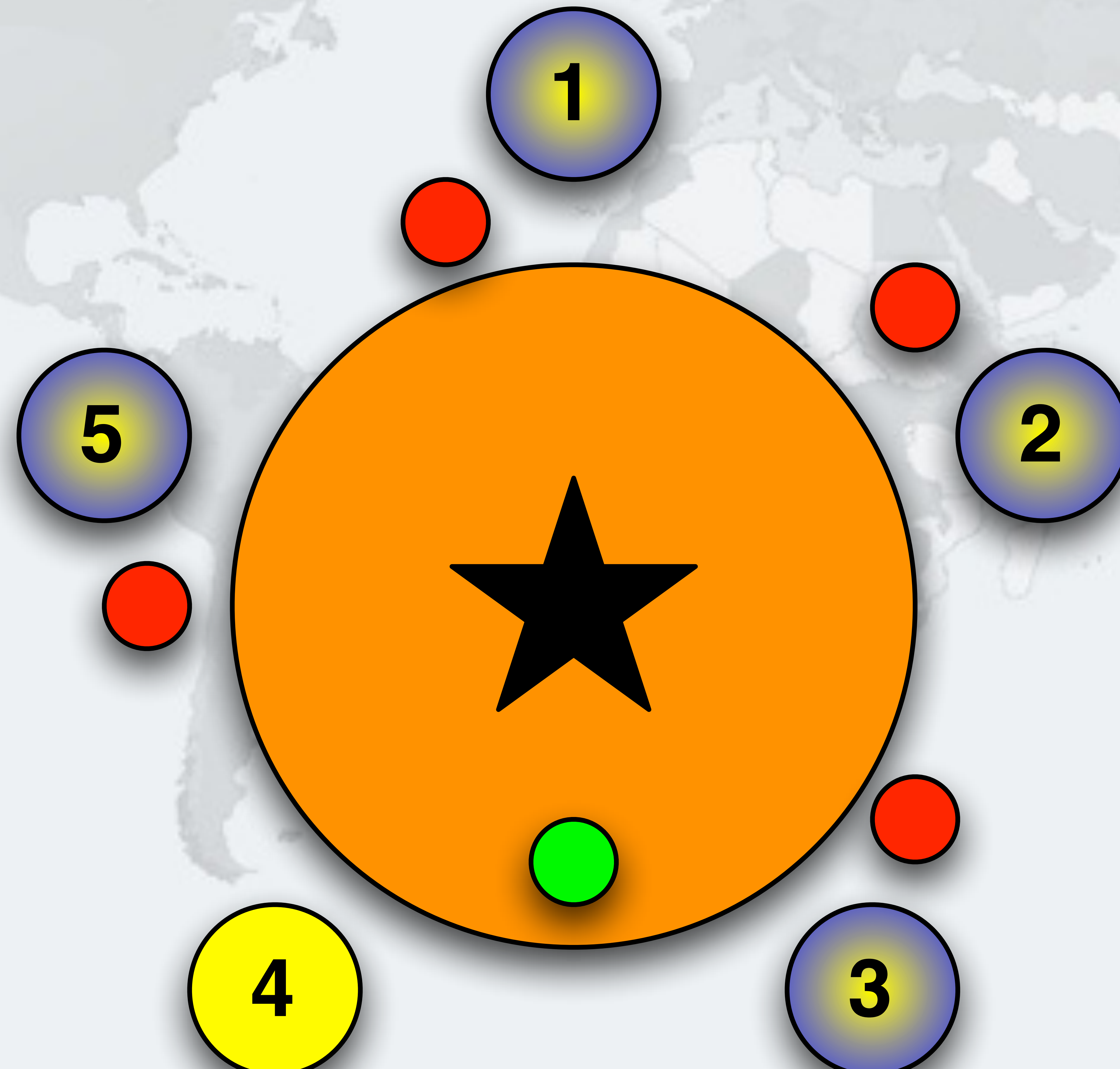
© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 2 wants to drink, takes right cup

© 2012-2017 Heinz Kabutz, All Rights Reserved

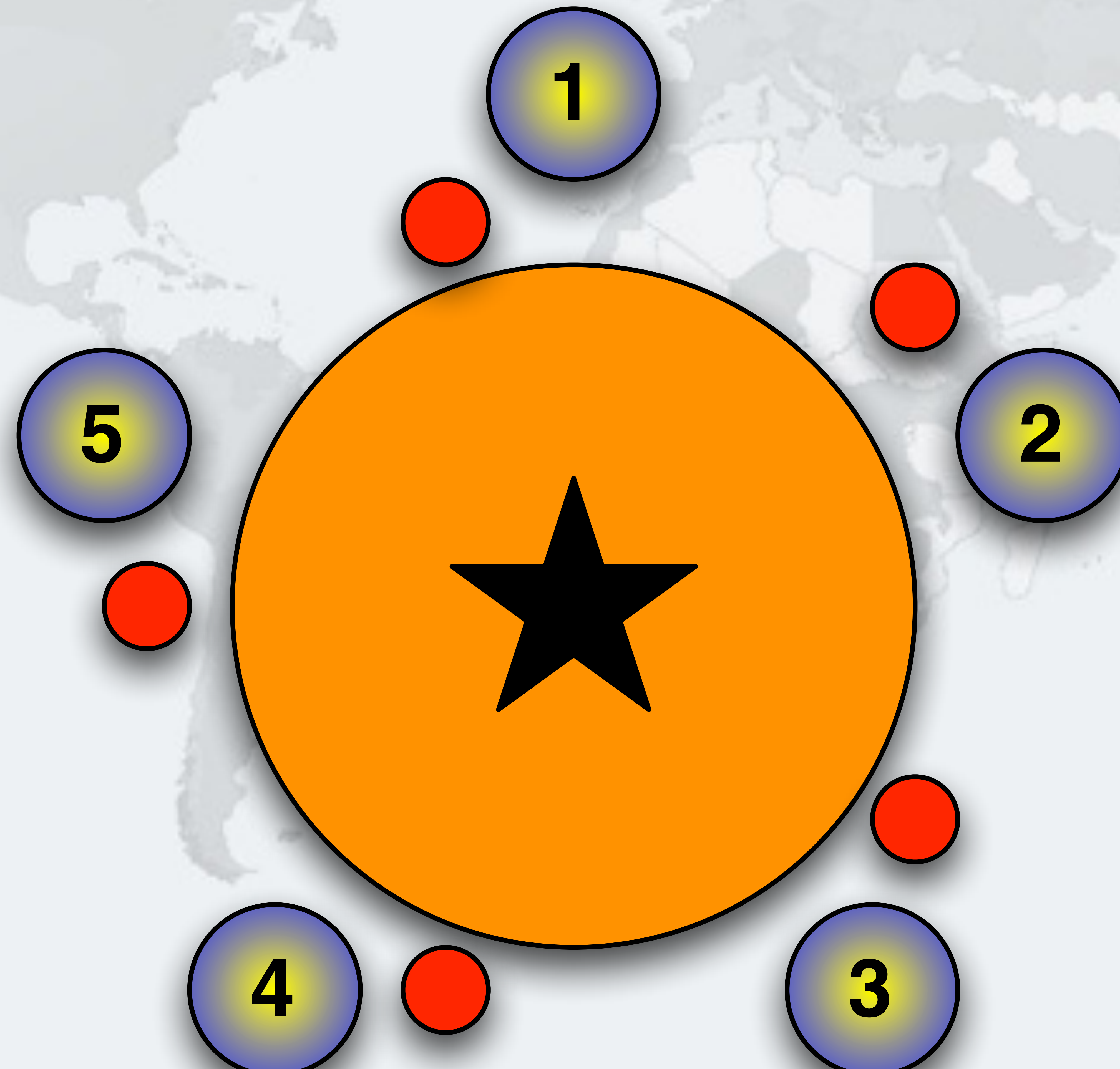
Javaspecialists.eu

Philosopher 3 wants to drink, takes right cup



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

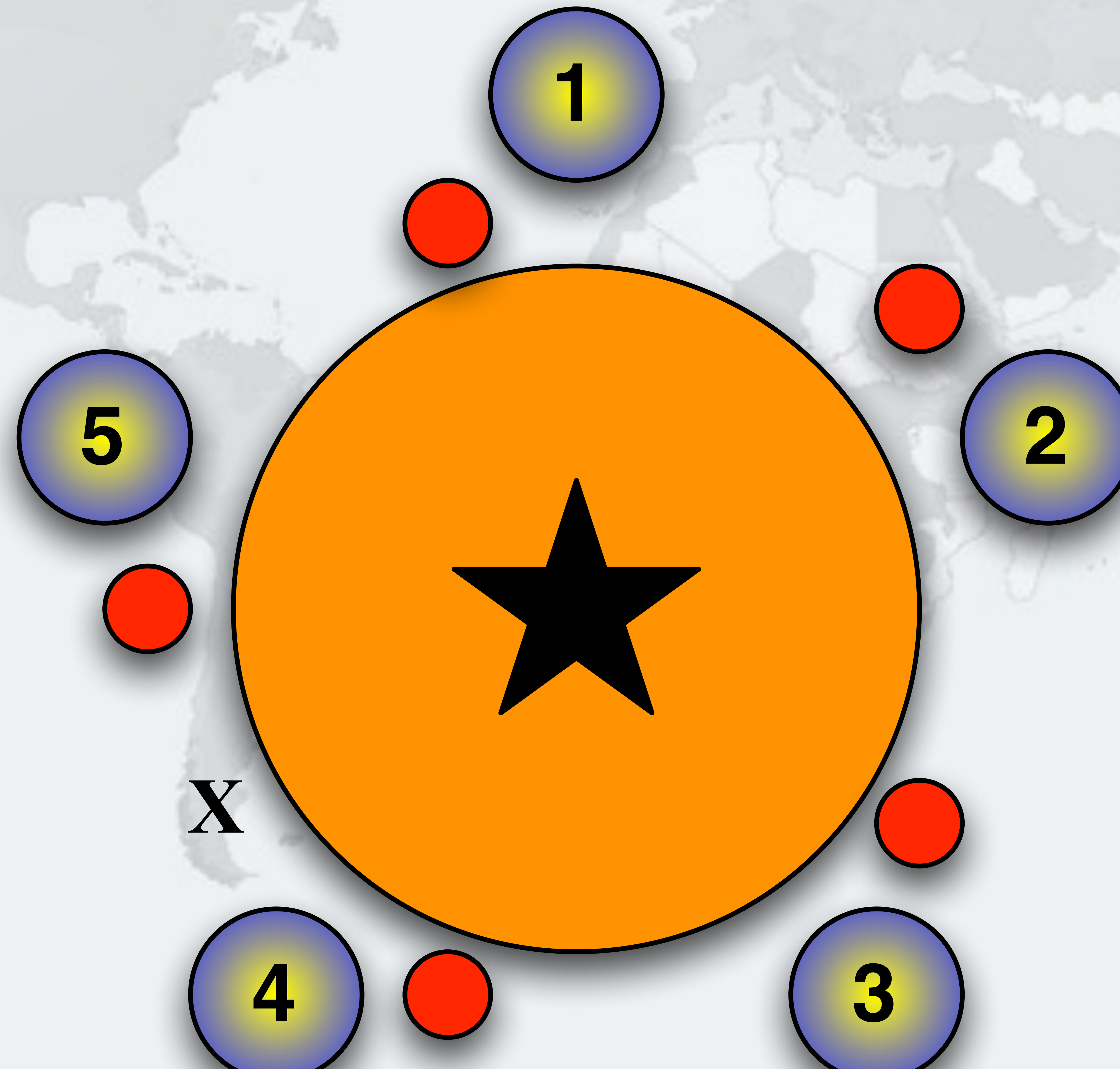
Philosopher 4 wants to drink, takes right cup



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Philosopher 4 tries to lock left, not available

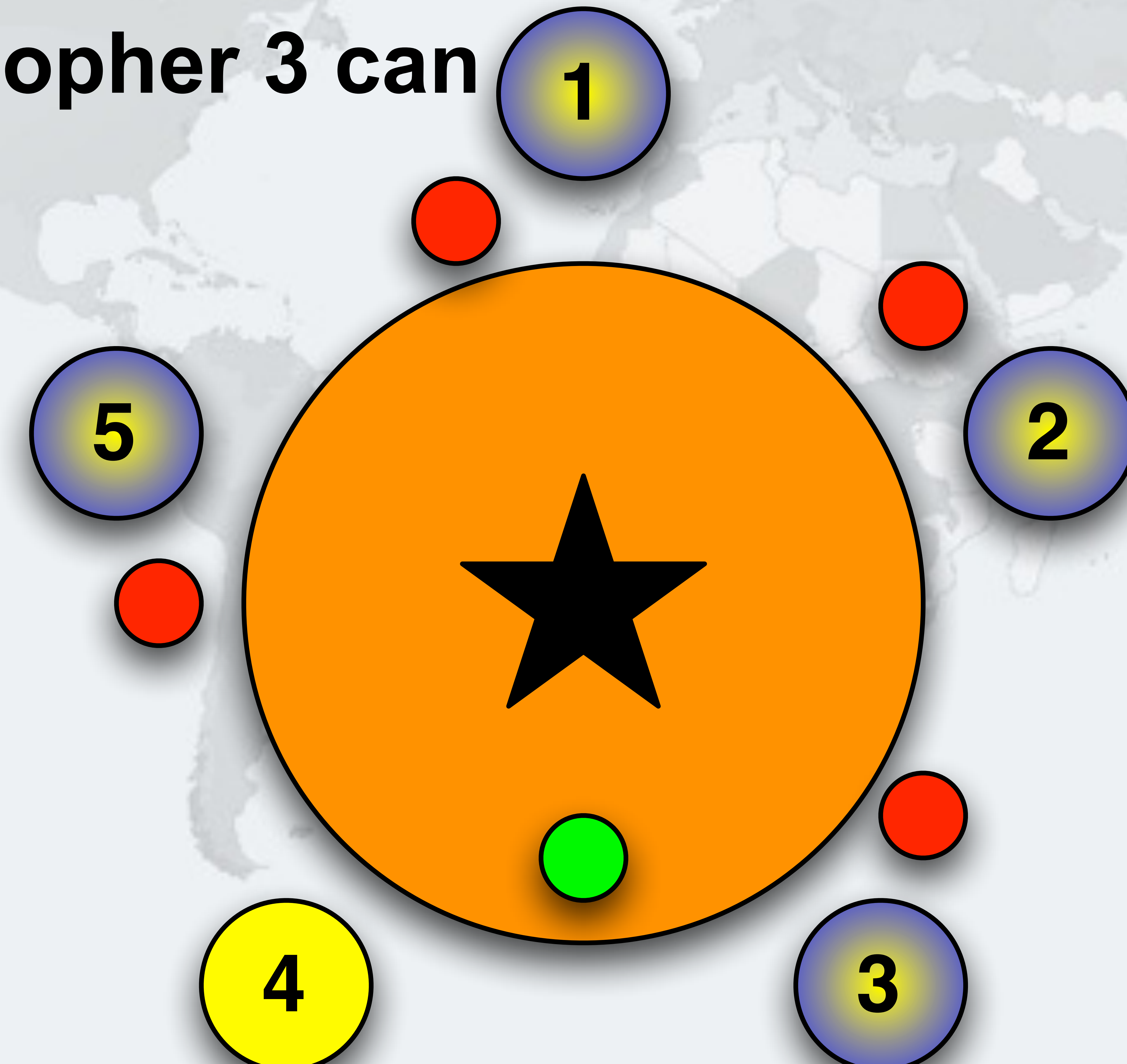


© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu

Philosopher 4 Unlocks right again

- **Now Philosopher 3 can drink**



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 2 Exercise

Deadlock resolution with tryLock()

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu
java training

Lab2: Solving Deadlock with tryLock()

- **Run Main class to trigger deadlock**
  - Capture a stack trace with `jstack -l pid`
  - Use Lock.tryLock() to avoid blocking on the inner lock
    - lock the right
    - tryLock the left
      - if success, then drink and unlock both
      - otherwise, unlock right only and retry
  - Verify that the deadlock has now disappeared

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 3: Resource Deadlock

Avoiding Liveness Hazards

© 2012-2017 Heinz Kabutz, All Rights Reserved

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**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Challenge

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

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

How to solve resource deadlocks

- **If you can reproduce the resource deadlock**
  - **Take a thread dump shortly *<u>before</u>* the deadlock**
  - **Take another dump *<u>after</u>* the deadlock**
  - **Compare the two dumps**
- **If you are already deadlocked**
  - **Take a few thread dumps**
  - **Look for threads that don't move, but should**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 3 Exercise

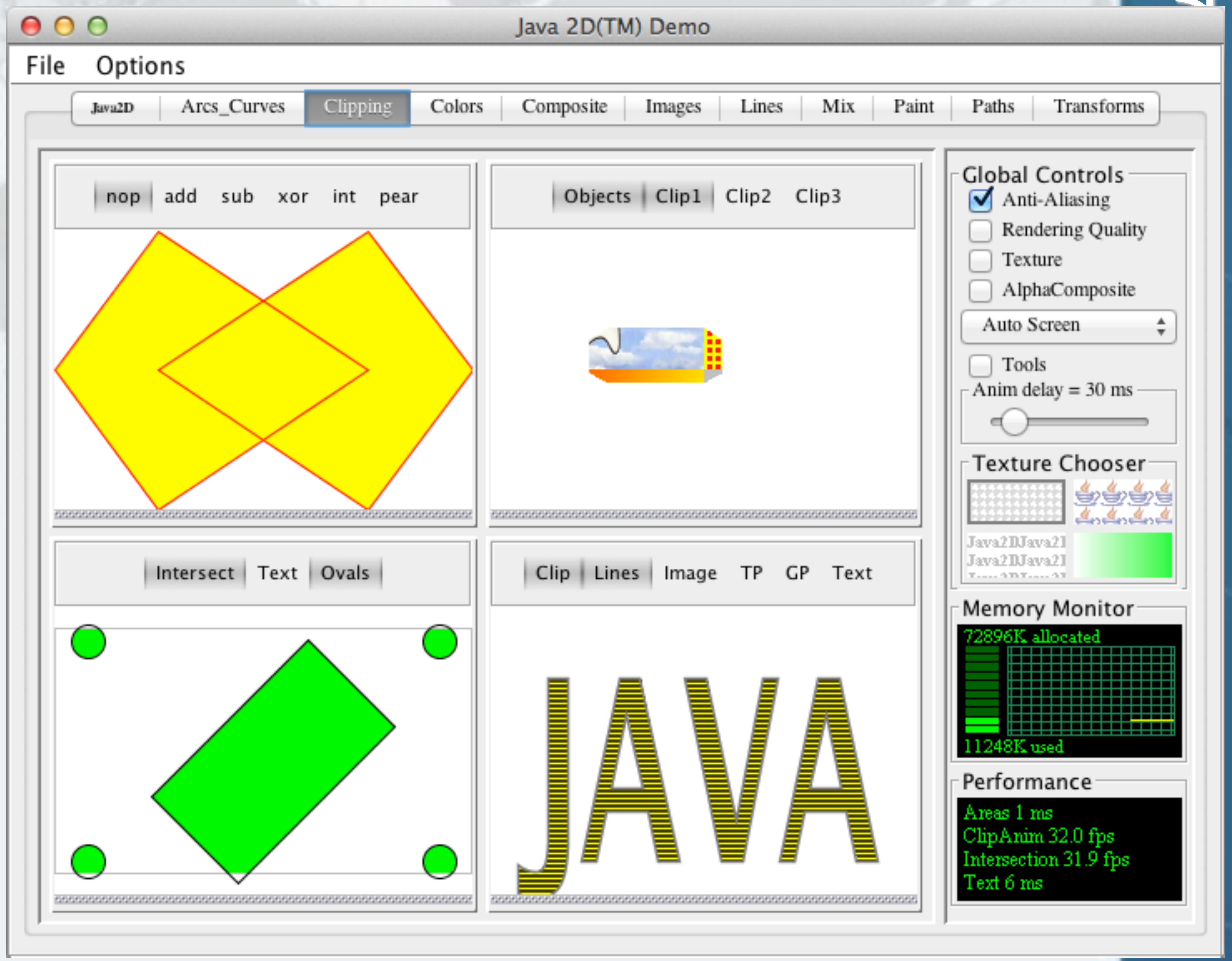
Resource Deadlock

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu
java training

Lab3 Resource Deadlock

- **Start our modified Java2Demo**
  - **Dump threads with `jstack -l`**
  - **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**



Javaspecialists.eu

© 2012-2017

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**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

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

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 4: Combining Your Skills

Avoiding Liveness Hazards

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu
java training

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 :-)**

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

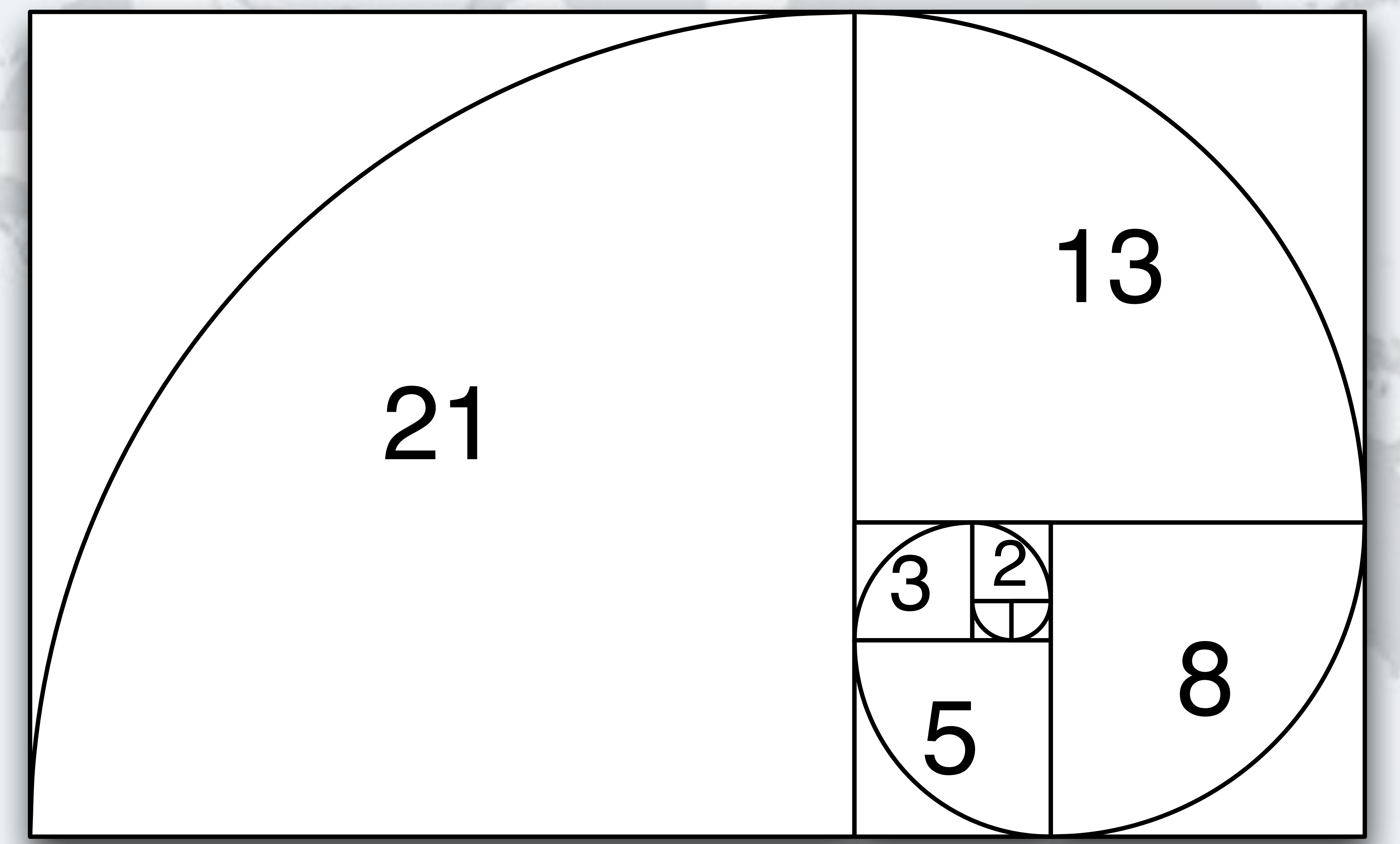
Lab 5: Speeding Up Fibonacci

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu
java training

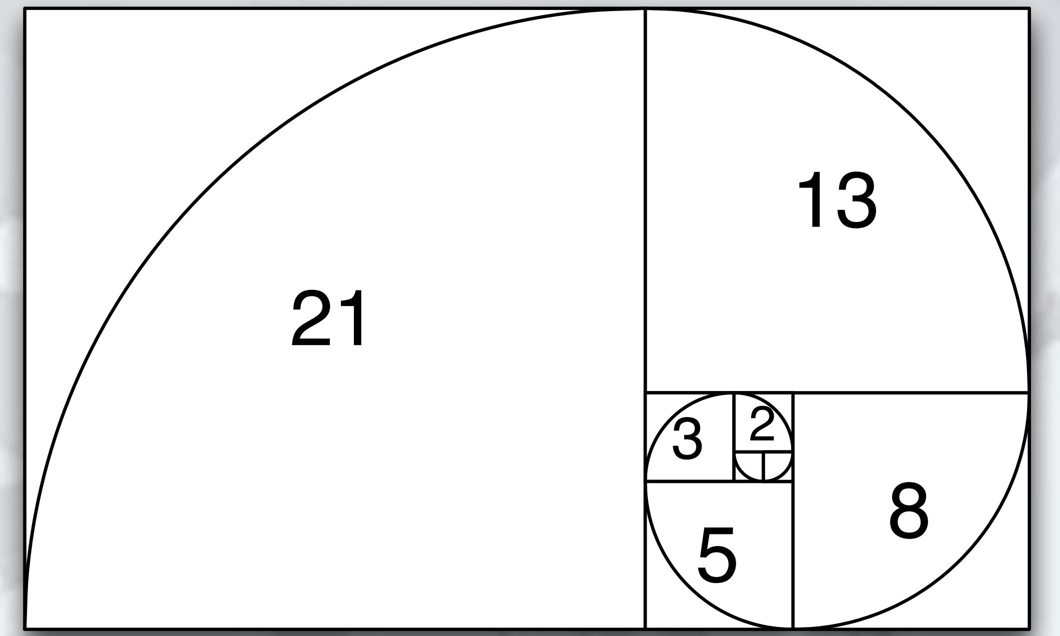
Lab 5: Speeding Up Fibonacci

- **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**



Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Exponential Algorithm



- **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)

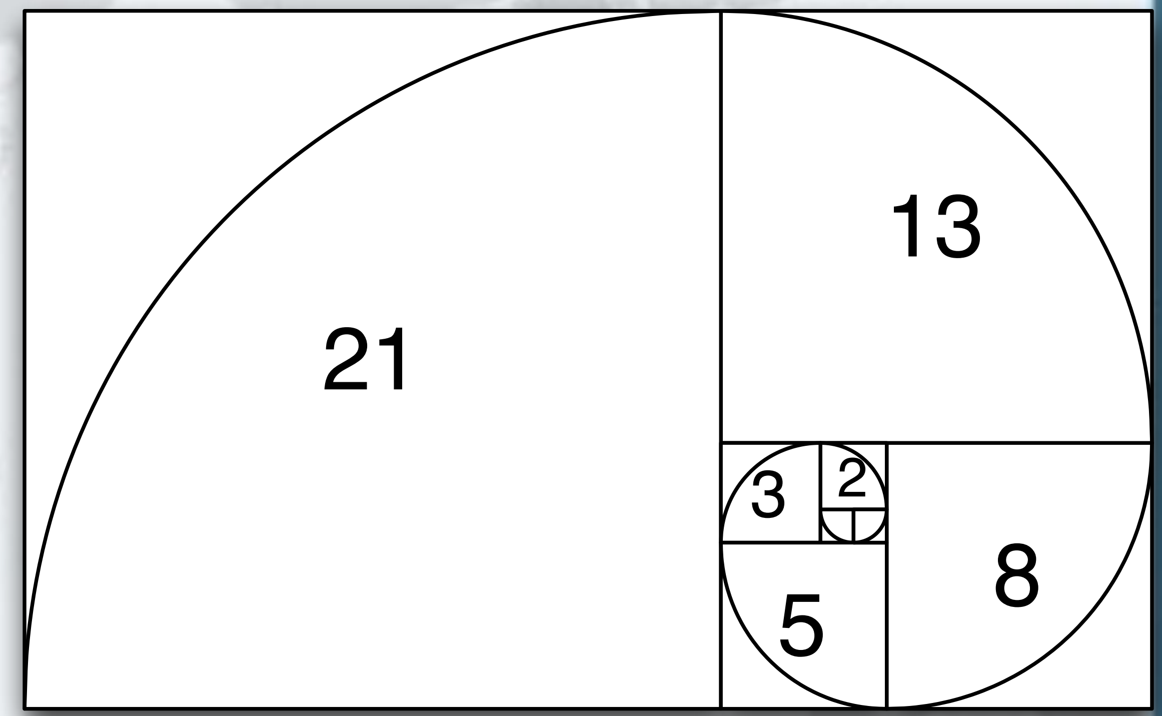
Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Linear Algorithm

- Instead of a recursive method, we could 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;
}
```



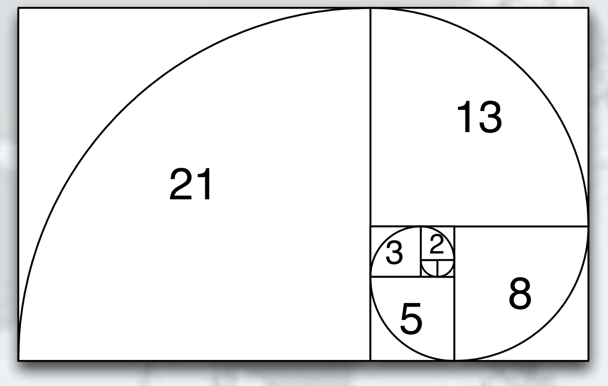
- This algorithm has linear time complexity
  - Solved f(1_000_000_000) in 1.7 seconds
    - However, the numbers overflow so the result is incorrect
    - We can use BigInteger, but its add() is also linear, so time is quadratic
    - We need a better algorithm

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

3rd Attempt Dijkstra's Sum of Squares

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

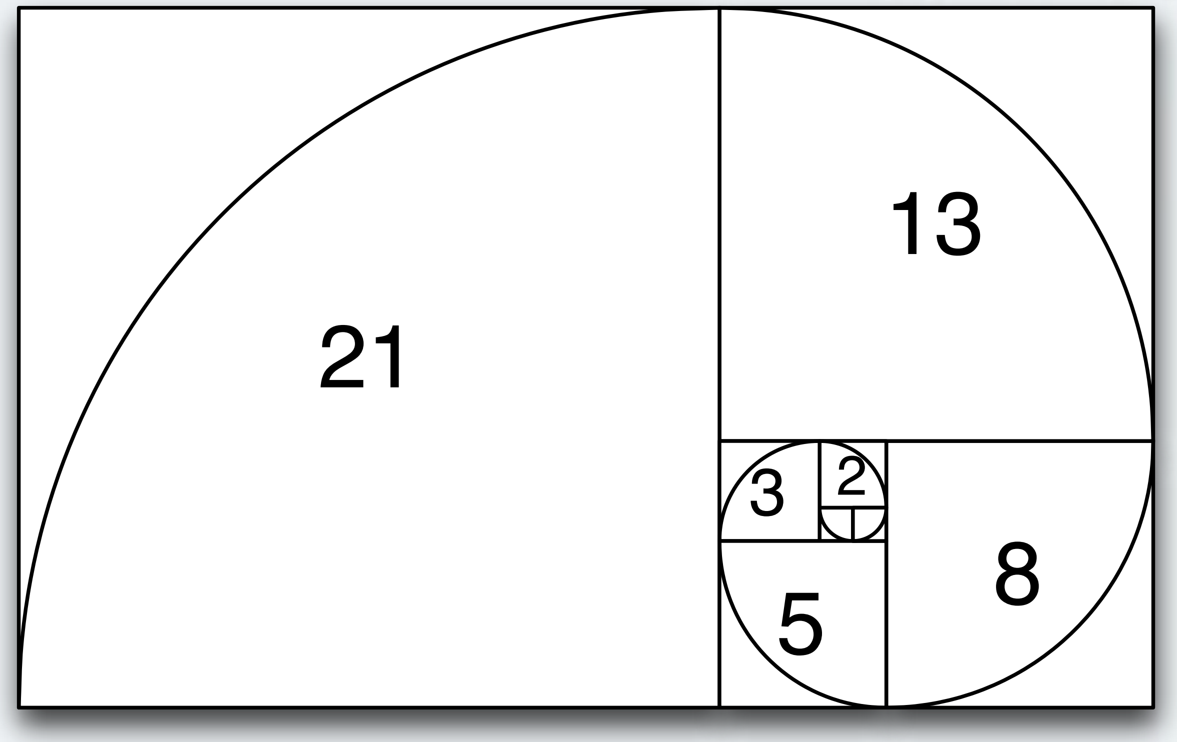


- **Logarithmic time complexity and can be parallelized**
  - Java 8 uses better BigInteger multiply() algorithms
    - Karatsuba complexity is $O(n^{1.585})$
    - 3-way Toom Cook complexity is $O(n^{1.465})$
    - Previous versions of Java had complexity $O(n^2)$
    - Single-threaded - we'll fix that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

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

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!**



Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 5.2: Parallelize your algorithm

- **We can parallelize by using common Fork/Join Pool**
  - **Next we fork() the 1st task, do the 2nd and then join 1st**

```
ForkJoinTask<BigInteger> f0_task = new RecursiveTask<BigInteger>() {
    protected BigInteger compute() {
        return f(half - 1);
    }
}.fork();
BigInteger f1 = f(half);
BigInteger f0 = f0_task.join();
```

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

Lab 5.3: Parallelize BigInteger

- **Using principles from lab 5.2, parallelize methods in eu.javaspecialists.performance.math.BigInteger**
  - **multiplyToomCook3()**
  - **squareToomCook3()**
- **These would probably not reach the threshold, so we won't parallelize them:**
  - **multiplyKaratsuba()**
  - **squareKaratsuba()**

Javaspecialists.eu
© 2012-2017 Heinz Kabutz, All Rights Reserved

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**

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

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

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Wrap Up

Avoiding Liveness Hazards

© 2012-2017 Heinz Kabutz, All Rights Reserved

Javaspecialists.eu
Java Training

Conclusion on Live(li)ness

- Concurrency is difficult, but good tools and techniques solve problems
- These are just a few that we use
- For more, sign up to
  - The Java Specialists' Newsletter
    - tinyurl.com/jfokus2017
    - Sign up before Feb 10th 2017 and enter a lucky draw

Javaspecialists.eu

© 2012-2017 Heinz Kabutz, All Rights Reserved

Finding and Solving Java Deadlocks

Dr Heinz M. Kabutz
heinz@kabutz.net
tinyurl.com/jfokus2017

© 2012-2017 Heinz Kabutz, All Rights Reserved