1)

The program **RAG** is a Java program that implements a resource allocation graph.

The class RAG has the following variables:

* numProcesses: The number of processes in the graph.
* numResources: The number of resources in the graph.
* allocation: A 2D array that stores the allocation of resources to processes. The first dimension of the array represents the process ID, and the second dimension represents the resource ID.
* request: A 2D array that stores the requests of processes for resources. The first dimension of the array represents the process ID, and the second dimension represents the resource ID.

The class RAG has the following **methods**:

* RAG(int numProcesses, int numResources): The constructor for the RAG class. It sets the numProcesses and numResources variables.
* setAllocation(int processId, int resourceId, int value): This method sets the allocation of resource resourceId to process processId to value.
* setRequest(int processId, int resourceId, int value): This method sets the request of process processId for resource resourceId to value.
* printGraph(): This method prints the resource allocation graph and the process request graph.

The main method of the program creates a new RAG object and sets the allocation and request of resources for some processes. Then, it calls the printGraph() method to print the resource allocation graph and the process request graph.

Here is an explanation of the output of the program:

* The first line of the output, Resource Status Graph:, indicates that the following lines will show the resource allocation graph.
* The second line, R0\tR1\tR2, shows the column headers for the resource allocation graph.
* The next four lines show the allocation of resources for each process. For example, the line P0: 1\t0\t0 means that process 0 is allocated 1 unit of resource 0, 0 units of resource 1, and 0 units of resource 2.
* The line Process Status Graph: indicates that the following lines will show the process request graph.
* The second line, P0\tP1\tP2\tP3, shows the column headers for the process request graph.
* The next four lines show the requests of resources for each process. For example, the line R0: 0\t-1\t1\t0 means that process 0 requests 0 units of resource 0, -1 units of resource 1, 1 unit of resource 2, and 0 units of resource 3.

2)  
The program **WFG** is a Java program that implements a wait-for graph.

The class WFG has the following variables:

* numProcesses: The number of processes in the graph.
* numResources: The number of resources in the graph.
* allocation: A 2D array that stores the allocation of resources to processes. The first dimension of the array represents the process ID, and the second dimension represents the resource ID.
* request: A 2D array that stores the requests of processes for resources. The first dimension of the array represents the process ID, and the second dimension represents the resource ID.
* waitfor: A 2D array that stores the wait-for graph. The first dimension of the array represents the process ID, and the second dimension represents the process ID that it is waiting for.

The class WFG has the following **methods**:

* WFG(int numProcesses, int numResources): The constructor for the WFG class. It sets the numProcesses and numResources variables.
* setAllocation(int processId, int resourceId, int value): This method sets the allocation of resource resourceId to process processId to value.
* setRequest(int processId, int resourceId, int value): This method sets the request of process processId for resource resourceId to value.
* convertWaitforGraph(): This method converts the resource allocation graph to a wait-for graph.
* printGraph(): This method prints the resource allocation graph, the process request graph, and the wait-for graph.

The main method of the program creates a new WFG object and sets the allocation and request of resources for some processes. Then, it calls the convertWaitforGraph() method to convert the resource allocation graph to a wait-for graph. Finally, it calls the printGraph() method to print the resource allocation graph, the process request graph, and the wait-for graph.

Here is an explanation of the output of the program:

* The first line of the output, Resource Status Graph:, indicates that the following lines will show the resource allocation graph.
* The second line, R0\tR1\tR2, shows the column headers for the resource allocation graph.
* The next four lines show the allocation of resources for each process. For example, the line P0: 1\t0\t0 means that process 0 is allocated 1 unit of resource 0, 0 units of resource 1, and 0 units of resource 2.
* The line Process Status Graph: indicates that the following lines will show the process request graph.
* The second line, P0\tP1\tP2\tP3, shows the column headers for the process request graph.
* The next four lines show the requests of resources for each process. For example, the line R0: 0\t1\t1\t1 means that process 0 requests 0 units of resource 0, 1 unit of resource 1, 1 unit of resource 2, and 1 unit of resource 3.
* The line Wait-for graph: indicates that the following lines will show the wait-for graph.
* The second line, P0\tP1\tP2\tP3, shows the column headers for the wait-for graph.
* The next four lines show the wait-for relationships between processes. For example, the line P0: 0 \t0 \t0 \t0 means that process 0 is not waiting for any other processes.

The wait-for graph is a directed graph that shows the relationship between processes that are waiting for resources that are allocated to other processes. In the wait-for graph, a directed edge from process P1 to process P2 means that process P1 is waiting for a resource that is allocated to process P2.

3)

The program **Deadlock** is a Java program that implements a deadlock detection algorithm.

The class Deadlock has the following variables:

* numProcesses: The number of processes in the graph.
* numResources: The number of resources in the graph.
* allocation: A 2D array that stores the allocation of resources to processes. The first dimension of the array represents the process ID, and the second dimension represents the resource ID.
* request: A 2D array that stores the requests of processes for resources. The first dimension of the array represents the process ID, and the second dimension represents the resource ID.
* waitfor: A 2D array that stores the wait-for graph. The first dimension of the array represents the process ID, and the second dimension represents the process ID that it is waiting for.

The class Deadlock has the following **methods**:

* Deadlock(int numProcesses, int numResources): The constructor for the Deadlock class. It sets the numProcesses and numResources variables.
* setAllocation(int processId, int resourceId, int value): This method sets the allocation of resource resourceId to process processId to value.
* setRequest(int processId, int resourceId, int value): This method sets the request of process processId for resource resourceId to value.
* convertWaitforGraph(): This method converts the resource allocation graph to a wait-for graph.
* printGraph(): This method prints the resource allocation graph, the process request graph, and the wait-for graph.
* isDeadlocked(): This method detects deadlocks by finding a cycle in the wait-for graph.
* hasCycle(int processId, List<Integer> visit, List<Integer> stack): This method recursively checks if there is a cycle in the wait-for graph starting from process processId.

The **main** method of the program creates a new Deadlock object and sets the allocation and request of resources for some processes. Then, it calls the convertWaitforGraph() method to convert the resource allocation graph to a wait-for graph. Finally, it calls the isDeadlocked() method to detect deadlocks. If a deadlock is detected, the program will print a message indicating that a deadlock has been detected. Otherwise, the program will print a message indicating that no deadlock has been detected.

4)

The program HRMDD is a Java program that implements a deadlock detection algorithm for two sites.

The class HRMDD has the following variables:

* numProcesses: The number of processes in each site.
* numResources: The number of resources in each site.
* allocation: A 2D array that stores the allocation of resources to processes. The first dimension of the array represents the process ID, and the second dimension represents the resource ID.
* request: A 2D array that stores the requests of processes for resources. The first dimension of the array represents the process ID, and the second dimension represents the resource ID.
* waitfor: A 2D array that stores the wait-for graph. The first dimension of the array represents the process ID, and the second dimension represents the process ID that it is waiting for.

The class HRMDD has the following methods:

* HRMDD(int numProcesses, int numResources): The constructor for the HRMDD class. It sets the numProcesses and numResources variables.
* setAllocation(int processId, int resourceId, int value): This method sets the allocation of resource resourceId to process processId to value.
* setRequest(int processId, int resourceId, int value): This method sets the request of process processId for resource resourceId to value.
* convertWaitforGraph(): This method converts the resource allocation graph to a wait-for graph.
* printGraph(): This method prints the resource allocation graph, the process request graph, and the wait-for graph.
* isDeadlocked(): This method detects deadlocks by finding a cycle in the wait-for graph.
* unionWaitforGraph(int[][] waitfor1, int[][] waitfor2): This method merges two wait-for graphs.

The main method of the program creates two HRMDD objects, one for each site. The setAllocation() and setRequest() methods are used to set the allocation and request of resources for each process. The convertWaitforGraph() method is used to convert the resource allocation graph to a wait-for graph. The printGraph() method is used to print the resource allocation graph, the process request graph, and the wait-for graph. The isDeadlocked() method is used to detect deadlocks. The unionWaitforGraph() method is used to merge two wait-for graphs.

The program outputs the following:

* The resource allocation graph for each site.
* The process request graph for each site.
* The wait-for graph for each site.
* Whether or not a deadlock has been detected.

**P0**

**R0**

**P1**

**R1**

**P2**

**R2**

**P0**

**P0**

**R0**

**P1**

**R1**

**R0**

Example:

**SITE 1**  **SITE 2**

Output ||-  
  
\*\*\*\*\*\*\*\*\*\*\*\*SITE 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Resource Status Graph:

R0 R1 R2

P0: 1 0 0

P1: 0 1 0

P2: 0 0 1

P3: 0 0 0

Process Status Graph:

P0 P1 P2 P3

R0: 0 0 0 1

R1: 1 0 1 0

R2: 0 1 0 0

Wait-for graph for SITE 1:

P0 P1 P2 P3

P0: 0 1 0 0

P1: 0 0 1 0

P2: 0 1 0 0

P3: 1 0 0 0

Deadlock detected

\*\*\*\*\*\*\*\*\*\*\*\*SITE 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Resource Status Graph:

R0 R1 R2

P0: 1 0 0

P1: 0 1 0

P2: 0 0 0

Process Status Graph:

P0 P1 P2 P3

R0: 0 0 0

R1: 1 0 0

R2: 0 1 0

Wait-for graph for SITE 2:

P0 P1 P2 P3

P0: 0 1 0

P1: 0 0 0

P2: 0 0 0

No deadlock detected

Final Wait-for Graph:

P0 -> P1

P1 -> P2

P2 -> P1

<-|| Deadlock detected ||->