(1)

Multithreading allows the execution of multiple parts of a program at the same time. These parts are known as threads and are lightweight processes available within the process. So multithreading leads to maximum utilization of the CPU by multitasking**.**

Some of the benefits of multithreaded programming are given as follows –

1. Resource Sharing: All the threads of a process share its resources such as memory, data, files etc. A single application can have different threads within the same address space using resource sharing.

2. Responsiveness: Program responsiveness allows a program to run even if part of it is blocked using multithreading. This can also be done if the process is performing a lengthy operation. For example - A web browser with multithreading can use one thread for user contact and another for image loading at the same time.

3. Utilization of Multiprocessor Architecture: In a multiprocessor architecture, each thread can run on a different processor in parallel using multithreading. This increases concurrency of the system. This is in direct contrast to a single processor system, where only one process or thread can run on a processor at a time.

4.Economy: It is more economical to use threads as they share the process resources. Comparatively, it is more expensive and time-consuming to create processes as they require more memory and resources. The overhead for process creation and management is much higher than thread creation and management.

(2)

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of our program, because it is the one that is executed when our program begins.

Properties :

1. It is the thread from which other “child” threads will be spawned.

2. Often, it must be the last thread to finish execution because it performs various shutdown actions.

The main thread is created automatically when our program is started. To control it we must obtain a reference to it. This can be done by calling the method currentThread( ) which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

When a Java thread is created, it inherits its priority from the thread that created it. You can also modify a thread's priority at any time after its creation using the setPriority method. Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class). The higher the integer, the higher the priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. Only when that thread stops, yields, or becomes not runnable for some

reason will a lower priority thread start executing. If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion.

The chosen thread will run until one of the following conditions is true:

* A higher priority thread becomes runnable.

* It yields, or its run method exits.

* On systems that support time-slicing, its time allotment has expired.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY

2. public static int NORM_PRIORITY

3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

(3)

```
// Java program to illustrate
// sleep() and join() method
import java.lang.*;
public class SleepDemo implements Runnable
{
        Thread t;
        public void run()
        {
                for (int i = 0; i < 3; i++)
                {
                System.out.println(Thread.currentThread().getName()   + " " + i);
                        try
                        {
                                Thread.sleep(10);                // thread to sleep for 10 milliseconds
                        }
                        catch (Exception e)
                        {
                                System.out.println(e);
```

```
                }
            }
        }
        public static void main(String[] aaa) throws Exception
        {
            Thread t = new Thread(new SleepDemo());
            t.start();                                  // call run() function
            Thread t2 = new Thread(new SleepDemo());
            t2.join(1000);                              // Waits for 1000ms this thread to die.
            t2.start();                                 // call run() function
            System.out.println("Joining after 1000"+ " mili seconds: ");
            System.out.println("Current thread: " + t2.getName());
            System.out.println("Is alive? " + t2.isAlive());          // Checks if this thread is alive
        }
}
```

Input:

Output:

```
Joining after 1000 mili seconds:
Current thread: Thread-1
Thread-0 0
Thread-1 0
Thread-0 1
Thread-1 1
Is alive? true
Thread-0 2
Thread-1 2
```