```java
//****************************************************************************
//
// File:    Automator.java
// Package: ---
// Unit:    Class Automator
//
//****************************************************************************

import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.List;

/**
 * This class automates many calls to the Chirp main method
 * by using command line arguments from an automation file.
 *
 * Each line in the file must either be commented out with
 * a '#', or be a valid command for Chirp.java.
 *
 * @author Jimi Ford (jhf3617)
 * @version 3-31-2015
 */
public class Automator {

    /**
     *
     * @param args command line arguments
     * args[0] = automation file
     */
    public static void main(String[] args) {
        if(args.length != 1) {
            usage();
        }
        try {
            List<String> lines = Files.readAllLines(Paths.get(args[0]),
                    Charset.defaultCharset());
            String[] lineArr;
            int lineCount = 0;
            boolean skip, comment;
            for (String line : lines) {
                ++lineCount;
                line = line.trim();
                lineArr = line.split(" ");
                skip = lineArr[0].equals(line);
                comment = lineArr[0].startsWith("#");
                if(skip || comment) {
                    if(comment) {
                        if(line.equals("#")) {
                            System.out.println();
                        } else {
                            System.out.println(line);
                        }
                    }
                    continue;
                }
                Chirp.main(lineArr);
            }
        } catch (IOException e) {
            error("Error reading automation file");
        }
    }

    /**
     * display usage message and exit
     */
    private static void usage() {
        System.err.println("usage: java Automator <automation file>");
        System.exit(1);
    }

    /**
     * print error message and call usage()
     * @param msg
     */
    private static void error(String msg) {
        System.err.println(msg);
        usage();
    }
}

```

```java
//****************************************************************************
//
// File:    CricketObserver.java
// Package: ---
// Unit:    Class CricketObserver
//
//****************************************************************************

/**
 * Class observes a group of crickets for a given number of time ticks and
 * keeps track of whether or not they have chirped or not.
 *
 * @author Jimi Ford (jhf3617)
 * @version 3-31-2015
 */
public class CricketObserver {

    /**
     * the number of crickets being observed
     */
    public final int crickets;

    /**
     * the number of time ticks observing for
     */
    public final int ticks;

    // private data members
    private boolean[][] chirps;

    /**
     * Construct a cricket observer
     * @param crickets the number of crickets to observe
     * @param ticks the number of time ticks observing for
     */
    public CricketObserver(int crickets, int ticks) {
        this.crickets = crickets;
        this.ticks = ticks;
        chirps = new boolean[ticks][crickets];
    }

    /**
     * called by a cricket to inform the observer that he has chirped
     * @param tick the time tick at which the cricket is chirping
     * @param n the unique identifier of the cricket
     */
    public void reportChirp(int tick, int n) {
        chirps[tick][n] = true;
    }

    /**
     * lookup a given time and cricket to see if it chirped at that moment
     * @param tick the moment in time to lookup
     * @param cricket the unique identifier of the cricket to check
     * @return true if it chirped
     */
    public boolean chirped(int tick, int cricket) {
        return chirps[tick][cricket];
    }

    /**
     * get the time tick at which all the crickets being observed synchronized
     * @return a number >= to 0 if they synchronized, -1 if they didn't
     */
    public int sync() {
        int row = 0;
        while(row < ticks) {
            if(sync(row)) return row;
            row++;
        }
        return -1;
    }

    /**
     * determine whether the crickets were synchronized at a given time tick or
     * not
     * @param tick the time tick to test
     * @return true if every cricket at this time tick chirped
     */
    private boolean sync(int tick) {
        boolean retval = true;
        for(int i = 0; i < crickets && retval; i++) {
            retval = chirps[tick][i];
        }
        return retval;
    }
}

```

```java
//******************************************************************************
//
// File:    Ticker.java
// Package: ---
// Unit:    Class Ticker
//
//******************************************************************************

/**
 * Class simulates a number of time ticks on a given network of crickets
 * @author Jimi Ford (jhf3617)
 * @version 3-31-2015
 */
public class Ticker {

    /**
     * tick a number of time ticks on a given network of crickets
     * @param g the network of crickets to tick
     * @param ticks the number of ticks to simulate
     */
    public static void tick(UndirectedGraph g, int ticks) {
        for(int i = 0; i < ticks; i++) {
            g.tick(i);
        }
    }
}

```

```java
//***************************************************************************
//
// File:    UndirectedEdge.java
// Package: ---
// Unit:    Class UndirectedEdge
//
//***************************************************************************

/**
 * Class UndirectedEdge represents an edge in a graph that connects two
 * vertices. It's important to note that the edge does not have a direction nor
 * weight.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class UndirectedEdge {

    // private data members
    private Cricket a, b;

    // future projects may rely on a unique identifier for an edge
    private final int id;

    /**
     * Construct an undirected edge
     * @param id a unique identifier to distinguish between other edges
     * @param a one vertex in the graph
     * @param b another vertex in the graph not equal to <I>a</I>
     */
    public UndirectedEdge(int id, Cricket a, Cricket b) {
        this.id = id;
        // enforce that a.n is always less than b.n
        if(a.n < b.n) {
            this.a = a;
            this.b = b;
        } else if(b.n < a.n) {
            this.a = b;
            this.b = a;
        } else {
            throw new IllegalArgumentException("Cannot have self loop");
        }
        this.a.addEdge(this);
        this.b.addEdge(this);
    }

    /**
     * Get the <I>other</I> vertex given a certain vertex connected to
     * this edge
     *
     * @param current the current vertex
     * @return the other vertex connected to this edge
     */
    public Cricket other(Cricket current) {
        if(current == null) return null;
        return current.n == a.n ? b : a;
    }
}
```

```java
//****************************************************************************
//
// File:    Chirp.java
// Package: ---
// Unit:    Class Chirp
//
//****************************************************************************

import java.io.IOException;

/**
 * Chirp runs a simulation of crickets chirping at night. The phenomenon we are
 * interested in studying is that some types of networks synchronize in how they
 * chirp. Based on the command line parameters, chirp tests the type of network
 * and determines what time the crickets syncrhonize.
 *
 * @author Jimi Ford (jhf3617)
 * @version 3-31-2015
 */
public class Chirp {

    private static final int GRAPH_TYPE_INDEX = 0,
                             NUM_VERTICES_INDEX = 1,
                             NUM_TICKS_INDEX = 2,
                             OUTPUT_IMAGE_INDEX = 3,
                             SEED_INDEX = 4,
                             K_INDEX = 4,
                             DE_INDEX = 4,
                             DE_SEED_INDEX = 5,
                             EDGE_PROBABILITY_INDEX = 5,
                             K_SEED_INDEX = 5,
                             REWIRE_PROBABILITY_INDEX = 6;

    /**
     * main method
     * @param args command line arguments
     */
    public static void main(String[] args) {
        if(args.length != 4 && args.length != 5 &&
                args.length != 6 && args.length != 7) usage();
        int crickets = 0, ticks = 0, k = 0, dE = 0;
        long seed = 0;
        double prob = 0;
        char mode;
        String outputImage = args[OUTPUT_IMAGE_INDEX];

        try {
            crickets = Integer.parseInt(args[NUM_VERTICES_INDEX]);
        } catch (NumberFormatException e) {
            error("<num vertices> must be a number");
        }
        try {
            ticks = Integer.parseInt(args[NUM_TICKS_INDEX]) + 1;
        } catch (NumberFormatException e) {
            error("<num ticks> must be numeric");
        }
        mode = args[GRAPH_TYPE_INDEX].toLowerCase().charAt(0);
        if(!(mode == 'c' || mode == 'r' || mode == 'k' ||
                mode == 's' || mode == 'f')) {
            error("<graph type> must be either 'c' for cycle, "
                    + "'r' for random, "
                    + "'k' for k-regular, "
                    + "'s' for small-world, "
                    + "'f' for scale-free");
        }
        UndirectedGraph g = null;
        CricketObserver o = new CricketObserver(crickets, ticks);
        switch(mode) {
        case 'r': // RANDOM GRAPH
            try {
                seed = Long.parseLong(args[SEED_INDEX]);
                prob = Double.parseDouble(args[EDGE_PROBABILITY_INDEX]);
                g = UndirectedGraph.randomGraph(
                        new Random(seed), crickets, prob, o);
            } catch(NumberFormatException e) {
                error("<seed> and <edge probability> must be numeric");
            } catch(IndexOutOfBoundsException e) {
                error("<seed> and <edge probability> must be included with "
                        + "random graph mode");
            }
            break;
        case 'c': // CYCLE GRAPH
            g = UndirectedGraph.cycleGraph(crickets, o);
            break;
        case 'k': // K-REGULAR GRAPH
            try {
                k = Integer.parseInt(args[K_INDEX]);
                g = UndirectedGraph.kregularGraph(crickets, k, o);
            } catch (NumberFormatException e) {
                error("<k> must be an integer");
            } catch (IllegalArgumentException e) {
                error("<k> must be < the number of crickets");
            }
            break;
        case 's': // SMALL WORLD GRAPH
            try {
                k = Integer.parseInt(args[K_INDEX]);
                prob = Double.parseDouble(args[REWIRE_PROBABILITY_INDEX]);
                seed = Long.parseLong(args[K_SEED_INDEX]);
                g = UndirectedGraph.smallWorldGraph(
                        new Random(seed), crickets, k, prob, o);
            } catch (NumberFormatException e) {
                error("<k> must be an integer < V, <rewire probability> "
                        + "must be a number "
                        + "between 0 and 1, and <seed> must be numeric");
            } catch (IllegalArgumentException e) {
                error("<k> must be < the number of crickets");
            }
            break;
        case 'f': // SCALE-FREE GRAPH
            try {
                dE = Integer.parseInt(args[DE_INDEX]);
                seed = Long.parseLong(args[DE_SEED_INDEX]);
                g = UndirectedGraph.scaleFreeGraph(
                        new Random(seed), crickets, dE, o);
            } catch (NumberFormatException e) {
                error("<dE> and <seed> must be numeric");
            } catch (IndexOutOfBoundsException e) {
                error("<dE> and <seed> must be supplied");
            }
        }

        g.vertices.get(0).forceChirp();
        Ticker.tick(g, ticks);



        try {
            ImageHandler.handle(o, outputImage);
        } catch (IOException e) {
            error("Problem writing image");
        }
        int sync = o.sync();
        String description;
        switch(mode) {
        case 'c': // CYCLE GRAPH
            description = "Cycle V = " + crickets +":";
            handleOutput(description,sync);
            break;
        case 'r': // RANDOM GRAPH
            description = "Random V = " + crickets +", p = " + prob + ":";
            handleOutput(description,sync);
            break;
        case 'k': // K-REGULAR GRAPH
            description = "K-regular V = " + crickets +", k = " + k + ":";
            handleOutput(description,sync);
            break;
        case 's': // SMALL-WORLD GRAPH
            description = "Small-world V = " + crickets + ", k = " + k +
                ", p = " + prob + ":";
            handleOutput(description,sync);
            break;
        case 'f': // SCALE-FREE GRAPH
            description = "Scale-free V = " + crickets +", dE = " + dE + ":";
            handleOutput(description,sync);
            break;
        }

    }

    /**
     * handle printing the results of the simulation
     * @param description the description of what kind of graph is being printed
     * @param sync time at which the network synchronized
     *       (-1 for not synchronized)
     */
    private static void handleOutput(String description, int sync) {
        System.out.print(description);
        if(sync >= 0) {
            System.out.println("\t"+" synchronized at t="+sync+".");
        } else {
            System.out.println("\t "+(char)27+"[31m"+  "did not synchronize." +
                    (char)27 + "[0m");
        }
    }

    /**
     * print an error message and call usage()
     * @param msg
     */
    private static void error(String msg) {
        System.err.println(msg);
        usage();
    }

    /**
     * usage message called when program improperly used
     */
    private static void usage() {
        System.err.println(
                "usage: java Chirp <graph type> <num vertices> <num ticks> "
                + "<output image> {(<seed> <edge probability>), or "
                + "(<k>), or "
                + "(<k> <seed> <rewire probability>), or "
                + "(<dE> <seed>)}");
        System.exit(1);
    }
}

```

```java
//******************************************************************************
//
// File:    Cricket.java
// Package: ---
// Unit:    Class Cricket
//
//******************************************************************************

/**
 * This class models a cricket that will chirp at time t + 2 if it hears a chirp
 * at time t. It inherits from vertex so that it can be connected to other
 * crickets through undirected edges.
 *
 * @author Jimi Ford (jhf3617)
 * @version 3-31-2015
 */
public class Cricket extends Vertex {

    private boolean[] chirp = new boolean[2];
    private boolean willChirp;
    private int currentTick = 0;
    private final CricketObserver observer;

    /**
     * Construct a cricket
     * @param n the unique integer identifier
     * @param o the cricket observer this cricket should report to
     */
    public Cricket(int n, CricketObserver o) {
        super(n);
        this.observer = o;
    }

    /**
     * force a cricket to chirp at the next time tick
     */
    public void forceChirp() {
        willChirp = chirp[0] = true;
    }

    /**
     * will chirp only if it is being forced to, or if it has heard a chirp
     * 2 time ticks ago
     */
    public void emitChirp() {
        if(willChirp) {
            willChirp = false;
            int n = super.degree();
            for(int i = 0; i < n; i++) {
                edges.get(i).other(this).hearChirp();
            }
            observer.reportChirp(currentTick, super.n);
        }
    }

    /**
     * hear another chirp from an adjacent cricket
     */
    private void hearChirp() {
        chirp[1] = true;
    }

    /**
     * simulate time passing by letting the cricket know what time it is
     *
     * @param tick the current time tick for this cricket
     */
    public void timeTick(int tick) {
        currentTick = tick;
        willChirp = chirp[0];
        chirp[0] = chirp[1];
        chirp[1] = false;
    }

    /**
     * determine if a given cricket is directly connected to this cricket
     * @param other the given cricket to check
     * @return true if this cricket as a single edge that connects the two
     */
    public boolean directFlight(Cricket other) {
        boolean retval = false;
        if(equals(other)) return true;
        int e = super.degree();
        Cricket o;
        for(int i = 0; i < e && !retval; i++) {
            o = super.edges.get(i).other(this);
            retval = o.equals(other);
        }
        return retval;
    }

    /**
     * determine if another object is equal to this cricket
     * @param o the other object
     * @return true if the other object is equal to this cricket
     */
    public boolean equals(Object o) {
        if( !(o instanceof Cricket)) {
            return false;
        }
        if(o == this) {
            return true;
        }
        Cricket casted = (Cricket) o;

        return casted.n == this.n;
    }
}

```

```java
//******************************************************************************
//
// File:    ImageHandler.java
// Package: ---
// Unit:    Class ImageHandler
//
//******************************************************************************

import java.io.BufferedOutputStream;


/**
 * Class takes care of saving the results of the simulation as an image
 *
 * @author Jimi Ford (jhf3617)
 * @version 3-31-2015
 */
public class ImageHandler {

    // private data members
    private static final byte SILENT = 0,
                              CHIRPED = 1,
                              SYNC = 2;

    /**
     *
     * @param o the cricket observer that holds the results of the simulation
     * @param out the name of the image file to save
     * @throws FileNotFoundException if there was an error writing to the given
     * file
     */
    public static void handle(CricketObserver o, String out)
            throws FileNotFoundException {
        AList<Color> palette = new AList<Color>();
        Color green = new Color().rgb(0, 255, 0);// green
        Color red = new Color().rgb(255, 0, 0); // red
        Color blue = new Color().rgb(0,0,255); // blue
        palette.addLast (green);
        palette.addLast (red);
        palette.addLast (blue);


        OutputStream imageout =
                new BufferedOutputStream (new FileOutputStream (new File(out)));
        IndexPngWriter imageWriter = new IndexPngWriter
                (o.ticks, o.crickets, imageout, palette);
        ByteImageQueue imageQueue = imageWriter.getImageQueue();
        byte[] bytes;
        boolean chirped;
        int sync = o.sync();
        for(int i = 0; i < o.ticks; i++) {
            bytes = new byte[o.crickets];
            for(int j = 0, cricket = 0; j < bytes.length; j++, cricket++) {
                if(i != sync) {
                    chirped = o.chirped(i, cricket);
                    bytes[j] = chirped ? CHIRPED : SILENT;
                } else {
                    bytes[j] = SYNC;
                }
            }
            try {
                imageQueue.put(i, bytes);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        try {
            imageWriter.write();
        } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}

```

```java
//******************************************************************************
//
// File:    UndirectedGraph.java
// Package: ---
// Unit:    Class UndirectedGraph
//
//******************************************************************************

import java.util.ArrayList;
import java.util.LinkedList;
import edu.rit.pj2.vbl.DoubleVbl;
import edu.rit.util.Random;

/**
 * Class UndirectedGraph represents an undirected graph meaning that if
 * there exists an edge connecting some vertex A to some vertex B, then
 * that same edge connects vertex B to vertex A.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class UndirectedGraph {

    // private data members
    private ArrayList<UndirectedEdge> edges;
    public ArrayList<Cricket> vertices;
    private int v;


    /**
     * Private constructor used internally by the static random graph
     * method
     * @param v the number of vertices in the graph
     */
    private UndirectedGraph(int v, CricketObserver o) {
        this.v = v;
        vertices = new ArrayList<Cricket>(v);
        edges = new ArrayList<UndirectedEdge>();
        for(int i = 0; i < v; i++) {
            vertices.add(new Cricket(i,o));
        }
    }

    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the id of the start vertex
     * @param goal the id of the goal vertex
     * @return the minimum distance between the two vertices
     */
    private int BFS(int start, int goal) {
        return BFS(vertices.get(start), vertices.get(goal));
    }

    /**
     * Perform a BFS to get the distance from one vertex to another
     *
     * @param start the reference to the start vertex
     * @param goal the reference to the goal vertex
     * @return the minimum distance between the two vertices
     */
    private int BFS(Cricket start, Cricket goal) {
        int distance = 0, verticesToProcess = 1, uniqueNeighbors = 0;
        LinkedList<Cricket> queue = new LinkedList<Cricket>();
        boolean[] visited = new boolean[v];
        visited[start.n] = true;
        Cricket current, t2;
        queue.add(start);
        while(!queue.isEmpty()) {
            current = queue.removeFirst();
            if(current.equals(goal)) {
                return distance;
            }
            for(int i = 0; i < current.degree(); i++) {
                t2 = current.getEdges().get(i).other(current);
                if(!visited[t2.n]) {
                    visited[t2.n] = true;
                    queue.add(t2);
                    uniqueNeighbors++;
                }
            }
            verticesToProcess--;
            if(verticesToProcess <= 0) {
                verticesToProcess = uniqueNeighbors;
                uniqueNeighbors = 0;
                distance++;
            }

        }
        return 0;
    }

    /**
     * Accumulate the distances of each pair of vertices into
     * a "running total" to be averaged
     *
     * @param thrLocal the reference to the "running total"
     * Prof. Alan Kaminsky's library handles averaging this
     * accumulated value.
     */
    public void accumulateDistances(DoubleVbl.Mean thrLocal) {
        for(int i = 0; i < v; i++) {
            for(int j = i + 1; j < v; j++) {
                int distance = BFS(i, j);
                // only accumulate the distance if the two vertices
                // are actually connected
                if(distance > 0) {
                    thrLocal.accumulate(distance);
                }
            }
        }
    }

    public void tick(int tick) {
        Cricket c;
        for(int i = 0; i < v; i++) {
```

```java
            c = vertices.get(i);
            c.timeTick(tick);
        }
        for(int i = 0; i < v; i++) {
            c = vertices.get(i);
            c.emitChirp();
        }
    }

    /**
     * Generate a random graph with a PRNG, a specified vertex count and
     * an edge probability
     *
     * @param prng Prof. Alan Kaminsky's Perfect Random Number Generator
     * @param v number of vertices to use
     * @param p edge probability between vertices
     * @return the randomly generated graph
     */
    public static UndirectedGraph randomGraph(Random prng, int v, double p,
            CricketObserver o) {
        UndirectedGraph g = new UndirectedGraph(v, o);
        UndirectedEdge edge;
        Cricket a, b;
        int edgeCount = 0;
        for (int i = 0; i < v; i++) {
            for (int j = i + 1; j < v; j++) {
                // connect edges
                // always order it `i` then `j`
                if(prng.nextDouble() <= p) {
                    a = g.vertices.get(i);
                    b = g.vertices.get(j);
                    edge = new UndirectedEdge(edgeCount++, a, b);
                    g.edges.add(edge);
                }
            }
        }
        return g;
    }

    public static UndirectedGraph cycleGraph(int v, CricketObserver o) {
        return kregularGraph(v, 1, o);
    }

    public static UndirectedGraph kregularGraph(int v, int k,
            CricketObserver o) {
        return smallWorldGraph(null, v, k, 0, o);
    }

    public static UndirectedGraph smallWorldGraph(Random prng, final int v,
            int k, double p, CricketObserver o) {
        UndirectedGraph g = new UndirectedGraph(v, o);
        UndirectedEdge edge;
        Cricket a, b, c;
        int edgeCount = 0;
        for(int i = 0; i < v; i++) {
            a = g.vertices.get(i);
            for(int j = 1; j <= k; j++) {
                b = g.vertices.get((i + j) % v);
        if(prng != null && prng.nextDouble() < p) {
          do {
            c = g.vertices.get(prng.nextInt(v));
          } while(c.n == a.n || c.n == b.n || a.directFlight(c));
          b = c;
        }
        edge = new UndirectedEdge(edgeCount++, a, b);
        g.edges.add(edge);
      }
    }
    return g;
  }

  public static UndirectedGraph scaleFreeGraph(Random prng, final int v,
      final int dE, CricketObserver o) {
    UndirectedGraph g = new UndirectedGraph(v, o);
//  boolean[]
    int edgeCount = 0;
    int c0 = prng.nextInt(v);
    int c1 = (c0 + 1) % v;
    int c2 = (c1 + 1) % v;
    Cricket a = g.vertices.get(c0), b = g.vertices.get(c1),
        c = g.vertices.get(c2);
    UndirectedEdge edge = new UndirectedEdge(edgeCount++, a, b);
    g.edges.add(edge);
    edge = new UndirectedEdge(edgeCount++, b, c);
    g.edges.add(edge);
    edge = new UndirectedEdge(edgeCount++, a, c);
    g.edges.add(edge);
    // we have 3 fully connected vertices now
    Cricket[] others = new Cricket[v-3];
    for(int other = 0, i = 0; i < v; i++) {
      if(i != c0 && i != c1 && i != c2) {
        others[other++] = g.vertices.get(i);
      }
    }
    // the rest are contained in others
    int[] prob;
    Cricket next, temp;
    ArrayList<Cricket> existing = new ArrayList<Cricket>();
    existing.add(a); existing.add(b); existing.add(c);
    for(int i = 0; i < others.length; i++) {
      next = others[i];
      existing.add(next);
      if(existing.size() <= dE) {
        for(int e = 0; e < existing.size(); e++) {
          temp = existing.get(e);
          if(next.equals(temp)) continue;
          edge = new UndirectedEdge(edgeCount++, temp, next);
          g.edges.add(edge);
        }
      } else {
        // potential bug - when do i add in the current vertex to the
        // probability distribution?
        int sumD = sumDeg(g);
        prob = new int[sumD];
        setProbabilityDistribution(g, prob);
        for(int e = 0; e < dE; e++) {
```

```
                    do {
                        int chosen = (int) Math.floor(prng.nextDouble() *
                                prob.length);
                        temp = g.vertices.get(prob[chosen]);
                    } while(next.directFlight(temp));
                    edge = new UndirectedEdge(edgeCount++, next, temp);
                    g.edges.add(edge);
                }
            }
        }

        return g;
    }

    private static void setProbabilityDistribution(UndirectedGraph g,
            int[] prob) {
        Vertex v;
        int degree = 0;
        int counter = 0;
        for(int i = 0; i < g.v; i++) {
            v = g.vertices.get(i);
            degree = v.degree();
            for(int j = counter; j < degree + counter; j++) {
                prob[j] = v.n;
            }
            counter += degree;
        }
    }

    private static int sumDeg(UndirectedGraph g) {
        int retval = 0;
        Vertex v;
        for(int i = 0; i < g.v; i++) {
            v = g.vertices.get(i);
            retval += v.degree();
        }
        return retval;
    }
}

```

```java
//****************************************************************************
//
// File:    Vertex.java
// Package: ---
// Unit:    Class Vertex
//
//****************************************************************************

import java.util.ArrayList;

/**
 * Class Vertex represents a single vertex in a graph. Vertices can be connected
 * to other vertices through undirected edges.
 *
 * @author Jimi Ford
 * @version 2-15-2015
 */
public class Vertex {

    // private data members
    protected ArrayList<UndirectedEdge> edges = new ArrayList<UndirectedEdge>();

    /**
     * The unique identifier for this vertex
     */
    public final int n;

    /**
     * Construct a vertex with a unique identifier <I>n</I>
     *
     * @param n the unique identifier to distinguish this vertex from
     *          all other vertices in the graph
     */
    public Vertex(int n) {
        this.n = n;
    }

    /**
     * Get the number of edges connected to this vertex
     *
     * @return the number of edges connected to this vertex
     */
    public int degree() {
        return edges.size();
    }

    /**
     * Get the reference to the collection of edges connected to
     * this vertex.
     *
     * @return the reference to the collection of edges
     */
    public ArrayList<UndirectedEdge> getEdges() {
        return this.edges;
    }

    /**
     * Add an edge to this vertex
     *
     * @param e the edge to add
     */
    public void addEdge(UndirectedEdge e) {
        this.edges.add(e);
    }

    /**
     * Compare another object to this one
     *
     * @param o the other object to compare to this one
     * @return true if the other object is equivalent to this one
     */
    public boolean equals(Object o) {
        if( !(o instanceof Vertex)) {
            return false;
        }
        if(o == this) {
            return true;
        }
        Vertex casted = (Vertex) o;

        return casted.n == this.n;
    }
}

```