Graphs:

- A graph is a data structure that has two types of elements, *vertices* and *edges*.
  - An edge is a connection between two vetices
  - If the connection is symmetric
    (in other words A is connected to B ⇔ B is connected to A),
    then we say the graph is *undirected*.
  - If an edge only implies one direction of connection, we say the graph is *directed*.
- The edges of a directed graph can be represented by ordered pairs, (A, B), where A and B are vertices of the graph. (A, B) is an edge means "A is connected to B"
- The edges of an undirected graph can be represented by unordered pairs {A, B} where A and B are vertices of the graph. {A, B} means "A and B are connected"
  (Note: I might sometimes write (A, B) for edges in an undirected graph with the understanding that in that case the pair is really unordered)
- Note: I sometimes refer to vertices as *nodes*.

- A ***path*** in a graph is a sequence of vertices of the graph, $V_1\ V_2 \ldots V_n$,
  where there is an edge from V_i to V_{i+1} $\forall$ i < n
- An undirected graph is said to be ***connected*** if there is a path between every pair of vertices in the graph.
- A path is called a ***cycle*** if it starts and ends at the same vertex and no edge is repeated.
- A ***tree*** is an connected, undirected graph that contains no cycles and has one vertex designated to be the ***root***.

- Some real-world applications of graphs are:
    1. Simple maps where the Vertices are cities and the edges are roads.
        a. For real world examples the vertices and edges might contain other information
        b. In this example the Cities might have names and the edges distances
    2. Friendship relations on FaceBook
        a. Here the vertices are the profiles and there is an edge between them if they are "friends"
    3. Air Travel, where the Vertices are airports and the edges are flights.
        a. What information might we want on the vertices and edges here?
- Which of these are directed graphs?
- Which of these graphs might have cycles?
- Which of these are connected?

- Another example is the "Kevin Bacon Game"
  - The idea is to connect any actor to Kevin Bacon
  - Here the vertices are actors
  - Two actors are connected if they appear in the same movie.
  - For example:
    - **John Lennon** was in
      - **A Hard Days Night** with
    - **Phil Collins** who was in
      - **Balto** with
    - **Kevin Bacon**
- There is an Oracle of Bacon site which finds these connections for any actor

- To solve the Kevin Bacon Game you need to be able to take two vertices of a graph and see if there is a path from one to the other.
  - A classic algorithm for doing this is a Depth First Search (DFS)
  - You have already seen DFS for trees in 120. For trees the algorithm is:
  - Given two vertices Start and Goal:
    - **`Boolean DFS (Start, Goal)`**
      - **`if Start == Goal then return true`**
      - **`for every vertex V connected to Start do`**
        - **`if DFS(V, Goal) then return true`**
      - **`return false`**
- This will work for trees but not for general graphs. Why not?

- The algorithm given on the previous slide can result in an infinite loop if the graph has a cycle.
- To modify it so that it works for all graphs we need to mark a node so that we don't explore it more than once:
    - **`Boolean DFA (Start, Goal)`**
        - **`if Start == Goal then return true`**
        - **`if Start is marked then return false`**
        - **`mark Start`**
        - **`for every vertex V connected to Start do`**
            - **`if DFS(V, Goal) return true`**
        - **`return false`**
- Note that when we start this algorithm all nodes must be unmarked.

- Some additional points you might want to think about:
    1. If the answer to the Kevin Bacon game is "yes", you're supposed to give the path from the actor to Kevin Bacon. The DFS as written doesn't do that. How could we modify it to give us the path?
    2. The Kevin Bacon score is the shortest path to Kevin Bacon.
        - Does the DFS find the shortest path?
        - The algorithm that finds the shortest path is the Breadth First Search (BFS) which is not as neat to write. Any idea how that would work?

- How to implement a graph in Java:
- Let's limit ourselves to primitives and the objects we've learned so far: arrays, Strings, HashMaps, and ArayLists.
- Obviously we want classes for vertices and edges.
- The graph class should include these, but how?
- The answer will depend on how you want to use the graph.
- Let's assume we want to use the graph for the Kevin Bacon Game.
- Since we want to get to a node quickly by the actor's name, let's store the nodes in a HashMap indexed by the name.
- Storing the edges in a HashMap doesn't make sense. What would we index on?

- We could store the edges in a linked list or in an ArrayList, but given a vertex we would have to search the whole thing to find that vertex's edges. ☹
- Instead, we can store the edges for each vertex together (in a linked list or an ArrayList).
- We can associate the edges with a vertex by either including the list of edges in the vertex object.