# Algorithms and Data Structures

*Exercise: Simple Graph exercise with Builder pattern*

## Introduction

In this exercise you will practice the development of a generic SimpleGraph data structure including a builder class, and the implementation of a method that determines whether the graph is a connected graph.

## Generalisation and extension of the Graph API.

Below you find the generic interface definition of a simple graph API for which you should build an implementation. Your implementation can handle any type V of vertices provided that all vertices are uniquely identified by some identifying key value, i.e.

a) Your vertex class implements the getId() method.
b) This getId() method returns a value of some generic type ID.

You will test your generic implementation both with using ID=String and ID=Double for the identification of vertices.

Your implementation of the graph construction methods addVertex and addEdge should be robust such that they will not add duplicate vertices to the graph with the same Id, and no duplicate edges between the same vertices. Edges are undirected, i.e. adding an edge between v1 and v2 makes v2 a neighbour of v1 and v1 an neigbour of v2.

Besides the basic graph construction and vertex access methods, the API requires implementation of an algorithm that detects whether the graph is connected or not:

a) **Boolean isConnected()** determines whether every vertex can be reached from every other vertex in the graph.
   *Hint:* pick any available start vertex, and count how many different vertices can be reached.

For convenient testing it is handy that you add a static local builder class to the graph implementation that supports building a simple graph by compact notation of method chaining. This builder class should provide addVertices and addEdges methods that can add multiple vertices and edges in one statement.

Below you find the full definition of the SImpleGraph and SimpleGraphBuilder API-s, the heading of the generic implementation class, and some main program code snippets for testing.

Graph API interfaces:

```
public interface Identifiable<ID> {
   ID getId();    // returns the ID of the identifiable object
}
```

```
public interface SimpleGraph<V extends Identifiable<ID>, ID> {
        // vertices can be any class that can be identified by a value of any type ID
        // the Identifiable interface requires implementation of a getId() method

        // adds the vertex if not already present; returns whether at least one new vertex was added
        boolean addVertex(V vertex);

        // retrieves the vertex specified by identifier id
        V getVertex(ID id);

        // adds an edge if not already present; returns whether a new edge was added
        boolean addEdge(V vertex1, V vertex2);
        boolean addEdge(ID vertexId1, ID vertexId2);

        // returns all vertices in the graph
        Collection<V> getVertices();

        // returns all neighbours connected to the given vertex
        Collection<V> getNeighbours(V vertex);
        Collection<V> getNeighbours(ID vertexId);

        // returns the total number of vertices in the graph
        int getNumVertices();

        // returns the total number of edges in the graph
        int getNumEdges();

        // produces a formatted string representation of the complete graph
        String getAdjacencyReport();

        // calculates whether the graph is connected
        boolean isConnected();
}
```

```
public interface SimpleGraphBuilder<V extends Identifiable<ID>, ID> {
        // add multiple vertices to the graph under construction
        SimpleGraphBuilder<V,ID> addVertices(V ...vertices);

        // connect multiple existing vertices as new neighbours
        // to an existing vertex in the graph under construction
        SimpleGraphBuilder<V,ID> addEdges(V vertex, V ...neighbours);
        SimpleGraphBuilder<V,ID> addEdges(ID vertexId, ID ...neighbourIds);

        // complete and deliver the groph under construction
        SimpleGraph<V,ID> build();
    }
```

Simple Undirected Graph implementation class heading:

```
public class SUGraph<V extends Identifiable<ID>, ID> implements SimpleGraph<V, ID> {
        // TODO select appropriate data structures to store vertices and connections


        public static class Builder<V extends Identifiable<ID>, ID>
                                        implements SimpleGraphBuilder<V,ID> {
          // create a num graph under construction
          private SUGraph<V,ID> suGraph = new SUGraph();


          // TODO implement builder methods
         }


}
```

Demo program that tests your graph construction and the isConnected method:

```
System.out.println("Welcome to the Simple Undirected Graph demo program\n");

    // Build a graph with vertices that are uniquely identified by a String identifier
    SimpleGraph<Vertex,String> graph = new SUGraph.Builder<Vertex,String>()
          .addVertices(new Vertex("A"), new Vertex("B"), new Vertex("C"),
                       new Vertex("D"), new Vertex("E"))
          .addEdges("A", "B","C")
          .addEdges("B", "C","D","E")
          .addEdges("C", "D","E")
          .addEdges("D", "E")
          .build();

    System.out.println(graph);
    System.out.println(graph.getAdjacencyReport());
    System.out.println("isConnected = " + graph.isConnected() + "\n");

    graph.addVertex(new Vertex("E"));      // duplicate test
    graph.addVertex(new Vertex("F"));
    graph.addVertex(new Vertex("G"));
    graph.addVertex(new Vertex("H"));
    graph.addEdge("F", "G");
    graph.addEdge("G", "H");
    graph.addEdge("H", "I");              // non-existing vertex test

    System.out.println(graph);
    System.out.println(graph.getAdjacencyReport());
    System.out.println("isConnected = " + graph.isConnected() + "\n");
```

Expected output:

```
Welcome to the Simple Undirected Graph demo program

SUGraph with 5 vertices and 16 neighbour connections
A: [C, B]
B: [A, D, C, E]
```

```
C: [A, D, B, E]
D: [C, B, E]
E: [D, C, B]
isConnected = true

SUGraph with 9 vertices and 22 neighbour connections
A: [C, B]
B: [A, D, C, E]
C: [A, D, B, E]
D: [C, B, E]
E: [D, C, B]
F: [G, H]
G: [F, H]
H: [G, F]
I: []
isConnected = false
```

Demo program that uses the Double type to identify a vertex:

```
System.out.println("\nTry the Double type for the vertex identifier:\n");
// Build a graph with vertices that are uniquely identified by a Double identifier
SimpleGraph<Vertex2,Double> graph2 = new SUGraph.Builder<Vertex2,Double>()
        .addVertices(new Vertex2(11.1), new Vertex2(22.2), new Vertex2(33.3),
                     new Vertex2(44.4), new Vertex2(55.5))
        .addEdges(11.1, 22.2, 33.3)
        .addEdges(22.2, 33.3, 44.4, 55.5)
        .addEdges(33.3, 44.4, 55.5)
        .addEdges(44.4, 55.5)
        .build();

System.out.println(graph2);
System.out.println(graph2.getAdjacencyReport());
System.out.println("isConnected = " + graph2.isConnected() + "\n");
```

Expected output:

```
Try the Double type for the vertex identifier:

SUGraph with 5 vertices and 16 neighbour connections
11.1: [22.2, 33.3]
22.2: [11.1, 44.4, 55.5, 33.3]
44.4: [22.2, 55.5, 33.3]
33.3: [11.1, 22.2, 44.4, 55.5]
55.5: [22.2, 44.4, 33.3]
isConnected = true
```