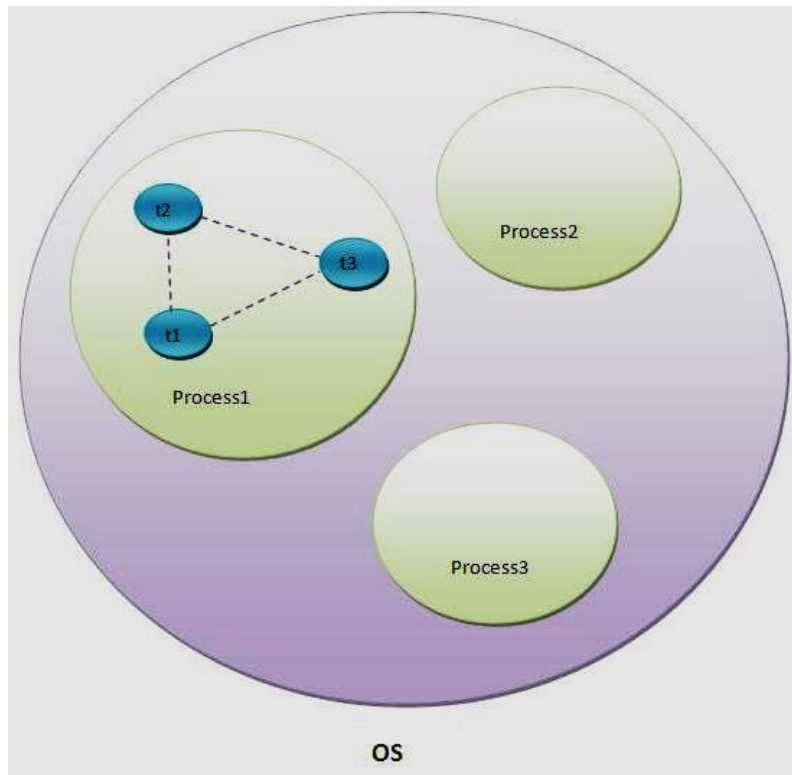


## CHAPTER - 3

### MULTITHREADING

#### THREAD:

A thread is a lightweight sub process, a 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 shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

#### MULTITHREADING:

**Multithreading in java** is a process of executing multiple threads simultaneously. Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process. Java Multithreading is mostly used in games, animation etc.

## Advantages of Java Multithreading:

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
2. We **can perform many operations together so it saves time**.
3. Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

## MULTITASKING:

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

### **1. Process-based Multitasking (Multiprocessing):**

- ❖ Each process have its own address in memory i.e. each process allocates separate memory area.
- ❖ Process is heavyweight.
- ❖ Cost of communication between the processes is high.
- ❖ Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

### **2. Thread-based Multitasking (Multithreading):**

- ❖ Threads share the same address space.
- ❖ Thread is lightweight.
- ❖ Cost of communication between the thread is low.

## THREAD LIFE CYCLE:

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state. But for better understanding the threads are explained in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

### **1. New:**

The thread is in new state if we create an instance of Thread class but before the invocation of start() method. It is also referred to as a **born thread**.

### **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:**

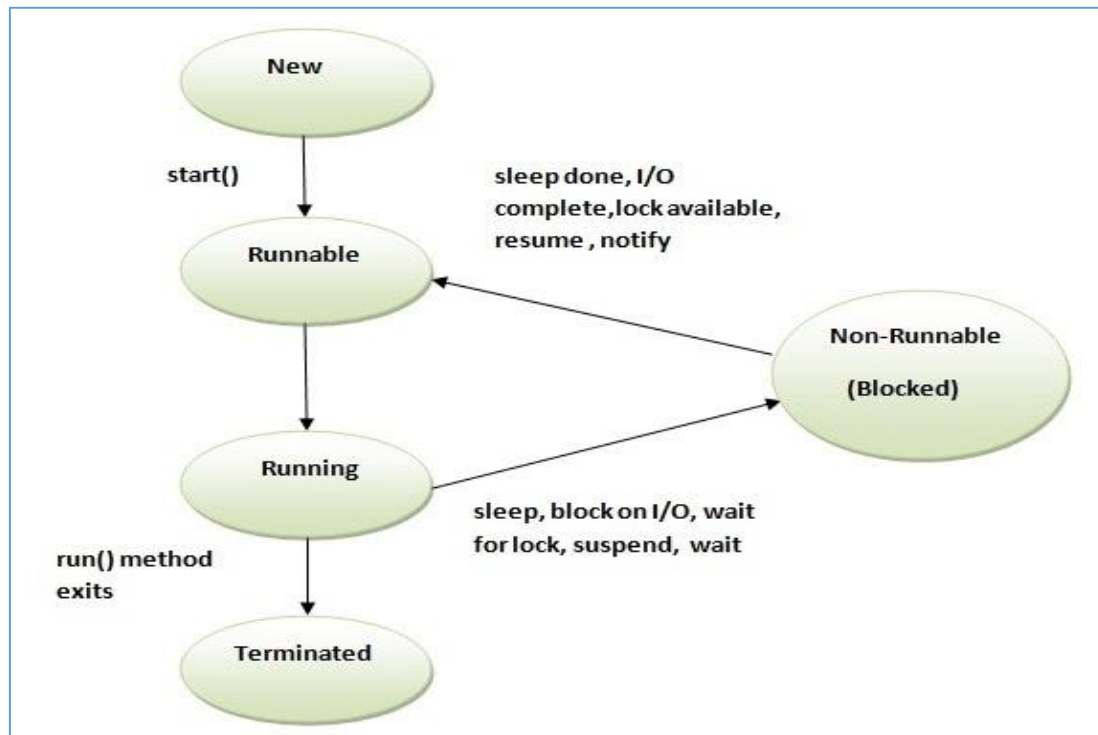
The thread is in running state if the thread scheduler has selected it.

### **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.



*Fig: Life Cycle of Thread*

## HOW TO CREATE THREAD:

*There are two ways to create a thread:*

### **1. Create a Thread by Implementing a Runnable Interface:**

If our class is intended to be executed as a thread then we can achieve this by implementing a Runnable interface. We need to follow three basic steps:

#### **STEP 1:**

As a first step, we need to implement a run() method provided by a Runnable interface. This method provides an entry point for the thread and we put our complete business logic inside this method. Following is a simple syntax of the run() method:

```
public void run( )
```

#### **STEP 2:**

As a second step, we will instantiate a Thread object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, **threadObj** is an instance of a class that implements the Runnable interface and **threadName** is the name given to the new thread.

### STEP 3:

Once a Thread object is created, we can start it by calling start() method, which executes a call to run() method. Following is a simple syntax of start() method:

```
void start();
```

#### *Example*

```
class RunnableDemo implements Runnable {
    private Thread t;
    private String threadName;

    RunnableDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }
    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }
    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}

public class TestThread {
    public static void main(String args[]) {
        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();
        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();
    }
}
```

## 2. Create a Thread by Extending a Thread Class

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

### STEP 1:

We need to override run( ) method available in Thread class. This method provides an entry point for the thread and we put our complete business logic inside this method. Following is a simple syntax of run() method:

```
public void run( )
```

### STEP 2:

Once Thread object is created, we can start it by calling start() method, which executes a call to run() method. Following is a simple syntax of start() method:

```
void start( );
```

### *Example*

```
class ThreadDemo extends Thread {
    private Thread t;
    private String threadName;
    ThreadDemo( String name) {
        threadName = name;
        System.out.println("Creating " + threadName );
    }
    public void run() {
        System.out.println("Running " + threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " + threadName + " interrupted.");
        }
        System.out.println("Thread " + threadName + " exiting.");
    }
    public void start () {
        System.out.println("Starting " + threadName );
        if (t == null) {
            t = new Thread (this, threadName);
            t.start ();
        }
    }
}
```

```

public class TestThread {
    public static void main(String args[]) {
        ThreadDemo T1 = new ThreadDemo( "Thread-1");
        T1.start();
        ThreadDemo T2 = new ThreadDemo( "Thread-2");
        T2.start();
    }
}

```

### THREAD METHODS:

Following is the list of important methods available in the 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 miliseconds):** 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.
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.
- 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).
- 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.

**23. public static boolean holdsLock(Object x):** Returns true if the current thread holds the lock on the given Object.

**24. public static void dumpStack():** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

### **PRIORITY OF A THREAD (THREAD PRIORITY):**

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.

### **3 Constants Defined In Thread Class:**

```
public static int MIN_PRIORITY
public static int NORM_PRIORITY
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.

### **Example of Priority of a Thread:**

```
class TestMultiPriority extends Thread{
    public void run(){
        System.out.println("running thread name is:"+Thread.currentThread().getName());
        System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
    }
    public static void main(String args[]){
        TestMultiPriority m1=new TestMultiPriority();
        TestMultiPriority m2=new TestMultiPriority();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

## THREAD SYNCHRONIZATION:

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. Java programming language provides a very handy way of creating threads and synchronizing their task.

### **Example:**

```
import java.util.Scanner;
class Account{
    private int balance;
    public Account(int bal){
        this.balance = bal;
    }
    public boolean isSufficientBalance(int amount){
        if(balance > amount){
            return(true);
        }
        else{
            return(false);
        }
    }
    public void withdraw(int wd){
        balance = balance - wd;
        System.out.println("Withdrawl Amount = " + wd);
        System.out.println("Current Balance = " + balance);
    }
}

class Customer implements Runnable{
    private final Account account;
    private String name;
    public Customer(Account account, String n){
        this.account = account;
        name = n;
    }
    @Override
    public void run(){
        Scanner scan = new Scanner(System.in);
        synchronized(account){
            System.out.println(name + ",Enter amount: ");
            int amount = scan.nextInt();
        }
    }
}
```



```

        if(account.isSufficientBalance(amount)){
            System.out.println(name);
            account.withdraw(amount);
        }
        else{
            System.out.println("Insufficient Balance");
        }
    }
}

public class ThreadExample{
    public static void main(String [] args){
        Account a1 = new Account(1000);
        Customer c1 = new Customer(a1,"Ramesh");
        Customer c2 = new Customer(a1,"Samir");
        Thread t1 = new Thread(c1);
        Thread t2 = new Thread(c2);
        t1.start();
        t2.start();
    }
}

```

➤ **Synchronized Method:**

If we declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

**Example:**

```
class Table{
    synchronized void printTable(int n){//synchronized method
        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);
    }
}

public class TestSynchronization2{
    public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}



```

### ➤ **Synchronized Block:**

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, then we can use synchronized block. If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

### ***Points to remember for Synchronized block***

-  Synchronized block is used to lock an object for any shared resource.
-  Scope of synchronized block is smaller than the method.

### **Syntax:**

```

synchronized (object reference expression) {
    //code block
}

```

### **Example:**

```

class Table{

```

```

void printTable(int n){
    synchronized(this){//synchronized block
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }
    }
}

//end of the method
}

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);
    }
}

public class TestSynchronizedBlock1{
    public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}

```

➤ **Static Synchronization:**

If we make any static method as synchronized, the lock will be on the class not on object.

### ***Problem without static synchronization***

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refer to a common object that has a single lock. But there

can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. I want no interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

**Example:**

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{
    synchronized static void printTable(int n){
        for(int i=1;i<=10;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){}
        }
    }
}

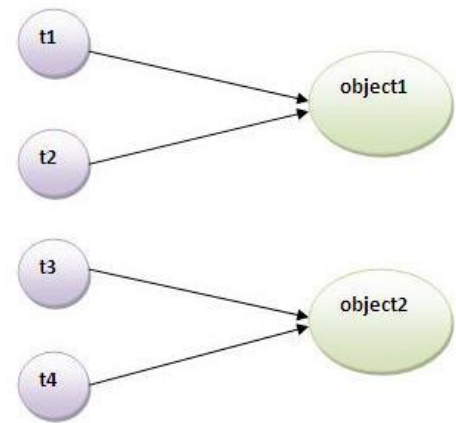
class MyThread1 extends Thread{
    public void run(){
        Table.printTable(1);
    }
}

class MyThread2 extends Thread{
    public void run(){
        Table.printTable(10);
    }
}

class MyThread3 extends Thread{
    public void run(){
        Table.printTable(100);
    }
}

class MyThread4 extends Thread{
    public void run(){
        Table.printTable(1000);
    }
}

public class TestSynchronization4{
    public static void main(String t[]){
        MyThread1 t1=new MyThread1();
        MyThread2 t2=new MyThread2();
        MyThread3 t3=new MyThread3();
        MyThread4 t4=new MyThread4();
        t1.start();
        t2.start();
```



```
t3.start();  
t4.start();  
}  
}
```