B. Tech. SE Semester-III

Object Oriented Programming with Java

Unit-5

Multithreaded Programming, IO Programming and Class Modeling

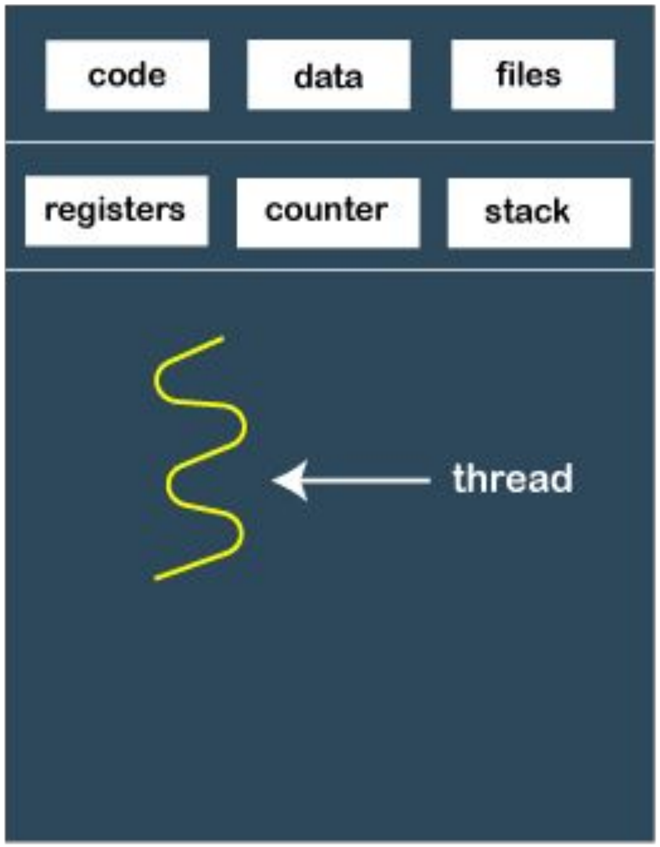
Subject Teacher : Dr. K. V. Metre

Contents :

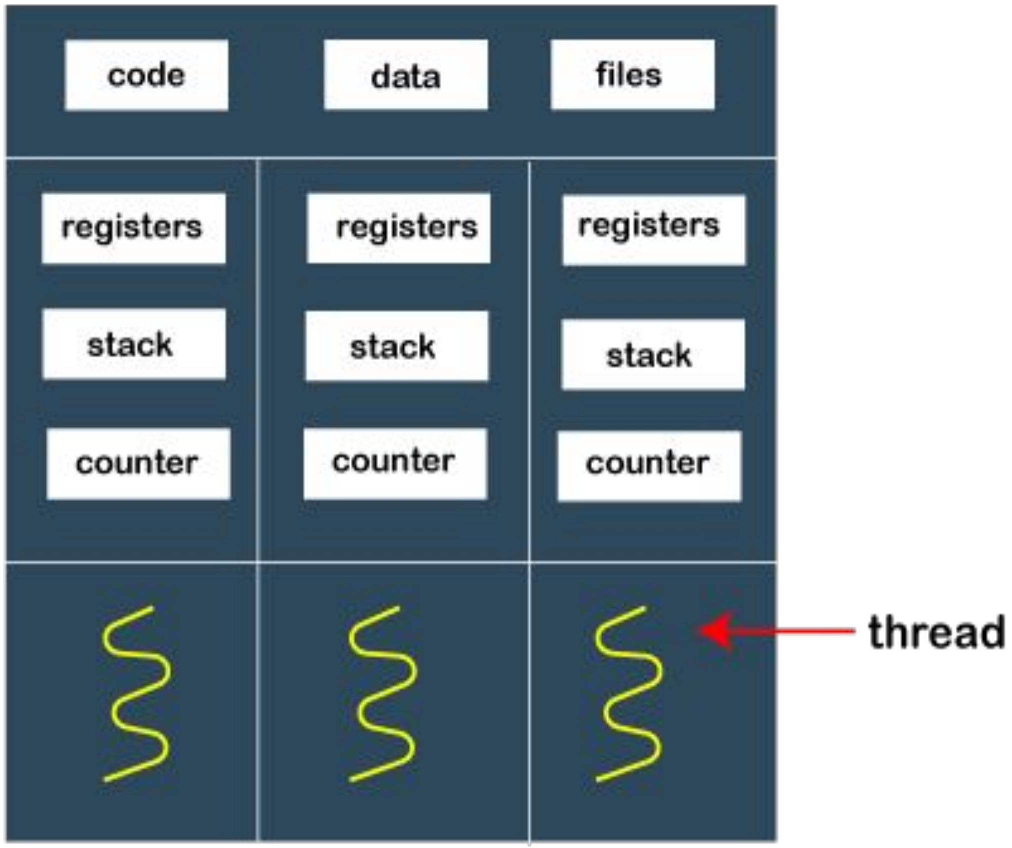
- **Multithreaded Programming:**
- **IO Programming:**
- **Class Modelling:**

Introduction to Multithreading

- Multithreading allows a program to split into multiple subprograms called threads that can run concurrently.
- **Multithreading in Java** is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.



Single-threaded process



Multi-threaded process

- Multithreading to multiprocessing is preferred because threads use a shared memory area.
- They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

1) **Process-based Multitasking (Multiprocessing)**
   - Each process has an address in memory., each process allocates a separate memory area.
   - A process is heavyweight.
   - Cost of communication between the process is high.
   - Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) **Thread-based Multitasking (Multithreading)**
   - Threads share the same address space.
   - A thread is lightweight.
   - Cost of communication between the thread is low.
   - Note: At least one process is required for each thread.

Difference between Process and thread

S	Process	Thread
1)	When a program is under execution, then it is known as a process.	A segment of a process is known as thread.
2)	Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fast because they are very lightweight.
3)	It is a heavy weight process.	It is a light weight process.
4)	Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory
5)	New process creation is more time taking as each new process takes all the resources.	A thread needs less time for creation.
6)	Processes can communicate with each other using inter-process communication only	threads can directly communicate with each other as they share the same address space.

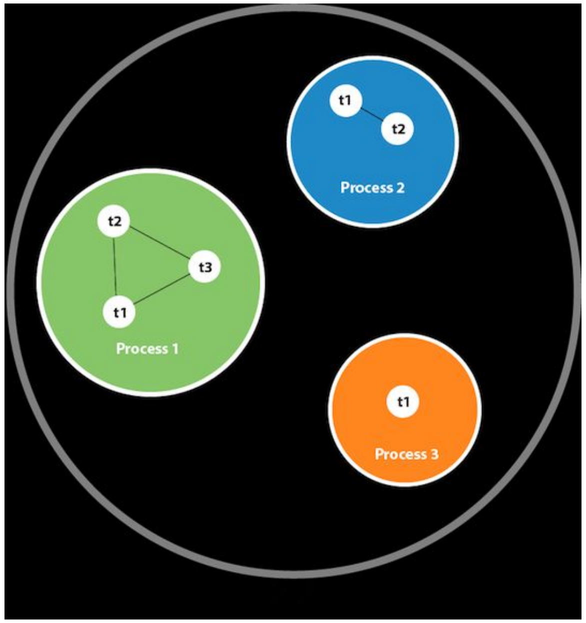
Advantages of Java Multithreading :

- It **doesn't block the user** because threads are independent and you can perform multiple operations concurrently

- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Thread in java

- A thread is a lightweight subprocess, the smallest unit of processing.
- It is a separate path of execution.
- Threads are independent.
- If there occurs exception in one thread, it doesn't affect other threads.
- It uses a shared memory area.



Java Thread class

- Java provides **Thread class** to achieve thread programming.
- Thread class provides constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface.

Java Thread class Methods

SN	Name of method	Decsription
1)	void start()	It is used to start the execution of the thread.
2)	Void run()	It is used to do an action for a thread.
3)	static void sleep(long)	It sleeps a thread for the specified amount of time.
4)	void stop()	It is used to stop the thread.
5)	int getPriority()	It returns the priority of the thread.
6)	void setPriority(int)	It changes the priority of the thread.
7)	long getId()	It returns the id of the thread.
8)	Boolean isAlive()	It tests if the thread is alive.

Java Thread class Methods

- **public Thread.State getState():** returns the state of the thread.
- **public void join():** waits for a thread to die.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.

Creation of a Thread

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().
- **public void run():** is used to perform action for a thread.

Java Thread Example by extending Thread class

```
class MyThread extends Thread{
     public void run(){
      System.out.println("thread is running...");
      }
      public static void main(String args[]){
      MyThread t1=new MyThread();
       t1.start();
 }
}
```

Output :

```
thread is running...");
```

Java Thread Example by implementing Runnable Interface

```
class MyThread implements Runnable{
    public void run(){
        System.out.println("thread is running...");
        }

    public static void main(String args[]){
         MyThread  m1=new  MyThread();
         Thread t1 =new Thread(m1);
         t1.start();
    }
    }
```

Output:

```
thread is running...
```

Life Cycle of a Thread

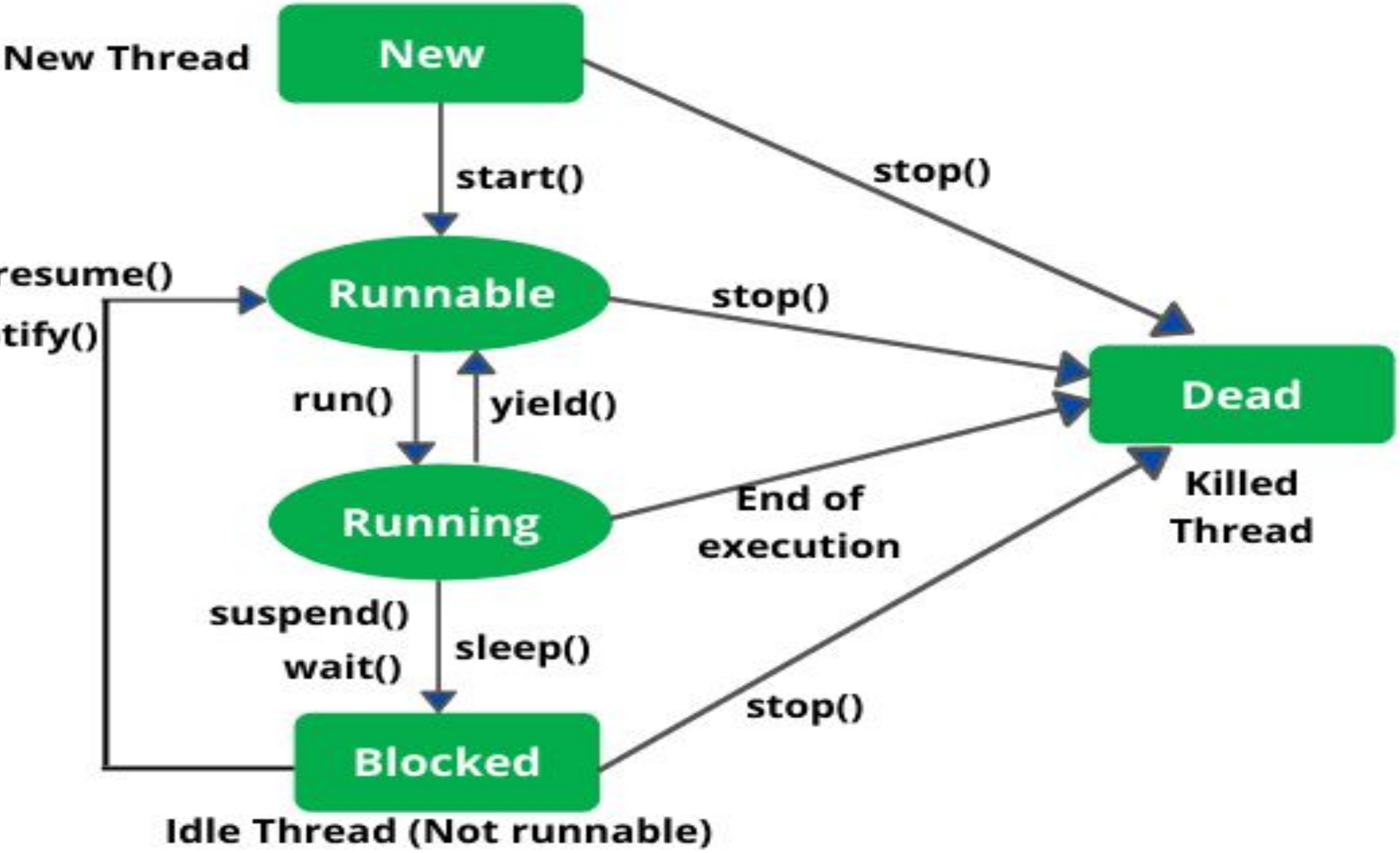


Fig: State Transition Diagram of a Thread

Life Cycle of a Thread

- **New**: When we create an instance of Thread class, the new thread is in the New state. It remains in this state until the program calls the thread's ‘start()’ method.

  MyThread t1 = new Myhread();’
- **Active (Runnable and Running)**: After calling the ‘start()’ method (i.e., t1.start()), the thread moves from the New state to the Runnable state. It's now ready to run and waiting to be selected for execution by the thread scheduler. Once the thread scheduler allocates CPU to the thread, it transitions to the Running state, and its ‘run()’ method starts to execute.
- **Blocked / Waiting / Timed Waiting**: In our ‘run()’ method, when we use t1.sleep(1000); method, which makes the thread sleep for 1 second. This places our thread in a Timed Waiting state. If running state of thread in Java needs to wait for another thread to release a resource, it will move to the Blocked/Waiting state.
- **Terminated/Dead**: Once the ‘run()’ method has completed its execution, the thread moves to the Terminated state OR t1.stop() method called

Thread Priority

- Each thread has a priority. Default value=5
- Priorities are represented by a number between 1 and 10.

3 constants defined in Thread class:

• public static int MIN_PRIORITY	1
• public static int NORM_PRIORITY	5
• public static int MAX_PRIORITY	10

- **public final int getPriority():**

  The java.lang.Thread.getPriority() method returns the priority of the given thread.
- **public final void setPriority(int newPriority):**

  The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority.
- The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

```
import java.lang.*;
public class MyThread extends Thread   {
    public void run()   {
         System.out.println("Inside the run() method");   }
     public static void main(String argvs[])  {
       MyThread  t1 = new  MyThread();
      MyThread t2 = new  MyThread();
      System.out.println("Priority of the thread t1 is : " + t1.getPriority());
       System.out.println("Priority of the thread t2 is : " + t2.getPriority());
          t2.setPriority(3);
        System.out.println("Priority of the thread t1 is : " + t1.getPriority());
System.out.println("Priority of the thread t2 is :" + t2.getPriority());
System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName());
System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
}  }
```

Output :

```
Priority of the thread t1 is : 5
Priority of the thread t2 is : 5
Priority of the thread t1 is : 5
Priority of thread t2 is : 3
Currently Executing The Thread :
main
Priority of the main thread is : 5
```

Thread Synchronization in Java

- Synchronization in Java is the capability to control the access of multiple threads to any shared resource.
- Using Java Synchronization, we will allow only one thread to access the shared resource.

Why use Synchronization?

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Types of Synchronization

- There are two types of synchronization
- Process Synchronization
- Thread Synchronization

Thread Synchronization in Java

Thread Synchronization :

- There are two types of thread synchronization mutual exclusive and inter-thread communication.

Concept of Lock in Java :

- Synchronization is built around an internal entity known as the lock or monitor.
- Every object has a lock associated with it.
- By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
- From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

```
class Table{
void printTable(int n){//method not synchronized
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }
    catch(Exception e){System.out.println(e);}
   } } }
class MyThread1 extends Thread{
      Table t;
      MyThread1(Table t){
             this.t=t;  }
       public void run(){
      t.printTable(5);
            } }
class MyThread2 extends Thread{
        Table t;
        MyThread2(Table t){
           this.t=t;  }
       public void run(){
        t.printTable(100);
         } }
```

```
class TestSynchronization1{
public static void main(String a
rgs[]){
Table obj = new Table();//only
one object
MyThread1 t1=new MyThread
1(obj);
MyThread2 t2=new MyThread
2(obj);
t1.start();
t2.start();
}
}
```

Output

```
5
100
10
200
15
300
20
400
25
500
```

Jnderstanding with static Synchronization

```
class Table {
synchronized static void printTable(int n)
{
for(int i=1;i<=5;i++){
System.out.println(n*i);
  try{
 Thread.sleep(400);
    }
 catch(Exception e){}              }   } }
  class MyThread1 extends Thread
{    public void run(){
   Table.printTable(5);
} }
 class MyThread2 extends Thread
{    public void run(){
Table.printTable(100);
 }
  }
```

```
public class TestSynchronization{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
t1.start();
t2.start();
}
}
```

```
1
2
3
4
5
100
200
300
400
500
```

Inter-Thread communication in Java

- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- It is implemented by following methods of **Object class**:
- wait()
- notify()
- notifyAll()

1) wait() method

- The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

  public final void wait()

2) notify() method

- The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

public final void notify()

3) notifyAll() method

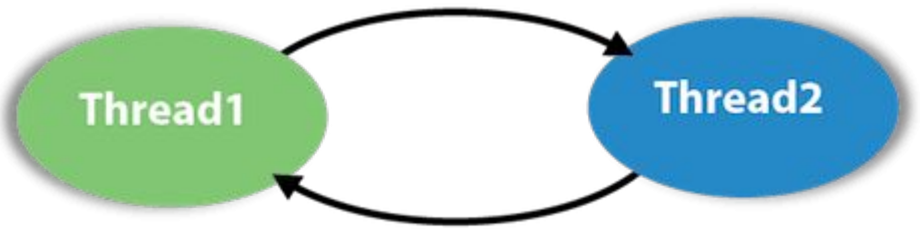
- Wakes up all threads that are waiting on this object's monitor.

  Syntax:

  public final void notifyAll()

Deadlock

- Deadlock in Java is a part of multithreading.
- Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.
- Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Deadlock

- Thread 1 locks A, waits for B
- Thread 2 locks B, waits for C
- Thread 3 locks C, waits for D
- Thread 4 locks D, waits for A

```
Runnable b2 = new Runnable() {
  public void run() {
    synchronized (b) {
      // Thread-2 have resource2 but need resource1 also
      synchronized (a) {
        System.out.println("In block 2");
```

Daemon Thread

- **Daemon thread in Java** is a service provider thread that provides services to the user thread.
- Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
- There are many java daemon threads running automatically e.g. finalizer etc.

Points to remember for Daemon Thread in Java :

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Daemon Thread

Methods for Java Daemon thread by Thread class

- The java.lang.Thread class provides two methods for java daemon thread

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current is daemon.

```
  public void run(){

if(Thread.currentThread().isDaemon()){//checking for daemon thread

          System.out.println("daemon thread work");
          }
      else{
            System.out.println("user thread work");
           }             }
   public static void main(String[] args){

TestDaemonThread t1=new TestDaemonThread();//creating thread
          TestDaemonThread t2=new TestDaemonThread();
          TestDaemonThread t3=new TestDaemonThread();
  t1.setDaemon(true);//now t1 is daemon thread

  t1.start();//starting threads
  t2.start();
  t3.start();
 }  }
```
Output:

```
daemon thread work
user thread work
```