Inter Thread Communication in Java using Wait Notify Example

Wait and notify methods in Java are used for inter-thread communication i.e. if one thread wants to tell something to another thread, it uses notify() and notifyAll() method of java.lang.Object. Classical example of wait and notify method is Producer Consumer design pattern, where One thread produce and put something on shared bucket, and then tell other thread that there is an item for your interest in shared object, consumer thread than pick than item and do his job, without wait() and notify(), consumer thread needs to be busy checking, even if there is no change in state of shared object.  
  
This brings an interesting point on using wait and notify mechanism, a call to notify() happens, when thread changed state of shared object i.e. in this case producer change bucket from empty to not empty, and consumer change state from non-empty to empty.  
Also [wait and notify method must be called from synchronized context](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html). Another important thing to keep in mind while calling them is, **using loop to check conditions instead of if block**.

In short, a waiting thread may woke up, without any change in it's waiting condition due to spurious wake up.  
  
For example, if a consumer thread, which is waiting because shared queue is empty, gets wake up due to a false alarm and try to get something from queue without further checking whether queue is empty or not than unexpected result is possible.

We have a shared Queue and [two threads](http://javarevisited.blogspot.sg/2013/02/how-to-join-multiple-threads-in-java-example-tutorial.html) called Producer and Consumer. Producer thread puts number into shared queue and Consumer thread consumes numbers from shared bucket. Condition is that once an item is produced, consumer thread has to be notified and similarly after consumption producer thread needs to be notified. This inter thread communication is achieved using wait and notify method. Remember [wait and notify method is defined in object class](http://javarevisited.blogspot.in/2012/02/why-wait-notify-and-notifyall-is.html), and they are must be called inside synchronized block.  
  
**package com.thread;**

**import java.util.LinkedList;**

**import java.util.Queue;**

**public class ProducerConsumerorInterThreadCommunication {**

**public static void main(String args[]) {**

**final Queue sharedQ = new LinkedList();**

**Thread producer = new Producer(sharedQ);**

**Thread consumer = new Consumer(sharedQ);**

**producer.start();**

**consumer.start();**

**}**

**}**

**class Producer extends Thread {**

**private final Queue sharedQ;**

**public Producer(Queue sharedQ) {**

**super("Producer");**

**this.sharedQ = sharedQ;**

**}**

**@Override**

**public void run() {**

**for (int i = 0; i < 4; i++) {**

**//If synchronized keyword is not used at method or block level we get java.lang.IllegalMonitorStateException because of race condition.**

**synchronized (sharedQ) {**

**//waiting condition - wait until Queue is not empty**

**while (sharedQ.size() >= 1) {**

**try {**

**System.*out*.println("Queue is full, waiting");**

**sharedQ.wait();**

**} catch (InterruptedException ex) {**

**ex.printStackTrace();**

**}**

**}**

**System.*out*.println("producing : " + i);**

**sharedQ.add(i);**

**sharedQ.notify();**

**}**

**}**

**}**

**}**

**class Consumer extends Thread {**

**private final Queue sharedQ;**

**public Consumer(Queue sharedQ) {**

**super("Consumer");**

**this.sharedQ = sharedQ;**

**}**

**@Override**

**public void run() {**

**while(true) {**

**synchronized (sharedQ) {**

**//waiting condition - wait until Queue is not empty**

**while (sharedQ.size() == 0) {**

**try {**

**System.*out*.println("Queue is empty, waiting");**

**sharedQ.wait();**

**} catch (InterruptedException ex) {**

**ex.printStackTrace();**

**}**

**}**

**int number = (int) sharedQ.poll();//Retrieves and removes the head of this queue,or returns null if this queue is empty.**

**System.*out*.println("consuming : " + number );**

**sharedQ.notify();**

**//termination condition**

**if(number == 3){break; }**

**}**

**}**

**}**

**}**

That's all on this simple example of **Inter thread communication in Java using wait and notify method**. You can see that both Producer and Consumer threads are communicating with each other and sharing data using shared Queue, Producer notifies consumer when there is an item ready for consumption and Consumer thread tells Producer once it's done with consuming. This is classical example of [Producer Consumer design pattern](http://javarevisited.blogspot.com/2012/02/producer-consumer-design-pattern-with.html) as well, which inherently involves inter-thread communication and data sharing between threads in Java.

**Why wait notify and notifyAll called from synchronized block or method in Java**

We use wait(), notify(), or notifyAll() method mostly for [inter-thread communication](http://javarevisited.blogspot.sg/2013/12/inter-thread-communication-in-java-wait-notify-example.html) in Java. One thread is waiting after checking a condition e.g. In the classic Producer-Consumer problem, the Producer thread waits if the buffer is full and Consumer thread notify Producer thread after it creates a space in the buffer by consuming an element.  
Calling [notify() or notifyAll()](http://java67.blogspot.com/2013/03/difference-between-wait-vs-notify-vs-notifyAll-java-thread.html) methods issues a notification to a single or multiple thread that a condition has changed and once notification thread leaves synchronized block, all the threads which are waiting fight for object lock on which they are waiting and lucky **thread returns from wait() method after reacquiring the lock** and proceed further.

Just to summarize we call **wait**(), **notify**() or **notifyAll**method in Java from synchronized method or synchronized block in Java to avoid:

1) **IllegalMonitorStateException in Java** which will occur if we don't call wait (), notify () or notifyAll () method from synchronized context.

2) **Any potential race condition between wait and notify method in Java**.

**Race condition in Java** is a type of concurrency bug or issue which is introduced in your program because  parallel execution of your program by multiple threads at same time, Since Java is a multi-threaded programming language hence risk of Race condition is higher in Java which demands clear understanding of what causes a race condition and how to avoid that. Anyway Race conditions are just one of hazards or risk presented by  use of multi-threading in Java just like [deadlock in Java](http://javarevisited.blogspot.com/2010/10/what-is-deadlock-in-java-how-to-fix-it.html). **Race conditions** occurs when two thread operate on same object without proper synchronization and there operation interleaves on each other. Classical **example of Race condition** is incrementing a counter since increment is not an atomic operation and can be further divided into three steps like read, update and write. if two [threads](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) tries to increment count at same time and if they read same value because of interleaving of read operation of one thread to update operation of another thread, one count will be lost when one thread overwrite increment done by other thread. atomic operations are not subject to race conditions because those operation cannot be interleaved. This is also [a popular multi-threading interview questions](http://javarevisited.blogspot.com/2011/07/java-multi-threading-interview.html) during core java interviews. In this article we will see **how to find race condition in Java** and  two sample code patterns which often causes race conditions in Java.

## **How to find Race Conditions in Java**

**Finding Race conditions** in any language is most difficult job and Java is no different, though since readability of Java code is very good and synchronized constructs are well defined heaps to find race conditions by code review. finding race conditions by unit testing is not reliable due to random nature of race conditions. since race conditions surfaces only some time your unit test may passed without facing any race condition. only sure shot way to find race condition is reviewing code manually or using code review tools which can alert you on potential race conditions based on code pattern and use of synchronization in Java. I solely rely on [code review](http://javarevisited.blogspot.com/2011/09/code-review-checklist-best-practice.html) and yet to find a suitable *tool for exposing race condition in java*.

## **Code Example of Race Condition in Java**

Based on my experience in Java synchronization and where we use synchronized keyword I found that two code patterns namely "**check and act**" and "**read modify write**" can suffer race condition if not synchronized properly. both cases rely on natural assumption that a single line of code will be atomic and execute in one shot which is wrong e.g. ++ is not atomic.

**"Check and Act" race condition pattern**

**classical example of "check and act" race condition in Java is getInstance() method of Singleton Class, remember that was one questions which we have discussed on 10 Interview questions on Singleton pattern in Java as "**[**How to write thread-safe Singleton in Java**](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html)**". getInstace() method first check for whether instance is null and than initialized the instance and return to caller. Whole purpose of Singleton is that getInstance should always return same instance of Singleton. if you call getInstance() method from two thread simultaneously its possible that while one thread is initializing singleton after null check, another thread sees value of \_instance reference variable as null (quite possible in java) especially if your object takes longer time to initialize and enters into critical section which eventually results in getInstance() returning two separate instance of Singleton. This may not happen always because a fraction of delay may result in value of \_instance updated in main memory. here is a code example**

**public Singleton getInstance(){**

**if(\_instance == null){   //race condition if two threads sees \_instance= null**

**\_instance = new Singleton();**

**}**

**}**

**an easy way to fix "check and act" race conditions is to synchronized keyword and enforce locking which will make this operation atomic and guarantees that block or method will only be executed by one thread and result of operation will be visible to all threads once**[**synchronized blocks**](http://javarevisited.blogspot.com/2011/04/synchronization-in-java-synchronized.html)**completed or thread exited form synchronized block.**

# [How to avoid deadlock in Java?](https://javarevisited.blogspot.com/2018/08/how-to-avoid-deadlock-in-java-threads.html)

How to avoid deadlock in Java? Is one of the popular [Java interview question](http://javarevisited.blogspot.sg/2015/10/133-java-interview-questions-answers-from-last-5-years.html) and flavor of the season for multi-threading, asked mostly at a senior level with lots of follow up questions. Even though the problem looks very basic but most of the Java developers get stuck once you start going deep.  
Interview questions start with, "**What is a deadlock?**"  
The answer is simple when two or more threads are waiting for each other to release the resource they need (lock) and get stuck for infinite time, the situation is called deadlock. It will only happen in the case of [multitasking](http://www.java67.com/2015/11/difference-between-multithreading-and-multitasking.html) or [multi-threading](https://javarevisited.blogspot.com/2018/06/top-5-java-multithreading-and-concurrency-courses-experienced-programmers.html).  
 **Object level lock(or Mutex) Vs Class level lock(or Mutex) in java**

In Java, a **synchronized** block of code can only be executed by one thread at a time. Also, java supports multiple threads to be executed concurrently. This may cause two or more threads to access the same fields or objects at same time.

Synchronization is the process which keeps all concurrent threads in execution to be in sync. Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory. When a method is declared as **synchronized**; the thread holds the monitor or [**lock**](https://howtodoinjava.com/java/multi-threading/how-to-use-locks-in-java-java-util-concurrent-locks-lock-tutorial-and-example/)**object** for that method’s object. If another thread is executing the synchronized method, your thread is blocked until that thread releases the monitor.

Please note that we can use synchronized keyword in the class on defined methods or blocks. synchronized keyword can not be used with variables or attributes in class definition.

## 1. Object level lock in Java

**Object level lock** is mechanism when we want to synchronize a **non-static method** or **non-static code block** such that only one thread will be able to execute the code block on given instance of the class. This should always be done **to make instance level data thread safe**.

Object level locking can be done as below :

|  |
| --- |
| Various ways for object level locking |
| **public class DemoClass**  **{**  **public synchronized void demoMethod(){}**  **}**    **or**    **public class DemoClass**  **{**  **public void demoMethod(){**  **synchronized (this)**  **{**  **//other thread safe code**  **}**  **}**  **}**    **or**    **public class DemoClass**  **{**  **private final Object lock = new Object();**  **public void demoMethod(){**  **synchronized (lock)**  **{**  **//other thread safe code**  **}**  **}**  **}** |

## 2. Class level lock in Java

**Class level lock** prevents multiple threads to enter in synchronized block in any of all available instances of the class on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads.

Class level locking should always be done **to make static data thread safe**. As we know that [**static**](https://howtodoinjava.com/java/basics/java-static-keyword/) keyword associate data of methods to class level, so use locking at static fields or methods to make it on class level.

|  |
| --- |
| Various ways for class level locking |
| **public class DemoClass**  **{**  **//Method is static**  **public synchronized static void demoMethod(){**    **}**  **}**    **or**    **public class DemoClass**  **{**  **public void demoMethod()**  **{**  **//Acquire lock on .class reference**  **synchronized (DemoClass.class)**  **{**  **//other thread safe code**  **}**  **}**  **}**    **or**    **public class DemoClass**  **{**  **private final static Object lock = new Object();**    **public void demoMethod()**  **{**  **//Lock object is static**  **synchronized (lock)**  **{**  **//other thread safe code**  **}**  **}**  **}** |

## 3. Object level lock vs class level lock – Important notes

1. Synchronization in Java guarantees that no two threads can execute a synchronized method, which requires same lock, simultaneously or concurrently.
2. synchronized keyword can be used only with methods and code blocks. These methods or blocks can be static or non-static both.
3. When ever a thread enters into Java synchronized method or block it acquires a lock and whenever it leaves synchronized method or block it releases the lock. Lock is released even if thread leaves synchronized method after completion or due to any Error or Exception.
4. Java synchronized keyword is **re-entrant** in nature it means if a synchronized method calls another synchronized method which requires same lock then current thread which is holding lock can enter into that method without acquiring lock.
5. Java synchronization will throw [NullPointerException](https://howtodoinjava.com/java/exception-handling/how-to-effectively-handle-nullpointerexception-in-java/) if object used in synchronized block is null. For example, in above code sample if lock is initialized as null, the “synchronized (lock)” will throw NullPointerException.
6. Synchronized methods in Java put a performance cost on your application. So use synchronization when it is absolutely required. Also, consider using synchronized code blocks for synchronizing only critical section of your code.
7. It’s possible that both static synchronized and non static synchronized method can run simultaneously or concurrently because they lock on different object.
8. According to the Java language specification you can not use synchronized keyword with constructor. It is illegal and result in compilation error.
9. Do not synchronize on non final field on synchronized block in Java. because reference of non final field may change any time and then different thread might synchronizing on different objects i.e. no synchronization at all.
10. Do not use String literals because they might be referenced else where in the application and can cause deadlock. String objects created with new keyword can be used safely. But as a best practice, create a new **private** scoped Object instance OR lock on the shared variable itself which we want to protect.

# Deadlock in Java Multithreading

[**synchronized**](http://quiz.geeksforgeeks.org/synchronized-in-java/) keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.  
It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called [**Deadlock**](http://quiz.geeksforgeeks.org/operating-system-process-management-deadlock-introduction/)

package com.thread;

//Java program to illustrate Deadlock

//in multithreading.

class Util

{

// Util class to sleep a thread

static void sleep(long millis)

{

try

{

Thread.*sleep*(millis);

}

catch (InterruptedException e)

{

e.printStackTrace();

}

}

}

//This class is shared by both threads

class Shared

{

// first synchronized method

synchronized void test1(Shared s2)

{

System.*out*.println("test1-begin");

Util.*sleep*(1000);

// taking object lock of s2 enters

// into test2 method

s2.test2(this);

System.*out*.println("test1-end");

}

// second synchronized method

synchronized void test2(Shared s1)

{

System.*out*.println("test2-begin");

Util.*sleep*(1000);

// taking object lock of s1 enters

// into test1 method

s1.test1(this);

System.*out*.println("test2-end");

}

}

class Thread1 extends Thread

{

private Shared s1;

private Shared s2;

// constructor to initialize fields

public Thread1(Shared s1, Shared s2)

{

this.s1 = s1;

this.s2 = s2;

}

// run method to start a thread

@Override

public void run()

{

// taking object lock of s1 enters

// into test1 method

s1.test1(s2);

}

}

class Thread2 extends Thread

{

private Shared s1;

private Shared s2;

// constructor to initialize fields

public Thread2(Shared s1, Shared s2)

{

this.s1 = s1;

this.s2 = s2;

}

// run method to start a thread

@Override

public void run()

{

// taking object lock of s2

// enters into test2 method

s2.test2(s1);

}

}

public class DeadlockDemo {

public static void main(String[] args)

{

// creating one object

Shared s1 = new Shared();

// creating second object

Shared s2 = new Shared();

// creating first thread and starting it

Thread1 t1 = new Thread1(s1, s2);

t1.start();

// creating second thread and starting it

Thread2 t2 = new Thread2(s1, s2);

t2.start();

// sleeping main thread

Util.*sleep*(2000);

}

}

Output : test1-begin

test2-begin

It is not recommended to run the above program with online IDE. We can copy the source code and run it on our local machine. We can see that it runs for indefinite time, because threads are in deadlock condition and doesn’t let code to execute. Now let’s see step by step what is happening there.

1. Thread t1 starts and calls test1 method by taking the object lock of s1.
2. Thread t2 starts and calls test2 method by taking the object lock of s2.
3. t1 prints test1-begin and t2 prints test-2 begin and both waits for 1 second, so that both threads can be started if any of them is not.
4. t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it become free. It will not release lock of s1 until it gets lock of s2.
5. Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 release the lock. t2 will also not release lock of s2 until it gets lock of s1.
6. Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.
7. As none of them is ready to release lock, so this is the Dead Lock condition.
8. When you will run this program, it will be look like execution is paused.

**Avoid Dead Lock condition**

We can avoid dead lock condition by knowing its possibilities. It’s a very complex process and not easy to catch. But still if we try, we can avoid this. There are some methods by which we can avoid this condition. We can’t completely remove its possibility but we can reduce.

* **Avoid Nested Locks :**This is the main reason for dead lock. Dead Lock mainly happens when we give locks to multiple threads. Avoid giving lock to multiple threads if we already have given to one.
* **Avoid Unnecessary Locks :**We should have lock only those members which are required. Having lock on unnecessarily can lead to dead lock.
* **Using thread join :**Dead lock condition appears when one thread is waiting other to finish. If this condition occurs we can use Thread.join with maximum time you think the execution will take.

**Important Points :**

* If threads are waiting for each other to finish, then the condition is known as Deadlock.
* Deadlock condition is a complex condition which occurs only in case of multiple threads.
* Deadlock condition can break our code at run time and can destroy business logic.
* We should avoid this condition as much as we can.

**1) Java Thread Example by extending Thread class**

**class Multi extends Thread{**

**public void run(){**

**System.out.println("thread is running...");**

**}**

**public static void main(String args[]){**

**Multi t1=new Multi();**

**t1.start();**

**}**

**}**

**Output:thread is running...**

**2) Java Thread Example by implementing Runnable interface**

**class Multi3 implements Runnable{**

**public void run(){**

**System.out.println("thread is running...");**

**}**

**public static void main(String args[]){**

**Multi3 m1=new Multi3();**

**Thread t1 =new Thread(m1);**

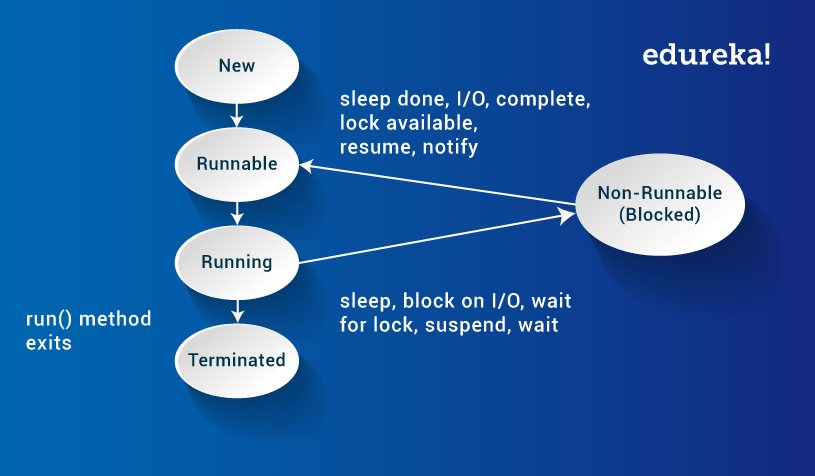
**t1.start();**

**}**

**}**

**Output:thread is running...**

* **New** - When we create an instance of Thread class, a thread is in a new state.
* **Running -**The Java thread is in running state.
* **Suspended** - A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
* **Blocked** - A Java thread can be blocked when waiting for a resource.
* **Terminated** - A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.



|  |  |
| --- | --- |
| **Method** | **Meaning** |
| getName | Obtain thread’s name |
| getPriority | Obtain 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 | Suspend a thread for a period of time |
| start | Start a thread by calling its run method |

|  |  |
| --- | --- |
| Method Signature | Description |
| String getName() | Retrieves the name of running thread in the current context in String format |
| void start() | This method will start a new thread of execution by calling run() method of Thread/runnable object. |
| void run() | This method is the entry point of the thread. Execution of thread starts from this method. |
| void sleep(int sleeptime) | This method suspend the thread for mentioned time duration in argument (sleeptime in ms) |
| void yield() | By invoking this method the current thread pause its execution temporarily and allow other threads to execute. |
| void join() | This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution |
| boolean isAlive() | This method will check if thread is alive or dead |