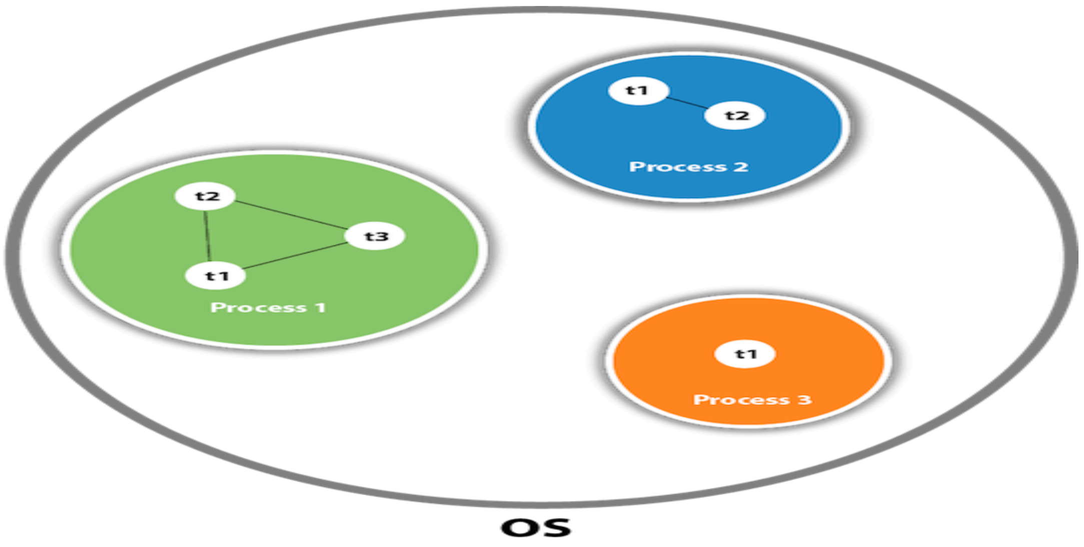
UNIT-III-PART-II

built-in exceptions, chained exceptions, creating own exception sub classes. Java thread model, thread priorities, synchronization, messaging, thread class and runnable interface, creating thread, creating multiple threads, thread priorities, synchronizing threads, interthread communication, thread life cycle.

What is Thread in java

A thread is a lightweight subprocess, the 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 uses a shared memory area.



a 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.

Introduction to Multithreading in Java

Multithreading is a concept of running multiple threads simultaneously. Thread is a lightweight unit of a process that executes in multithreading environment.

A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process)

Advantages of Multithreading

Multithreading **reduces** the CPU **idle time** that increase overall performance of the system.

Since thread is lightweight process then it takes **less memory**

Multitasking

Multitasking is a process of performing multiple tasks simultaneously.

We can understand it by computer system that perform multiple tasks like: writing data to a file, playing music, downloading file from remote server at the same time.

Multitasking can be achieved either by using multiprocessing or multithreading.

Multitasking by using multiprocessing involves multiple processes to execute multiple tasks simultaneously whereas Multithreading involves multiple threads to executes multiple tasks.

Why Multithreading ?

Thread has many advantages over the process to perform multitasking.

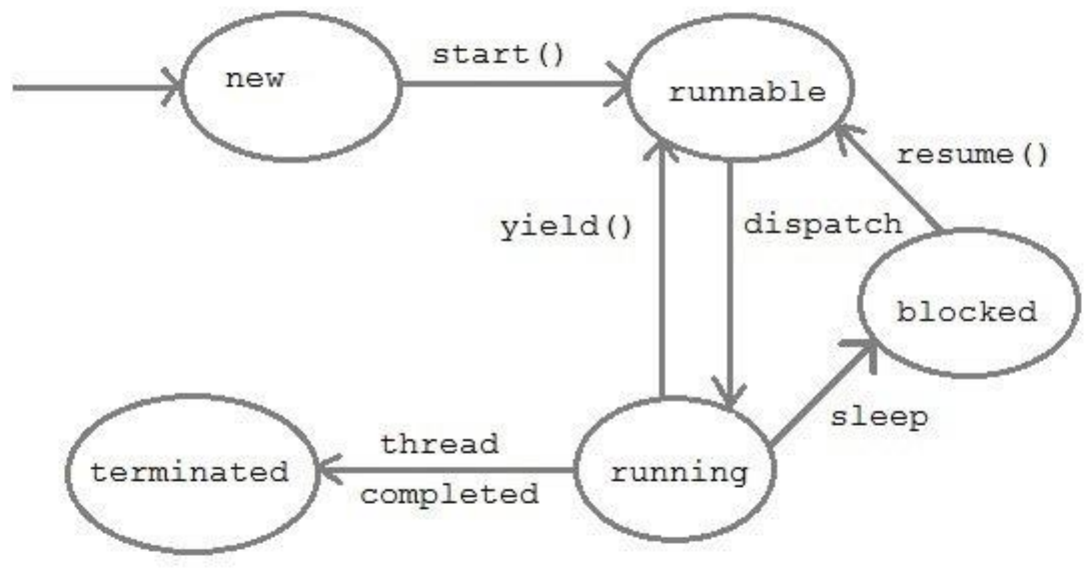
Process is heavy weight, takes more memory and occupy CPU for longer time that may lead to performance issue with the system.

To overcome these issue process is broken into small unit of independent sub-process.

These sub-process are called threads that can perform independent task efficiently.

Life cycle of a Thread

Like process, thread have its life cycle that includes various phases like: new, running, terminated etc. we have described it using the below image.



1. **New :** A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. **Runnable :** After invocation of start() method on new thread, the thread becomes runnable.
3. **Running :** A thread is in running state if the thread scheduler has selected it.
4. **Waiting :** A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
5. **Terminated :** A thread enter the terminated state when it complete its task.

Daemon Thread

Daemon threads is a low priority thread that provide supports to user threads.

These threads can be user defined and system defined as well. Garbage collection thread is one of the system generated daemon thread that runs in background.

These threads run in the background to perform tasks such as garbage collection.

Daemon thread does allow JVM from existing until all the threads finish their execution.

Thread Class Methods

Thread class also defines many methods for managing threads. Some of them are,

Method	Description
setName()	to give thread a name
getName()	return thread's name
getPriority()	return thread's priority
isAlive()	checks if thread is still running or not
join()	Wait for a thread to end

Method	Description
run()	Entry point for a thread
sleep()	suspend thread for a specified time
start()	start a thread by calling run() method

Creating a thread in Java

To implement multithreading, Java defines two ways by which a thread can be created.

- By implementing the **Runnable** interface.
- By extending the **Thread** class.

Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface, the class needs to implement the run() method.

```
public void run()
```

```
class MyThread implements Runnable
{
   public void run()
   {
      System.out.println("concurrent thread started running..");
   }
}


class MyThreadDemo
{
   public static void main(String args[])
   {
      MyThread mt = new MyThread();
      Thread t = new Thread(mt);
      t.start();
   }
}
```

Extending Thread class

This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class.

```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("concurrent thread started running..");
        }
}


classMyThreadDemo
{
        public static void main(String args[])
        {
                MyThread mt = new  MyThread();
                mt.start();
        }
}
```

o/p:-

```
concurrent thread started running..
```

What if we call run() method directly without using start() method?

In above program if we directly call run() method, without using start() method,

```
public static void main(String args[])
{
        MyThread mt = new MyThread();
        mt.run();
}
```

Can we Start a thread twice?

No, a thread cannot be started twice. If you try to do so, IllegalThreadStateException will be thrown.

```
public static void main(String args[])
{
        MyThread mt = new MyThread();
        mt.start();
        mt.start();             //Exception thrown
}
```

When a thread is in running state, and you try to start it again, or any method try to invoke that thread again using start() method, exception is thrown.

Example of thread with join() method

In this example, we are using join() method to ensure that thread finished its execution before starting other thread. It is helpful when we want to executes multiple threads based on our requirement.

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

```
r1
```

```
r2
r1
r2
```

Sleeping Thread

To sleep a thread for a specified time, Java provides sleep method which is defined in Thread class.

It always pause the current thread execution. Any other thread can interrupt the current thread in sleep, in that case InterruptedException is thrown.

Syntax:-

```
    sleep(long millis)throws InterruptedException
```

In this example, we are sleeping threads by using sleep method. Each thread will sleep for 1500 milliseconds and then resume its execution.

Example:-

```
import java.lang.Thread;
import java.io.*;
  public class TestSleepMethod2
{
   // main method
public static void main(String argvs[])
{
  try
{
for (int j = 0; j < 5; j++)
{

// The main thread sleeps for the 1000 milliseconds, which is 1 sec
// whenever the loop runs
Thread.sleep(1000);
  // displaying the value of the variable
System.out.println(j);
}
}
catch (Exception expn)
{
// catching the exception
System.out.println(expn);
}
}
}
```

Output:

```
0
1
2
3
4
```

Naming Thread

Each thread in Java has its own name which is set by the JVM by default.

we can get name of a thread by calling getName() method of Thread class. If we wish to set new name of the thread then **setName()** method can be used.

Thread Name

lets first fetch the thread name set by the JVM. It is default name so initially we cannot predict it.

Ex:-

```
public class MyThread extends Thread
{

    public void run()
    {
      System.out.println("thread running...");
    }
    public static void main(String[] args)
    {
       MyThread t1=new MyThread();

       t1.start();
       System.out.println("thread name: "+t1.getName());
    }
}
```

o/p:-

```
thread running...
thread name: Thread-0
```

Thread Priorities

Priority of a thread describes how early it gets execution and selected by the thread scheduler.

In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler.

The priority is given by the JVM or by the programmer itself explicitly.

The range of the priority is between 1 to 10

MIN_PRIORITY

It holds the minimum priority that can be given to a thread. The value for this is 1.

NORM_PRIORITY

It is the default priority that is given to a thread if it is not defined. The value for this is 0.

MAX_PRIORITY

It is the maximum priority that can be given to a thread. The value for this is 10.

hread Priority

If we don't set thread priority of a thread then by default it is set by the JVM. In this example, we are getting thread's default priority by using the getPriority() method.

Example:-

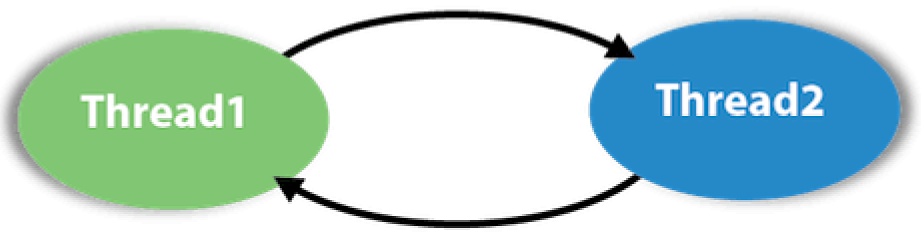
```
class MyThread extends Thread
{
        public void run()
        {
                System.out.println("Thread Running...");
        }

        public static void main(String[]args)
        {
                MyThread p1 = new MyThread();
                MyThread p2 = new MyThread();
                MyThread p3 = new MyThread();
                p1.start();
                System.out.println("P1 thread priority : " + p1.getPriority());
                System.out.println("P2 thread priority : " + p2.getPriority());
                System.out.println("P3 thread priority : " + p3.getPriority());


        }
}
```

o/p:-

```
P1 thread priority : 5
Thread Running...
P2 thread priority : 5
P3 thread priority : 5
```

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.



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.

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);}
   }

 }
}

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();
}
}
```

Output:

```
5
 10
 15
 20
 25
 100
 200
 300
 400
 500
```

Using Synchronized block

If want to synchronize access to an object of a class or only a part of a method to be synchronized then we can use synchronized block for it.

It is capable to make any part of the object and method synchronized.

Example

In this example, we are using synchronized block that will make the display method available for single thread at a time.

```
class First
{
  public void display(String msg)
  {
    System.out.print ("["+msg);
    try
    {
      Thread.sleep(1000);
    }
    catch(InterruptedException e)
    {
      e.printStackTrace();
    }
    System.out.println ("]");
  }
}

class Second extends Thread
{
  String msg;
  First fobj;
  Second (First fp,String str)
  {
    fobj = fp;
    msg = str;
    start();
  }
  public void run()
  {
    synchronized(fobj)      //Synchronized block
    {
      fobj.display(msg);
    }
  }
}
```

```
public class MyThread
{
 public static void main (String[] args)
 {
  First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second (fnew,"new");
  Second ss2 = new Second(fnew, "programmer");
 }
}
```

o/p:-

```
[welcome]
[new]
[programmer]
```

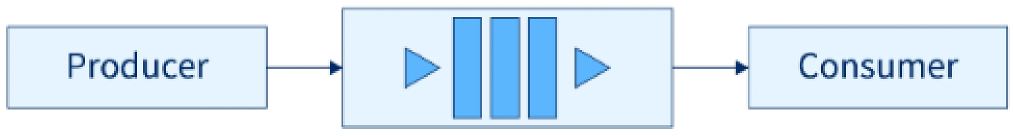
Interthread Communication

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.

The producer-consumer problem in Java (also known as the bounded-buffer problem) is a classic multi-process synchronization problem, in which we try to achieve synchronization between more than one process.

In the producer-consumer problem, there are two processes: the producer and the consumer. These processes share a common buffer of fixed size, which is used as a queue. The producer's task is to generate data (item) and put it into the buffer. The consumer's task is to consume data by removing it from the queue.



introduction to producer consumer problem in java

The problem/complexities in the Producer-Consumer problem:

The producer should not produce any data when the buffer is full.

The producer in this case should wait until the consumer consumes data and some space is cleared in the buffer.

The consumer can consume data only when the buffer is not empty. If the buffer is empty, the consumer cannot consume data (i.e., cannot remove data from the queue).

Access to the memory buffer should not be given simultaneously to both the producer and consumer.

A real-life example of this can be a factory that produces some items and stores them in its storage. The producer will only produce items when there is space left in the storage. If the storage is full, the producer will wait for the consumer to consume items before starting production again. Similarly, the consumer cannot consume an item if the storage is empty. The consumer will wait until there is an item in the storage to consume.

Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

Test.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
```