Detect cycle in an undirected graph

```
import java.util.*;
```

- **`import java.util.*`**: Imports all utility classes from the Java library, including `ArrayList`, `List`, and others, which are needed for creating the graph structure.

```
public class GfG {
```

- **`class GfG`**: Defines the main class named `GfG` (short for GeeksforGeeks, commonly used for coding examples).

```
// A recursive function that
// uses visited[] and parent to detect
// cycle in subgraph reachable from vertex v.
static boolean isCyclicUtil(int v, List<List<Integer>> adj,
                            boolean[] visited, int parent) {
```

- **`isCyclicUtil`**: This is a recursive helper function used to check whether a cycle exists in the part of the graph that can be reached from vertex `v`.
  - **Parameters**:
    - `int v`: The current vertex.
    - `List<List<Integer>> adj`: The adjacency list representing the graph.
    - `boolean[] visited`: Array to mark visited nodes.
    - `int parent`: Tracks the parent of the current vertex to avoid false cycle detection.

```
// Mark the current node as visited
visited[v] = true;
```

- **`visited[v] = true`**: Marks the current node `v` as visited to avoid revisiting it.

```
// Recur for all the vertices adjacent to this vertex
for (int i : adj.get(v)) {
```

- **`for (int i : adj.get(v))`**: Iterates over all vertices adjacent to vertex `v`.

```
// If an adjacent vertex is not visited,
// then recur for that adjacent
if (!visited[i]) {
    if (isCyclicUtil(i, adj, visited, v))
        return true;
}
```

- **`if (!visited[i])`**: If the adjacent vertex `i` has not been visited, recursively call `isCyclicUtil` for that vertex.
- **`if (isCyclicUtil(i, adj, visited, v)) return true;`**: If the recursive call returns `true`, a cycle has been found, so return `true`.

```
// If an adjacent vertex is visited and
// is not parent of current vertex,
// then there exists a cycle in the graph.
else if (i != parent)
    return true;
```

- **`else if (i != parent)`**: If the adjacent vertex `i` is already visited and is not the parent of the current vertex `v`, a cycle is detected (a back edge exists).
- **`return true`**: Return `true` since a cycle is found.

```
return false;
}
```

- **`return false`**: If no cycle is detected, return `false` to indicate that the current path has no cycle.

```
// Returns true if the graph contains a cycle, else false.
static boolean isCyclic(int V, List<List<Integer>> adj) {
```

- **`isCyclic`**: This is the main function to detect if there is a cycle in the undirected graph.
    - **Parameters**:
        - `int V`: The number of vertices.
        - `List<List<Integer>> adj`: The adjacency list representing the graph.

```
// Mark all the vertices as not visited
boolean[] visited = new boolean[V];
```

- **`boolean[] visited = new boolean[V];`**: Initializes a `visited` array to keep track of whether a vertex has been visited.

```
// Call the recursive helper function to detect cycle in different DFS
trees
for (int u = 0; u < V; u++) {
```

- **`for (int u = 0; u < V; u++)`**: Loop through all vertices in the graph.

```
// Don't recur for u if it is already visited
if (!visited[u]) {
    if (isCyclicUtil(u, adj, visited, -1))
        return true;
}
```

- **`if (!visited[u])`**: For any unvisited vertex `u`, call the recursive helper function `isCyclicUtil` to check for a cycle. If a cycle is found, return `true`.
- **`isCyclicUtil(u, adj, visited, -1)`**: The parent is passed as `-1` since there is no parent for the first vertex in DFS traversal.

```
return false;
}
```

- **`return false`**: If no cycle is detected after checking all vertices, return `false`.

Driver Code (Main Function)

```
// Driver program to test above functions
public static void main(String[] args) {
    int V = 3;
    List<List<Integer>> adj = new ArrayList<>(V);
```

- **`public static void main`**: The main method where the execution of the program begins.
- **`int V = 3`**: The graph contains 3 vertices.
- **`List<List<Integer>> adj = new ArrayList<>(V);`**: Creates an adjacency list to store the graph's edges. The graph has `V` vertices.

```
for (int i = 0; i < V; i++) {
    adj.add(new ArrayList<>());
}
```

- **Initialize adjacency list**: Creates empty adjacency lists for each vertex (0, 1, and 2).

```
// Add edges to the graph
adj.get(1).add(0);
adj.get(0).add(1);
adj.get(0).add(2);
adj.get(2).add(0);
adj.get(1).add(2);
adj.get(2).add(1);
```

- **Adding edges to form the graph**:
  - 1 → 0 and 0 → 1 (edge between vertex 1 and 0),
  - 0 → 2 and 2 → 0 (edge between vertex 0 and 2),
  - 1 → 2 and 2 → 1 (edge between vertex 1 and 2).

    - This forms a cycle in the graph.

```
System.out.println(isCyclic(V, adj) ? "Contains cycle" : "No Cycle");
```

- **Cycle detection**: Calls the `isCyclic` function to check if the graph contains a cycle. If `true`, it prints "Contains cycle"; otherwise, it prints "No Cycle".

Another Graph Example

```
V = 3;
List<List<Integer>> adj2 = new ArrayList<>(V);
for (int i = 0; i < V; i++) {
    adj2.add(new ArrayList<>());
}
adj2.get(0).add(1);
adj2.get(1).add(0);
adj2.get(1).add(2);
adj2.get(2).add(1);
```

- **Another graph definition**: Creates a new adjacency list for a second graph with 3 vertices and adds edges:
    - 0 → 1 and 1 → 0 (edge between vertex 0 and 1),
    - 1 → 2 and 2 → 1 (edge between vertex 1 and 2).
    - This graph does not contain a cycle.

```
System.out.println(isCyclic(V, adj2) ? "Contains Cycle" : "No Cycle");
```

- **Cycle detection for the second graph**: Checks if the second graph contains a cycle and prints the result.