Marwadi
education foundation

Unit – 12
Concurrency

Prepared By
Prof. Ravikumar Natarajan
Assistant Professor, CE Dept.

KNOWLEDGE IS THE CURRENCY
FOR THE 21st CENTURY

Contents

1. Concurrency
2. Thread states and life cycle,
3. Creating and Executing threads with the Executor Framework,
4. Thread synchronization

Concurrency

- Concurrency is the ability to run several programs or several parts of a program in parallel.
- If a time consuming task can be performed asynchronously or in parallel, this improves the throughput and the interactivity of the program.
- A modern computer has several CPU's or several cores within one CPU.
- Two basic units of execution,
- Processes – it has self contained execution environment. Has its own memory space.
- Threads – Threads exist within process – every process has at least one. Threads share process's resources including memory and open files.

What is Thread?

- Thread is a tiny program running continuously. It is called as light weight process. Any single path of execution is called thread.
- A thread is also called flow of execution.
- Switching between threads automatically done by JVM

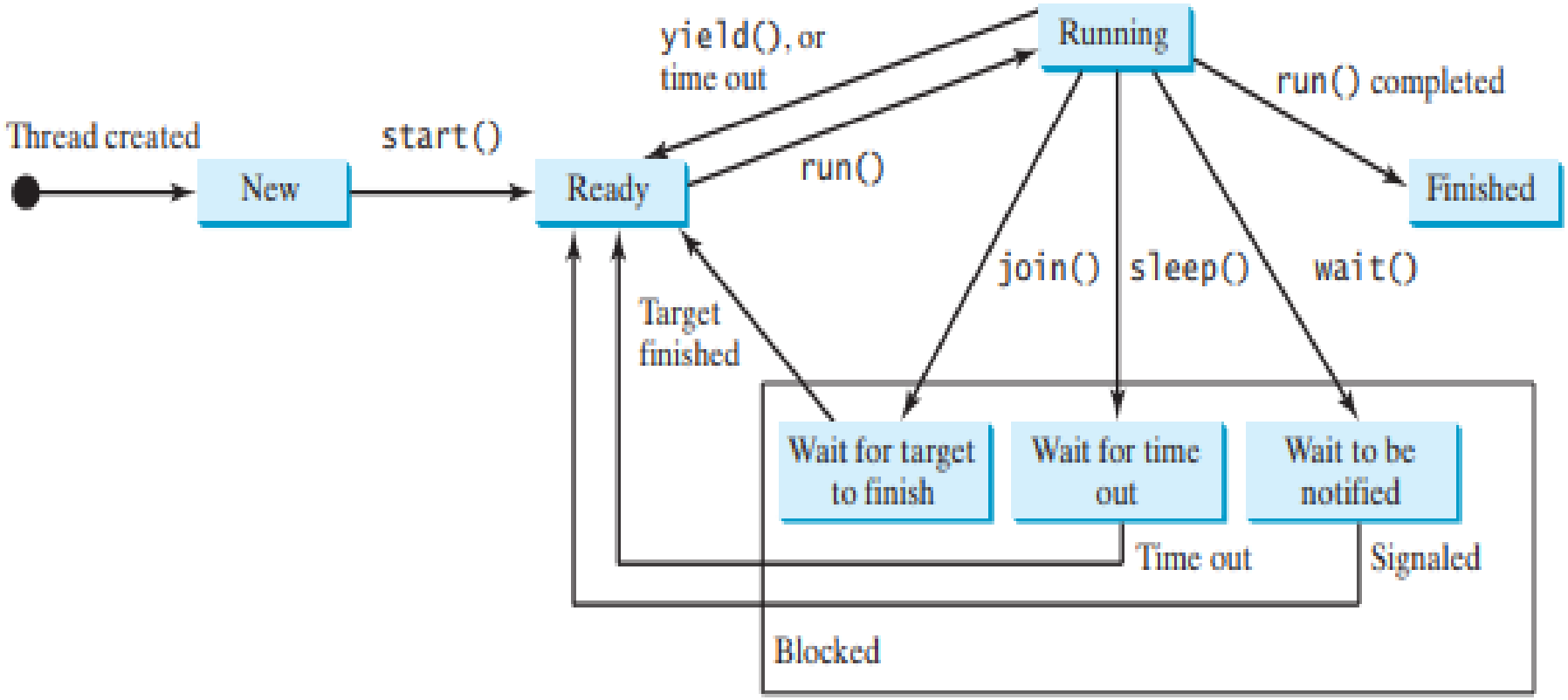
S.No	Thread	Process
1.	It is a light-weight process	It is a heavy-weight process
2	Threads do not require separate address space. It runs in the address space of process to which it belongs to.	Each process requires separate address space to execute.

Multithreading vs Multitasking

S.No	Multithreading	Multitasking
1.	Thread is a fundamental unit of multithreading.	Process is a fundamental unit of multiprocessing environment.
2.	Multiple parts of a single program gets executed in multithreading envi.	Multiple programs get executed in multiprocessing envi.
3.	During multithreading the processor switches b/w multiple threads of the program.	During multiprocessing the processor switches b/w multiple programs.
4.	It is cost effective bcz cpu can be shared among multiple threads at a time.	It is expensive because when a particular process uses cpu other processes has to wait.
5.	Highly efficient	Less efficient
6.	It helps in developing application programs.	It helps in developing OS programs.

Thread states and life cycle

- **A thread state indicates the status of thread.**
- Tasks are executed in threads. Threads can be in one of five states: New, Ready, Running, Blocked, or Finished



Thread states and life cycle

- When a thread is newly created, it enters the New state. After a thread is started by calling its start() method, it enters the Ready state. A ready thread is runnable but may not be running yet. The operating system has to allocate CPU time to it.
- When a ready thread begins executing, it enters the Running state. A running thread can enter the Ready state if its given CPU time expires or its yield() method is called.

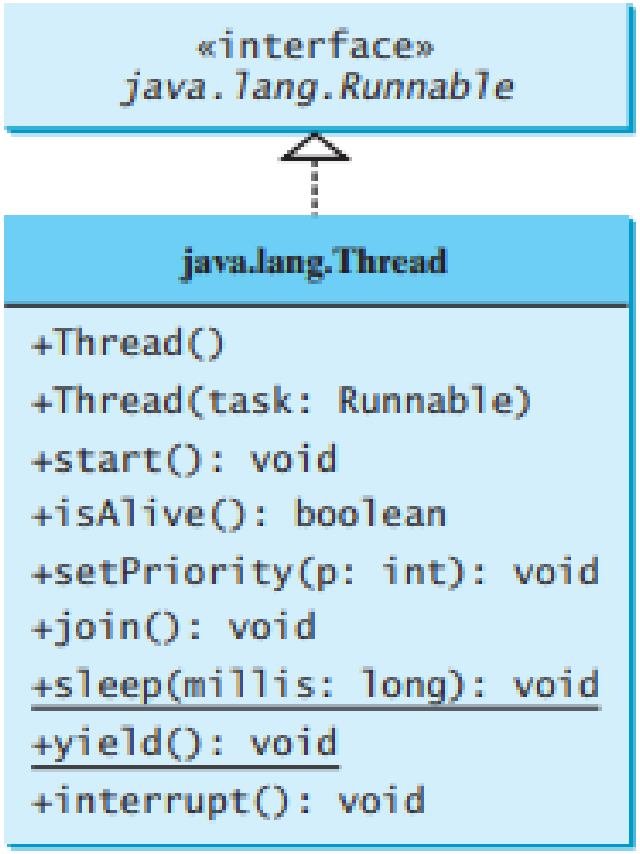
- A thread can enter the Blocked state (i.e., become inactive) for several reasons. It may have invoked the join(), sleep(), or wait() method. It may be waiting for an I/O operation to finish. A blocked thread may be reactivated when the action inactivating it is reversed. For
- example, if a thread has been put to sleep and the sleep time has expired, the thread is reactivated and enters the Ready state.

- Finally, a thread is Finished if it completes the execution of its run() method.
- The isAlive() method is used to find out the state of a thread. It returns true if a thread is in the Ready, Blocked, or Running state; it returns false if a thread is new and has not started or if it is finished.

- The interrupt() method interrupts a thread in the following way: If a thread is currently in the Ready or Running state, its interrupted flag is set; if a thread is currently blocked, it is awakened and enters the Ready state, and a java.lang.InterruptedException is thrown.

The Thread Class

- The Thread class contains the constructors for creating threads for tasks and the methods for controlling threads.



Creates an empty thread.	
Creates a thread for a specified task.	
Starts the thread that causes the `run()` method to be invoked by the JVM.	
Tests whether the thread is currently running.	
Sets priority p (ranging from 1 to 10) for this thread.	
Waits for this thread to finish.	
Puts a thread to sleep for a specified time in milliseconds.	
Causes a thread to pause temporarily and allow other threads to execute.	
Interrupts this thread.	

Creating Threads

Two approaches,

1. Using Thread class
2. Using Runnable interface

The **run()** method is the most important method in any thread programming. Using this method thread behavior can be implemented.

Syntax:

```
public void run()
{
//statements
}
```

Creating Threads


```
//Example using Thread class
class mythread extends Thread
{
public void run()
{
System.out.println("Thread is created!!");
}
}
class threaddemo
{
public static void main(String args[])
{
mythread t= new mythread();
t.start();
}
}
```

```
//Example using Runnable interface
class mythread implements Runnable
{
public void run()
{
System.out.println("Thread is created!!");
}
}
class Main extends mythread
{
public static void main(String args[])
{
Main obj = new Main();
Thread t= new Thread(obj);
t.start();
}
}
```

Thread Synchronization

- Java Synchronization allows only one thread to access the shared resource. This will overcome problems like i) Thread interference and ii) Memory consistency errors. iii) Inconsistency problems.
- The synchronization concept is based on monitor, which is lock and unlock. When a thread owns this monitor other thread cannot access the resources. Other threads will be in waiting state.

Two ways to achieve synchronization,
1) Using synchronized methods
2) Using synchronized blocks or statements

Rules:

1. **Constructors**, **Classes and variables cannot be synchronized**
2. Each object has one lock
3. A thread can acquire more than one lock.
4. When synchronization applied it is called critical section (one thread process can access resource at a time).

Thread Synchronization


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

Inter-Thread Communication – Cooperation among Threads – Producer-Consumer Problem


```
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 of " +amount+"
completed...");
}
```

```
synchronized void deposit(int amount)
{
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("Deposit of " + amount+ "
completed... ");
            notify();
            }
}

class InterThread
{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(500);}
                        }.start();

new Thread(){
public void run(){c.deposit(12000);}
                        }.start();   } }
```

Deadlock and Semaphore

Marwadi
education foundation

//For your reference

Semaphores can be used to restrict the number of threads that access a shared resource.

Deadlocks can be avoided by using a proper resource ordering

Creating and Executing threads with the Executor Framework

- Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks.

- In server application, it creates a new thread each time a request arrives and service this new request in the newly created thread.

- **Disadvantage:** A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Creating and Executing threads with the Executor Framework

- As active threads consume system resources, a JVM creating too many threads at the same time can cause the system to run out of memory.
- This necessitates the need to limit the number of threads being created.
- Solution: ThreadPool
- A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.
- Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

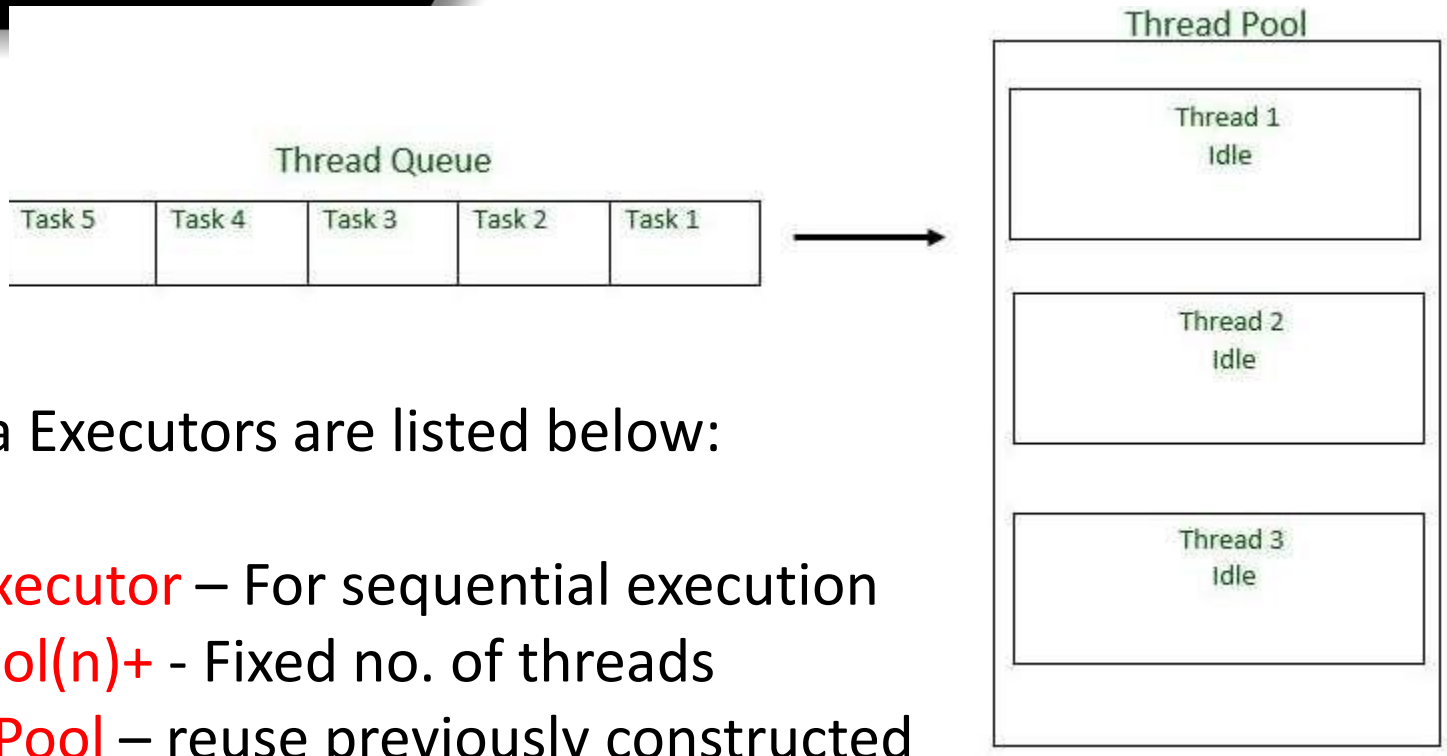
Creating and Executing threads with the Executor Framework

- Java provides the Executor framework which is centered around the
- **Executorinterface**, its sub-interface–**ExecutorService** and the class **ThreadPoolExecutor**, which implements both of these interfaces.

- By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.

- To use thread pools, we first create a object of Executor Service and pass a set of tasks to it. **ThreadPoolExecutor** class allows to set the core and maximum pool size. The runnables that are run by a particular thread are executed sequentially.

Creating and Executing threads with the Executor Framework



Some types of Java Executors are listed below:

1. SingleThreadExecutor – For sequential execution
2. FixedThreadPool(n)+ - Fixed no. of threads
3. CachedThreadPool – reuse previously constructed
4. ScheduledExecutor – to run at regular interval

Syntax: ExecutorService executor = Executors.new **SingleThreadExecutor**();

Creating and Executing threads with the Executor Framework

Marwadi education foundation

- One of the main advantages of using this approach is when you want to process 100 requests at a time, but do not want to create 100 Threads for the same, so as to reduce JVM overload.

- You can use this approach to create a ThreadPool of 10 Threads and you can submit 100 requests to this ThreadPool. ThreadPool will create maximum of 10 threads to process 10 requests at a time.

- After process completion of any single Thread, ThreadPool will internally allocate the 11th request to this Thread and will keep on doing the same to all the remaining requests.

Queries???

END OF UNIT - 12