# UNIT-IV

**Multithreading and Files: Introduction, Thread Lifecycle, Creation of Threads, Thread Priorities, Thread Synchronization, Communication between Threads. Reading Data from Files and Writing Data to Files, Random Access Files**

**Multithreading** is a process of executing multiple threads simultaneously.

A Thread is a light weight process, a smallest unit of processing. Multithreading helps to increase the throughput. In multithreading, the threads will share a common memory area so the context switching between the threads will take less time

## Advantages of Java Multithreading

1) The users are not blocked because threads are independent, and we can perform multiple operations at same time

2) As such the threads are independent, the other threads won't get affected if one thread meets an exception.

**Threads can be created in two ways:**

1. Extending the Thread class
2. Implementing the Runnable Interface

## Life cycle of a Thread (Thread States)

The life cycle of thread consists of 5 different states.

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

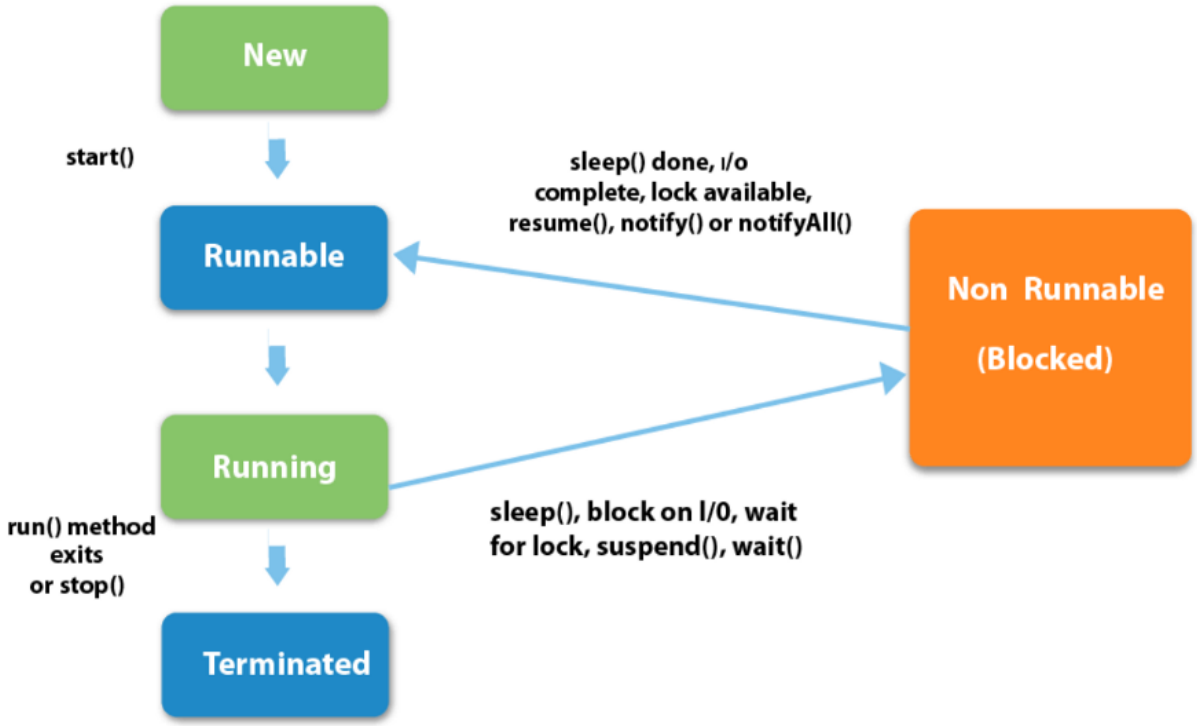
1. **New:** A thread is said to be in new state, if you create an instance of Thread class and start() method is *not yet invoked*.

2. **Runnable:** In this stage, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, when to run the thread.

3. **Running:** When the thread starts executing, then the state is changed to "running" state. The scheduler selects one thread from the thread pool, and it starts executing.

4. **Blocked/Non Runnable :** thread is considered to be in the blocked state when it is suspended, sleeping, or waiting for some time in order to satisfy some condition.

5. **Terminated:** This is the state when the thread is terminated. When the thread completes execution of its "run" method, it goes into the "terminated" state.




## Creating Threads

### 1. Creating Thread by extending the Thread class

We should create a class by extending the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

Example :

Java Thread Example by extending Thread class

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

## Constructors of Thread class:

1. Thread()
2. Thread(String name)
3. Thread(Runnable r)
4. Thread(Runnable r, String name)

## methods of Thread class:

1. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
2. **public void run():** is used to perform action for a 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 milliseconds):** 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 boolean isDaemon():** tests if the thread is a daemon thread.
15. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
16. **public void interrupt():** interrupts the thread.
17. **public boolean isInterrupted():** tests if the thread has been interrupted.
18. **public void suspend():** is used to suspend the thread(depricated).
19. **public void resume():** is used to resume the suspended thread(depricated).
20. **public void stop():** is used to stop the thread(depricated).
21. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.

## 2. Creating Thread by implementing Runnable interface:

Threads can also be created by implementing the Runnable interface. Runnable interface have only one method named run(), with the help of Runnable interface the class can work as thread and also we can extend any other class

**public void run():** is used to perform action for a thread.

following are the steps to create a new thread using the Runnable interface:

1. The first step is to create a Java class that implements the Runnable interface.
2. The second step is to override the run() method of the Runnable() interface in the class.
3. Now, pass the Runnable object as a parameter to the constructor of the Thread class
4. Finally, invoke the start method of the Thread object.

**Example:**

```
class RunnableThread implements Runnable
{
        public void run()
        {
                System.out.println("thread created using Runnable Interface");
        }
        public static void main(String args[])
        {
                RunnableThread m1=new RunnableThread ();
                RunnableThread t1 =new RunnableThread (m1);
                t1.start();
        }
}
```

## Priority of a Thread (Thread Priority):

Each thread will have a priority. Priorities are represented by a number between 1, 5 and 10. In most cases, thread schedular 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.

### constants defined in Thread class for Priorities:

1. public static int MIN_PRIORITY - 1
2. public static int NORM_PRIORITY - 5
3. public static int MAX_PRIORITY - 10

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 for priority of a Thread:

```
class ExamplePriority 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[])
    {
        ExamplePriority m1=new ExamplePriority ();
        ExamplePriority m2=new ExamplePriority ();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

**Output:**

```
running thread name is:Thread-0
running thread priority is:10

running thread name is:Thread-1
running thread priority is:1
```

## Thread Scheduler in Java:

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.

## Java Synchronization

Synchronization is a process of handling resource accessibility by multiple thread requests. The main purpose of synchronization is to avoid thread interference. At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent inconsistency problem.

Thread Synchronization

thread synchronization can be done with the help of

Synchronized method.

Synchronized block.

Static synchronization.

## 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.

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 args[]){
```

```
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

## Java synchronized method

If you 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 of java synchronized method
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);}
 }
}
}
```

### Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you 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 to use synchronized block

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

```
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
}
```

## Static Synchronization

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

## Example of static synchronization

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();
}
}
```

## Deadlock in java

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.

## Example of Deadlock in java

```
class TestDeadlockExample1 {
 public static void main(String[] args) {
 final String resource1 = "CSection";
 final String resource2 = "DSection";
 // t1 tries to lock resource1 then resource2
 Thread t1 = new Thread() {
 public void run() {
 synchronized (resource1) {
 System.out.println("Thread 1: locked resource 1");

 try { Thread.sleep(100);} catch (Exception e) {}

 synchronized (resource2) {
System.out.println("Thread 1: locked resource 2");
 }
 }
 }
 };

 // t2 tries to lock resource2 then resource1
 Thread t2 = new Thread() {
 public void run() {
 synchronized (resource2) {
 System.out.println("Thread 2: locked resource 2");

 try { Thread.sleep(100);} catch (Exception e) {}

 synchronized (resource1) {
 System.out.println("Thread 2: locked resource 1");
 }
 }
```

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

Output: Thread 1: locked resource

## Inter-thread communication in Java

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
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**

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.
The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.
Method Description
public final void wait()throws InterruptedException waits until object is notified.
public final void wait(long timeout)throws InterruptedException waits for the specified amount of time

**2) 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()

wait(), notify() and notifyAll() methods belongs to Thread class ?

No, they belong to Object class

Inter thread communication in java

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();
} }
```

Output: going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed