# EECS 3311
# Software Design
# Lab 1 (100 points), Version 1

Instructors: Song Wang, Ilir Dema
Release Date: Jan 18, 2022

**Due: Jan 31, 2022, midnight**

All your lab submissions must be compilable on the department machines, using **Java 1.8**. It is then crucial that should you choose to work on your own machine, you are responsible for testing your project before submitting it for grading. This lab is intended to help you get familiar with the basic OOP design principles.

Check the Amendments section of this document regularly for changes, fixes, and clarifications.

Ask questions on the course forum on the eClass site.

## 1 Policies

- Your (submitted or un-submitted) solution to this assignment (which is not revealed to the public) remains the property of the EECS department. Do not distribute or share your code in any public media (e.g., a non-private Github repository) in any way, shape, or form before you get the permission from your instructors.

- You are required to **work on your own for this lab**. No group partners are allowed.
- When you submit your solution, you claim that it is solely your work. Therefore, it is considered as an violation of academic integrity if you copy or share any parts of your code or documentation.
- When assessing your submission, the instructor and TA may examine your doc/code, and suspicious submissions will be reported to the department/faculty if necessary. We do not tolerate academic dishonesty, so please obey this policy strictly.

- You are entirely responsible for making your submission in time.

- You may submit multiple times prior to the deadline: only the last submission before the deadline will be graded.
- Practice submitting your project early even before it is in its final form.
- No excuses will be accepted for failing to submit shortly before the deadline.
- Back up your work periodically, so as to minimize the damage should any sort of computer failures occur. You can use a **private** Github repository for your labs/projects.
- The deadline is strict with no excuses.
- Emailing your solutions to the instruction or TAs will not be acceptable.

## Amendments

- Jan 24, starter code amendment

```
/**
 * @return true if graph is empty, false otherwise.
 */
public boolean isEmpty() {
// TODO: Complete this method
// Hint: An empty graph contains zero vertices
return true; // this line needs to be rewritten
}
```

- Jan 20, pseudocode amendment

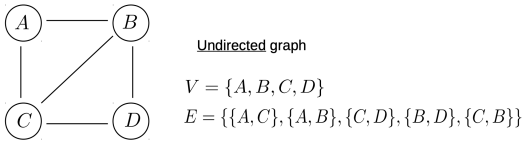
```
// open is a Java Queue, containing lists of vertices
open.insert(<initial>) // note <initial> is a list containing the initial vertex

while (! open.isEmpty()) {
      n = open.remove();  // get the list of vertices at the top of the queue
      if ([last vertex of n is equal to the destination vertex] // we are done!
          return n // n is the required path
     for m in in the list of adjacent vertices of the last vertex of n {
        if m is not in n {
          let q be a copy of the list n
          add m to q
          add q to the open queue
       }
     }
}
```

# 2 Problem

Let's consider a simple map, with four cities, labelled A,B,C,D, connected with routes between them (figure below). Notably, there is no direct route from A to D, however one can travel in several ways from A to D, for example, one may follow the path A-C-D, or A-B-C-D, or A-B-D.

Usually, such problems, are modeled using a data structure called **graph**.

A **graph** is a set of objects, called **vertices**, connected to each other by **edges**. In many practical cases, such as the example below, the *vertices* store some information (i.e name of a city), and *edges* indicate that one can traverse from one end (vertex) of the edge to the other end (vertex). Sometimes this traversal can be one directional (like an one-way road), or bidirectional. If the traversal is one-directional, the graph is called *oriented (directed)*. Otherwise, if the traversal is bi-directional, the graph is called *unoriented (undirected)*. The example in the figure below, depicts an undirected graph with 4 vertices and 5 edges:

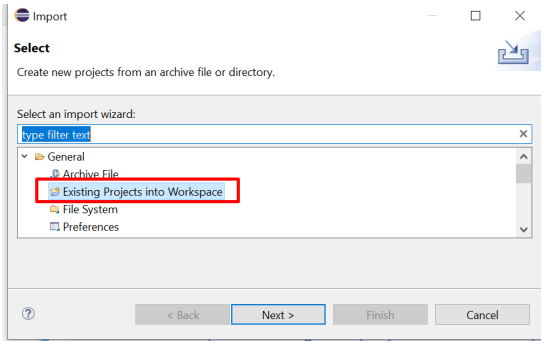



In this lab, we will tackle the problem of representing a graph using Java, and also develop a basic graph search algorithm, known as *breadth first search* (often shortened to **BFS**). The BFS searching algorithm is always able to find a path from a starting vertex to a destination vertex on a connected graph (a graph is connected it there exists a path from any vertex to any other vertex). The solution may not be optimal - this means, the solution path

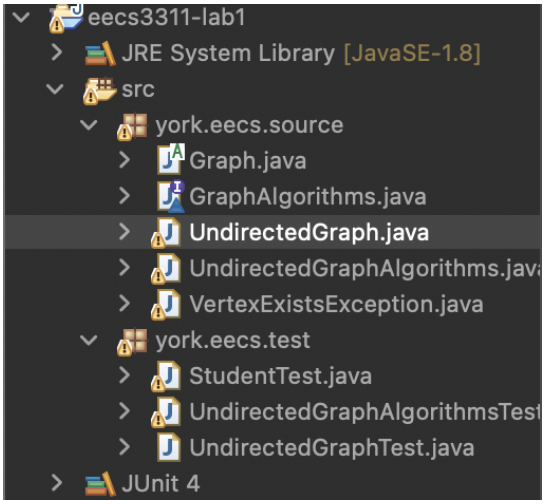
given by the BFS may not necessarily be the shortest (or, in general, most economical) path. However it is very simple, and gives us a good way to explore many basic Object Oriented Design concepts, illustrated with lots of Java functionality.

# 3 Getting Started

- Go to the course eClass page for your section. Under Lab 1, download the file **EECS3311_Lab1.zip** which contains the starter project for this lab.
- Unzip the file, and you should see a directory named **eecs3311-lab1**. It's a Eclipse project.
- You can import this project into your Eclipse as an General Java project.




- The start project should have the following structure and **on default does compile.** In case your imported copy does not compile, please make sure you are using Java 1.8.




# 4 The design

Let's explain a little bit the design of our solution. A general OO design principle is the dependency inversion, which states that designs must be based on abstractions. Applied to our problem, this principle dictates that our graph data structure must be able to accommodate generalized data types, and also must be usable in most popular graph algorithms.

## 4.1 `Graph` abstract class

Hence, we will make our **Graph** data type *abstract class*, and parametrize based on the information that will be stored on each vertex. This means, we must be able to create graphs of objects such as:

**employees**, where a vertex stores an employee object, and an edge establishes a hierarchy between them, i.e. a wroker and a supervisor.

- Example: `Graph<Employee> gEmp = new Graph<>();`

**locations**, where a vertex stores a location information (i.e. address) and an edge indicates a direct route from a location to the next location.

- Example: `Graph<City> cityNavigation = new Graph<>();`

... and many more!

A fundamental property of the objects stored on vertices of a graph, is that they need to be *comparable* to each other: we need to be able to tell when two such objects are equal, and often when one object is "bigger" than another in some sense (i.e., employee A accomplishes more sales than empolyee B, etc.).

Having in mind this description, please familiarize yourself with our abstract class `Graph`, part of the starter code.

Keep in mind, this class adheres to another equally important design principle, known as *single responsbility principle*, which states "One class must have one and only one reason to change".

## 4.2 `GraphAlgorithms` interface

In general, the algorithms describe ways the data structures behave when manuipuated to solve problems. For example, a simple algorithm used to compute the sum of an array, is iterating through its elements using a loop, and accumulating them sum into a variable (so called *accumulator pattern*).

The majority of graph algorithms visit nodes (or edges, or both) of a graph, and in the process, perform a computation. The details of the implementation differ, based on general properties of the graph (i.e. directed or undirected graphs). Therefore, the abstraction process, performed on the process of writing graph algorithms, leads to a general purpose interface, called `GraphAlgorithms` interface in our design. Please check it out carefully.

In general, such interfaces may contain many abstarct methods declaring many graph algorithms. In this assignment, we will implement only one of them, called BFS (Breadth First Search). Here is the pseudocode for this algorithm:

```
// open is a Java Queue, containing the vertices we need to check in the next step
// visited is a Java ArrayList, containing vertices that we have already visited.
// Its purpose is to avoid going back to a vertex already checked (guess what happens if we do ...)
// parent is a Java Map, whose keys are vertices, and values thier parents.
// Its purpose is to allow us construct the path from initial to destination.
open.insert(<initial>)
visited = <empty list>
path = <empty list>
while (! open.isEmpty()) {
      n = open.remove();  // remove the to vertex from the top of the queue
      // add n to the list of visited nodes
      if (n==destination) // we are done!
          { // add n to the path
            // break from the loop}
for m in n.adjacent() {
     // if m is not in visited,
     // add m to the open,
     // add the pair (m, n) to the parent map}
}
// at this point,  construct and return the path
```

# 5 Your Tasks

## 5.1 Complete the `UndirectedGraph` and `UndirectedGraphAlgorithms` Classes

- You are expected to write valid implementations in the **UndirectedGraph** and **UndirectedGraphAlgorithms** classes. Each instance of "TODO:" in these two classes indicates which implementation you have to complete.
- Your are no allowed to import external libraries libraries with the exception of the imports that are already present in the starter code.
- Study the **UndirectedGraphTest** and **UndirectedGraphAlgorithmsTest** class carefully: it shows how **UndirectedGraph** and **UndirectedGraphAlgorithms** expected to work.
- You must not change any of the methods, parameters, or statements in the **UndirectedGraphTest** and **UndirectedGraphAlgorithmsTest** classes.

## 5.2 Write Your Own Test Cases

In the **StudentTest** class, you are required to add as many tests as you judge necessary to test the correctness of **UndirectedGraph** and **UndirectedGraphAlgorithms**.

- You must add at least 10 test cases in **StudentTest**, and all your tests must pass. (In fact, you should write as many as you think is necessary.) Each test case must test properties and/or behaviours of at least one member of **UndirectedGraph** and **UndirectedGraphAlgorithms** classes.
- You will not be assessed by the quality or completeness of your tests (i.e., we will only check that you have at least 10 tests and all of them pass). However, write tests for yourself so that your software will pass all tests that we run to assess your code.

# 6 Submission

To submit:

- Close Eclipse
- Zip your lab1 project with name 'EECS3311_Lab1.zip'.
- By the due date and time, submit on eclass.