Inter Thread Communication

Inter-thread communication in Java is a technique through which multiple threads communicate with each other.

It provides an efficient way through which more than one thread communicate with each other by reducing CPU idle time. CPU idle time is a process in which CPU cycles are not wasted.

When more than one threads are executing simultaneously, sometimes they need to communicate with each other by exchanging information with each other. A thread exchanges information before or after it changes its state.

How to achieve Inter thread communication in Java

Inter thread communication in Java can be achieved by using three methods provided by Object class of java.lang package. They are:

1. **wait()**
2. **notify()**
3. **notifyAll()**

These methods can be called only from within a synchronized method or synchronized block of code otherwise, an exception named **IllegalMonitorStateException** is thrown.

wait() Method in Java

wait() method in Java notifies the current thread to give up the monitor (lock) and to go into sleep state until another thread wakes it up by calling notify() method. This method throws InterruptedException.

Note:

1. A monitor is an object which acts as a lock. It is applied to a thread only when it is inside a synchronized method.

2. Only one thread can use monitor at a time. When a thread acquires a lock, it enters the monitor.

3. When a thread enters into the monitor, other threads will wait until first thread exits monitor.

4. A lock can have any number of associated conditions.

notify() Method in Java

The notify() method wakes up a single thread that called wait() method on the same object. If more than one thread is waiting, this method will awake one of them.

notifyAll() Method in Java

The notifyAll() method is used to wake up all threads that called wait() method on the same object.
The thread having the highest priority will run first.

```
public class A
{
int i;
     synchronized void deliver(int i)
     {
       this.i = i;
       System.out.println("Data Delivered: " +i);
     }
     synchronized int receive()
     {
       System.out.println("Data Received: " + i);
       return i;
     }
}
```

```
public class Thread1 extends Thread
{
 A obj;
 Thread1(A obj)
 {
  this.obj = obj;
 }
public void run()
{
 for(int j = 1; j <= 5; j++){
   obj.deliver(j);
  }
 }
}
```

```
public class Thread2 extends Thread
```

```
{
A obj;
Thread2(A obj)
{
 this.obj = obj;
}
public void run()
{
for(int k = 0; k <= 5; k++){
 obj.receive();
}
 }
}


public class NoCommunication
{
public static void main(String[] args)
{
A obj = new A();
Thread1 t1 = new Thread1(obj);
Thread2 t2 = new Thread2(obj);
  t1.start();
  t2.start();
 }
}
```

```
// In this program, we will understand how to use wait and notify.
// It is the most efficient way for thread communication.
public class A
      {
       int i;
       boolean flag = false; // flag will be true when data production is over.
      synchronized void deliver(int i)
      {
       if(flag)
       try
       {
        wait(); // Wait till a notification is received from Thread2. There will be no
wastage of time.
       }
       catch(InterruptedException ie)
       {
        System.out.println(ie);
       }
         this.i = i;
         flag = true; // When data production is over, it will store true into flag.
         System.out.println("Data Delivered: " +i);
         notify(); // When data production is over, it will notify Thread2 to use it.
 }
synchronized int receive()
{
if(!flag)
try {
 wait(); // Wait till a notification is received from Thread1.
```

```
}
catch(InterruptedException ie){
 System.out.println(ie);
}
 System.out.println("Data Received: " + I);
  flag = false; // It will store false into flag when data is received.
  notify(); // When data received is over, it will notify Thread1 to produce next dat
a.
  return i;
 }
}
public class Thread1 extends Thread
{
 A obj;
 Thread1(A obj)
 {
  this.obj = obj;
 }
public void run()
{
for(int j = 1; j <= 5; j++){
 obj.deliver(j);
  }
}}
public class Thread2 extends Thread
{
A obj;
Thread2(A obj)
{
 this.obj = obj;
}
public void run()
{
for(int k = 0; k <= 5; k++){
 obj.receive();
}
 }}
public class Communication
{
public static void main(String[] args)
{
 A obj = new A(); // Creating an object of class A.

// Creating two thread objects and pass reference variable obj as parameter to Thread1
and Thread2.
Thread1 t1 = new Thread1(obj);
Thread2 t2 = new Thread2(obj);
// Run both threads.
  t1.start();
  t2.start();
 }
}
```

Why wait(), notify() and notifyAll() methods are defined inside Object class, not Thread class?

This is because all these three methods are related to lock and Java provides lock at Object level, not at the Thread level. Therefore, they are defined in Object class.

Difference between wait() and sleep() in Java

Object.wait() and Thread.sleep() are entirely two different methods in Java that are used in two different contexts. Here, we have listed a few differences between these two methods.

1. wait() method is always called from synchronized block otherwise, it will throw IllegalMonitorStateException whereas sleep() method can be called from any code block.

2. wait() method releases the acquired lock while sleep() method does not release the lock.

3. wait() is called on Object whereas sleep() is called on Thread.

4. waiting thread can be awakened by invoking notify()/notifyAll() methods while sleeping thread cannot be awakened.

5. wait is a non-static method while sleep is a static method.

Daemon Thread

Daemon thread in Java is a service thread that provides some services to other threads or objects.

It executes continuously without any interruption to provide services to one or more user threads.

It typically runs in the background. Once the daemon thread started, it continues to provide that service until the application terminates. Its termination is automatic.

The garbage collector of JVM process always runs in the background for freeing memory of unused objects. If the garbage collector is only thread running, JVM does not continue the execution process and it exits.

Key Points to Remember for Daemon Thread

1. Daemon thread has no other role in life rather than to serve user threads.

2. Since the life span of daemon thread depends on the user thread, therefore, it provides services to user threads in the background as long as the program is running.

3. Daemon thread is a low-priority thread.

4.Java supports two general types of threads: user threads (normal threads) and daemon threads. The difference between two is that a user thread will continue to execute until its run() method ends while a daemon thread will be automatically ended when all user threads of a program have ended.

5. If only daemon thread is running a program and all user threads have completed their execution, JVM will exit from the program because daemon thread will be ended automatically.

```
public class MyDaemon implements Runnable
{
public void run()
{
// Checking whether a thread is Daemon or not
   if(Thread.currentThread().isDaemon()) {
       System.out.println(Thread.currentThread() + " is a daemon thread");
   }
   else {
      System.out.println(Thread.currentThread() + " is a user (normal) thread");
   }
}
public static void main(String[] args)
{
  MyDaemon obj = new MyDaemon();
  Thread t1 = new Thread(obj, "Thread 1");
    t1.setDaemon(true); // Set to daemon.

  Thread t2 = new Thread(obj, "Thread 2");
   t1.start(); // Execution starts.
   t2.start();

  System.out.println("Main thread ending");
 }
}
```

```
public class MyDaemon implements Runnable
{
public void run()
{
  System.out.println(Thread.currentThread() + " is a daemon thread");
}
public static void main(String[] args)
{
```

```
MyDaemon obj = new MyDaemon();
Thread t1 = new Thread(obj, "Thread 1");

  t1.start(); // Execution starts.
  t1.setDaemon(true); // It will throw IllegalThreadStateException.

  System.out.println("Main thread ending");
 }
}
```

Volatile Keyword in Java

Volatile keyword in Java is a non-access modifier that can be applied with a variable declaration.

It tells the compiler the value of variable may change at anytime.

Volatile Variable in Java

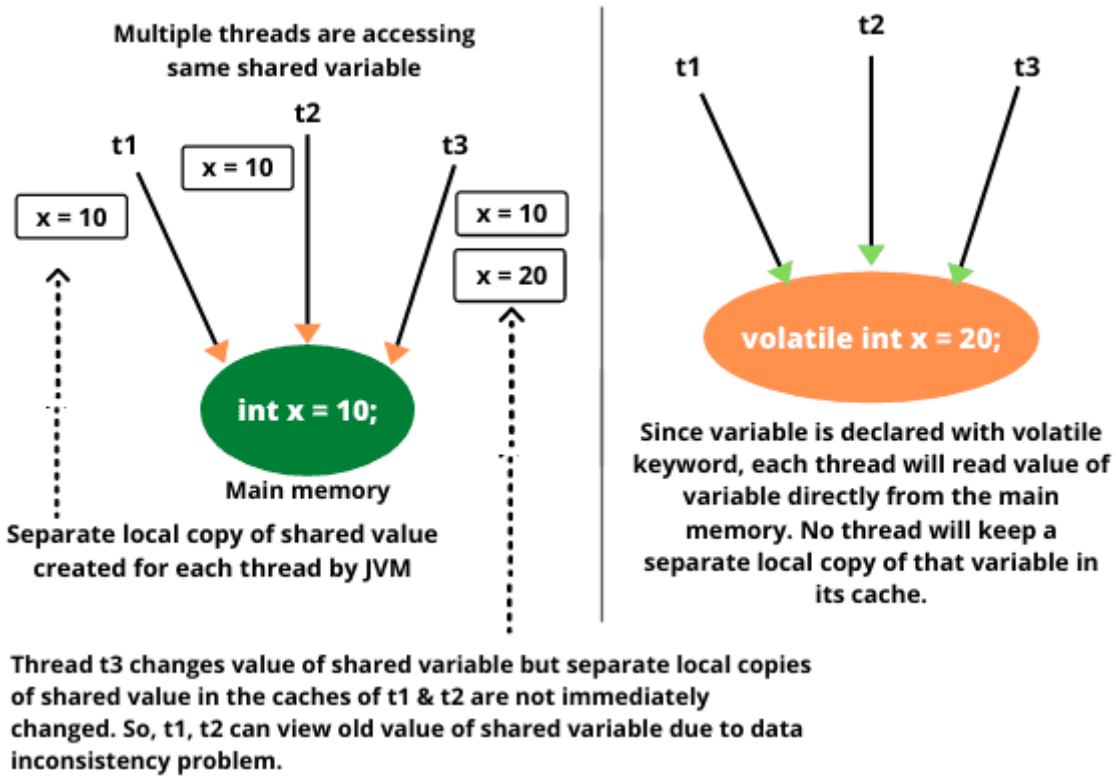
Java allows threads to access the shared variable.

When a thread changes the value of shared variable, it is possible that another thread may see the old value of shared variable for some time. This is because threads are allowed to cache
shared data.

That is, each thread keeps its own separate local copy of the shared value. When a thread changes the value of shared variable, the separate local copies of shared value in the caches of
other threads are not immediately changed.

Therefore, other threads can continue to view the old value of shared variable due to data inconsistency problems.

However, in the case of a synchronized method or block, threads are forced to update their caches with the most current values of the variables. So, it is always safe while using shared variable inside synchronized code.



When to use Volatile Keyword?

There are the following reasons to use volatile keyword in Java programs. They are:

1.A volatile keyword is only useful when we are working a multi-threaded environment and the value of variables is to be shared among threads. There is no use of volatile keyword in the non-multithreading environment.

2. We can use a volatile variable to stop a thread by using the value of variable as a flag. If the flag is set then thread will continue to run. If another thread clears the flag, thread will
stop.

Since two threads share the flag, so the thread will get its updated value from the main memory on every read.

3. It can be used as an alternative way of performing synchronization in Java.

4.All threads will read the updated value of volatile variable after completion of the write operation. If we do not use the volatile keyword, different threads may read different values.

```
public class A extends Thread
{
// Declare a volatile variable named flag with initial value true.
 public volatile boolean flag = true;

@Override
public void run(){
 System.out.println("Thread starts running");

// Since the flag is volatile, so for every read, thread will read its latest value fr
om main memory.
 while(flag)
 {
  try{
  System.out.println("Thread going to sleep");
  Thread.sleep(1000);
  }
 catch(InterruptedException ie){
  ie.printStackTrace();
 }
  }
 System.out.println("Thread stopped...");
}
public void stopThread(){
  this.flag = false;
 }
}
public class MyThread {
public static void main(String[] args) {
 A a = new A();
 Thread t = new Thread(a);
 t.start(); // Thread started.

try {
 Thread.sleep(3000); // Main thread will sleep for 3 seconds.
}
catch(InterruptedException ie){
 ie.printStackTrace();
}
// Stop thread.
// Call stopThread() using reference variable a.
   a.stopThread();
  }
}
```