CSE 373, Winter 2013
Homework Assignment #8 (25 points)
Six Degrees of Kevin Bacon

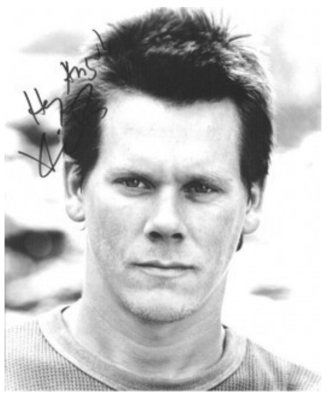
Due Friday, March 15, 2013, 11:30 PM

This program focuses on implementation of graph path searching algorithms and using those algorithms to perform searches on interesting data. Turn in files named `BaconNumberFinder.java` and `SearchableGraph.java` from the Homework section of the course web page.

You will also need to download the support .jar archives, .java files, and input .txt files from the Homework section of the course web site; place them in your Java project and add to it. If you are using Eclipse, place .java files in your project's `src/` folder and .txt files in the project root folder (the parent of `src/`). We do not guarantee that the provided tests are exhaustive; perform additional testing of your own before you submit your work.

Part A (Six Degrees of Kevin Bacon) Description:

In the first part of this assignment, you will use a provided graph implementation to solve the "Six Degrees of Kevin Bacon" problem. Kevin Bacon, a well-known actor, inspired a college movie game called Six Degrees of Kevin Bacon, which is centered on finding the *Bacon number* of an arbitrary actor or actress. The Bacon number of an actor or actress is determined by the following rules:



1. Kevin Bacon himself has a Bacon number of 0 (zero).
2. The Bacon number of any other actor is defined to be the minimum of the Bacon numbers of all others with whom the actor appeared in a movie, plus 1.

Almost every actor in Hollywood can be successfully linked to Kevin Bacon in 6 steps or fewer, hence the name "Six Degrees of Kevin Bacon". In fact, the majority of actors have a Bacon number of 2 or 3. The higher the Bacon number of an actor, the less connected they are to other actors.

(Notably, Bacon is not the most linkable actor. That honor currently goes to Dennis Hopper. The average Hopper number is 2.743. By contrast, the average Bacon number is 2.951.) More information about the Six Degrees of Kevin Bacon is available on Wikipedia at http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon. You can play an interactive web version of the game at http://oracleofbacon.org/.

Finding an actor's Bacon number and path to Kevin Bacon are tasks that can be solved by a computer. Movie data can be represented a graph of actors, with edges connecting pairs of actors who appear in movies together. Common graph path searching algorithms such as breadth-first search can discover an actor's Bacon number and path.

For this part, you are given a JAR archive `Graph.jar` that you should attach to your project. You must also attach the Guava library, which is used by `Graph.jar`. See the course web site if you need help attaching JARs to your project.

The provided JAR archive contains a complete implementation of a `SearchableGraph` class that implements a provided `Graph` interface. This class represents a graph of vertices and edges that can be weighted or unweighted, directed or undirected, and uses type parameters to store information about each vertex and edge as desired. You will use this graph class to solve the Kevin Bacon problem (as motivation to see that the graph class is useful for problem solving). In the next part of this assignment you will re-implement parts of the graph class yourself.

You are given a complete provided main client program named `MainKevinBacon.java` file that creates a `SearchableGraph` from a file of actors and movies. Your job is to write a file `BaconNumberFinder.java` that uses the methods of the searchable graph object to search the graph for the actor's Bacon number, along with the chain of films in which the actor appeared with each person to reach Kevin Bacon. The Bacon path is the shortest path between the actor and Kevin Bacon. There might be multiple equally shortest paths; it does not matter which is chosen by your graph.

Here's an example log of execution (user console input is in bold); your output should match exactly:

```
Welcome to CSE 373 Six Degrees of Kevin Bacon.
If you tell me an actor's name, I'll connect them to Kevin Bacon
through the movies they've appeared in.  I bet your actor has
a Kevin Bacon number of less than six!

Movies file name to open (Enter for movies.txt)? movies.txt
Show runtimes of each search (Y/N, Enter for N)? N
Percentage of movies to include (Enter for 100)? 100
Reading input file ...

Actor's name? Edward Norton
Path from Edward Norton to Kevin Bacon:
Edward Norton was in "Fight Club (1999)" with Brad Pitt
Brad Pitt was in "Ocean's Eleven (2001)" with Julia Roberts
Julia Roberts was in "Flatliners (1990)" with Kevin Bacon
Edward Norton's Bacon number is 3
```

The exact constructor and method your `BaconNumberFinder` class must have are the following:

```
public class BaconNumberFinder {
    public BaconNumberFinder(Graph<String, String> actors) {...}
    public void findBaconNumber(String actor) {...}
}
```

The main client program calls your `findBaconNumber` method, passing the actor that the user typed (such as "Edward Norton" above). You are to print all of the output of the movie chain leading from the actor back to Kevin Bacon as shown in the log, with one movie per line, in that exact format, followed by the actor's Bacon number. If the actor has no path to Kevin Bacon, print "No path found." If the user types the name of an actor that is not found in the graph, you program should print "No such actor." See the course web site for several test case logs and match their output exactly.

The provided JAR gives you an interface `Graph<V, E>` and a class `SearchableGraph<V, E>` that implements the interface. The generic parameter types `V` and `E` represent what type of information you want to associate with each vertex and each edge respectively in the graph. For example, if you want to create a graph of Facebook friends, where each vertex is a Facebook user name (a string) and each edge represents a friendship, you might want to store in each edge the date at which the two people became friends. So you could create a graph such as the following:

```
Graph<String, Date> facebook = new SearchableGraph<String, Date>();
facebook.addVertex("Marty");
facebook.addVertex("Jessica");
facebook.addVertex("Stuart");
facebook.addVertex("Rich");
facebook.addEdge("Marty", "Jessica", new Date(2001, 05, 08));
facebook.addEdge("Marty", "Stuart", new Date(1999, 09, 19));
facebook.addEdge("Stuart", "Rich", new Date(1981, 07, 14));
```

With that same graph, you could examine the friends of a given user with code such as the following:

```
for (String friend : facebook.neighbors("Marty")) {
    ...
}
```

You could find out about paths between particular Facebook users with code such as the following:

```
List<String> path = facebook.shortestPath("Rich", "Marty");
// [Rich, Stuart, Marty]

boolean reachable = facebook.isReachable("Jessica", "Rich");   // true
```

Here are the methods available in the `Graph` interface and `SearchableGraph` class. Since there are so many, we provide full documentation on the class web site. Refer to those documents for details of each method's behavior.

```
public SearchableGraph<V, E>()   // undirected, unweighted
public SearchableGraph<V, E>(boolean directed, boolean weighted)
public void addEdge(V v1, V v2)
public void addEdge(V v1, V v2, E e)
public void addEdge(V v1, V v2, int weight)
public void addEdge(V v1, V v2, E e, int weight)
public void addVertex(V v)
public void clear()
public void clearEdges()
public boolean containsEdge(E e)
public boolean containsEdge(V v1, V v2)
public boolean containsVertex(V v)
public int cost(List<V> path)
public int degree(V v)
public E edge(V v1, V v2)
public int edgeCount()
public Collection<E> edges()
public int edgeWeight(V v1, V v2)
public int inDegree(V v)
public boolean isDirected()
public boolean isEmpty()
public boolean isReachable(V v1, V v2)         // DFS
public boolean isWeighted()
public List<V> minimumWeightPath(V v1, V v2)   // Dijkstra's algorithm
public Set<V> neighbors(V v)
public int outDegree(V v)
public void removeEdge(E e)
public void removeEdge(V v1, V v2)
public void removeVertex(V v)
public List<V> shortestPath(V v1, V v2)        // BFS
public String toString()
public String toStringDetailed()
public int vertexCount()
public Set<V> vertices()
```

Part B (SearchableGraph) Description:

In the second part of this assignment, you will re-implement the `SearchableGraph` class that you used in Part A. You do not need to rewrite the entire graph implementation; you must only rewrite three path-searching algorithms: depth-first search (DFS), breadth-first search (BFS), and Dijkstra's algorithm.

We provide you a superclass named `AbstractGraph` that implements all of the `Graph` methods other than the following three, which you must write yourself. The idea is that `AbstractGraph` contains implementation of basics like collections of vertices and edges. You can call all of the other methods on the superclass graph as appropriate to help you implement the remaining path search behavior. As mentioned previously, please refer to the detailed `Graph` and `AbstractGraph` documentation on the class web site to see all of the assets available to you from the superclass.

The internal graph representation in the superclass is an "adjacency map". An adjacency map is essentially an adjacency matrix, except that a nested `Map` or a Guava `Table` is used that connects pairs of vertices to their associated edges. This representation has similar benefits to an adjacency matrix, for example, the graph will have constant (O(1)) expected runtime for common operations such as adding/retrieving vertices and edges, or getting collections of vertices and neighbors. It also avoids the O(V^2) memory that would be used by a full 2D array matrix representation.

You must write the following two constructors and three methods. You may define additional `private` methods as needed. All methods must run in the average runtime specified. You should not add any fields to your class. None of your methods should modify the state of the graph, in terms of the set of vertices and edges that it stores.

`public` **`SearchableGraph`**`()`

In this constructor you should initialize a new undirected, unweighted, empty graph. All you need to do is to call the corresponding zero-argument constructor from the superclass.

`public` **`SearchableGraph`**`(boolean directed, boolean weighted)`

In this constructor you should initialize a new empty graph that can be directed or undirected, weighted or unweighted. All you need to do is to call the corresponding two-argument constructor from the superclass.

`public boolean` **`isReachable`**`(V v1, V v2)`

In this method you should explore the graph to see whether there is any path that leads from the given starting vertex `v1` to the given ending vertex `v2`, and if so, return `true`. If there is not any path from `v1` to `v2`, you should return `false`. You should use the **depth-first search (DFS)** algorithm to search for the path. By definition, any vertex can reach itself, so if the same vertex is passed as `v1` and `v2`, your method should return `true`. This method should be $O(V + E)$, which is the runtime of DFS. If either of the vertices passed is `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph, you should throw an `IllegalArgumentException`.

The version of depth-first search shown in the lecture slides carries along a list of vertices in the path so far and adds / removes from that list on each recursive call along the way, so that the actual path will be known once it is found. In this assignment, though, you do not need to output the path, just return whether or not there is a path, so you do not need to pass along any such list in your code for `isReachable`.

`public List<V>` **`shortestPath`**`(V v1, V v2)`

In this method you should explore the graph to find the path with the least number of vertices that leads from the given starting vertex `v1` to the given ending vertex `v2`, and return that path as a list of its vertices. You should use the **breadth-first search algorithm (BFS)** to find the path. If `v1` and `v2` are the same, the shortest path from a vertex to itself should be a one-element list containing only that vertex. This method should be $O(V + E)$, which is the runtime of BFS. If there is not any path from `v1` to `v2`, you should return `null`. If either of the vertices passed is `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph, you should throw an `IllegalArgumentException`.

`public List<V>` **`minimumWeightPath`**`(V v1, V v2)`

In this method you should explore the graph to find the path with the lowest cost (total weight) that leads from the given starting vertex `v1` to the given ending vertex `v2`, and return that path as a list of its vertices You should use **Dijkstra's algorithm** to find the path. If `v1` and `v2` are the same, the minimum weight path from a vertex to itself should be a one-element list containing only that vertex. This method should be $O(E \log V)$ for sparse graphs, which is the runtime of Dijkstra's algorithm. If there is not any path from `v1` to `v2`, you should return `null`. If either of the vertices passed is `null`, you should throw a `NullPointerException`. If either of the vertices passed is not a part of the graph, you should throw an `IllegalArgumentException`.

See the course lecture slides and/or the textbook for the pseudo-code of each of these three graph search algorithms. Following that pseudo-code closely will lead you to a correct solution.

Some of the path-searching algorithms expect you to keep track of information about each vertex, such as whether or not it has been visited yet, or the current cost to reach it, or a 'previous' vertex to point back to from that vertex. To help you implement such functionality, the abstract superclass provides an internal structure associated with each vertex called a `Vertex` object (not to be confused with the generic type `V`). A `Vertex` object provides the following relevant methods:

`public int` **`cost`**`()`
`public void` **`setCost`**`(int cost)`
	Gets or sets a 'cost' associated with reaching this vertex. This is useful when implementing Dijkstra's algorithm.

`public V` **`previous`**`()`
`public void` **`setPrevious`**`(V previous)`
	Gets or sets a 'previous' reference from this vertex to some other vertex that should come before it in a path. This is useful when implementing breadth-first search and Dijkstra's algorithm.

`public boolean` **`visited`**`()`
`public void` **`setVisited`**`(boolean visited)`
	Gets or sets a flag indicating whether this vertex has been visited yet by your algorithm. Useful for all three path algorithms.

`public void` **`clear`**`()`	Wipes out any data set previously in this `Vertex` object, such as any previous vertex, cost, etc., and sets visited flag back to false.

The abstract superclass has a method `vertexInfo` that accepts a parameter of type `V` and returns the `Vertex` object corresponding to that vertex. The superclass creates and maintains such `Vertex` objects for you. For example, if you have a `V` object named `v1` and you want to mark `v1` as visited and set its cost to 42, you could write the following code:

```
Vertex<V> info = vertexInfo(v1);
info.setVisited(true);
info.setCost(42);
```

Setting and examining information about the vertices is useful to help you write the three path-searching algorithms. But the information you set in the vertices is not automatically wiped after you are done with the algorithm. So you should make sure to clear out all of the vertex objects at the start of each path-searching algorithm method. To do this, call the abstract graph superclass's `clearVertexInfo` method:

```
clearVertexInfo();
```

When you finish this part of the assignment, you can go run your Kevin Bacon programs with your own searchable graph class. This provides a good test of your graph code. There are other test programs posted on the class web site.

Hints:

- In Part B, we suggest implementing the three path-searching algorithms in the order shown in this handout. DFS and BFS are easier to implement than Dijkstra's algorithm.
- Dijkstra's algorithm is supposed to use a priority queue of vertices to visit, ordered in ascending order of cost. You can use a `java.util.PriorityQueue` to do this. You will need to write an inner `Comparator` that compares vertices by cost, looking up their `Vertex` info as needed. If you modify the cost of a vertex, you should remove that vertex from the priority queue and re-add it to repair the ordering.
- You can print the state of the graph at any time using its `toString` method, and to see even more detail about every vertex and edge and associated info for each, try calling the `toStringDetailed` method instead, which returns a longer string with lots of information about the graph's state.
  `System.out.println(this.toStringDetailed());`

Style Guidelines and Grading:

We will grade your program's behavior and output to give you an **External Correctness** score. You can use the course web site's Output Comparison Tool to check your output for each part of the assignment.

We will also grade your code quality (**Internal Correctness**). There are many general Java coding styles that you should follow, such as naming, indentation, avoiding redundancy, etc. For a list of these, please see the **Style Guide** document posted on the class web site. You should follow all of its guidelines as appropriate on this and future assignments.

Do not declare a value as a private field unless it is necessary to retain it as part of the state of your object. Your searchable graph class should not need any fields.

You must also write **efficient code** and match the expected **Big-Oh** demands of each method. Relaxing the Big-Oh might make it easier to implement the functionality; but the whole point is implement the graph efficiently. So this will be a grading focus. If you implement each path searching algorithm following the descriptions from class and the textbook, you should be okay. A common source of inefficient code is calling bulky methods as helpers when it is not necessary to do so, or unnecessarily looping over data structures (such as looping over every vertex or edge) more than necessary.

Implement each path-searching method using the algorithm described: DFS, BFS, and Dijkstra's. If you do not do so, you will lose points. Since your three path-searching methods use very different algorithms, and since each algorithm is expensive to run, none of the three methods should call any of the others as a helper, because this would be inefficient.

Watch out for **redundant code**. If you are performing identical or very similar commands repeatedly, factor out the common code and logic into a helper method, loop, or other facility to remove the redundancy.

In your method comment headers, comment what exceptions (if any) are thrown by each method and under what conditions they are thrown. In addition to headers on each class and each method, also make sure to include inline comments next to any complex or tricky code, briefly explaining its purpose. These inside-method comments are good places to explain details of your implementation that might not be appropriate to put in method or class headers.