Java Multithreading

1. What do you mean by Multithreading? Why is it important?

Ans: Multithreading means multiple threads and is considered one of the most important features of Java. As the name suggests, it is the ability of a CPU to execute multiple threads independently at the same time but share the process resources simultaneously. Its main purpose is to provide simultaneous execution of multiple threads to utilize the CPU time as much as possible. It is a Java feature where one can subdivide the specific program into two or more thread to make the execution of the program fast and easy.

2. What are the benefits of using Multithreading?

Ans: There are various benefits of multithreading as given below:

- Allow the program to run continuously even if a part of it is blocked.
- Improve performance as compared to traditional parallel programs that use multiple processes.
- Allows to write effective programs that utilize maximum CPU time.
- Improves the responsiveness of complex applications or programs.
- Increase use of CPU resources and reduce costs of maintenance.
- Saves time and parallelism tasks.
- If an exception occurs in a single thread, it will not affect other threads as threads are independent.
- Less resource intensive than executing multiple processes at the same time.

3. What is Thread in Java?

Ans: A Thread is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the Thread concept in Java. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.

Another benefit of using thread is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as Multithreading.

4. **What are the two ways of implementing thread in Java?**

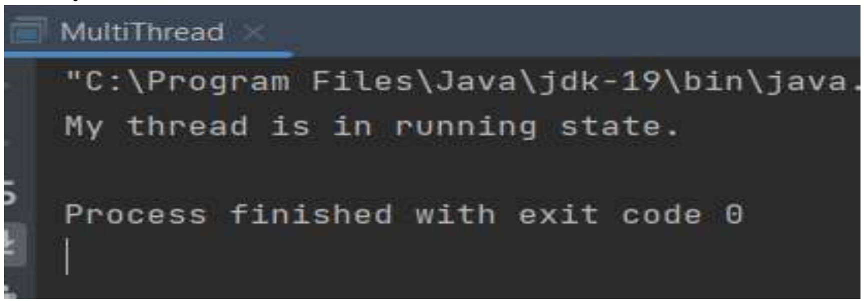
Ans: There are basically two ways of implementing thread in java as given bellow.

1. **Extending the Thread class-**
   Example:

```
package Thread;

class MultiThread extends Thread
{
    public void run()
    {
        System.out.println("My thread is in running state.");
    }
    public static void main(String args[])
    {
        Multithread obj=new Multithread();
        obj.start();
    }
}
```

Output:



```
"C:\Program Files\Java\jdk-19\bin\java.
My thread is in running state.

Process finished with exit code 0
```

2. **Implementing Runnable Interface in java**
   Example:

```
package Thread;

public class MultiThreadEx2 implements Runnable{
    public void run()
    {
        System.out.println("My thread is in running state.");
    }
    public static void main(String args[])
    {
        MultiThreadEx2 obj=new MultiThreadEx2();
        Thread Obj =new Thread(obj);
        Obj.start();
    }
}
```

Output: My thread is in running state.

5. What's the difference between thread and process?

1. Process means any program is in execution.

   Thread means a segment of a process.

2. The process takes more time to terminate.

   The thread takes less time to terminate.

3. The Process takes more time for creation.

   Thread takes less time for creation.

4. The Process also takes more time for context switching.

   Thread takes less time for context switching.

5. The process is less efficient in terms of communication.

   Thread is more efficient in terms of communication.

6. Multiprogramming holds the concepts of multi-process.

   We don't need multi programs in action for multiple threads because a single process consists of multiple threads.

7. The process is isolated.

   Threads share memory.

8. The process is called the heavyweight process.

   A Thread is lightweight as each thread in a process shares code, data, and resources.

9. Process switching uses an interface in an operating system.

   Thread switching does not require calling an operating system and causes an interrupt to the kernel.

10. If one process is blocked then it will not affect the execution of other processes

    But If a user-level thread is blocked, then all other user-level threads are blocked.

11. The process has its own Process Control Block, Stack, and Address Space.

    Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.

12. Changes to the parent process do not affect child processes.

    Since all threads of the same process share address space and other resources so any changes to the main thread may affect the behaviour of the other threads of the process.

13. A system call is involved in it.

    But in Thread No system call is involved, it is created using APIs.

14. The process does not share data with each other.

    Threads share data with each other.

6. How can we create daemon threads?

Ans: We can create daemon threads in java using the thread class setDaemon(true). It is used to mark the current thread as daemon thread or user thread. isDaemon() method is generally used to check whether the current thread is daemon or not. If the thread is a daemon, it will return true otherwise it returns false.

Example:

Program to illustrate the use of setDaemon() and isDaemon() method.

```
package Thread;

public class DaemonThread extends Thread
{
   public DaemonThread(String name){
       super(name);
   }
   public void run()
   {
   // Checking whether the thread is Daemon or not
       if(Thread.currentThread().isDaemon())
       {
           System.out.println(getName() + " is Daemon thread");
       }
       else
       {
           System.out.println(getName() + " is User thread");
       }
   }
   public static void main(String[] args)
   {
       DaemonThread t1 = new DaemonThread("t1");
       DaemonThread t2 = new DaemonThread("t2");
       DaemonThread t3 = new DaemonThread("t3");
   // Setting user thread t1 to Daemon
       t1.setDaemon(true);
   // starting first 2 threads
       t1.start();
       t2.start();
   // Setting user thread t3 to Daemon
       t3.setDaemon(true);
       t3.start();
     }
}
```

Output:

```
t3 is Daemon thread

t1 is Daemon thread

t2 is User thread
```

7. What are the wait() and sleep() methods?

Ans: wait(): As the name suggests, it is a non-static method that causes the current thread to wait and go to sleep until some other threads call the notify () or notifyAll() method for the object's monitor (lock). It simply releases the lock and is mostly used for inter-thread communication. It is defined in the object class, and should only be called from a synchronized context.

Example:

```
synchronized(monitor)

{

monitor.wait()  Here Lock Is Released By Current Thread

}
```

sleep(): As the name suggests, it is a static method that pauses or stops the execution of the current thread for some specified period. It doesn't release the lock while waiting and is mostly used to introduce pause on execution. It is defined in thread class, and no need to call from a synchronized context.

Example:

```
synchronized(monitor)

{

Thread.sleep(1000);  Here Lock Is Held By The Current Thread

//after 1000 milliseconds, current thread will wake up, or after we call that
is interrupt() method

}
```