Objective: To design a CPU scheduler for simulatin a feq CPU Scheduling policies.

```
import java.util.*;

class Process {
    int pid, arrival, burst, remaining;
    int start = -1, completion, waiting, turnaround, response;
}

public class CPUScheduling {
    static void fcfs(Process[] p, int n) {
        int time = 0;
        double awt = 0, att = 0, art = 0;

        System.out.println("\nGantt Chart:");
        System.out.print("|");

        for (int i = 0; i < n; i++) {
            if (time < p[i].arrival)
                time = p[i].arrival;

            p[i].start = time;
            p[i].response = p[i].start - p[i].arrival;

            time += p[i].burst;

            p[i].completion = time;
            p[i].turnaround = p[i].completion - p[i].arrival;
            p[i].waiting = p[i].turnaround - p[i].burst;

            System.out.print(" P" + p[i].pid + " |");

            awt += p[i].waiting;
            att += p[i].turnaround;
            art += p[i].response;
        }

        System.out.println("\nAverage Waiting Time = " + (awt / n));
        System.out.println("Average Turnaround Time = " + (att / n));
        System.out.println("Average Response Time = " + (art / n));
    }
```

```
static void roundRobin(Process[] p, int n, int tq) {
    int time = 0, completed = 0;
    double awt = 0, att = 0, art = 0;

    System.out.println("\nGantt Chart:");
    System.out.print("|");

    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= time && p[i].remaining > 0) {
                if (p[i].start == -1) {
                    p[i].start = time;
                    p[i].response = p[i].start - p[i].arrival;
                }

                System.out.print(" P" + p[i].pid + " |");

                if (p[i].remaining > tq) {
                    time += tq;
                    p[i].remaining -= tq;

                } else {
                    time += p[i].remaining;
                    p[i].remaining = 0;
                    p[i].completion = time;
                    p[i].turnaround = p[i].completion - p[i].arrival;
                    p[i].waiting = p[i].turnaround - p[i].burst;
                    awt += p[i].waiting;
                    att += p[i].turnaround;
                    art += p[i].response;
                    completed++;
                }
            }
        }
    }

    System.out.println("\nAverage Waiting Time = " + (awt / n));
    System.out.println("Average Turnaround Time = " + (att / n));
    System.out.println("Average Response Time = " + (art / n));

}
```

```
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Process[] p = new Process[5];

        int[] arrival = {0, 2, 4, 6, 8};
        int[] burst = {3, 6, 4, 5, 2};

        for (int i = 0; i < 5; i++) {
            p[i] = new Process();
            p[i].pid = i + 1;
            p[i].arrival = arrival[i];
            p[i].burst = burst[i];
            p[i].remaining = burst[i];
        }

        while (true) {
            System.out.println("\n1. FCFS");
            System.out.println("2. Round Robin");
            System.out.println("5. Exit");
            System.out.print("Enter choice: ");

            int choice = sc.nextInt();

            switch (choice) {
                case 1:
                    fcfs(p, 5);
                    break;
                case 2:
                    System.out.print("Enter Time Quantum: ");
                    int tq = sc.nextInt();
                    roundRobin(p, 5, tq);
                    break;
                case 5:
                    System.exit(0);
                default:
                    System.out.println("Invalid choice");
            }

        }

    }

}
```

Objective: Implementation of Banker's algorithm to avoid deadlock

```
public class BankersAlgorithm {
   public static void main(String[] args) {
      int n = 5, m = 4;
      int[][] alloc = {{0, 0, 1, 2}, {2, 0, 0, 0}, {0, 0, 3, 4}, {2, 3, 5, 4}, {0, 3, 3, 2}};
      int[][] max = {0, 0, 1, 2}, {2, 7, 5, 0}, {6, 6, 5, 6}, {4, 3, 5, 6}, {0, 6, 5, 2}};
      int[] avail = {0, 1, 0, 4};
      int[][] need = new int[n][m];

      for (int i = 0; i < n; i++)
         for (int j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
      System.out.println("Need Matrix:");

      for (int i = 0; i < n; i++) {
         for (int j = 0; j < m; j++)
            System.out.print(need[i][j] + " ");
         System.out.println();

      }

      boolean[] finish = new boolean[n];
      int[] safeSeq = new int[n];
      int count = 0;

      while (count < n) {
         boolean found = false;
         for (int i = 0; i < n; i++) {
            if (!finish[i]) {
               int j;
               for (j = 0; j < m; j++)
                  if (need[i][j] > avail[j])
                     break;
               if (j == m) {
                  for (int k = 0; k < m; k++)
                     avail[k] += alloc[i][k];
                  safeSeq[count++] = i;
                  finish[i] = true;
                  found = true;
               }
            }
         }
         if (!found) break;
      }

      if (count == n) {
         System.out.print("System is in safe state.\nSafe Sequence: ");
         for (int i = 0; i < n; i++)
            System.out.print("P" + (safeSeq[i] + 1) + " ");
      } else {
         System.out.println("System is not in safe state.");
      }
   }
}
```