```
/******************************************************************************
 *  Compilation:  javac Graph.java
 *  Execution:    java Graph input.txt
 *
 *  A graph, implemented using an array of sets.
 *  Parallel edges and self-loops allowed.
 *
 *  % java Graph tinyG.txt
 *  13 vertices, 13 edges
 *  0: 6 2 1 5
 *  1: 0
 *  2: 0
 *  3: 5 4
 *  4: 5 6 3
 *  5: 3 4 0
 *  6: 0 4
 *  7: 8
 *  8: 7
 *  9: 11 10 12
 *  10: 9
 *  11: 9 12
 *  12: 11 9
 *
 *  % java Graph mediumG.txt
 *  250 vertices, 1273 edges
 *  0: 225 222 211 209 204 202 191 176 163 160 149 114 97 80 68 59 58 49 44 24 15
 *  1: 220 203 200 194 189 164 150 130 107 72
 *  2: 141 110 108 86 79 51 42 18 14
 *  ...
 *
 ******************************************************************************/

import java.util.NoSuchElementException;
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner;
import java.util.StringTokenizer;

public class Graph {
    private static final String NEWLINE = System.getProperty("line.separator");

    private final int V;
    private int E;
    private Queue<Integer>[] adj;

    /**
     * Initializes an empty graph with {@code V} vertices and 0 edges.
     * param V the number of vertices
     *
     * @param  V number of vertices
     * @throws IllegalArgumentException if {@code V < 0}
     */
    public Graph(int V) {
        if (V < 0) throw new IllegalArgumentException("Number of vertices must be
nonnegative");
        this.V = V;
        this.E = 0;
        adj = (Queue<Integer>[]) new Queue[V];
        for (int v = 0; v < V; v++) {
            adj[v] = new Queue<Integer>();
        }
    }

    /**
     * Initializes a graph from the specified input stream.
     * The format is the number of vertices <em>V</em>,
```

```
     * followed by the number of edges <em>E</em>,
     * followed by <em>E</em> pairs of vertices, with each entry separated by
whitespace.
     *
     * @param  in the input stream
     * @throws IllegalArgumentException if the endpoints of any edge are not in
prescribed range
     * @throws IllegalArgumentException if the number of vertices or edges is
negative
     * @throws IllegalArgumentException if the input stream is in the wrong format
     */
    public Graph(String arg) {
        try {
            File myinput = new File(arg);
            Scanner myReader = new Scanner(myinput);
            String data;
            StringTokenizer st;

            data = myReader.nextLine();
            this.V = Integer.parseInt(data);
            if (V < 0) throw new IllegalArgumentException("number of vertices in a
Graph must be nonnegative");
            adj = (Queue<Integer>[]) new Queue[V];
            for (int v = 0; v < V; v++) {
                adj[v] = new Queue<Integer>();
            }
            data = myReader.nextLine();
            int E = Integer.parseInt(data);
            if (E < 0) throw new IllegalArgumentException("number of edges in a
Graph must be nonnegative");
            for (int i = 0; i < E; i++) {
                data = myReader.nextLine();
                st = new StringTokenizer(data);
                int v = Integer.parseInt(st.nextToken());
                int w = Integer.parseInt(st.nextToken());
                validateVertex(v);
                validateVertex(w);
                addEdge(v, w);
            }
        }
        catch (Exception e) {
            throw new IllegalArgumentException("invalid input format in Graph
constructor", e);
        }
    }


    /**
     * Returns the number of vertices in this graph.
     *
     * @return the number of vertices in this graph
     */
    public int V() {
        return V;
    }

    /**
     * Returns the number of edges in this graph.
     *
     * @return the number of edges in this graph
     */
    public int E() {
        return E;
    }

    // throw an IllegalArgumentException unless {@code 0 <= v < V}
```

```
    private void validateVertex(int v) {
        if (v < 0 || v >= V)
            throw new IllegalArgumentException("vertex " + v + " is not between 0
and " + (V-1));
    }

    /**
     * Adds the undirected edge v-w to this graph.
     *
     * @param  v one vertex in the edge
     * @param  w the other vertex in the edge
     * @throws IllegalArgumentException unless both {@code 0 <= v < V} and {@code
0 <= w < V}
     */
    public void addEdge(int v, int w) {
        validateVertex(v);
        validateVertex(w);
        E++;
        adj[v].enqueue(w);
        adj[w].enqueue(v);
    }


    /**
     * Returns the vertices adjacent to vertex {@code v}.
     *
     * @param  v the vertex
     * @return the vertices adjacent to vertex {@code v}, as an iterable
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public Iterable<Integer> adj(int v) {
        validateVertex(v);
        return adj[v];
    }

    /**
     * Returns the degree of vertex {@code v}.
     *
     * @param  v the vertex
     * @return the degree of vertex {@code v}
     * @throws IllegalArgumentException unless {@code 0 <= v < V}
     */
    public int degree(int v) {
        validateVertex(v);
        return adj[v].size();
    }


    /**
     * Returns a string representation of this graph.
     *
     * @return the number of vertices <em>V</em>, followed by the number of edges
<em>E</em>,
     *         followed by the <em>V</em> adjacency lists
     */
    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(V + " vertices, " + E + " arestas " + NEWLINE);
        for (int v = 0; v < V; v++) {
            s.append(v + ": ");
            for (int w : adj[v]) {
                s.append(w + " ");
            }
            s.append(NEWLINE);
        }
        return s.toString();
```

```
    }


    /**
     * Unit tests the {@code Graph} data type.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {
        Graph G = new Graph(args[0]);
        System.out.println(G);
    }

}
```