**Dining Philosophers in JPF**

-

**Team Guide:**

Dr. Richard Carver

**Members:**

Anirudh Arora (G00894977)

Saurabh Patil (G00907573)

Shreyas Raut (G00886535)

Somesh Chadda (G00963429)

**University:**  
George Mason University  
CS - 706

Contents

[Problem Definition](#h.tcxqyfct0y89)

[Introduction](#h.miorcztca8ky)

[Program Design](#h.tw3o9jvvcst7)

[Solution 1](#h.l4m5dwr343r4)

[Solution 2](#h.tswkfg6mmcc8)

[Java Path Finder Design](#h.6iyk30tf6u07)

[JPF Model Checking](#h.11xaox66pe5r)

[Testing](#h.jm5zesnek1ww)

[JPF Features and Classification](#h.e01cqbbq2kxh)

[Analysis of Test result:](#h.tmfl8g1kp2in)

[Programs](#h.9ijzrigir54k)

[Solution 1](#h.opqoeuqybmul)

[Solution 2](#h.s64lrm3k7dk7)

[References](#h.364jy7jcrt3c)

### Problem Definition

Install and use the NASA Java Pathfinder. Implement Dining Philosopher’s using Odd philosopher solution (Left Handed philosopher) and using steward (n-1 philosopher) solution.

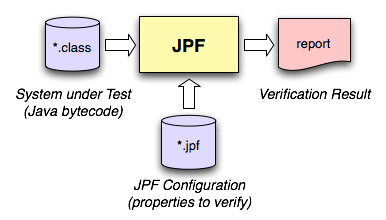
### Introduction

Java Pathfinder (JPF) is a system to verify executable Java bytecode programs. JPF was developed at the NASA Ames Research Center and open sourced in 2005.

Today, JPF is a swiss army knife for all sort of runtime based verification purposes.

For a concrete example, consider the Dining Philosopher's problem. When we run JPF on the Dining Philosopher's code, we get a textual trace which finally shows the deadlocked state. This textual trace can be presented in a more intuitive manner. It is desirable to see not only the individual states but also the history of execution that led to the deadlock. At the user’s direction, the tree of execution choices made by JPF’s scheduler should also be viewable in an intuitive manner.

The answer used to be simple: "JPF is an explicit state software model checker for Java™ bytecode".



JPF is a runtime that is configured to a combination of different components. JPF was designed so that it is easy to extend. We for now, restrict ourselves here to what the jpf core is, but keep in mind it is only primus inter pares among JPF components.

The JPF core is a Virtual Machine (VM) for Java™ bytecode, which means it is a program which you give Java programs to execute. It is used to find defects in these programs, so you also need to give it the properties to check for as input. JPF gets back to you with a report that says if the properties hold and/or which verification artifacts have been created by JPF for further analysis (like test cases).

### 

### 

### Program Design

For a concrete example, consider the Dining Philosopher's problem. In this there are n philosophers who spend their time eating and thinking. They sit at a table with n seats. A bowl of rice sits in the center of the table. There is one fork between each pair of philosophers. When a philosopher is hungry, she picks up the two forks that are next to her one at a time. When she gets her forks, she holds them until she is finished eating. Then she puts down her forks one at a time and goes back to thinking.

Solutions to the dining philosophers problem are required to be free from deadlock and starvation. A classical deadlock situation is created in solutions that use a hold-and-wait policy. This policy allows a philosopher to hold one fork, which she is not willing to relinquish, while waiting on another fork.

#### **Solution 1**

In this solution all n philosophers pick up their forks in the same order. Except that one philosopher is designated as the “odd” philosopher. The odd philosopher picks up her right

fork first (instead of her left fork). This solution is deadlock-free, and

it is starvation free. But this solution does not satisfy maximal-parallelism.

**Pseudo code:**

if (args.length > 0){

nPhilosophers = Integer.parseInt(args[0]);

}

//Verify.beginAtomic();

Fork[] forks = new Fork[nPhilosophers];

for (int i = 0; i < nPhilosophers; i++) {

forks[i] = new Fork();

}

for (int i = 0; i < nPhilosophers; i++) {

Philosopher p = new Philosopher(forks[i], forks[(i + 1) % nPhilosophers]);

if(i == nPhilosophers -1 )

p = new Philosopher(forks[(i + 1) % nPhilosophers], forks[i]);

p.start();

}

#### **Solution 2**

In this solution there are (n − 1) philosophers are allowed to sit at a table that has n seats. Whenever a philosopher seats a counter is updated to keep the count on the number of philosopher who are sitting so that we are aware of the number of seats available. This counter is updated before the philosopher can pick up forks. However this solution does not satisfy maximum parallelism. Since it is possible that two neighboring philosophers hold a single fork but are unwilling to let each other eat.

**Pseudo code:**

static protected class Steward {

int sitting;

int max;

public Steward(int ms) { sitting = 0; max = ms; }

synchronized void sitDown() {

while(sitting == max) {

try { wait(); }

catch(InterruptedException e) {}

}

++sitting;

}

synchronized void getUp() {

--sitting;

notify();

}

}

### 

### 

### 

### Java Path Finder Design

### 

It is implemented in Java itself, so it doesn’t run as fast as normal Java. It is a VM running on top of a VM. While execution semantics of JPF are a little hardwired - the VM instruction set is represented by a set of classes that can be replaced.

JPF can identify points in your program from where execution could proceed differently, then systematically explore all of them. This means JPF (theoretically) executes all paths through your program, not just one like a normal VM. Typical choices are different scheduling sequences or random values, but again JPF allows you to introduce your own type of choices like user input or state machine events.

The number of paths can grow out of handle. This is called the *state explosion problem*. The first line of defense employed by JPF is ***state matching***: each time JPF reaches a choice point, it checks if it has already seen a similar program state, in which case it can safely abandon this path, ***backtrack*** to a previous choice point that has still unexplored choices, and proceed from there. JPF can restore program states, which is like telling a debugger "go back 100 instructions".

These features are normally used to find defects in the program you want to verify. The kind of defects depends on how you configure JPF. The core checks for defects that can be identified by the VM without you having to specify any property: deadlocks and unhandled exceptions (which also include Java assert expressions).These are called *non-functional* properties, and no application should violate them. In JPF doesn't stop you can define your own properties, which is mostly done with so called ***listeners***, little "plugins" that let you closely monitor all actions taken by JPF, like executing single instructions, creating objects, reaching a new program state and many more.One additional feature that comes in handy in case JPF finds a defect is the availability of the complete execution history that leads to the bug, down to every executed bytecode instruction if you need it. We call this a program ***trace***, and it is invaluable to find out what really caused the defect

In short JPF executes normal Java bytecode programs and can store, match and restore program states. Its primary application has been Model checking of concurrent programs, to find defects such as data races and deadlocks.

#### **JPF Model Checking**

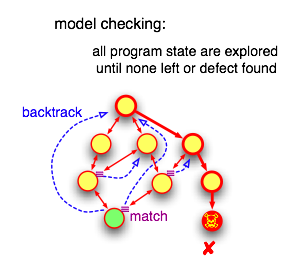
JPF does not just test our program for defects, it does much more, at least when used as a model checker.

Software testing is an empirical set of techniques where you execute your program with a number of inputs in order to find out if it behaves correctly.

Testing techniques differ on how we choose the input and on how much knowledge about the system under test (SUT) and its execution environment we assume (black/grey/white box), which especially affects how we can define and check correct behavior. This involves a lot of educated guesses. We usually compensate this by performing "enough" tests - which would be the next guess.

### 

*​Model Checking* as a *​Formal Method* does not depend on guesses. If there is a violation of a given specification, model checking will find it. Model checking is supposed to be a rigorous method that exhaustively explores all possible SUT behaviors.Model checking doesn't stop until it has checked all data combinations or has found an error.



We will illustrate the differences using an example:

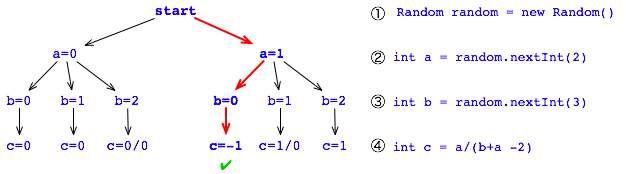
|  |
| --- |
| **import** java.util.Random**;**  **public** **class** **Rand** **{**  **public** **static** **void** **main** **(**String**[]** args**)** **{**  Random random **=** **new** Random**(**42**);** *// (1)*    **int** a **=** random**.**nextInt**(**2**);** *// (2)*  System**.**out**.**println**(**"a=" **+** a**);**    *//... lots of code here*    **int** b **=** random**.**nextInt**(**3**);** *// (3)*  System**.**out**.**println**(**" b=" **+** b**);**    **int** c **=** a**/(**b**+**a **-**2**);** *// (4)*  System**.**out**.**println**(**" c=" **+** c**);**   **}** **}** |

#### **Testing**

Executing this program with a normal Java VM yields something like the following output. If we don't provide an explicit seed when creating the Random object in (1), the result is going to differ between runs, but every run will choose just a single 'a' and 'b' value (i.e. print just a single "a=.." and "b=.." line.

|  |
| --- |
| **>** java Rand a**=**1  b**=**0  c**=-**1 **>** |

Here’s a graphical representation of all the ways our program could be executed, and how it actually was executed in our test run. The nodes of the graph represent "program states", and the edges "transitions" the execution could take from a certain state



Model Checking ¶

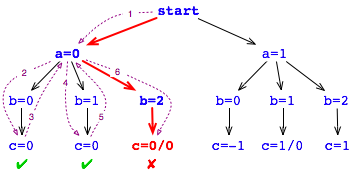
Enter JPF which does not just consider single values for 'a' and 'b', but looks at all possible choices

#### 

|  |
| --- |
| > bin/jpf +cg.enumerate\_random=true Rand JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center ====================================================== system under test application: /Users/cserver/tmp/Rand.java  ====================================================== search started: 5/23/07 11:49 PM a=0  b=0  c=0  b=1  c=0  b=2  ====================================================== error #1 gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty java.lang.ArithmeticException: division by zero  at Rand.main(Rand.java:15) .... > |

By specifying "+vm.enumerate\_random=true" we told JPF to consider all possible values for expressions (2) and (3). JPF starts with "a=0", then picks "b=0", which yields a valid "c=0" value. But instead of terminating like a normal VM, JPF recognized that there are more choices left, so it "backtracks" to (3) and picks "b=1". Again, no problem here with computing "c=0". Back to (3), JPF now tries "b=2", which of course spells disaster for our little program when executing (4), as can be seen by the following error report.JPF also keeps the complete "trace" (execution path) how it got to this error (denoted by the red arrows), which means we don't have to debug the program to find out.

Here is a graphical representation of this process.



#### 

#### 

#### **JPF Features and Classification**

JPF is not just a model checker: it is a JVM that can be used for model checking. Some of the basic model checking traits are:

**Explicit State** model checking is JPF's standard mode of operation. It means JPF keeps track of concrete values of local variables, stackframes, heap objects and thread states. Other than the intentionally different scheduling behavior, JPF should produce the same results like a normal JVM. Of course it is slower (it is a JVM running on top of a JVM, doing a lot of additional work).

**Symbolic Execution** means that JPF can perform program execution with symbolic values obtained from how a certain variable was used along the current path of execution (e.g. “x > 0 && x < 43”). Moreover, JPF can even mix concrete and symbolic execution, or switch between them. **State**

**Matching** is a key mechanism to avoid unnecessary work. The execution state of a program mainly consists of heap and thread-stack snapshots. While JPF executes, it checks every new state if it already has seen an equal one, in which case there is no use to continue along the current execution path, and JPF can backtrack to the nearest unexplored non-deterministic choice. What variables contribute to state matching, and how state matching is performed can be controlled by the user.

**Backtracking** means that JPF can restore previous execution states, to see if there are unexplored choices left. For instance, if JPF reaches a program end state, it can walk backwards to find different possible scheduling sequences that have not been executed yet. While this theoretically can be achieved by re-executing the program from the beginning, backtracking is a much more efficient mechanism if state storage is optimized.

**Partial Order Reduction** is what JPF employs to minimize context switches between threads that do not result in interesting new program states. This is done on-the-fly, i.e. without prior analysis or annotation of the program, by examining which instructions can have inter-thread effects. While this is fast, it cannot handle the “diamond case”, since it cannot look ahead of the current execution.

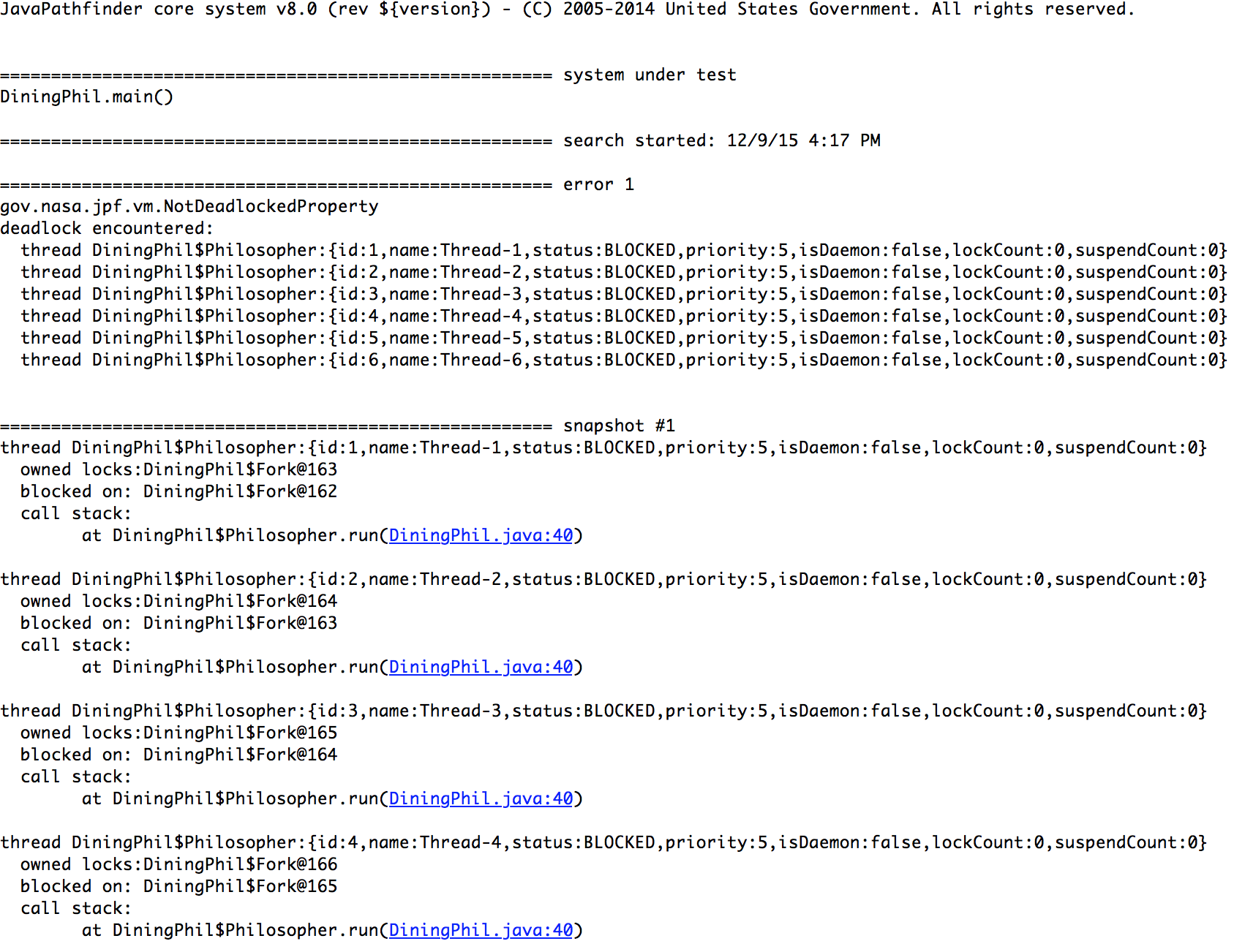
### 

### 

### Analysis of Test result:

**Default Philosopher solution:**

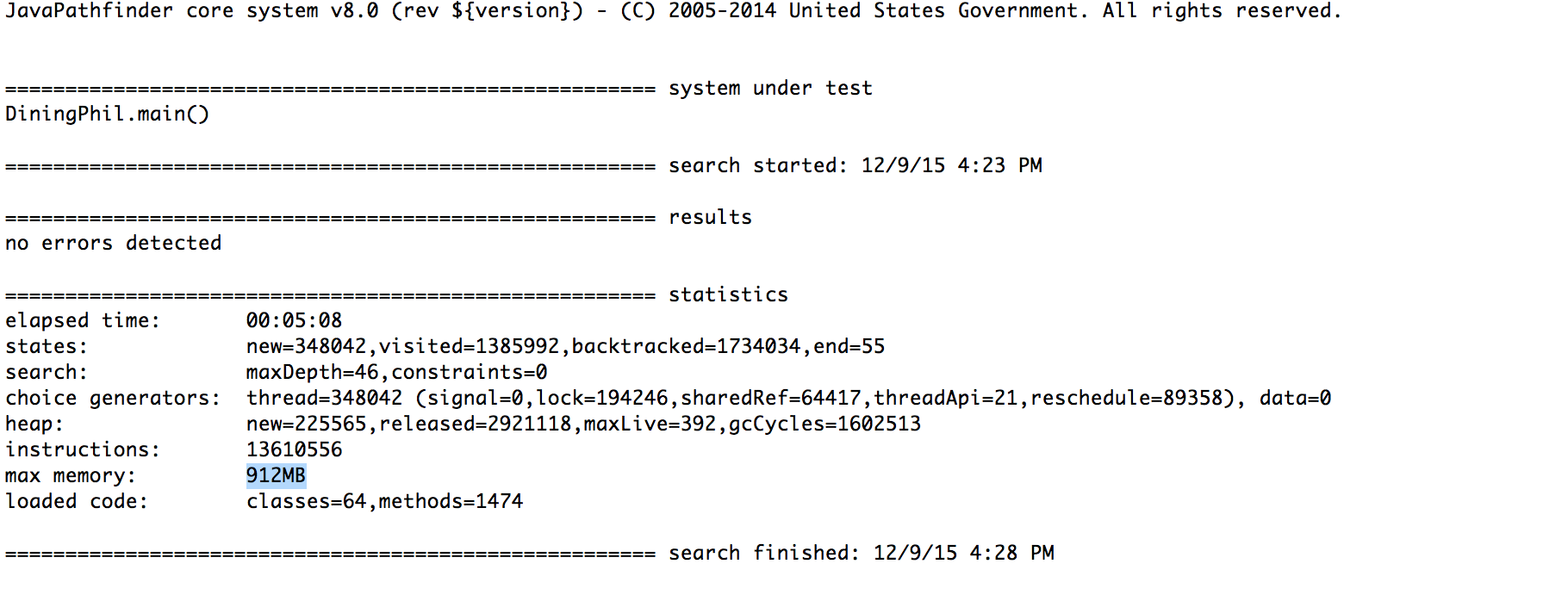
Consider the base case where every philosopher when hungry pick up the fork to his left. There will be a deadlock when all are hungry and pick up their left fork.





A deadlock is detected as shown in the output in ‘marked in blue’ shows the line of code which caused the deadlock. in this case it is waiting for the right fork.

**Left handed philosopher solution:**

Here the code is based on the 3rd solution given in the book. Hence we will not encounter a deadlock.   


**N-1 philosopher Problem:**

This is based on the N-1 solution(solution 2 in the textbook). as we know there will be no dead lock. hence the JPF does not encounter a deadlock.

### 

Thus it could be said that the JPF is a handy tool in analysing a Concurrent Program.

### 

### 

### 

### Programs

#### **Solution 1**

public class DiningPhil1 {

// a fork -- just provides an object

static protected class Fork {

public int id;

public Fork(int i) { id = i; }

}

// a philosopher

static class Philosopher extends Thread {

Fork left;

Fork right;

public Philosopher(int i, Fork left, Fork right) {

id = i;

this.left = left;

this.right = right;

//start();

}

public void run() {

// think!

synchronized (left) {

synchronized (right) {

// eat!

}

}

}

}

static int nPhilosophers = 6;

// the main method

public static void main(String[] args) {

if (args.length > 0){

nPhilosophers = Integer.parseInt(args[0]);

}

//Verify.beginAtomic();

Fork[] forks = new Fork[nPhilosophers];

for (int i = 0; i < nPhilosophers; i++) {

forks[i] = new Fork();

}

for (int i = 0; i < nPhilosophers; i++) {

Philosopher p = new Philosopher(i,forks[i], forks[(i + 1) % nPhilosophers]);

if(i == nPhilosophers -1 )

p = new Philosopher(forks[(i + 1) % nPhilosophers], forks[i]); // n-1 philosopher is odd

p.start();

}

//Verify.endAtomic();

}

}

#### **Solution 2**

public class DiningPhil2 {

static protected class Fork {

public int id;

public Fork(int i) { id = i; }

}

static protected class Steward {

int sitting; /\*\* the number or philosophers sitting at the table currently \*/

int max; /\*\* the max number of philosophers allowed to sit at any one time \*/

public Steward(int ms) { sitting = 0; max = ms; }

/\*\* this procedure is called by philosophers. It returns when the

\* calling philosopher is allowed by the steward to sit at the

\* table \*/

synchronized void sitDown() {

while(sitting == max) {

try { wait(); }

catch(InterruptedException e) {}

}

++sitting;

}

/\*\* this procedure informs the steward that the calling philosopher

\* is done eating and is leaving the table \*/

synchronized void getUp() {

--sitting;

notify();

}

}

static protected class Phil extends Thread

{

public Steward steward;

public Fork leftFork,rightFork;

public int id;

public Phil(int i,Steward b,Fork lf,Fork rf) {

steward = b;

id = i;

leftFork = lf;

rightFork = rf;

}

public void run()

{

while(true) {

steward.sitDown();

synchronized(leftFork) {

synchronized(rightFork) {

/\*\* eat \*/

}

}

steward.getUp();

}

}

}

static int nPhilosophers = 10;

public static void main(String [] args)

{

if(args.length > 0) {

Steward steward = new Steward(nPhilosophers);

Fork [] forks = new Fork [nPhilosophers];

Phil [] phils = new Phil [nPhilosophers];

for(int i = 0;i < nPhilosophers;++i) forks[i] = new Fork(i);

for(int i = 0;i < nPhilosophers;++i) phils[i] = new Phil(i,steward,forks[i],forks[(i + 1) % nPhilosophers]);

for(int i = 0;i < nPhilosophers;++i) phils[i].start();

} else System.out.println("usage: jpf DP <philosopher num>");

}

}

#### 

### References

[1] <http://javapathfinder.sourceforge.net/>

[2]<http://babelfish.arc.nasa.gov/>

[3] Modern Multithreading : Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs 1st Edition by [Richard H. Carver](http://www.amazon.com/Richard-H.-Carver/e/B001ITYIR8/ref=dp_byline_cont_book_1) (Author), [Kuo-Chung Tai](http://www.amazon.com/s/ref=dp_byline_sr_book_2?ie=UTF8&text=Kuo-Chung+Tai&search-alias=books&field-author=Kuo-Chung+Tai&sort=relevancerank) (Author)