MODULE 9 - SHEET 1

```
// This is programm is due to Dr P. Robinson

public class BufferExample
 { public static void main(String[] args)
    { Buffer buf = new Buffer();
      Consumer cons = new Consumer(buf);
      cons.start();
      try
       { for (int i = 1; i<=7; i++)
            buf.put(i);
       }
      catch (InterruptedException e)
       { System.out.printf("Producer Interrupted!");
       }
      cons.stop();
    }
 }

class Buffer
 { private int value;
   private boolean full = false;

   public synchronized void put(int i) throws InterruptedException
    { while (this.full)
         this.wait();
      this.value = i;
      this.full = true;
      this.notifyAll();
    }

   public synchronized int get() throws InterruptedException
    { while (!this.full)
         this.wait();
      this.full = false;
      this.notifyAll();
      return this.value;
    }
 }

class Consumer extends Thread
 { private Buffer buffer;

   public Consumer(Buffer b)
    { this.buffer = b;
    }

   public void run()
    { while (true)
       { try
  { System.out.printf("Found %d%n", this.buffer.get());
         }
        catch (InterruptedException e)
```

```
 { System.out.printf("Interrupted while consuming!");
 }
      }
   }
}
```

MODULE 9 - SHEET 2

Five philosophers (numbered 0, 1, 2, 3 and 4) sit round a circular table in the middle of which is a bowl of food. Between each adjacent pair of philosophers there is a single fork. The forks are also numbered 0, 1, 2, 3 and 4.

The philosophers spend most of their time thinking but, every now and then, a given philosopher feels hungry and fancies eating some of the food. In order to eat, a philosopher needs to use BOTH the fork on the left AND the fork on the right. Of course, either or both forks may be in use and the philosopher has to wait until the forks are free.

To avoid deadlock, the philosphers follow a fixed strategy. Philospher n (where n is 0, 1, 2 or 3) always picks up fork n+1 before fork n and, after eating, always puts down fork n before fork n+1. Philosopher 4 operates differently and always picks up fork 4 before fork 0 and, after eating, always puts down fork 0 before fork 4.

Write a program which will simulate the behaviour of the philosophers. Each philosopher should run as a separate thread and each fork should be a separate object incorporating synchronised methods get and put (which a philosopher used for acquiring and releasing the fork).

Typical output might be as follows...

```
  Philosopher 0   Philosopher 1   Philosopher 2   Philosopher 3   Philosopher 4

                                  wants some food
                                  acquires fork 3
                                  acquires fork 2
                                  starts his meal
                                  has become full
                                  releases fork 2
                                  releases fork 3
                                  resumes thought
                  wants some food
                  acquires fork 2
                  acquires fork 1
                  starts his meal
                  is still eating
                                                                  wants some food
                                                                  acquires fork 4
                                                                  acquires fork 0
                                                                  starts his meal
                                                                  is still eating
 wants some food
 awaiting fork 1
                                                  wants some food
                                                  awaiting fork 4
                  is still eating
                                                                  has become full
                                                                  releases fork 0
```

```
                                                releases fork 4
                                                resumes thought
                                        acquires fork 4
                                        acquires fork 3
                                        starts his meal
                                        is still eating
                has become full
                releases fork 1
                releases fork 2
                resumes thought
acquires fork 1
acquires fork 0
starts his meal
has become full
releases fork 0
releases fork 1
resumes thought
                                        has become full
                                        releases fork 3
                                        releases fork 4
                                        resumes thought
                        wants some food
                        acquires fork 3
                        acquires fork 2
                        starts his meal
                        has become full
                        releases fork 2
                        releases fork 3
                        resumes thought
                                                wants some food
                                                acquires fork 4
                                                acquires fork 0
                                                starts his meal
```

```
public class EvalIntro
 { public static void main(String[] args)
    { Node tree1 = new NumNode(6);
      Node tree2 = new OpNode(Operator.PLUS, new NumNode(2), new NumNode(5));
      Node tree3 = new OpNode(Operator.MULT, tree1, tree2);

      System.out.printf("tree1: %d%n", tree1.eval());
      System.out.printf("tree2: %d%n", tree2.eval());
      System.out.printf("tree3: %d%n", tree3.eval());
    }
 }

class Operator
 { public static final int PLUS  = 0;
   public static final int MINUS = 1;
   public static final int MULT  = 2;
   public static final int DIV   = 3;
 }

abstract class Node
 { public abstract int eval();
 }

class NumNode extends Node
 { private int value;

   public NumNode(int n)
    { this.value = n;
    }

   public int eval()
    { return this.value;
    }
 }

class OpNode extends Node
 { private int op;
   private Node first, second;

   public OpNode(int p, Node f, Node s)
    { this.op = p;
      this.first = f;
      this.second = s;
    }

   public int eval()
    { switch(this.op)
       { case Operator.PLUS:  return this.first.eval() + this.second.eval();
         case Operator.MINUS: return this.first.eval() - this.second.eval();
         case Operator.MULT:  return this.first.eval() * this.second.eval();
         case Operator.DIV:   return this.first.eval() / this.second.eval();
         default:             return 0x80000000;
       }
```

```
    }
}
```

```
public class CroquetB
 { private static int count=0;
   private static int[] plan = new int[28];
   private static boolean[] alreadyPlayed = new boolean[193];
   private static int[] lawnCount = new int[516];
   private final static int p1 = 1, p2 = 2, p3 = 4, p4 =  8,
                            p5 =16, p6 =32, p7 =64, p8 =128;
   private final static int maxgame = 28;

   public static void main(String[] args)
    { for (int i=0; i<=192; i++)
         alreadyPlayed[i] = false;
      for (int i=0; i<=515; i++)
         lawnCount[i] = 0;

      plan[0] = p1 | p2;  alreadyPlayed[p1|p2] = true;
      plan[1] = p3 | p4;  alreadyPlayed[p3|p4] = true;
      plan[2] = p5 | p6;  alreadyPlayed[p5|p6] = true;
      plan[3] = p7 | p8;  alreadyPlayed[p7|p8] = true;

      lawnCount[(p1<<2)+0] = 1;  lawnCount[(p2<<2)+0] = 1;
      lawnCount[(p3<<2)+1] = 1;  lawnCount[(p4<<2)+1] = 1;
      lawnCount[(p5<<2)+2] = 1;  lawnCount[(p6<<2)+2] = 1;
      lawnCount[(p7<<2)+3] = 1;  lawnCount[(p8<<2)+3] = 1;

      tryIt(4);
      System.out.printf("There are %d solutions%n", count);
    }

   private static void tryIt(int game)
    { if (game == maxgame)
       { count++;
         printOut();
         System.exit(0);
         return;
       }
      int imposs = plan[game-4];
      for (int i = game & 0x1C; i<game; i++)
         imposs |= plan[i];
      int poss = ~imposs & 0xFF;
      int lawn = game%4;
      int[] player = new int[6];
      int k = -1;
      while (poss!=0)
       { int maybe = poss & -poss;
         if (lawnCount[(maybe<<2)+lawn] != 2)
   { player[++k] = maybe;
           }
         poss &= ~maybe;
       }
      for (int i=0; i<k; i++)
         for (int j=i+1; j<=k; j++)
  { int pi = player[i];
```

```
            int pj = player[j];
            int pair = pi | pj;
            if (!(alreadyPlayed[pair]))
             { plan[game] = pair;
               alreadyPlayed[pair] = true;
               lawnCount[(pi<<2)+lawn]++;
               lawnCount[(pj<<2)+lawn]++;
               tryIt(game+1);
               alreadyPlayed[pair] = false;
               lawnCount[(pi<<2)+lawn]--;
               lawnCount[(pj<<2)+lawn]--;
             }
}
  }
```

```
   private static void printOut()
    { for (int lawn=0; lawn<4; lawn++)
       { for (int game=lawn; game<maxgame; game += 4)
            System.out.printf("%s  ", match(plan[game]));
         for (int p=p1<<2; p<=p8<<2; p<<=1)
            System.out.printf("%2d", lawnCount[p+lawn]);
         System.out.printf("%n");
       }
    }

   private static String match(int pair)
    { String s = "";
      for (int p = pair & -pair; pair != 0; p = pair & -pair)
       { switch(p)
          { case p1:  s += 1;  break;
            case p2:  s += 2;  break;
            case p3:  s += 3;  break;
            case p4:  s += 4;  break;
            case p5:  s += 5;  break;
            case p6:  s += 6;  break;
            case p7:  s += 7;  break;
            case p8:  s += 8;
          }
         pair &= ~p;
       }
      return s;
    }
 }

// The above program attempts to count the number of ways in which
// 8 croquet players can compete by pairs on 4 lawns over 7 rounds
// subject to extra restrictions...

// The extra restrictions are that each player must play twice on
// three lawns and once on the remaining lawn and no player may play
// two successive games on the same lawn.

// There are so many solutions that the program will take a long
// time to run to completion.  The program assumes (without loss of
// generality) that in the first round the pairs are 12, 34, 57 and
// 68.  The output gives the schedules and lawn counts.  The first
// solution happens to be:
//
//           12  35  14  27  48  57  68   2 2 1 2 2 1 2 2
//           34  17  26  18  67  38  25   2 2 2 1 1 2 2 2
//           56  28  37  45  23  16  47   1 2 2 2 2 2 2 1
//           78  46  58  36  15  24  13   2 1 2 2 2 2 1 2
```

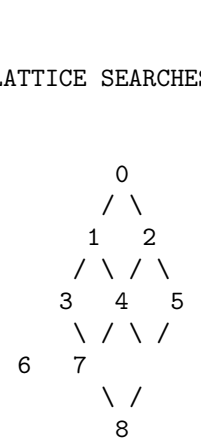
```
public class SolitaireA
 { private static int count=0;

   private static int[] triple = {007000, 000700, 000340, 000034, 000016, 000007,
                                  001104, 012200, 002210, 064000, 024400, 004420,
                                  004204, 002102, 022100, 001041, 011040, 051000};

   private static int[] ta =     {006000, 000600, 000300, 000030, 000014, 000006,
                                  001100, 012000, 002200, 060000, 024000, 004400,
                                  000204, 000102, 002100, 000041, 001040, 011000};

   private static int[] tb =     {003000, 000300, 000140, 000014, 000006, 000003,
                                  000104, 002200, 000210, 024000, 004400, 000420,
                                  004200, 002100, 022000, 001040, 011000, 050000};

   private static final int first = 037777, last = 040000;

   public static void main(String[] args)
    { long start = System.currentTimeMillis();
      tryIt(first);
      long timeSpent = System.currentTimeMillis() - start;
      System.out.printf("There are %d solutions%n", count);
      System.out.printf("Done in %d milliseconds%n", timeSpent);
    }

   private static void tryIt(int state)
    { if (state == last)
       { count ++;
         return;
       }
      for (int i=0; i<18; i++)
       { int x = state & triple[i];
         if (x == ta[i] || x == tb[i])
    tryIt(state ^ triple[i]);
       }
    }
 }

// The above program inefficiently counts the number of ways of solving
// the Triangular Solitaire Problem which is played on a board which has
// 15 holes:
//                                  1
//                                2   3
//                              4   5   6
//                            7   8   9  10
//                         11  12  13  14  15
//
// The program correctly discovers that there are 6816 solutions but, when
// timed, took 25009 milliseconds.
```

MODULE 9 - SHEET 7

TREE SEARCHES AND LATTICE SEARCHES

QUESTION

Given the network on the right, how many ways are there of getting from 0 to 8 going downwards only?

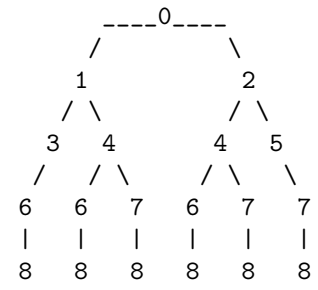


CLASSIC 8-QUEENS APPROACH

```
private static void tryIt(state)
 { if (last state)
    { count++;
      return;
    }
   note possibilities;
   for (each possibility)
      tryIt(new state)
 }
```

ANALYSIS

```
tryIt(0)  (1,2)
   tryIt(1)  (3,4)
      tryIt(3)  (6)
         tryIt(6)  (8)
            tryIt(8)    -> count++
      tryIt(4)  (6,7)
         tryIt(6)  (8)
            tryIt(8)    -> count++
         tryIt(7)  (8)
            tryIt(8)    -> count++
   tryIt(2)  (4,5)
      tryIt(4)  (6,7)
         tryIt(6)  (8)
            tryIt(8)    -> count++
         tryIt(7)  (8)
            tryIt(8)    -> count++
      tryIt(5)  (7)
         tryIt(7)  (8)
            tryIt(8)    -> count++
```



19 calls of tryIt

REVISED APPROACH

```
    private static int[] SA = {-1,-1,-1,-1,-1,-1,-1,-1,+1};

    private static int tryIt(state)
     { int score = SA[state];
       if (score<0)
        { score = 0;
          note possibilities;
          for (each possibility)
             score += tryIt(new state);
          SA[state] = score;
        }
       return(score);
     }
```

ANALYSIS

```
    tryIt(0)  -1,0,3,6  (1,2)        SA[0] = 6  ^6    13 calls of tryIt
       tryIt(1)  -1,0,1,3  (3,4)     SA[1] = 3  ^3
          tryIt(3)  -1,0,1  (6)      SA[3] = 1  ^1
             tryIt(6)  -1,0,1  (8)   SA[6] = 1  ^1
                tryIt(8)                        ^1
          tryIt(4)  -1,0,1,2  (6,7)  SA[4] = 2  ^2
             tryIt(6)                           ^1
             tryIt(7)  -1,0,1  (8)   SA[7] = 1  ^1
                tryIt(8)                        ^1
       tryIt(2)  -1,0,2,3  (4,5)     SA[2] = 3  ^3
          tryIt(4)                              ^2
          tryIt(5)  -1,0,1  (7)      SA[5] = 1  ^1
             tryIt(7)                           ^1
```