**Practical No. 3**

**Aim:** To implement Byzantine Algorithm

**Theory:**

In [fault-tolerant computer systems](https://en.wikipedia.org/wiki/Fault-tolerant_computer_systems), and in particular [distributed computing](https://en.wikipedia.org/wiki/Distributed_computing) systems, Byzantine fault tolerance (BFT) is the characteristic of a system that tolerates the class of failures known as the [Byzantine Generals' Problem](https://en.wikipedia.org/wiki/Byzantine_fault_tolerance#Byzantine_Generals.27_Problem), which is a generalized version of the [Two Generals' Problem](https://en.wikipedia.org/wiki/Two_Generals%27_Problem) – for which there is an unsolvability proof. The phrases interactive consistency or source congruency have been used to refer to Byzantine fault tolerance, particularly among the members of some early implementation teams. It is also referred to as error avalanche, Byzantine agreement problem, Byzantine generals problem and Byzantine failure. Byzantine failures are considered the most general and most difficult class of failures among the [failure modes](https://en.wikipedia.org/wiki/Failure_cause). The so-called fail-stop failure mode occupies the simplest end of the spectrum. Whereas fail-stop failure model simply means that the only way to fail is a [node](https://en.wikipedia.org/wiki/Node_(computer_science)) crash, detected by other nodes, Byzantine failures imply no restrictions, which means that the failed node can generate arbitrary data, pretending to be a correct one, which makes fault tolerance difficult.

**Byzantine General’s Problem**

Byzantine refers to the Byzantine Generals' Problem, an agreement problem (described by [Leslie Lamport](https://en.wikipedia.org/wiki/Leslie_Lamport), Robert Shostak and Marshall Pease in their 1982 paper, ["The Byzantine Generals Problem"](http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf))in which a group of generals, each commanding a portion of the [Byzantine army](https://en.wikipedia.org/wiki/Byzantine_army), encircle a city. These generals wish to formulate a plan for attacking the city. In its simplest form, the generals must only decide whether to attack or retreat. Some generals may prefer to attack, while others prefer to retreat. The important thing is that every general agrees on a common decision, for a half-hearted attack by a few generals would become a [rout](https://en.wikipedia.org/wiki/Rout) and be worse than a coordinated attack or a coordinated retreat.

The problem is complicated by the presence of traitorous generals who may not only cast a vote for a suboptimal strategy, they may do so selectively. For instance, if nine generals are voting, four of whom support attacking while four others are in favor of retreat, the ninth general may send a vote of retreat to those generals in favor of retreat, and a vote of attack to the rest. Those who received a retreat vote from the ninth general will retreat, while the rest will attack (which may not go well for the attackers). The problem is complicated further by the generals being physically separated and having to send their votes via messengers who may fail to deliver votes or may forge false votes.

Byzantine fault tolerance can be achieved if the loyal (non-faulty) generals have a unanimous agreement on their strategy. Note that there can be a default vote value given to missing messages. For example, missing messages can be given the value <Null>. Further, if the agreement is that the <Null> votes are in the majority, a pre-assigned default strategy can be used (e.g., retreat).

The typical mapping of this story onto computer systems is that the computers are the generals and their digital communication system links are the messengers.

**Implementation:**

1. Write a java program containing a logical error which won’t show during compilation, but is technically wrong as it’ll give wrong results.
2. Compile the program.
3. Execute it. The difference in result due to the logical error will be noticed easily. Note the difference.

**Program:**

import java.io.\*;

public class GGByzantine

{

public static void main(String a[]) throws IOException

{

double s1,s2,s3,s4,s5,per;

int num,choice=10,fact;

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

while(choice!=0)

{

System.out.println("Enter your choice");

System.out.println("0. Exit");

System.out.println("1. Calculate result");

System.out.println("2. Calculate factorial");

choice = Integer.parseInt(br.readLine());

switch(choice)

{

case 0:

System.out.println("The program will now exit in 2 seconds");

try

{

Thread.sleep(2000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

break;

case 1:

System.out.println("You chose to calculate result");

System.out.println("Enter the marks of all subjects of a student(out of 100) separated by end of line: ");

s1 = Double.parseDouble(br.readLine());

s2 = Double.parseDouble(br.readLine());

s3 = Double.parseDouble(br.readLine());

s4 = Double.parseDouble(br.readLine());

s5 = Double.parseDouble(br.readLine());

per=result(s1,s2,s3,s4,s5);

System.out.println("The student has scored "+per+"%");

break;

case 2:

System.out.println("You chose to calculate factorial");

System.out.println("Enter the number to calculate it's factorial");

num = Integer.parseInt(br.readLine());

fact = factorial(num);

System.out.println("Factorial of "+num+" is "+fact);

break;

default:

System.out.println("Please enter a proper choice");

}

}

}

static double result(double s1,double s2,double s3,double s4,double s5)

{

double total,res;

total = s1 + s2 + s3 + s4 + s5;

res = total\*500/100; //logical error, actual formula: res = total\*100/500

return res;

}

static int factorial(int num)

{

int fact=0,i; //logical error, fact should be initialised to 1, not 0

for(i=1;i<=num;i++)

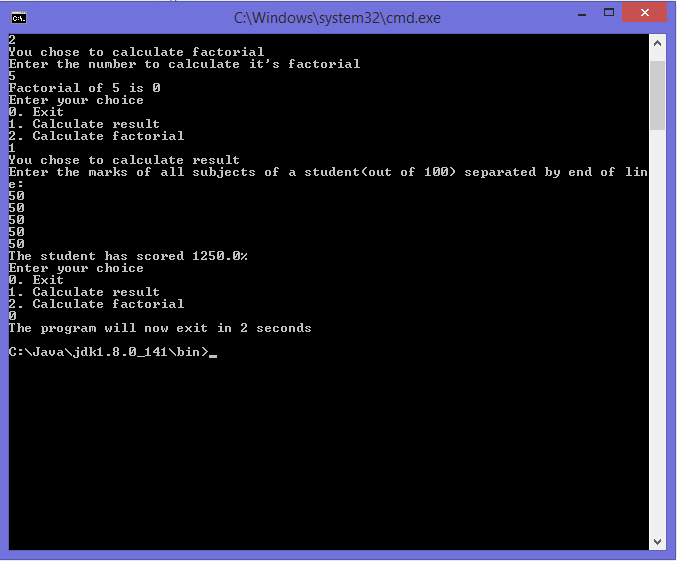
fact=fact\*i;

return fact;

}

}

**Output:**

****

**Conclusion:** The basic concept of Byzantine Fault tolerance has been understood and a sample program depicting the same has been made. The logical error in the program caused wrong output and it has been noted.