```java
package graphs;

public class Vertex {
    /** Used as a unique identifier for vertex to allow
     * array-based implementation.
     * Set and get index are only allowed within subclasses
     */
    protected int index;

    public Vertex( ) {
        super();
        index = -1;
    }

    protected int getIndex() {
        return index;
    }

    protected void setIndex( int i ) {
        this.index = i;
    }

    public String toString() {
        return "vertex(" + index + ")";
    }

}
```

```java
package graphs;

import java.util.Iterator;

/** Directed or undirected graph - the main difference is in
 * the treatment of edges.
 * @author Ian Hayes
 */
public interface Graph<V extends Vertex, E> extends Iterable<V>{

    /** Class used for returning edges in iterator */
    public class AdjacentEdge<V,E> {
        public V target;
        public E edgeInfo;

        public AdjacentEdge( V target, E edgeInfo ) {
            super();
            this.target = target;
            this.edgeInfo = edgeInfo;
        }
    }

    /** @return the size of the graph */
    public int size();

    /** Adds a vertex to the graph
     * @param v vertex to be added
     * @requires v is not already in a graph
     */
    public void addVertex( V v );

    /** Add an edge to the graph
     * @param u source vertex
     * @param v target vertex
     * @param e edge information
     * @requires both u and v are already vertices of the graph
     */
    public void addEdge( V u, V v, E e );

    /** Check is an edge exits from u to v
     * @param u source vertex
     * @param v target vertex
     * @requires both u and v are vertices in the graph
     * @return true if and only if graph has edge from u to v */
    public boolean hasEdge( V u, V v );

    /** @return an iterator over the vertices of the graph */
    public Iterator<V> iterator();

    /** @return iterator over the list of vertices adjacent to vertex v */
    public Iterable<AdjacentEdge<V,E>> adjacent( V v );

}
```

```java
package graphs;

/** Implementations are directed graphs.
 * @author Ian Hayes */
public interface DGraph<V extends Vertex, E> extends Graph<V, E> {

}
```

```java
package graphs;

/** Implementations are undirected graphs. */
public interface UGraph<V extends Vertex, E> extends Graph<V,E> {

}
```

```java
package graphs;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Iterator;

/** This class implements the common parts of directed and undirected
 * graphs using an adjacency list representation.
 * Only adding edges is different between the two.
 * @author Ian Hayes
 *
 * @param <V extends Vertex> type of information stored with each vertex
 * @param <E> type of information stored with each edge
 */
abstract class GraphAdj<V extends Vertex,E>
        implements Graph<V,E> {

    /** A graph is represented by a list of Ventry elements,
     * each of which contains a vertex and a list of adjacent edges.
     */
    protected List<VEntry> graph;

    protected class VEntry {
        V source;
        List<AdjacentEdge<V,E>> edges;

        VEntry( V v ) {
            super();
            this.source = v;
            this.edges = new LinkedList<AdjacentEdge<V,E>>();
        }
    }
    /** Constructor for an empty graph */
    public GraphAdj() {
        super();
        // Array implementation to allow efficient lookup of vertices
        graph = new ArrayList<VEntry>();
    }
    /** @return the number of vertices in the graph */
    public int size() {
        return graph.size();
    }
    /** Add a vertex to the graph
     * @param v vertex to be added
     * @requires v is not already in a graph
     */
    public void addVertex( V v ) {
        /** Check if vertex has been added to a graph */
        assert v.getIndex() == -1;
        v.setIndex( graph.size() );
        graph.add( new VEntry( v ) );
    }
    /** Internal check that a vertex is actually in the graph
     * @return true if and only if v is in this graph  */
    protected boolean hasVertex( V v ) {
        int i = v.getIndex();
        return 0 <= i && i < graph.size() && graph.get(i).source == v;
    }
    /** Adding an edge is different for directed and undirected graphs
     * and hence the method is abstract in this class.
     * @param u source vertex
```

```java
     * @param v target vertex
     * @param e edge information
     * @requires both u and v are already vertices of the graph
     */
    public abstract void addEdge( V u, V v, E e );

    /** Check is graph has an edge from u to v
     * @param u source vertex
     * @param v target vertex
     * @requires u and v are vertices of the graph
     * @return true if and only if the graph has an edge from u to v
     */
    public boolean hasEdge( V u, V v ) {
        assert hasVertex(u) && hasVertex(u);
        // Search the edges adjacent to u for vertex v
        for( AdjacentEdge<V,E> e : adjacent( u )  ) {
            if( e.target == v ) {
                return true;
            }
        }
        return false;
    }
    /** Allow iteration over the vertices of a graph
     * @return an iterator over the vertices
     */
    public Iterator<V> iterator() {
        return new Vertices();
    }
    /** Allow iteration over the edges adjacent to a vertex
     * @param u source vertex
     * @return edges adjacent to u as an Iterable
     */
    public Iterable<AdjacentEdge<V,E>> adjacent( V u ) {
        return graph.get(u.getIndex()).edges;
    }
    /** The iterator over vertices uses the list iterator over the entries
     *  in the graph and selects the vertex from the entry.
     */
    private class Vertices implements Iterator<V> {

        private Iterator<VEntry> entryIterator;

        public Vertices() {
            super();
            entryIterator = graph.iterator();
        }
        public boolean hasNext() {
            return entryIterator.hasNext();
        }
        public V next() {
            return entryIterator.next().source;
        }
        /** Removal of vertices is not supported */
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

```java
package graphs;

/** Implementation of a directed graph using adjacency lists
 * @author Ian Hayes
 */
public class DGraphAdj<V extends Vertex, E>
    extends GraphAdj<V, E>
    implements DGraph<V, E>
{

    public DGraphAdj() {
        super();
    }
    /** Add an edge to the graph
     * As this is a directed graph it is added from u to v only
     * @param u source vertex
     * @param v target vertex
     * @param e edge information
     * @requires both u and v are already vertices of the graph
     */
    public void addEdge( V u, V v, E e ) {
        assert hasVertex(u) && hasVertex(v);
        graph.get(u.getIndex()).edges.add(
                new Graph.AdjacentEdge<V,E>(v,e) );
    }
}
```

```java
package graphs;

/** Implementation of an undirected graph using adjacency lists
 * @author Ian Hayes
 */
public class UGraphAdj<V extends Vertex, E>
    extends GraphAdj<V,E>
    implements UGraph<V,E> {

    public UGraphAdj() {
        super();
    }

    public void addEdge( V u, V v, E e ) {
        graph.get(u.getIndex()).edges.add(
                new AdjacentEdge<V,E>(v,e) );
        graph.get(v.getIndex()).edges.add(
                new AdjacentEdge<V,E>(u,e) );
    }
}
```

```java
package graphs;

import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

/** This class implements the common parts of directed and undirected
 * graphs using an adjacency matrix representation.
 * Only adding edges is different between the two.
 * @author Ian Hayes
 *
 * @param <VertexInfo> type of information stored with each vertex
 * @param <EdgeInfo> type of information stored with each edge
 */
abstract class GraphMatrix<V extends Vertex, E> implements Graph<V,E> {

    protected E[][] edge;
    protected List<V> graph;
    int maxSize;

    public GraphMatrix(int maxSize) {
        super();
        this.maxSize = maxSize;
        graph = new ArrayList<V>(maxSize);
        edge = (E[][]) new Object[maxSize][maxSize];
//      for ( int i = 0; i < maxSize; i++ ) {
//          edge[i] = (Edge<Vertex<VI>,EI>[]) new Object[maxSize];
//      }
    }

    private class Vertices implements Iterator<V> {
        int currentVertex;

        public Vertices() {
            super();
            currentVertex = -1;
        }

        public boolean hasNext() {
            return currentVertex < graph.size()-1;
        }

        public V next() {
            currentVertex++;
            return graph.get(currentVertex);
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }

    private class Edges
        implements Iterator<AdjacentEdge<V,E>>,
                   Iterable<AdjacentEdge<V,E>> {
        int currentVertex;
        int currentEdge;

        public Edges( V u ) {
            super();
            currentVertex = u.getIndex();
            currentEdge = -1;
```

```java
        }

        public Iterator<AdjacentEdge<V,E>> iterator() {
            return this;
        }

        private void findNext() {
            do {
                currentEdge++;
            } while( currentEdge < graph.size() &&
                    edge[currentVertex][currentEdge] == null);
            currentEdge--;
        }

        public boolean hasNext() {
            findNext();
            return currentEdge < graph.size()-1;
        }

        public AdjacentEdge<V,E> next() {
            assert currentEdge < graph.size()-1;
            findNext();
            currentEdge++;
            return new AdjacentEdge<V,E>(
                    graph.get(currentVertex),
                    edge[currentVertex][currentEdge]);
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }

    public int size() {
        return graph.size();
    }

    public void addVertex( V v ) {
        int size = graph.size();
        v.setIndex(size);
        graph.add( v );
        for(int i = 0; i <= size; i++) {
            edge[size][i] = null;
            edge[i][size] = null;
        }
    }

//    private Vertex<VI> getVertex( int i ) {
//        return graph.get(i);
//    }

    private boolean hasVertex( int i ) {
        return 0 <= i && i < graph.size();
    }

    public abstract void addEdge( V u, V v, E e );

    public boolean hasEdge( V u, V v ) {
        assert hasVertex(u.getIndex()) && hasVertex(v.getIndex());
        return edge[u.getIndex()][v.getIndex()] != null;
    }
    public Iterator<V> iterator() {
        return new Vertices();
    }

    public Iterable<AdjacentEdge<V,E>> adjacent( V u ) {
        return new Edges( u );
    }
}
```

```
package graphs;

/** Implementation of a directed graph using adjacency matrix
 * @author Ian Hayes
 */
public class DGraphMatrix<V extends Vertex, E>
    extends GraphMatrix<V,E>
    implements DGraph<V,E>
{

    public DGraphMatrix(int size) {
        super(size);
    }

    public void addEdge( V u, V v, E e ) {
        assert e != null;
        edge[u.getIndex()][v.getIndex()] = e;
    }
}
```

```
package graphs;

/** Implementation of a directed graph using adjacency matrix
 * @author Ian Hayes
 */
class UGraphMatrix<V extends Vertex, E>
extends GraphMatrix<V,E>
implements UGraph<V,E>
{

    public UGraphMatrix(int size) {
        super(size);
    }

    public void addEdge( V u, V v, E e ) {
        assert e != null;
        edge[u.getIndex()][v.getIndex()] = e;
        edge[v.getIndex()][u.getIndex()] = e;
    }
}
```