UNIT IV
MULTITHREADING AND GENERIC PROGRAMMING

Thread:

- A thread is a single sequential (separate) flow of control within program. Sometimes, it is called an execution context or light weight process.

Multithreading

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. ht weight process.

Multitasking

Executing several tasks simultaneously is called multi-tasking.

- There are 2 types of multi-tasking
- 1. Process-based multitasking
- 2. Thread-based multi-tasking

Process-based multi-tasking

Executing various jobs together where each job is a separate independent operation is called process-based multi-tasking.

Thread-based multi-tasking

Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread-based multitasking and each independent part is called Thread.

It is best suitable for the programmaticlevel. The main goal of multi-tasking is to make or do a better performance of the system by reducing response time

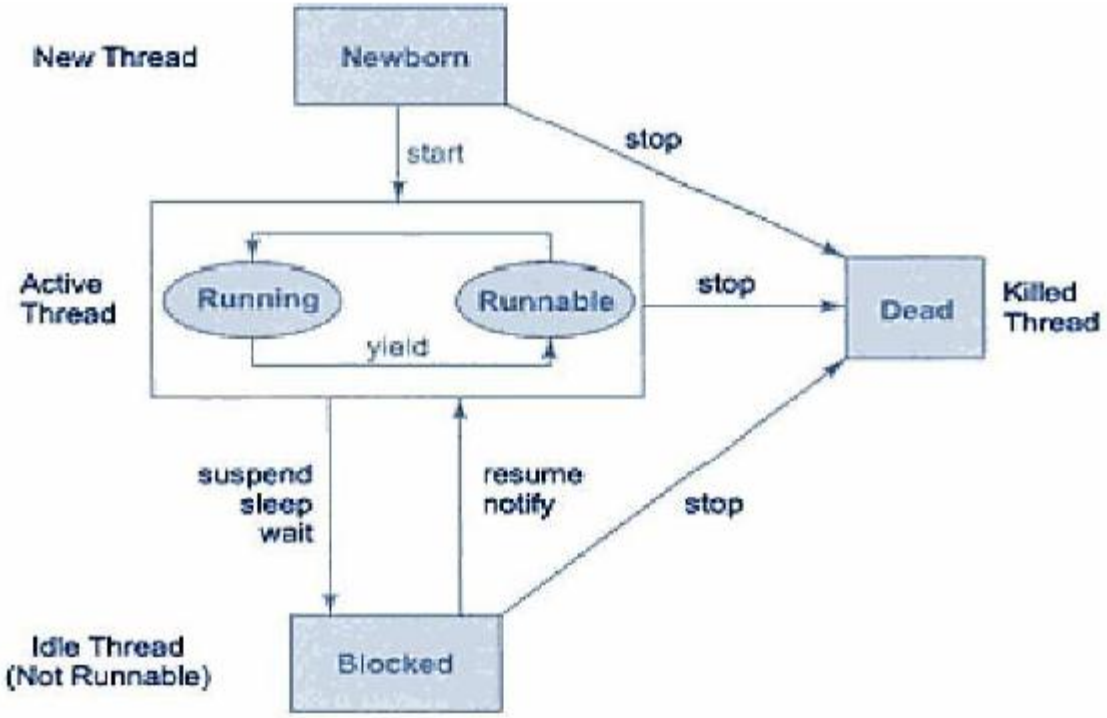
Multithreading	Multitasking
Multithreading is to execute multiple threads in a process concurrently. **Execution**	Multitasking is to run multiple processes on a computer concurrently.
In Multithreading, the CPU switches between multiple threads in the same process	In Multitasking, the CPU switches between multiple processes to complete the execution.
Resource Sharing	
In Multithreading, resources are shared among multiple threads in a process	In Multitasking, resources are shared among multiple processes
Complexity Multithreading is light-weight and easy to create.	Multitasking is heavy-weight and harder to create

Life Cycle of Thread

- A thread can be in any of the five following states

1. **Newborn State**
2. **Runnable State**
3. **Running State**
4. **Blocked State**
5. **Dead State**

Life Cycle of Thread



The Main Thread

- When we run any java program, the program begins to execute its code starting from the main method.
- The JVM creates a thread to start executing the code present in main method. This thread is called as main thread.
- Although the main thread is automatically created, you can control it by obtaining a reference to it by calling currentThread() method.

Two important things to know about main thread are,

- It is the thread from which other threads will be produced.
- main thread must be always the last thread to finish execution.

```
class MainThread
{
 public static void main(String[] args)
 {
  Thread t1=Thread.currentThread();
  t.setName("MainThread");
  System.out.println("Name of thread is "+t1);
 }
}
```

Output: Name of thread is Thread[MainThread,5,main]

Creating Threads

- Threading is a facility to allow multiple tasks to run concurrently within a single process.
- Threads are independent, concurrent execution through a program, and each thread has its own stack.

In Java, There are two ways to create a thread:

1) By extending Thread class.
2) By implementing Runnable interface.

1. Create Thread by 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()

2) Java Thread Example by implementing Runnable interface

```
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 Class

- 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.

1) Java Thread Example by extending Thread class

```
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();
 }
}
```

```
Output:thread is running...
```

Thread class

- Thread class provide constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface

Constructors of Thread class:

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

Thread priority:

- Each thread have a priority. Priorities are represented by a number between 1 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.

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.

Commonly used methods of Thread class

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

public void start(): starts the execution of the thread.JVM calls the run() method on the thread.

public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

public void join(): waits for a thread to die.

public void join(long miliseconds): waits for a thread to die for the specified miliseconds.

public int getPriority(): returns the priority of the thread.

public int setPriority(int priority): changes the priority of the thread.

public String getName(): returns the name of the thread.

public void setName(String name): changes the name of the thread.

public Thread currentThread(): returns the reference of currently executing thread.

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.

Starting a thread:

- **start() method** of Thread class is used to start a newly created thread. It performs following tasks:A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Synchronization in Java

- Synchronization in java is the capability *to control the access of multiple threads to any shared resource.*
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Types of Synchronization

There are two types of synchronization

- Process Synchronization
- Thread Synchronization

Why use Synchronization

The synchronization is mainly used to

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

Thread Synchronization

- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusive
  - Synchronized method.
  - Synchronized block.
  - static synchronization.
- Cooperation (Inter-thread communication in java

Mutual Exclusive

- Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:
- by synchronized method
- by synchronized block
- by static synchronization

Synchronized block.

- 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.

Threads and Synchronization

Multitasking

- • *Multitasking* means that you can have several
- processes running at same time, even if only
- one processor.
- • Can run a browser, VM, powerpoint, print job,
- etc.
- • All modern operating systems support
- multitasking
- • On a single processor system, multitasking is
- an illusion projected by operating system

Threads

- • Inside each process can have several threads
- • Each thread represents its own flow of logic
- • gets separate runtime stack
- • Modern operating systems support threading
- too; more efficient than separate processes
- • Example of threading in a browser:
- • separate thread downloads each image on a page
- (could be one thread per image)
- • separate thread displays HTML
- • separate thread allows typing or pressing of stop
- button
- • makes browser look more responsive

Threads in C/C++

- • Threads are not part of C or C++
- • Have to write different code for each operating
- systems
- • Difficult to port

Threads in Java

- • Part of language
- • Same code for every Java VM
- • Simpler than in most other languages
- • Still very difficult:
- • When running multiple threads, there is
- nondeterminism, even on same machine
- • Often hard to see that your code has bugs
- • Requires lots of experience to do good designs

Threads in the Virtual Machine

- **VM has threads in background**
- **• VM alive as long as a "legitimate thread" still**
- **around (illegitimate threads are "daemons")**
- **• GUI programs will start separate thread to**
- **handle events once frame is visible**
- main thread
- garbage collector
- event thread
- (once container is visible)

Thread Class

- Use Thread class in java.lang
- • Two most important instance methods:
- • start: Creates a new thread of execution in the
- VM; then, invokes run in that thread of execution;
- current thread also continues running
- • run: explains what the thread should do
- • Thread is not abstract, so there are default
- implementations
- • start does what is described above; should be final
- method (but isn't)
- • run returns immediately

Creating A Do Nothing Thread

- • The following code creates a Thread object,
- then starts a second thread.
- public static void main(String[] args) {
- Thread t = new Thread( );
- t.start( ); // now two threads, both running
- System.out.println(“main continues”);
- }
- • In code above:
- • First line creates a Thread object, but main is the
- only running thread
- • Second line spawns a new VM thread. Two threads
- are now active.
- • main thread continues at same time as new t

Getting Thread to Do Something

- • Option #1: extend Thread class, override run
- method
- class ThreadExtends extends Thread {
- public void run( ) {
- for(int i = 0; i < 1000; i++)
- System.out.println("ThreadExtends " + i);
- }
- }
- class ThreadDemo {
- public static void main(String[] args) {
- Thread t1 = new ThreadExtends( );
- t1.start( );
- for(int i = 0; i < 1000; i++)
- System.out.println("main " + i);
- }
- }

Alternative to Extending Thread

- **No multiple inheritance; might not have an**
- **extends clause available**
- **• Might not model an IS-A relationship**
- **• Really just need to explain to Thread what**
- **run method to use**
- **• Obvious function object pattern**
- **• run is encapsulated in standard Runnable interface**
- **• implement Runnable; send an instance to Thread**
- **constructor**
- **• preferred solution**

Alternative #2: Using Runnable

```
class ThreadsRunMethod implements Runnable {
public void run( ) {
for( int i = 0; i < 1000; i++ )
System.out.println( "ThreadsRunMethod " + i );
}
}
class ThreadDemo {
public static void main( String[] args ) {
Thread t2 = new Thread ( new ThreadsRunMethod( ) );
t2.start( );
for( int i = 0; i < 1000; i++ )
System.out.println( "main " + i );
}
}
```

Anonymous Implementation

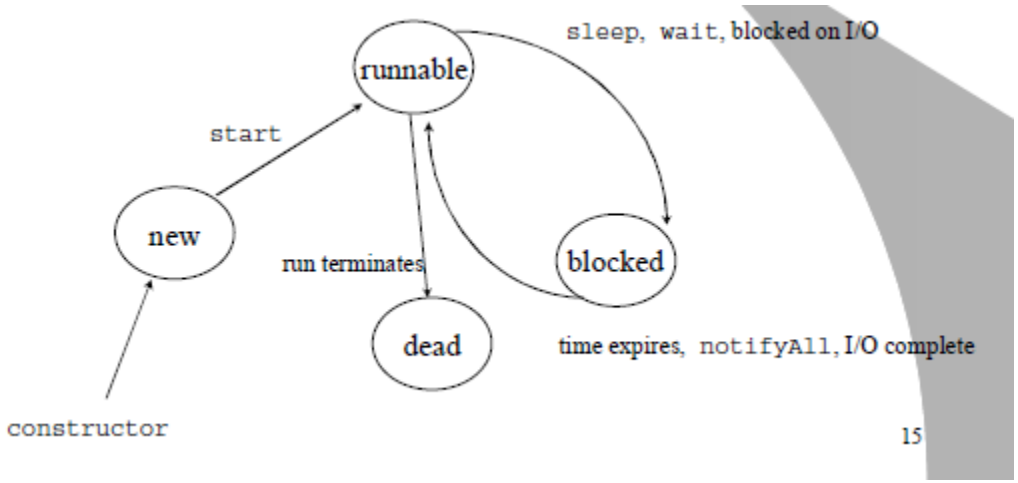
- **May see the Runnable implemented as an**
- **anonymous class in other people's code**
- **class ThreadDemo {**
- **public static void main(String[] args) {**
- **Thread t3 = new Thread (new Runnable( ) {**
- **public void run( ) {**
- **for(int i = 0; i < 1000; i++)**
- **System.out.println("ThreadAnonymous " + i);**
- **}**
- **}**
- **);**
- **t3.start( );**
- **for(int i = 0; i < 1000; i++)**
- **System.out.println("main " + i);**
- **}**
- **}**

Common Mistake #1

- **You should NEVER call run yourself**
- **• will not create new VM thread**
- **• will not get separate stack space**
- **• will invoke run in the current thread**
- **• start don't run**

Thread States

- **Thread is not runnable until start is called**
- **• Thread can only unblock if cause of blocking is**
- **Resolved**



Summary

- • Threading is an essential part of Java and any real
- program. Easier in Java than elsewhere
- • tells you how hard it is elsewhere
- • Follow the rules
- • start don't run
- • don't rely exclusively on priorities
- • no public data
- • synchronize mutators, maybe accessors
- • leave critical section only after object is restored
- • no sleeping in synchronized block
- • use wait/notifyAll pattern (or await/signalAll)
- • obtain monitors in same order