```
import java.util.ArrayList;
import java.util.List;
```

- **`import` statements**: These import the `ArrayList` and `List` classes from the Java utility library. They are used for creating dynamic lists.

```
class GfG {
```

- **`class GfG`**: This defines a class named `GfG` (short for GeeksforGeeks, where this type of code is often shared) which will contain methods for detecting cycles in a graph.

```
// Utility function to detect cycle in a directed graph
private static boolean isCyclicUtil(List<List<Integer>> adj, int u,
                                    boolean[] visited, boolean[] recStack) {
```

- **`isCyclicUtil`**: This is a recursive helper function that checks whether there's a cycle starting from vertex `u`.
- **Parameters**:
  - `List<List<Integer>> adj`: The adjacency list representing the graph.
  - `int u`: The current node being processed.
  - `boolean[] visited`: Array tracking whether a node has been visited.
  - `boolean[] recStack`: Array tracking nodes in the current recursive stack (part of the DFS traversal).

java
Copy code

```
if (!visited[u]) {
```

- **`if (!visited[u])`**: If the current node `u` has not been visited, continue processing it. Otherwise, skip this node since it has already been processed.

```
// Mark the current node as visited
// and part of recursion stack
visited[u] = true;
recStack[u] = true;
```

- **`visited[u] = true`**: Mark the current node as visited.
- **`recStack[u] = true`**: Mark the current node as being part of the recursion stack to detect back edges (which form cycles).

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

- **`for (int x : adj.get(u))`**: For each adjacent node `x` connected to node `u` in the graph, iterate through its neighbors.

```
if (!visited[x] && isCyclicUtil(adj, x, visited, recStack)) {
    return true;
} else if (recStack[x]) {
    return true;
}
```

- **`if (!visited[x] && isCyclicUtil(adj, x, visited, recStack))`**: If the adjacent node `x` has not been visited, recursively call `isCyclicUtil` to check if visiting `x` leads to a cycle.
- **`else if (recStack[x])`**: If the adjacent node `x` is already in the recursion stack (`recStack[x]` is true), it means there's a back edge, which indicates a cycle. Return `true`.

```
}
// Remove the vertex from recursion stack
recStack[u] = false;
return false;
```

- **`recStack[u] = false`**: Remove the node `u` from the recursion stack once all its neighbors have been processed.
- **`return false`**: Return `false` if no cycle was detected starting from this node.

```
// Function to detect cycle in a directed graph
```

```
public static boolean isCyclic(List<List<Integer>> adj, int V) {
    boolean[] visited = new boolean[V];
    boolean[] recStack = new boolean[V];
```

- **`isCyclic`**: This is the main function that detects if there is a cycle in the graph.
- **`boolean[] visited`**: Array to track if a node has been visited.
- **`boolean[] recStack`**: Array to track the recursion stack.

```
// Call the recursive helper function to detect cycle in different DFS
trees
for (int i = 0; i < V; i++) {
    if (!visited[i] && isCyclicUtil(adj, i, visited, recStack)) {
        return true;
    }
}
```

- **`for (int i = 0; i < V; i++)`**: Iterate through each node in the graph.
- **`if (!visited[i] && isCyclicUtil(adj, i, visited, recStack))`**: If the node has not been visited, call `isCyclicUtil` to check for cycles from that node.
- **`return true`**: If any cycle is detected, return `true`.

```
return false;
}
```

- **`return false`**: If no cycle is found after checking all nodes, return `false`.

```
// Driver function
public static void main(String[] args) {
    int V = 4;
    List<List<Integer>> adj = new ArrayList<>();
```

- **`public static void main`**: The main method where the program execution begins.
- **`int V = 4`**: The number of vertices in the graph is set to 4.

- **`List<List<Integer>> adj = new ArrayList<>()`**: Create an adjacency list to represent the graph with `V` vertices.

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

- **`for (int i = 0; i < V; i++)`**: Initialize each element of the adjacency list as a new `ArrayList`.

```
// Adding edges to the graph
adj.get(0).add(1);
adj.get(0).add(2);
adj.get(1).add(2);
adj.get(2).add(0);
adj.get(2).add(3);
adj.get(3).add(3);
```

- **Adding edges**: The edges of the graph are defined by adding directed edges between vertices:
    - 0 → 1, 0 → 2, 1 → 2, 2 → 0, 2 → 3, and 3 → 3 (a self-loop on node 3).

```
// Function call
if (isCyclic(adj, V)) {
    System.out.println("Contains cycle");
} else {
    System.out.println("No Cycle");
}
```

- **`isCyclic(adj, V)`**: Call the `isCyclic` function to check if the graph contains a cycle.
- **Output**: Print whether the graph contains a cycle or not based on the result.