Synchronization in Java

Threads can communicate among themselves with the help of inter-thread communication methods. They are :

**sleep()-** causes a running thread to go into the **blocked(i.e not-runnable)** state. When the time specified as the argument of **sleep()** method is elapsed, the blocked thread will return to the runnable state.

**suspend()-** this method causes a running thread to go into the **blocked(i.e not-runnable**) state and remain in that state until further orders. When the **resume()** method is executed, the blocked thread will return to the runnable state.

**wait()-** this method causes a running thread to go into the **blocked(i.e not-runnable)** state and remain in that state until a specified condition occurs. When the **notify()** method is executed, the blocked thread will return to the runnable state.

To block a thread use any of the three methods

**sleep()**

**suspend()**

**wait()**

To revoke a blocked method

**resume()**

**notify()**

|  |
| --- |
| class ItemQ  {  int n;  synchronized intget()  {  System.out.println("Got: "+n);  return n;  }  synchronized void put(int n)  {  this.n=n;  System.out.println("Put: "+n);  }  }  class Producer extends Thread  {  ItemQ q;  Producer(ItemQ q1)  {  this.q=q1;  }  public void run()  {  inti=0;  while(true)  {  try  {  q.put(i++);  }  catch(Exception e)  {}  }  }  }  class Consumer extends Thread  {  ItemQ q;  Consumer(ItemQ q1)  {  this.q=q1;  }  public void run()  {  while(true)  {  try  {  q.get();  }  catch(Exception e)  {}  } //while  } //run  }  class producer\_consumer4  {  public static void main(String args[])  {  ItemQ q=new ItemQ();  Producer p1=new Producer(q);  Consumer c1=new Consumer(q);  p1.start();  c1.start();  System.out.println("Press control-C to stop");  }  } |

|  |
| --- |
| Put: 1  Got: 1  Got: 1  Got: 1  Got: 1  Got: 1  Put: 2  Put: 3  Put: 4  Put: 5  Put: 6  Put: 7  Got: 7 |

Although the **put( )**and **get( )** methods on **Q** are synchronized, nothing stops the producer

from overrunning the consumer, nor will anything stop the consumer from consuming the

same queue value twice. Thus, you get the erroneous output shown here (the exact output

will vary with processor speed and task load):

As you can see, after the producer put 1, the consumer started and got the same 1 five times

in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer

have a chance to consume them.

The proper way to write this program in Java is to use **wait( )**and **notify( )** to signal in

both directions, as shown here:

|  |
| --- |
| //producer consumer with wait() and notify()  class ItemQ  {  int n;  boolean flag=false;  synchronized intremove()  {  if(!flag)  {  try  {  wait();  }  catch(InterruptedException e)  {  System.out.println("Interrupted Exception caught");  } //catch  } //if  System.out.println("item removed: "+n);  flag=false;  notify();  return n;  } //remove  synchronized void add(int n)  {  if(flag)  {  try  {  wait();  }  catch(InterruptedException e)  {  System.out.println("Interrupted Exception caught");  }  }  this.n=n;  flag=true;  System.out.println("item added: "+n);  notify();  } //add  } // ItemQ  class Producer extends Thread  {  ItemQ q;  Producer(ItemQ q1)  {  this.q=q1;  }  public void run()  {  inti=0;  while(true)  {  q.add(i++);  } //while  } //run  } //Producer  class Consumer extends Thread  {  ItemQ q;  Consumer(ItemQ q1)  {  this.q=q1;  }  public void run()  {  while(true)  {  q.remove();  } //while  } //run  } //Consumer  class producer\_consumer5  {  public static void main(String args[])  {  ItemQ q=new ItemQ();  Producer p1=new Producer(q);  Consumer c1=new Consumer(q);  p1.start();  c1.start();  System.out.println("Press control-C to stop");  }  } |

Inside **remove( )**, **wait( )** is called. This causes its execution to suspend until the **Producer**

notifies you that some data is ready. When this happens, execution inside **remove( )**resumes.

After the data has been obtained, **remove( )**calls **notify( )**. This tells **Producer** that it is okay to

put more data in the queue. Inside **add( )**, **wait( )** suspends execution until the **Consumer**

has removed the item from the queue. When execution resumes, the next item of data is put

in the queue, and **notify( )**is called. This tells the **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

|  |
| --- |
| E:\javaprgs\thread>java -cp . producer\_consumer5  Press control-C to stop  item added: 0  item removed: 0  item added: 1  item removed: 1  item added: 2  item removed: 2  item added: 3  item removed: 3  item added: 4  item removed: 4  item added: 5  item removed: 5  item added: 6  item removed: 6  item added: 7  item removed: 7  item added: 8  item removed: 8  item added: 9  item removed: 9  item added: 10  item removed: 10  item added: 11  item removed: 11 |

Another example for inter-thread communication

|  |
| --- |
| class Chat  {  boolean flag=false;  public synchronized void Answer(String msg)  {  if(!flag)  {  try  {  wait();  } //try  catch(InterruptedException e)  {  System.out.println("error is "+e);  } //catch  } //if  System.out.println(msg);  flag=false;  notify();  }//Answer  public synchronized void Question(String msg)  {  if(flag)  {  try  {  wait();  } //try  catch(InterruptedException e)  {  System.out.println("error is "+e);  } //catch  } //if  System.out.println(msg);  flag=true;  notify();  }//Question  }//Chat  class T1 implements Runnable  {  Chat m;  String[] s1={"hi","how are you?","i am also doing fine"};  public T1(Chat m1)  {  this.m=m1;  new Thread(this,"Question").start();  }  public void run()  {  for(inti=0;i<s1.length;i++)  m.Question(s1[i]);  } //run  } //T1  class T2 implements Runnable  {  Chat m;  String[] s2={"hi","I am good, what about you","Great"};  public T2(Chat m1)  {  this.m=m1;  new Thread(this,"Answer").start();  }  public void run()  {  for(inti=0;i<s2.length;i++)  m.Answer(s2[i]);  } //run  } //T2  public class producer\_consumer6  {  public static void main(String args[])  {  Chat m=new Chat();  new T1(m);  new T2(m);  }  } |

Output

|  |
| --- |
| :\javaprgs\thread>javac producer\_consumer6.java  E:\javaprgs\thread>java -cp . producer\_consumer6  hi  hi  how are you?  I am good, what about you  i am also doing fine  Great |

Suspending , resuming and stopping of threads

**Understandings suspend, resume, and stop methods in java thread**

* **step() method in Thread** – This method terminates the thread execution. Once a thread is stopped, it cannot be restarted with the start() method, because stop() method terminates the execution of a thread.
* **suspend() method in Thread** – If you want to stop the thread execution and start it again when a certain event occurs. In this case, suspend() method allows a thread to temporarily cease execution.
* **resume() method in Thread** – resume() method works with suspend() method. Resume() method allows the suspended thread to start again.

|  |
| --- |
| //program for suspending, resuming and stopping threads  class newThread implements Runnable  {  String name;  Thread t;  newThread(String tname)  {  name=tname;  t=new Thread(this,name);  System.out.println("New thread: "+t);  t.start();  }  public void run()  {  try  {  for(int i=15;i>0;i--)  {  System.out.println(name + ": "+i);  Thread.sleep(200);  }  }  catch(InterruptedException e)  {  System.out.println(name+" interrupted");  } //catch  System.out.println(name+ "exiting");  }  }  class suspendresume  {  public static void main(String args[])  {  newThread ob1=new newThread("one");  newThread ob2=new newThread("two");  try  {  Thread.sleep(1000);  ob1.t.suspend();  System.out.println("Suspending thread one");  Thread.sleep(1000);  ob1.t.resume();  System.out.println("Resuming thread one");  ob2.t.suspend();  System.out.println("Suspending thread two");  Thread.sleep(1000);  ob2.t.resume();  System.out.println("Resuming thread two");  }  catch(InterruptedException e)  {  System.out.println("Main thread interrupted");  }  try  {  System.out.println("waiting for threads to finish");  ob1.t.join();  ob2.t.join();  }  catch(InterruptedException e)  {  System.out.println("Main thread interrupted");  }  System.out.println("Main thread exiting");  }  } |

Output

|  |
| --- |
| E:\javaprgs\threads>java suspendresume  New thread: Thread[one,5,main]  New thread: Thread[two,5,main]  one: 15  two: 15  one: 14  two: 14  one: 13  two: 13  one: 12  two: 12  one: 11  two: 11  Suspending thread one  two: 10  two: 9  two: 8  two: 7  two: 6  Resuming thread one  Suspending thread two  one: 10  one: 9  one: 8  one: 7  one: 6  Resuming thread two  waiting for threads to finish  two: 5  one: 5  two: 4  one: 4  two: 3  one: 3  two: 2  one: 2  two: 1  one: 1  twoexiting  oneexiting  Main thread exiting |

In this program a newThread class which implements Runnable interface is created. It has a constructor which assigns value for thread name is created. This is responsible for starting the thread. run() method is created which runs the created thread. In this the name of the thread along with the value given in for loop is printed. A sleep() method is called which delay the for loop for 200 milliseconds.

In the main program two objects say ob1 and ob2 for the class newThread is created. Now the two threads started running. In the main program sleep() method is called with 1000 milliseconds. At this time the newThread ob1 method and ob2 method runs (two threads) printing the name of thread with a uniform interval of 200 milliseconds. After 1000 millisecond the ob1 is suspended and main thread again sleeps for 1000 milliseconds. At this time the ob2 runs. After this ob1 is resumed and it starts running. Now ob2 is suspended and again main thread sleeps for 1000 milliseconds. Now ob1 is resumed and so on

Example for stopping thread

|  |
| --- |
| import java.util.Date;  class counterThread implements Runnable  {  private int counter=0;  public void run()  {  for(int i=0;i<10;i++)  {  counter++;  System.out.println("counter is : "+counter+" -- current time is : "+ new Date());  try  {  Thread.sleep(500);  if(counter ==5)  {  Thread.currentThread().stop();  } //if  } //try  catch(InterruptedException e)  {  System.out.println(e);  } //catch  } //for  } //run  public int getCounter()  {  return counter;  }  } //class counterThread  //main prgoram  public class counterdemo  {  public static void main(String args[])  {  counterThread counter=new counterThread();  Thread t1=new Thread(counter);  t1.start();  for(int i=0;i<10;i++)  {  try  {  System.out.println("Main thread: "+i);  Thread.sleep(1000);  if(i==5)  {  t1.resume();  }  } //try  catch(Exception e)  {  System.out.println(e);  } //catch  } //for  }  } |

Output

|  |
| --- |
| E:\javaprgs\threads>java counterdemo  Main thread: 0  counter is : 1 -- current time is : Fri Feb 23 10:21:23 IST 2018  counter is : 2 -- current time is : Fri Feb 23 10:21:23 IST 2018  Main thread: 1  counter is : 3 -- current time is : Fri Feb 23 10:21:24 IST 2018  counter is : 4 -- current time is : Fri Feb 23 10:21:24 IST 2018  Main thread: 2  counter is : 5 -- current time is : Fri Feb 23 10:21:25 IST 2018  Main thread: 3  Main thread: 4  Main thread: 5  Main thread: 6  Main thread: 7  Main thread: 8  Main thread: 9 |

In the above program when the value of i inside for loop reaches the thread is stopped. Till that the main and child thread are running. After stopping of child thread only the main thread runs. This cannot be resumed as it is stopped. Even resume is given it won’t work.

In the below program it shows that when a thread is suspended it can be resumed.

|  |
| --- |
| import java.util.Date;  class counterThread implements Runnable  {  private int counter=0;  public void run()  {  for(int i=0;i<10;i++)  {  counter++;  System.out.println("counter is : "+counter+" -- current time is : "+ new Date());  try  {  Thread.sleep(500);  if(counter ==5)  {  Thread.currentThread().suspend();  } //if  } //try  catch(InterruptedException e)  {  System.out.println(e);  } //catch  } //for  } //run  public int getCounter()  {  return counter;  }  } //class counterThread  //main prgoram  public class counterdemo1  {  public static void main(String args[])  {  counterThread counter=new counterThread();  Thread t1=new Thread(counter);  t1.start();  for(int i=0;i<10;i++)  {  try  {  System.out.println("Main thread: "+i);  Thread.sleep(1000);  if(i==5)  {  t1.resume();  }  } //try  catch(Exception e)  {  System.out.println(e);  } //catch  } //for  }  } |

|  |
| --- |
| E:\javaprgs\threads>java counterdemo1  Main thread: 0  counter is : 1 -- current time is : Fri Feb 23 10:27:05 IST 2018  counter is : 2 -- current time is : Fri Feb 23 10:27:05 IST 2018  Main thread: 1  counter is : 3 -- current time is : Fri Feb 23 10:27:06 IST 2018  counter is : 4 -- current time is : Fri Feb 23 10:27:06 IST 2018  Main thread: 2  counter is : 5 -- current time is : Fri Feb 23 10:27:07 IST 2018  Main thread: 3  Main thread: 4  Main thread: 5  Main thread: 6  counter is : 6 -- current time is : Fri Feb 23 10:27:11 IST 2018  counter is : 7 -- current time is : Fri Feb 23 10:27:11 IST 2018  Main thread: 7  counter is : 8 -- current time is : Fri Feb 23 10:27:12 IST 2018  counter is : 9 -- current time is : Fri Feb 23 10:27:12 IST 2018  Main thread: 8  counter is : 10 -- current time is : Fri Feb 23 10:27:13 IST 2018  Main thread: 9 |

Deadlock

**Deadlock**

A special type of error that you need to avoid that relates specifically to multitasking is

*deadlock,* which occurs when two threads have a circular dependency on a pair of synchronized

objects. For example, suppose one thread enters the monitor on object X and another thread

enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y,

it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized

method on X, the thread waits forever, because to access X, it would have to release its own

lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for

two reasons:

• In general, it occurs only rarely, when the two threads time-slice in just the right way.

• It may involve more than two threads and two synchronized objects. (That is, deadlock

can occur through a more convoluted sequence of events than just described.)

|  |
| --- |
| public class testdeadlock1  {  public static void main(String args[])  {  String resource1="I year";  String resource2="II year";  Thread t1=new Thread()  {  public void run()  {  synchronized(resource1)  {  System.out.println("Thread1: locked resource1 "+resource1);  try  {  Thread.sleep(100);  }  catch(Exception e)  {  System.out.println("exception occured is "+e);  }  synchronized(resource2)  {  System.out.println("Thread1: locked resource2 "+resource2);  }  }  } //run  }; //thread t1  Thread t2=new Thread()  {  public void run()  {  synchronized(resource2)  {  System.out.println("Thread2: locked resource2 "+resource2);  try  {  Thread.sleep(100);  }  catch(Exception e)  {  System.out.println("exception occured is "+e);  }  synchronized(resource1)  {  System.out.println("Thread2: locked resource1 " + resource1);  }  }  } //run  }; //thread t2  t1.start();  t2.start();  }  } |

Output

|  |
| --- |
| E:\javaprgs\thread>java -cp . testdeadlock1  Thread1: locked resource1 I year  Thread2: locked resource2 II year |