# MultiThreading

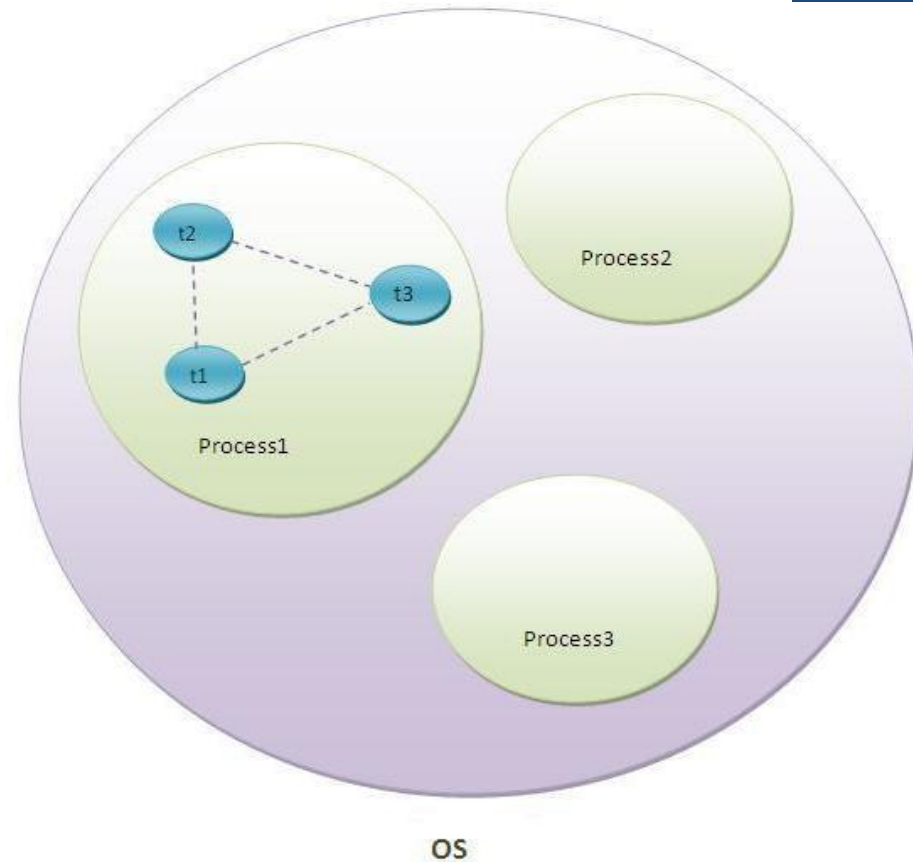**SMITA SANKHE**

**Assistant Professor**

**Department of Computer Engineering**

# What is Thread?

- A thread is a lightweight sub process having an independent path of execution within a program.
- As threads are independent, if exception occurs in one thread, it doesn't affect other threads.
- All threads shares a common memory area.



In this figure, thread is executed inside the process. **There can be multiple processes inside the OS and one process can have multiple threads.**

# Multitasking

- Multitasking is a process of executing multiple tasks simultaneously. Multitasking  is performed to utilize the CPU time efficiently.

- Multitasking can be achieved by two ways:
    1) **Process-based Multitasking (Multiprocessing)**
    2) **Thread-based Multitasking (Multithreading)**

# Thread Lifecycle
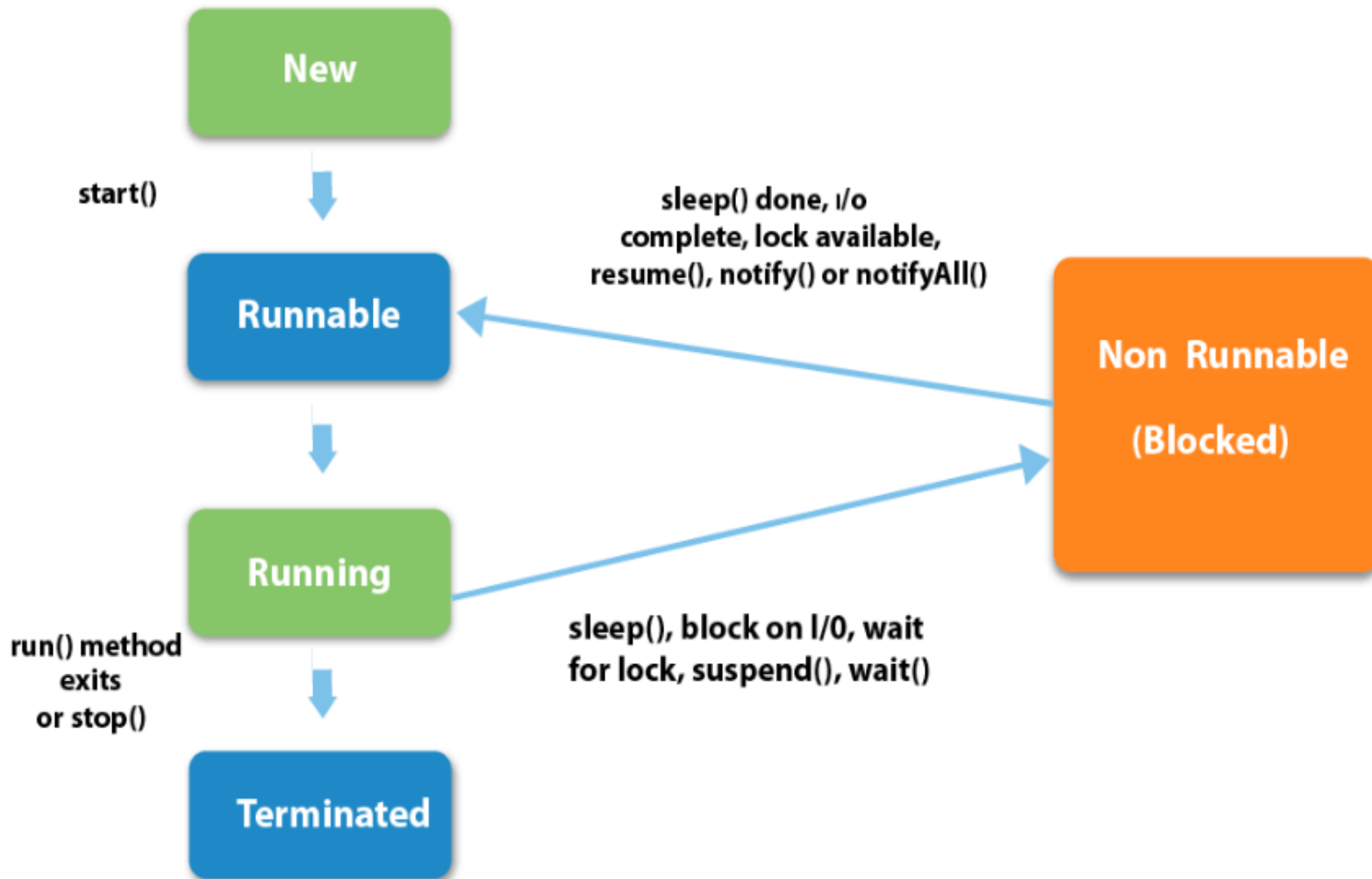
# Thread Lifecycle (contd..)

**1) New**

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

**2) Runnable**

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

**3) Running**

The thread is in running state if the thread scheduler has selected it.

**4) Non-Runnable (Blocked)**

This is the state when the thread is still alive, but is currently not eligible to run.

**5) Terminated**

A thread is in terminated or dead state when its run() method exits.

# Creating Threads

- You can create a thread by instantiating an object of type **Thread**.

- Java defines two ways in which this can be accomplished:
  - You can implement the **Runnable** interface.
  - You can extend the **Thread** class, itself.

# Constructors of Thread class

- Thread()

- Thread(String name)

- Thread(Runnable r)

- Thread(Runnable r,String name)

# Methods of Thread class

1.  **public void run():**

    is used to perform action for a thread.

2.  **public void start():**

    starts the execution of the thread. JVM calls the run() method on the thread

3.  **public void sleep(long milliseconds):**

    Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds

4.  **public void join():**

    waits for a thread to die

5.  **public void join(long miliseconds):**

    waits for a thread to die for the specified milliseconds

# Methods of Thread class (contd..)

**6. public int getPriority():**

        returns the priority of the thread

**7. public int setPriority(int priority):**

        changes the priority of the thread.

**8. public String getName():**

        returns the name of the thread.

**9. public void setName(String name):**

        changes the name of the thread.

**10. public Thread currentThread():**

        returns the reference of currently executing thread.

**11. public int getId():**

        returns the id of the thread.

# Methods of Thread class (contd..)

**12. public Thread.State getState():**

       returns the state of the thread

**13. public boolean isAlive():**

       tests if the thread is alive

**14. public void yield():**

       causes the currently executing thread object to temporarily pause and allow other threads to execute

**15. public void suspend():**

       is used to suspend the thread(depricated)

**16. public void resume():**

       is used to resume the suspended thread(depricated).

**17. public void stop():**

       is used to stop the thread(depricated).

# Methods of Thread class (contd..)

**18. public boolean isDaemon():**

       tests if the thread is a daemon thread

**19. public void setDaemon(boolean b):**

       marks the thread as daemon or user thread.

**20. public void interrupt():**

       interrupts the thread.

**21. public boolean isInterrupted():**

       tests if the thread has been interrupted.

**22. public static boolean interrupted():**

       tests if the current thread has been interrupted.

# 1. Implementing Runnable

- The easiest way to create a thread is to create a class that implements the **Runnable** interface.

- To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

<p style="text-align:center;color:red;"><strong>public void run( )</strong></p>

- **run( )** can call other methods, use other classes, and declare variables, just like the main thread

- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**.

- The **start( )** method is shown here:

<p style="text-align:center;color:red;"><strong>void start( )</strong></p>

**If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitly create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.**

# Example

```java
class Multi3 implements Runnable{
    public void run(){
    System.out.println("thread is running...");
  }


    public static void main(String args[]){
    Multi3 m1=new Multi3();
    Thread t1 =new Thread(m1);
    t1.start();
    }
}
```

**Output:**
thread is running

# 2. Extending Thread

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.

- The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

# Example

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

# Thread Scheduler

- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- Only one thread at a time can run in a single process.
- The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

i.    **Preemptive scheduling**

   highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence

ii.    **Time slicing**

   a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# Sleep Method in Thread:

- In some scenarios you would like a thread to stop executing the code for a period of time and then start again.

- *Sleep method* in Thread tells the currently executing thread to sleep for specified amount of time in *MilliSeconds*.

- It can throw InterruptedException . So it should be embedded in the try catch block

- There is no guarantee that the thread will go to **Sleep state** the moment it is executed and no guarantee that the thread will sleep for specified amount of time. The thread scheduler can wake it up any time.

- Once the thread completes or out of its sleep state, it can move to Running or Runnable state.

# Example: sleep method

```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
   try{Thread.sleep(500);}
    catch(InterruptedException e)
         {System.out.println(e);}
   System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();

  t1.start();
  t2.start();
 } }
```

Output:
1
1
2
2
3
3
4
4

# Example: 2

```
class TestCallRun2 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
   try{Thread.sleep(500);}catch(InterruptedException e){System.out.println
(e);}
   System.out.println(i);
  }
 }
 public static void main(String args[]){
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();

  t1.run();
  t2.run();
 }
}
```

**Output:**
1
2
3
4
5
1
2
3
4
5

Note: normal object not thread object

# join() method

- waits for a thread to die.

In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

- **<span style="color:red">Syntax:</span>**

public void join()throws InterruptedException

public void join(long milliseconds)throws InterruptedException

# Example:

```
class TestJoinMethod1 extends Thread{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e){System.out.println(e);}
  System.out.println(i);
  }
 }
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();
 TestJoinMethod1 t2=new TestJoinMethod1();
 TestJoinMethod1 t3=new TestJoinMethod1();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

Output:

1
2
3
4
5
1
1
2
2
3
3
4
4
5
5

# Using isAlive( ) and join( )

- In the preceding examples to allow main thread to finish last , is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread.

- However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended?

- Fortunately, **Thread** provides a means by which you can answer this question.

# Using isAlive( ) and join( )

- Two ways exist to determine whether a thread has finished.
-  First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

<div align="center">

**final boolean isAlive( )**

</div>

- The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.
- While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

<div align="center">

**final void join( ) throws InterruptedException**

</div>

- This method waits until the thread on which it is called terminates.

# Example:

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                Thread.sleep(500);
                }
        catch(InterruptedException ie)
        {
                // do something
        }
        System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();
                t2.start();
                System.out.println(t1.isAlive());
                System.out.println(t2.isAlive());
        }
}
```

**Output:**

r1

true

true

r1

r2

r2