Que. Difference between throw and throws in Java
Ans.

No.	Throw	Throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions.

6)	e.g.
	void m(){
	throw new ArithmeticException("sorry");
	
	}
	
		e.g.
		public void method( )throws IOException, SQLException
		{
		……
		}

Que. Differentiate checked and unchecked exception.
Ans. Exceptions that are checked at compile-time are called checked exceptions. These exceptions are explicitly handled in the code itself with the help of try-catch blocks. Checked exceptions are extended from the java.lang.Exception class.
The exceptions that are not checked at compile time are called unchecked exceptions. These exceptions are not essentially handled in the program code, instead the JVM handles such exceptions. Unchecked exceptions are extended from the java.lang.RuntimeException class.

Que. Explain Exception Handling with suitable example.
Ans. Errors can be classified into two categories :

- Compile-time errors
- Run-time errors

An **exception** is an abnormal condition that arises in a code sequence at **run time**.

➔If the exception is not caught and handled properly, the interpreter will display an error message and will terminate the program. If we want the program to continue with the exception, then we should try to catch the exception object thrown by the error condition and then display an appropriate message. This task is known **as exception Handling.**

➔ Java exception handling is managed via five keywords:
try, catch, throw, throws, and finally.

➔ Program statements that we want to monitor for exceptions are contained within a try block. If an exception occurs within the **try block**, it is thrown.

➔ Our Code can catch this exception (using **catch**) and handle it in some rational manner. (Prof. Viral S. Patel)

➔ System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**.

➔ Any exception that is thrown out of a method must be specified as such by a **throws**

clause.

➔ Any code that absolutely must be executed before a method returns is put in a **finally block.**

➔**Throwable** is the top of the exception class.
Throwable have two subclasses.
One is **Exception** and other is **Error.**
The important subclass of Exception is **RuntimeException**

➔Exception in Java can be categorized into two types :

- **Unchecked exception** : The exceptions that are not checked at compile time are called unchecked exceptions. These exceptions are not essentially handled in the program code, instead the JVM handles such exceptions. Unchecked exceptions are extended from the **java.lang.RuntimeException** class.
- **Checked exception** : Exceptions that are checked at compile-time are called checked exceptions. These exceptions are explicitly handled in the code itself with the help of try-catch blocks. Checked exceptions are extended from the **java.lang.Exception** class. (Prof. Viral S. Patel)

➔Example :

```
class ThrowsDemo
{
    static void throwOne() throws IllegalAccessException
    {
         System.out.println("Inside throwOne.");
         throw new IllegalAccessException("demo");
    }

    public static void main(String args[])
    {
         try
         {
             throwOne();
         }
(        catch (IllegalAccessException e)
         {
             System.out.println("Caught " + e);
         }
         finally
         {
             System.out.println("Inside finally");
         }
    }
}
```

Output :

```
inside throwOne
caught java.lang.IllegalAccessException: demo
Inside finally
```

Que. Explain Chained Exception.

Ans. Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception.

For example, consider a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero but the **actual cause of exception was an I/O error** which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Constructors Of Throwable class Which support chained exceptions in java :

1. **Throwable(Throwable cause)** :- Where cause is the exception that causes the current exception.
2. **Throwable(String msg, Throwable cause)** :- Where msg is the exception message and cause is the exception that causes the current exception.

Methods Of Throwable class Which support chained exceptions in java :

1. **getCause()** method :- This method returns actual cause of an exception.
2. **initCause(Throwable cause)** method :- This method sets the cause for the calling exception.

Example of using Chained Exception:

```
import java.io.*;
class MyChainedException
{
   public static void main(String[] args)
   {
      try
      {
         ArithmeticException a = new ArithmeticException("Top Level Exception.");
         a.initCause(new IOException("IO cause."));

         throw a;
      }
      catch(ArithmeticException ae)
      {
         System.out.println("Caught : " + ae);
         System.out.println("Actual cause: "+ ae.getCause());
      }
   }
}
```

Output :

```
Caught: java.lang.ArithmeticException: Top Level Exception.
Actual cause: java.io.IOException: IO cause.
```

Exception Chained Using Methods :

Example:

```
class E1 extends Exception
{
}

class E2 extends Exception
{
}

class E
{
  public static void main(String args[])
  {
   try
   {
     E2 e2 = new E2();
     e2.initCause(new E1());
     throw e2;
   }
   catch(Exception e2)
   {
    System.out.println("Caught :"+e2);
    System.out.println("Actual cause :"+e2.getCause());
   }
  }
}
```

Output:

```
Caught:E2
Actual cause:E1

/* Asst. Prof. Viral S. Patel */
```

Exception Chained Using Constructors :

Example:

```
class E1 extends Exception
{
}

class E2 extends Exception
{
    E2(Throwable cause)
    {
      super(cause);
    }
    E2(String message, Throwable cause)
    {
      super(message, cause);
    }
}

class E
{
  public static void main(String args[])
  {
   try
   {
     E1 e1 = new E1();
     E2 e2 = new E2(e1);
     // E2 e2 = new E2("second exception",e1);
     throw e2;
   }
   catch(Exception e2)
   {
     System.out.println("Caught:"+e2);
     System.out.println("Actual cause:"+e2.getCause());
   }
  }
}
```

Output: (for calling one parameter constructor)

```
Caught:E2: E1
Actual cause:E1
```

[

Output: (for calling two parameters constructor)

```
Caught:E2 : second exception
Actual cause:E1
```

]

Que. Explain user defined exception in java.

Ans. In java we can create our **own exception class** and throw that exception using throw keyword. These exceptions are known as user-defined or custom exceptions. For that we have to

extends Exception class or RuntimeException class. We can override **toString( )** method of Object class to display error message which give description of exception during println( ) method is called.

Example : User Define Checked Exception

```
class GreaterTenException extends Exception
{
   private int detail;
   GreaterTenException(int a)
   {
     detail = a;
   }
   public String toString()
   {
     return "GreaterTenException["+detail+"]";
   }
}

class Demo
{
 static void takeSmallerTen(int a) throws
GreaterTenException
   {
     if(a<10)
       System.out.println(a);
     else
       throw new GreaterTenException(a);
   }

   public static void main(String args[])
   {
      try
      {
        takeSmallerTen(11);
      }
      catch(Exception e)
      {
        System.out.println(e);
      }
   }
}
```

Output:

```
GreaterTenException[11]

/* Asst. Prof. Viral S. Patel */
```

Example : User Define Unchecked Exception

```
import java.io.*;
class GreaterTenException extends RuntimeException
{
   private int detail;
   GreaterTenException(int a)
   {
     detail = a;
   }
   public String toString()
   {
     return "GreaterTenException["+detail+"]";
   }
}

class Demo
{
   static void takeSmallerTen(int a)
   {
     if(a<10)
       System.out.println(a);
     else
       throw new GreaterTenException(a);
   }

   public static void main(String args[])
   {
      int a=0;
      BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));

      try
      {
           System.out.print("Enter value (value<10):");
           a = Integer.parseInt(br.readLine());
           takeSmallerTen(a);
      }
      catch(Exception e)
      {
           System.out.println(e);
      }
   }
}
```

Output:

```
Enter value (value<10):11
GreaterTenException[11]
```

Que. Give difference between Multithreading and Process based Multitasking.
Ans.

Thread based Multitasking (Multithreading)	Process based Multitasking
It is a programming concept in which a program or a process is divided into two or more subprograms or threads that are executed at the same time in parallel.	It is an operating system concept in which multiple programs or processes are performed their task simultaneously. (Prof. Viral S. Patel)
It supports execution of multiple parts of a single program simultaneously	It supports execution of multiple programs simultaneously.
The processor has to switch between different parts or threads of a program	The processor has to switch between different programs or processes.
It is highly efficient	It is less efficient in comparison to multithreading
A thread is the smallest unit in multithreading	A program or process is the smallest unit in a multitasking environment.
It helps in developing efficient programs.	It helps in developing efficient operating system.
It is cost-effective in case of context switching.	It is expensive in case of context switching.

Que. What is Thread ? Give methods of thread class.

Ans. Thread is basically a lightweight sub-process, a smallest unit of processing. A thread have its own execution stack and program counter. Multithreading in java is a process of executing multiple threads simultaneously. (Prof. Viral S. Patel)

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

Que. Give any Four Thread class Constructors.

Ans.

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

Que. How can we create thread (or multiple threads) in java ?
Ans. We can create thread by creating object of Thread class. Two ways in which this can be done:

1) **By implement the Runnable interface.**
2) **By extend the Thread class.**

➔To implement Runnable, a class need only implement a single method called run( ).

public void run( )

Inside run( ), we can write the code which executed by the new thread.
A class that implements Runnable have to create object of Thread class.

We can use one of the following constructors of Thread class in the concept of implement Runnable interface.

Thread(Runnable threadOb);
Thread(Runnable threadOb, String threadName);

After creating object of Thread, we can call **start( )** method which call to run( ) method to execute the thread. (Prof. Viral S. Patel)

Example :

```
class MyThread implements  Runnable
{
  Thread t;

  MyThread(String threadName)
  {
    t = new Thread(this,threadName);
    t.start();
  }

  public void run()
  {
      …..
  }
}
```

```
class Demo
{
  public static void main(String args[])
  {
     MyThread t1 = new MyThread("Child1");
     MyThread t2 = new MyThread("Child2");
  }
}
```

➔The second way to create a new class that **extends Thread class** and then create an object of that class. The extending class must **override the run( ) method.**

```
class MyThread extends Thread
{
  MyThread(String threadName)
  {
    super(threadName);
    start();
  }

  public void run()
  {
      …..
  }
}
```

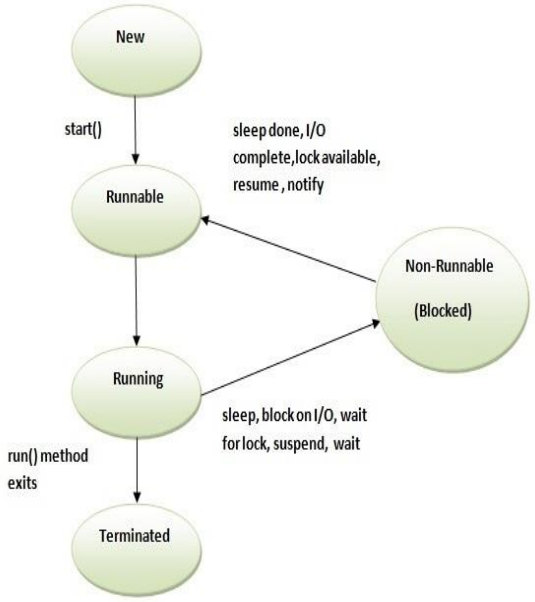
```
class Demo
{
  public static void main(String args[])
  {
     MyThread t1 = new MyThread("Child1");
     MyThread t2 = new MyThread("Child2");
  }
}
```

Note : super call to Thread class's constructor which is **Thread(String threadName);**

Que. Which methods are inherited in Thread class from class java.lang.Object ?
Ans. equals, finalize, notify, notifyAll, wait methods are inherited in Thread class from class java.lang.Object

Que. Explain LifeCycle of Thread.
Ans. A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.
➔But for better understanding the threads, we are generally explaining it in the 5 states. The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable (ready-to-run)
3. Running
4. Non-Runnable (Blocked,Wait,Sleep)
5. Terminated (Dead)



1) New
The thread is in new state if you create an instance of Thread class but before the invocation of start() method. (Prof. Viral S. Patel)
2) Runnable
The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
➔**yield()** is used to give the other threads of the same priority a chance to execute i.e. causes current running thread to move to runnable state.
➔**notify()**: This wakes up threads that called wait() on the same object and moves the thread to ready state.
➔**notifyAll()**: This wakes up all the threads that called wait() on the same object.

3) Running
   The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)
   This is the state when the thread is still alive, but is currently not eligible to run.
   ➔When **wait()** method is invoked on an object, the thread executing that code gives up its lock on the object immediately and moves the thread to the wait state.
   ➔**sleep()** is used to pause a thread for a specified period of time

5) Terminated
   A thread is in terminated or dead state when its run() method exits

Que. Explain Synchronization in Multithreading.
Ans.
➔ If process (or method) is **Asynchronous** then it does not guarantee thread-safety. As resource inside the asynchronous method accessed by multiple threads can be brought in inconsistent state. (Prof. Viral S. Patel)
➔When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization.**
➔Key to synchronization is the concept of the **monitor (also called a semaphore)**. A monitor is an object that is used as a **mutually exclusive lock**, or **mutex**.
➔As long as the thread holds the monitor, no other thread can enter the synchronized section of code. other threads are said to be waiting for the monitor.
➔In java **'synchronized'** keyword is used with the **method** to enter the thread in monitor.
➔It is also possible to mark a **block** of code as synchronized .

synchronized void method( )
{
……. // code here is synchronized
}
	synchronized (lock-object)
	{
	…… //code here is synchronized
	}

➔Example of Synchronization using synchronized method and synchronized block:

```
class MyResource
{

synchronized void useRes()
 {
   System.out.print("[");

   try
   {
     Thread.sleep(1000);
   }
   catch(Exception e)
   {
   }

   System.out.print("]");
 }

}
```

```
class MyThread implements
Runnable
{
  Thread t;
  MyResource res;

  MyThread(MyResource r)
  {
    res = r;
    t = new Thread(this);
    t.start();
  }

  public void run()
  {
      res.useRes();
  }

}
```

```
class Demo
{
  public static void main(String args[])
  {
     MyResource ob = new MyResource();

     MyThread t1 = new MyThread(ob);
     MyThread t2 = new MyThread(ob);
  }
}

Output:
[ ] [ ]


Note: If we not write synchronized
keyword before void users() method then
output will be [[]].
```

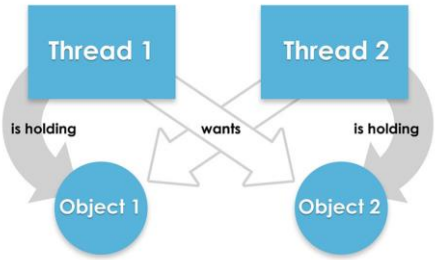
```
class MyResource
{

 void useResource()
 {
   System.out.print("[");

   try
   {
     Thread.sleep(1000);
   }
   catch(Exception e)
   {
   }

   System.out.print("]");
 }

}
```

```
class MyThread implements
Runnable
{
  Thread t;
  MyResource res;
  MyThread(MyResource r)
  {
    res = r;
    t = new Thread(this);
    t.start();
  }
  public void run()
  {
    synchronized(res)
    {
      res.useResource();
    }
  }
}
```

```
class Demo
{
  public static void main(String args[])
  {
     MyResource ob = new MyResource();

     MyThread t1 = new MyThread(ob);
     MyThread t2 = new MyThread(ob);
  }
}

Output:
[ ] [ ]
```

➔Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.
➔A Java multithreaded program may suffer from the **deadlock condition because the synchronized keyword,** 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.

Que. Explain Deadlock in java.
Ans. (Prof. Viral S. Patel)
An interesting situation may occur when two or more threads are waiting to gain control of a resource of each other. A Java multithreaded program may suffer from the **deadlock condition because the synchronized keyword,** 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.**



We can see deadlock in synchronized method concept.

```
Thread A
…..
resource1.method1(resource2);  // hold resource1
…..
synchronized void method1(resource2)
{
      ……
      try { Thread.sleep(100);} catch (Exception e) {}
      resource2.function2( );   // need resource2
      ……
}
synchronized void method2( )
{
…..
}
```

```
Thread B
…….
resource2.function1( resource1);  //hold resource2
…..
synchronized void function1(resource1)
{
      ……...
      try { Thread.sleep(100);} catch (Exception e) {}
      resource1.method2( );   // need resource1
      ……
}
synchronized void function2( )
{
…..
}
```

We can also see deadlock in synchronized block concept. Example :

Thread A	Thread B
…..	…..
synchronized(**resource1**)	synchronized(**resource2**)
{	{
……	…….
try { Thread.sleep(100);} **catch** (Exception e) {}	**try** { Thread.sleep(100);} **catch** (Exception e) {}
synchronized(**resource2**)	synchronized(**resource1**)
{	{
……	…….
}	}
}	}
…..	…..

Example of Deadlock : (Prof. Viral S. Patel)

```
class Res1
{
   synchronized void task1(Res2 r2)
   {
        System.out.print("[");
        try
        {
         Thread.sleep(1000);
        }
        catch(Exception e)
        {
        }
        r2.fun2();
        System.out.print("]");
   }
   synchronized void task2()
   {
        System.out.println("tast2 of Res1");
   }
}
```

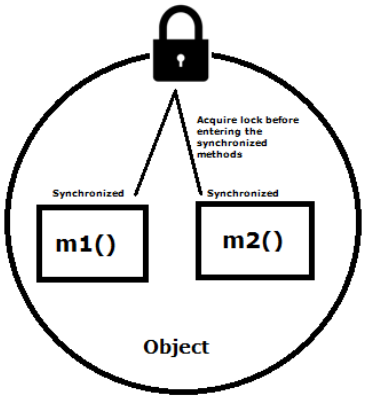
```
class Res2
{
   synchronized void fun1(Res1 r1)
   {
        System.out.print("(");
        Try
        {
         Thread.sleep(1000);
        }
        catch(Exception e)
        {
        }
        r1.task2();
        System.out.print(")");
   }
   synchronized void fun2()
   {
        System.out.println("fun2 of Res2");
   }
}
```

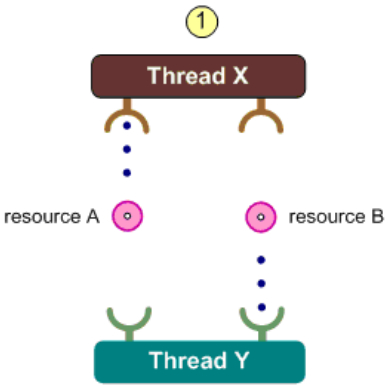
```
class MyThread implements Runnable
{
  Thread t;
  Res1 re1;
  Res2 re2;

  MyThread(Res1 r1, Res2 r2)
  {
        re1 = r1; re2 = r2;
        t = new Thread(this);
        t.start();
  }

  public void run()
  {
   if(t.getName().equals("Thread-0"))
     re1.task1(re2); //Thread-0 hold re1 & need re2
   else
     re2.fun1(re1); //Thread-1 hold re2 & need re1
  }
}
```

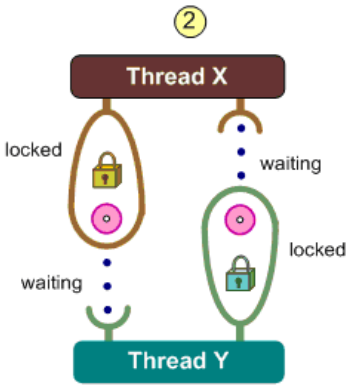
```
class Demo
{
   public static void main(String args[])
   {
        Res1 ob1 = new Res1();
        Res2 ob2 = new Res2();

        MyThread t1 = new MyThread(ob1,ob2);
        MyThread t2 = new MyThread(ob1,ob2);
   }
}

Output:
[(
```

Note : Thread-0 hold re1 and need re2 to complete its task1. Same as other side Thread-1 hold res2 and need res1 to complete its fun1. So deadlock generate after display [(.

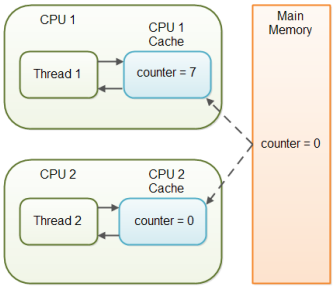
Que. How mutex can be achieved in thread ?
Ans. Mutex in thread at a time only one thread lock the particular region of code. If another thread want to lock the same region of code it must have to wait until first lock release. This can be **achieved by java synchronized method or synchronized block**.







Que. Write down the use of volatile keyword.
Ans. Volatile is used to indicate that a variable's value will be modified by different threads and achieving synchronization in java in some cases, like visibility. Without volatile keyword different reader thread may see different values as compiler re-order the code, free to cache value instead of always reading from main memory. So volatile keyword in java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.



Que. Explain Inter-Thread Communication.
Ans. Inter-thread communication can be defined as the **exchange of messages between two or more threads**. The transfer of messages takes place before or after the change of state of thread. For example, an active thread may notify to another suspended thread just before switching to the suspend state. Java implements inter-thread communication with the help of following three methods :

final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )

wait() : Tells the calling thread to give up the monitor and Sends the calling thread into the sleep mode. This thread can now be activated only by notify or notifyall() methods.

One can also specify the time for which the thread has to wait. The desired waiting time period is specified as an argument to the wait() method.

notify() : Resumes the first thread that went into the sleep mode. (Prof. Viral S. Patel)

notifyall() : Resumes all the threads that are in sleep mode. The execution of these threads happens as per priority. (Prof. Viral S. Patel)

These methods are declared within **Object class**. Since the methods are declared as final they can not be overridden.

```
class MyResource
{
 int n;
 boolean valueSet = false;

 synchronized void put(int n)
 {
  if(valueSet)
   try { wait(); } catch(Exception e){…}

  this.n = n;
  valueSet = true;
  System.out.println("Put: " + n);
  notify();
 }


 synchronized void get()
 {
  if(!valueSet)
    try { wait(); } catch(Exception e){…}

  System.out.println("Got: " + n);
  valueSet = false;
  notify();
 }


}
```

```
class Producer implements Runnable
{
  MyResource res;
  Thread t;
  Producer(MyResource r)
  {
    res = r;
    t = new Thread(this, "Producer");
    t.start();
  }
  public void run()
  {
    for(int i=0;i<3;i++)
    {
      res.put(i*2);
    }
  }
}

class Consumer implements Runnable
{
  MyResource res;
  Thread t;
  Consumer(MyResource r)
  {
    res = r;
    t = new Thread(this, "Consumer");
    t.start();
  }
  public void run()
  {
    for(int i=0;i<3;i++)
    {
      res.get();
    }
  }
}
```

```
class PCFixed
{
   public static void main(String args[])
   {
   MyResource ob = new MyResource();
   new Producer(ob);
   new Consumer(ob);
   }
}
```

Output :

```
Put: 0
Got: 0
Put: 2
Got: 2
Put: 4
Got: 4
```

Que. Explain join( ) and isAlive( ) methods in Thread concept.
Ans. Two ways exist to determine whether a thread has finished.

First, we can call isAlive( ) on the thread. This method is defined by Thread, and its general form is :

final boolean isAlive( )

The isAlive( ) method returns true if the thread upon which it is called is still running.

It returns false otherwise.

While isAlive( ) is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called join( ), shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. (Prof. Viral S. Patel)

Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

final void join(long miliseconds) throws InterruptedException

```
class MyThread implements Runnable
{
    Thread t;

    MyThread(String tnam)
    {
       t = new Thread(this,tnam);
       t.start();
    }

    public void run()
    {
       for(int i=0;i<3;i++)
       {
   System.out.println(t.getName()+":"+i);
       }
    }

}
```

Output :

```
main start
c1 isAlive :true
c2 isAlive :true
c1:0
c2:0
c1:1
c2:1
c1:2
c2:2
c1 isAlive :false
c2 isAlive :false
main exit
```

```
class Demo
{
    public static void main(String args[])
    {
        System.out.println("main Start");
        MyThread t1 = new MyThread("c1");
        MyThread t2 = new MyThread("c2");

 System.out.println("c1 isAlive :" + t1.t.isAlive());
 System.out.println("c2 isALive :" + t2.t.isAlive());

        try
        {
           t1.t.join();
           t2.t.join();
        }
        catch(InterruptedException e)
        {
           System.out.println("Error caught");
        }

 System.out.println("c1 isAlive :" + t1.t.isAlive());
 System.out.println("c2 isALive :" + t2.t.isAlive());

          System.out.println("main exit");
    }
}
```

Note: we can see c1 and c2 thread exit before main thread exit. main wait until c1 and c2 complete their task because threads c1 and c2 call to join( ) method in main thread.

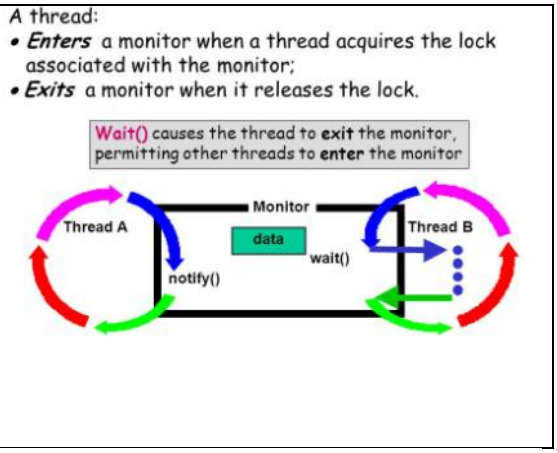
Que. What is Daemon Thread in Java ?
Ans. Daemon thread in java is **service provider thread** that provides services to the user thread. Its life depend on the mercy of user threads i.e. **when all the user threads dies, JVM terminates this thread automatically.** It is a **low priority thread**. There are many java daemon threads running automatically e.g. gc, finalizer etc.

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

No.	Method	Description
1)	public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
2)	public boolean isDaemon()	is used to check that current thread is daemon.

Que. Define monitor.
Ans.

The monitor is a control mechanism can think of a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time. Each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.



Que. Difference between wait and sleep method in Java Thread concept.
Ans.

No.	wait( )	sleep( )
1)	wait( ) method releases the lock during synchronization.	sleep( ) method doesn't release the lock during synchronization.
2)	It is a method of Object class.	It is the method of Thread class.
3)	It is the non-static method.	It is the static method. So it can call by class name like Thread.sleep(1000).
4)	To start thread again from wait( ), Object have to call notify( ) or notifyAll( ) method.	In sleep( ), thread gets start after specified time (ms/sec) interval or by interrupt.

Que. What is the function of yield( ) method in java ?
Ans. yield() pauses the current thread to give a chance to execute other threads of the same priority i.e. causes current running thread to move to runnable state.