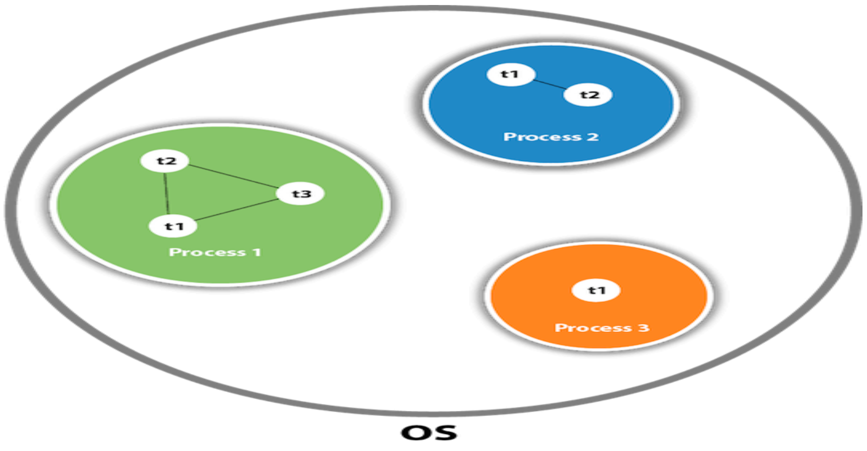
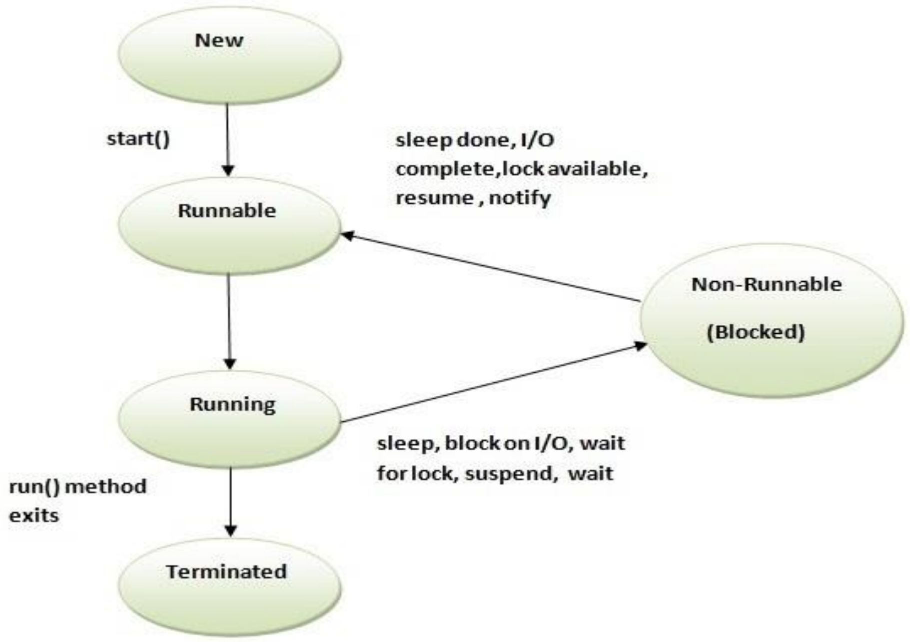
Multithreading

- **A thread is a lightweight subprocess, the smallest unit of processing. It has 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**

1. The main purpose of multithreading is to provide **simultaneous execution** of two or more parts of a program for maximum utilization the CPU time. A multithreaded program contains two or more parts that can run **concurrently**. Each such part of a program called thread.



3. **Life Cycle of a thread:** A thread can be in one of the following states:


New
start()
Runnable
sleep done, I/O complete, lock available, resume , notify
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

Life Cycle of a thread

New Thread: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and has not started to execute.

Runnable State: A thread that is ready to run is moved to a runnable state. In this state, a thread might be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.

A multi-threaded program allocates a fixed amount of time to each individual thread. Each thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

Blocked/Waiting state: When a thread is temporarily inactive, then it's in one of the following states:

- Blocked
- Waiting
- Timed Waiting: A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

Terminated State: A thread terminates because of either of the following reasons:

- Because it exits normally. This happens when the code of the thread has been entirely executed by the program.
- Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

Multitasking vs Multithreading vs Multiprocessing vs parallel processing

Multitasking: Ability to execute more than one task at the same time is known as multitasking.
Multithreading: We already discussed about it. It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.
Multiprocessing: It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.
Parallel Processing: It refers to the utilization of multiple CPUs in a single computer system.

Main Thread

```
class MainThreadTest
  {
     public static void main(String args[])
     {
        Thread t = Thread.currentThread();
          System.out.println(t);
     }
  }                          O/P:  Thread[main,5,main]
```

main thread(main), Thread Priority(5), Thread Group(main)

```
class MainThreadTest2
  {
     public static void main(String args[])
     {
        Thread t = Thread.currentThread();
        t.setName("My Thread");
        System.out.println(t);
     }
  }
```

O/P: Thread[My Thread,5,main]

Creating a thread in Java:

There are two ways to create a thread in Java:

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

Commonly used Constructors of Thread class:

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

Before we begin with the programs(code) of creating threads, let's have a look at these methods of Thread class. We have used few of these methods in the example below.

- getName(): It is used for Obtaining a thread's name
- getPriority(): Obtain a thread's priority
- isAlive(): Determine if a thread is still running
- join(): Wait for a thread to terminate

- run(): Entry point for the thread
- sleep(500) suspend a thread for a period of time
- start(): start a thread by calling its run() method

Method 1: Thread creation by extending Thread class

Example 1:

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

Output: **My thread is in running state.**

Method 2: Thread creation by implementing Runnable Interface

A Simple Example

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

Output: My thread is running…

Thread priorities

- Thread priorities are the integers which decide how one thread should be treated with respect to the others.
- Thread priority decides when to switch from one running thread to another, process is called context switching
- A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
- To set the priority of the thread setPriority() method is used which is a method of the class Thread Class.
- In place of defining the priority in integers, we can use MIN_PRIORITY, NORM_PRIORITY or MAX_PRIORITY.

  - ➢ public static int MIN_PRIORITY = 1
  - ➢ public static int NORM_PRIORITY = 5
  - ➢ 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.

```
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[])
     {     Thread t = Thread.currentThread();
   System.out.println(t);
TestMultiPriority m1=new TestMultiPriority();
  TestMultiPriority m2=new TestMultiPriority();
   m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
  }
}
```

```
o/p :   Thread[main,5,main]

        running thread name is:Thread-1
        running thread name is:Thread-0
        running thread priority is:10
        running thread priority is:1
```

Daemon Thread

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.
There are many java daemon threads running automatically e.g. gc, finalizer etc.

Note:

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

Methods for Java Daemon thread by Thread class

The java.lang.Thread class provides two methods for java daemon thread.

Method	Description
public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
public boolean isDaemon()	is used to check that current is daemon.

```
public class TestDaemonThread1 extends Thread{
 public void run()
{
 if(Thread.currentThread().isDaemon())//checking for daemon thread
        {   System.out.println("daemon thread work");   }
  else
        {    System.out.println("user thread work");     }
 }
 public static void main(String[] args)
{    TestDaemonThread1 t1=new TestDaemonThread1();
      TestDaemonThread1 t2=new TestDaemonThread1();
      TestDaemonThread1 t3=new TestDaemonThread1();
      t1.setDaemon(true);                  //now t1 is daemon thread
      t1.start();  //starting threads
      t2.start();
```

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

```
                            o/p: daemon thread work
                                 user thread work
                                 user thread work
```

Methods: isAlive() and join()

- In all the practical situations main thread should finish last else other threads which have spawned from the main thread will also finish.
- To know whether the thread has finished we can call isAlive() on the thread which returns true if the thread is not finished.
- Another way to achieve this by using join() method, this method when called from the parent thread makes parent thread wait till child thread terminates.
- These methods are defined in the Thread class.
- We have used isAlive() method in the above examples too.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

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

```
// Problem without Synchronization
class Table{
void printTable(int n){//method not synchronized
  for(int i=1;i<=10;i++){
    System.out.println(n*i);
    try
          {         Thread.sleep(500);       }
     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 WithOutSynchronization
{
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();
}
}
```

```
                                O/P:    5       100     10      200     15      300     20
                                        400     25      500     30      600     35      700
                                        40      800     45      900     50      1000
```

Solution of Above Problem

```
class MyTable
     {
        synchronized static void display_Table(int n)
                                                   //synchronized method
             {
          for(int i=1;i<=10;i++)
                 {
           System.out.print("\t"+n*i);
                       try{        Thread.sleep(500);       }
                       catch(Exception e)   {System.out.println(e.getMessage());}
                 }
              }                                  //end of the method
           }
      class MyThread1 extends Thread
         {   MyTable t;
              MyThread1(MyTable t)
             {    this.t=t;  }
            public void run()
              {
                  t.display_Table(5);
               }
            }
            class MyThread2 extends Thread
            {
            MyTable t;
            MyThread2(MyTable t) {   this.t=t;  }
            public void run(){    t.display_Table(10);  }
            }
            class Th_Synchronization
            {
            public static void main(String args[]){
            MyTable obj = new MyTable();          //only one object
            MyThread1 t1=new MyThread1(obj);
            MyThread2 t2=new MyThread2(obj);
            t1.start();
            t2.start();
            }
            }
```

```
O/P: 5    10    15    20    25    30    35
     40   45    50    10    20    30    40
      50   60    70    80    90    100
```

Java Program to print ODD & EVEN Numbers

```
class MyOddEven
{
  synchronized static void display_No(int n)        //synchronized method
      {
   for(int i=1;i<=20;i++)  {
           System.out.print("\t"+n);
               n=n+2;
                   try{        Thread.sleep(500);       }
```

```
              catch(Exception e)   {System.out.println(e.getMessage());}
         }
 }//end of the method
}

class MyThread1 extends Thread{
MyOddEven t;
MyThread1(MyOddEven t){
this.t=t;
}
public void run(){
t.display_No(1);
}
}
class MyThread2 extends Thread
{
MyOddEven t;
MyThread2(MyOddEven t) {   this.t=t;  }
public void run(){    t.display_No(2);  }
}
class Th_Synchronization_EVEN_ODD
{
public static void main(String args[]){
MyOddEven obj = new MyOddEven();
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

O/P

```
D:\YMS\java_program>java Th_Synchronization_EVEN_ODD
      1    3    5    7    9    11    13    15    17    19    21    23    25    27    29
31    33    35    37    39     2    4    6    8    10    12    14    16    18    20
22    24    26    28    30    32    34    36    38    40
```

Inter-thread communication

P1 (t1) notify() then wait() concurrent P2(t2) wait() then notify()

Lock in Java: Synchronization is built around an internal entity known as the lock or monitor. Every object has an 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.

- Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.
- When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.
- To implement the synchronous behavior java has synchronous method. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. All the other threads then wait until the first thread come out of the synchronized block.
- When we want to synchronize access to objects of a class which was not designed for the multithreaded access and the code of the method which needs to be accessed synchronously is not available with us, in this case we cannot add the synchronized to the appropriate methods. In java we have the solution for this, put the calls to the methods (which needs to be synchronized) defined by this class inside a synchronized block in following manner.

```
synchronized(object)
{
   // statement to be synchronized
}
 ……………..AND
synchronized      RType   method_name()
{
   // statement to be synchronized
}
```

Inter-thread Communication

We have few methods through which java threads can communicate with each other. These methods are wait(), notify(), notifyAll(). All these methods can only be called from within a synchronized method.

- To understand synchronization java has a concept of monitor. Monitor can be thought of as a box which can hold only one thread. Once a thread enters the monitor all the other threads have to wait until that thread exits the monitor.
- wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- notify() wakes up the first thread that called wait() on the same object. notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

Example 1: ITC

```
   class PC
    {    int count;

         boolean flag=false;

         public  synchronized void Give(int count)

           {     while(flag)

                 {

                  try{ wait();}catch(Exception e){}

                 }

                 this.count=count;

                System.out.println("Given"+count);

                flag=true;

                notify();

            }

        public synchronized  void Take()

           {     while(!flag)

                      {

                           try{wait();} catch(Exception e){}

                       }

             System.out.println("Taken"+count);

                flag=false;

                notify();
            }
      }
class Producer implements Runnable
      {       PC pc;
                 Producer(PC pc )
               {    this.pc=pc;
                    Thread t =new Thread(this,"Producer Thread");
                    t.start();
               }
         public void run()
              {      int i=0;
                    while(true)
                    {    pc.Give(++i) ;
                        try{  Thread.sleep(1000);  }  catch(Exception ie){}
                     }
              }
       }
class Consumer implements Runnable
      {       PC pc;
                 Consumer(PC pc )
               {    this.pc=pc;
                    Thread t =new Thread(this,"Consumer Thread");
                     t.start();
               }
         public void run()
              {
                    while(true)
                     {    pc.Take() ;
                        try {   Thread.sleep(1000);  } catch(Exception e){}
                     }
```

```
                }
            }
        class ITC

            {   public static void main (String a[])

                        PC obj=   new PC();

                            new Producer(obj);

                            new Consumer(obj);

                }

            }
```

```
                                        O/P: Given1
                                             Taken1
                                             Given2
                                             Taken2
                                             ……………infinite……times
```

Example 2:ITC

```
class Chat {
  boolean flag = false;

  public synchronized void Question(String msg) {
    if (flag) {
      try {
        wait();
      } catch (InterruptedException e)
                  {     System.out.println( e);         }
    }
    System.out.println(msg);
    flag = true;
    notify();
  }
  public synchronized void Answer(String msg) {
    if (!flag) {
                try {    wait();      }
                catch (InterruptedException e)  {   System.out.println( e);     }
              }
    System.out.println(msg);
    flag = false;
    notify();
  }
}
class T1 implements Runnable {
  Chat m;
  String[] s1 = { "Hi", "How are you ?", "I am also doing fine!", "Bye-Bye" };
 T1(Chat m) {
    this.m = m;
    new Thread(this).start();
  }
  public void run() {
    for (int i = 0; i < s1.length; i++) {
      m.Question(s1[i]);
    }
  }
}
class T2 implements Runnable {
  Chat m;
  String[] s2 = { "Hello ", "I am good, what about you?", "Great! bye...." };
   T2(Chat m) {
    this.m = m;
    new Thread(this).start();
  }
  public void run() {
    for (int i = 0; i < s2.length; i++) {
      m.Answer(s2[i]);
    }
  }
}
public class I_Thread_C {
  public static void main(String[] args) {
    Chat m = new Chat();
    new T1(m);
    new T2(m);
  }
}
```

O/P:

```
Hi
Hello
How are you ?
I am good, what about you?
I am also doing fine!
Great! bye....
Bye-Bye
```

Deadlock in Thread

```
public class  Deadlock
{             public static Object pen=new Object();
              public static Object copy=new Object();

       public static void main(String a[])
         {
                    new Thread1().start();
                    new Thread2().start();
         }
  private static class Thread1 extends Thread
    {      public void run()                          // thread1 tries to lock pen then copy
             {   synchronized(pen)
                    {   System.out.println("thread 1 Holding Pen");
                                   try      {sleep(5000);}
                                   catch(Exception e)
                                          {e.getMessage();}
                         System.out.println("thead 1 Waiting for Copy");
                  synchronized(copy)
                               {
                                   System.out.println("thread 1 Holding Copy.........");
                               }
                    }
           }
     }
  private static class Thread2 extends Thread
    {      public void run()            // thread2 tries to lock copy then pen

             {   synchronized(copy)
                {  System.out.println("thread2 Holding Copy");
                    try {sleep(1000);}   catch(Exception e) {e.getMessage();}
                   System.out.println("thread 2 Waiting for Pen");
                   synchronized(pen)               {System.out.println("thread    2    Holding
Pen.........");}
             }
         }
     }
}
```

O/P:

```
thread 1 Holding Pen
thread2 Holding Copy
thread 1 Waiting for Copy
thread 2 Waiting for Pen
```