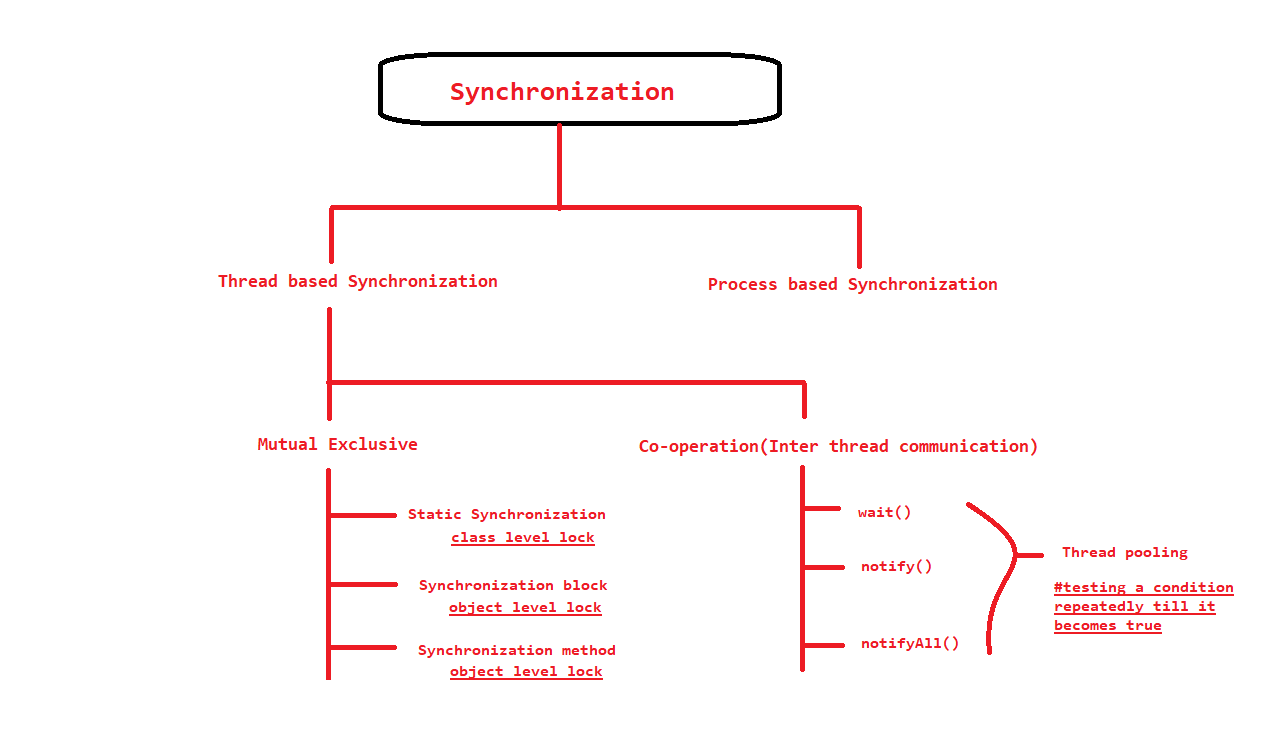
Inter-thread Communication in Java



**What is Polling and what are problems with it?**  
The process of testing a condition repeatedly till it becomes true is known as polling.

Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain action is taken. This waste many CPU cycles and makes the implementation inefficient.  
For example, in a classic queuing problem where one thread is producing data and other is consuming it.

**How Java multi threading tackles this problem?**  
To avoid polling, Java uses three methods, namely, **wait(), notify() and notifyAll().**  
All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

* **wait()-**It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
* **notify()-**It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.
* **notifyAll()-**It wakes up all the threads that called wait() on the same object.

**A simple Java program to demonstrate the three methods-**  
Please note that this program might only run in offline IDEs as it contains taking input at several points.

|  |
| --- |
| // Java program to demonstrate inter-thread communication  // (wait(), join() and notify()) in Java  import java.util.Scanner;  public class threadexample  {      public static void main(String[] args)                             throws InterruptedException      {          final PC pc = new PC();            // Create a thread object that calls pc.produce()          Thread t1 = new Thread(new Runnable()          {              @Override              public void run()              {                  try                  {                      pc.produce();                  }                  catch(InterruptedException e)                  {                      e.printStackTrace();                  }              }          });            // Create another thread object that calls          // pc.consume()          Thread t2 = new Thread(new Runnable()          {              @Override              public void run()              {                  try                  {                      pc.consume();                  }                  catch(InterruptedException e)                  {                      e.printStackTrace();                  }              }          });            // Start both threads          t1.start();          t2.start();            // t1 finishes before t2          t1.join();          t2.join();      }        // PC (Produce Consumer) class with produce() and      // consume() methods.      public static class PC      {          // Prints a string and waits for consume()          public void produce()throws InterruptedException          {              // synchronized block ensures only one thread              // running at a time.              synchronized(this)              {                  System.out.println("producer thread running");                    // releases the lock on shared resource                  wait();                    // and waits till some other method invokes notify().                  System.out.println("Resumed");              }          }            // Sleeps for some time and waits for a key press. After key          // is pressed, it notifies produce().          public void consume()throws InterruptedException          {              // this makes the produce thread to run first.              Thread.sleep(1000);              Scanner s = new Scanner(System.in);                // synchronized block ensures only one thread              // running at a time.              synchronized(this)              {                  System.out.println("Waiting for return key.");                  s.nextLine();                  System.out.println("Return key pressed");                    // notifies the produce thread that it                  // can wake up.                  notify();                    // Sleep                  Thread.sleep(2000);              }          }      }  } |

Output:

producer thread running

Waiting for return key.

Return key pressed

Resumed

As monstrous as it seems, it really is a piece of cake if you go through it twice.

1. In the main class a new PC object is created.
2. It runs produce and consume methods of PC object using two different threads namely t1 and t2 and wait for these threads to finish.

Lets understand how our produce and consume method works.

* First of all, use of synchronized block ensures that only one thread at a time runs. Also since there is a sleep method just at the beginning of consume loop, the produce thread gets a kickstart.
* When the wait is called in produce method, it does two things. Firstly it releases the lock it holds on PC object. Secondly it makes the produce thread to go on a waiting state until all other threads have terminated, that is it can again acquire a lock on PC object and some other method wakes it up by invoking notify or notifyAll on the same object.
* Therefore we see that as soon as wait is called, the control transfers to consume thread and it prints -“Waiting for return key”.
* After we press the return key, consume method invokes notify(). It also does 2 things- Firstly, unlike wait(), it does not releases the lock on shared resource therefore for getting the desired result, it is advised to use notify only at the end of your method. Secondly, it notifies the waiting threads that now they can wake up but only after the current method terminates.
* As you might have observed that even after notifying, the control does not immediately passes over to the produce thread. The reason for it being that we have called Thread.sleep() after notify(). As we already know that the consume thread is holding a lock on PC object, another thread cannot access it until it has released the lock. Hence only after the consume thread finishes its sleep time and thereafter terminates by itself, the produce thread cannot take back the control.
* After a 2 second pause, the program terminates to its completion.

If you are still confused as to why we have used notify in consume thread, try removing it and running your program again. As you must have noticed now that the program never terminates.  
The reason for this is straightforward-When you called wait on produce thread, it went on waiting and never terminated. Since a program runs till all its threads have terminated, it runs on and on.  
There is a second way round this problem. You can use a second variant of wait().

void wait(long timeout)

This would make the calling thread sleep only for a time specified.