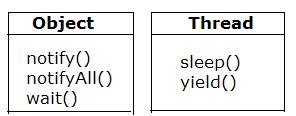
**Introduction to Threads**

Multithreading refers to two or more tasks executing concurrently within a single *program. A thread is an independent path of execution within a program. Many threads can run concurrently within a program. Every thread in Java is created and controlled by the****java.lang.Thread class****. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.*

***Multithreading has several advantages over Multiprocessing such as;***

* *Threads are lightweight compared to processes*
* *Threads share the same address space and therefore can share both data and code*
* *Context switching between threads is usually less expensive than between processes*
* *Cost of thread intercommunication is relatively low that that of process intercommunication*
* *Threads allow different tasks to be performed concurrently.*

***The following figure shows the methods that are members of the Object and Thread Class.***

**

Some Important Methods defined in**java.lang.Thread**are shown in the table:

|  |  |  |
| --- | --- | --- |
| **Method** | **Return Type** | **Description** |
| currentThread( ) | Thread | Returns an object reference to the thread in which it is invoked. |
| getName( ) | String | Retrieve the name of the thread object or instance. |
| start( ) | void | Start the thread by calling its runmethod. |
| run( ) | void | This method is the entry point to execute thread, like the main method for applications. |
| sleep( ) | void | Suspends a thread for a specified amount of time (in milliseconds). |
| isAlive( ) | boolean | This method is used to determine the thread is running or not. |
| activeCount( ) | int | This method returns the number of active threads in a particular thread group and all its subgroups. |
| interrupt( ) | void | The method interrupt the threads on which it is invoked. |
| yield( ) | void | By invoking this method the current thread pause its execution temporarily and allow other threads to execute. |
| join( ) | void | This method and **join(long millisec)** Throws InterruptedException.  These two methods are invoked on a thread. These are not returned until either the thread has completed or it is timed out respectively. |

**Thread Creation**

There are two ways to create thread in java;

* Implement the Runnable interface (java.lang.Runnable)
* By Extending the Thread class (java.lang.Thread)

**Implementing the Runnable Interface**

**The Runnable Interface Signature**

public interface Runnable {

void run();

}

One way to create a thread in java is to implement the Runnable Interface and then instantiate an object of the class. We need to override the run() method into our class which is the only method that needs to be implemented. The run() method contains the logic of the thread.

**The procedure for creating threads based on the Runnable interface is as follows:**

1. A class implements the Runnable interface, providing the run() method that will be executed by the thread. An object of this class is a Runnable object.

2. An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.

3. The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned.

4. The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception.

Below is a program that illustrates instantiation and running of threads using the runnable interface instead of extending the Thread class. To start the thread you need to invoke the **start()** method on your object.

|  |
| --- |
| class RunnableThread implements Runnable {  Thread runner;  public RunnableThread() {  }  public RunnableThread(String threadName) {  runner = new Thread(this, threadName); // (1) Create a new thread.  System.out.println(runner.getName());  runner.start(); // (2) Start the thread.  }  public void run() {  //Display info about this particular thread  System.out.println(Thread.currentThread());  }  }  public class RunnableExample {  public static void main(String[] args) {  Thread thread1 = new Thread(new RunnableThread(), "thread1");  Thread thread2 = new Thread(new RunnableThread(), "thread2");  RunnableThread thread3 = new RunnableThread("thread3");  //Start the threads  thread1.start();  thread2.start();  try {  //delay for one second  Thread.currentThread().sleep(1000);  } catch (InterruptedException e) {  }  //Display info about the main thread  System.out.println(Thread.currentThread());  }  } |

**Output**

thread3  
Thread[thread1,5,main]  
Thread[thread2,5,main]  
Thread[thread3,5,main]  
Thread[main,5,main]private

[**Download**](http://www.javabeginner.com/runnable-thread.zip)Runnable Thread Program Example

This approach of creating a thread by implementing the Runnable Interface must be used whenever the class being used to instantiate the thread object is required to extend some other class.

**Extending Thread Class**

The procedure for creating threads based on extending the Thread is as follows:

1. A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.

2. This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.

3. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

Below is a program that illustrates instantiation and running of threads by extending the Thread class instead of implementing the Runnable interface. To start the thread you need to invoke the **start()** method on your object.

|  |
| --- |
| class XThread extends Thread {  XThread() {  }  XThread(String threadName) {  super(threadName); // Initialize thread.  System.out.println(this);  start();  }  public void run() {  //Display info about this particular thread  System.out.println(Thread.currentThread().getName());  }  }  public class ThreadExample {  public static void main(String[] args) {  Thread thread1 = new Thread(new XThread(), "thread1");  Thread thread2 = new Thread(new XThread(), "thread2");  // The below 2 threads are assigned default names  Thread thread3 = new XThread();  Thread thread4 = new XThread();  Thread thread5 = new XThread("thread5");  //Start the threads  thread1.start();  thread2.start();  thread3.start();  thread4.start();  try {  //The sleep() method is invoked on the main thread to cause a one second delay.  Thread.currentThread().sleep(1000);  } catch (InterruptedException e) {  }  //Display info about the main thread  System.out.println(Thread.currentThread());  }  } |

Output

Thread[thread5,5,main]  
thread1  
thread5  
thread2  
Thread-3  
Thread-2  
Thread[main,5,main]

[**Download**](http://www.javabeginner.com/java-thread-example.zip) **Java Thread Program Example**

When creating threads, there are two reasons why implementing the Runnable interface may be preferable to extending the Thread class:

* Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface  
  has this option.
* A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

An example of an anonymous class below shows how to create a thread and start it:

( new Thread() {

public void run() {

for(;;) System.out.println(“Stop the world!”);

}

}

).start();

Page2)

**2)Thread Synchronization**

With respect to multithreading, **Synchronization** is a process of controlling the access of shared resources by the multiple threads in such a manner that only one thread can access a particular resource at a time.

In non synchronized multithreaded application, it is possible for one thread to modify a shared object while  
another thread is in the process of using or updating the object’s value. Synchronization prevents such type  
of data corruption which may otherwise lead to dirty reads and significant errors.  
Generally critical sections of the code are usually marked with synchronized keyword.

**Examples** of using Thread Synchronization is in “The Producer/Consumer Model”.

Locks are used to synchronize access to a shared resource. A lock can be associated with a shared resource.  
Threads gain access to a shared resource by first acquiring the lock associated with the object/block of code.  
At any given time, at most only one thread can hold the lock and thereby have access to the shared resource.  
A lock thus implements mutual exclusion.

The object lock mechanism enforces the following rules of synchronization:

A thread must acquire the object lock associated with a shared resource, before it can enter the shared resource.

The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with the shared resource.

If a thread cannot immediately acquire the object lock, it is blocked, that is, it must wait for the lock to become available.

* When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished.  
  If another thread is waiting for this object lock, it can proceed to acquire the lock in order to gain access to the shared resource.

Classes also have a class-specific lock that is analogous to the object lock. Such a lock is actually a lock on the java.lang.Class object associated with the class.

Given a class A, the reference A.class denotes this unique Class object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.

There can be 2 ways through which synchronized can be implemented in Java:

* synchronized methods
* synchronized blocks

Synchronized statements are same as synchronized methods. A synchronized statement can only be executed after a thread has acquired the lock on the object/class referenced in the synchronized statement.

**Synchronized Methods**

Synchronized methods are methods that are used to control access to an object. A thread only executes a synchronized method after it has acquired the lock for the method’s object or class. If the lock is already held by another thread, the calling thread waits. A thread relinquishes the lock simply by returning from the synchronized method, allowing the next thread waiting for this lock to proceed. Synchronized methods are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently. This is called a race condition. It occurs when two or more threads simultaneously update the same value, and as a consequence, leave the value in an undefined or inconsistent state. While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait until it gets the lock. This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object. Such a method can invoke other synchronized methods of the object without being blocked. The non-synchronized methods of the object can of course be called at any time by any thread.

**Below is an example shows how synchronized methods and object locks are used to coordinate access to a common object by multiple threads. If the ‘synchronized’ keyword is removed, the message is displayed in random fashion.**

|  |
| --- |
| public class SyncMethodsExample extends Thread {  static String[] msg = { "Beginner", "java", "tutorial,", ".,", "com",  "is", "the", "best" };  public SyncMethodsExample(String id) {  super(id);  }  public static void main(String[] args) {  SyncMethodsExample thread1 = new SyncMethodsExample("thread1: ");  SyncMethodsExample thread2 = new SyncMethodsExample("thread2: ");  thread1.start();  thread2.start();  boolean t1IsAlive = true;  boolean t2IsAlive = true;  do {  if (t1IsAlive && !thread1.isAlive()) {  t1IsAlive = false;  System.out.println("t1 is dead.");  }  if (t2IsAlive && !thread2.isAlive()) {  t2IsAlive = false;  System.out.println("t2 is dead.");  }  } while (t1IsAlive || t2IsAlive);  }  void randomWait() {  try {  Thread.currentThread().sleep((long) (3000 \* Math.random()));  } catch (InterruptedException e) {  System.out.println("Interrupted!");  }  }  public synchronized void run() {  SynchronizedOutput.displayList(getName(), msg);  }  }  class SynchronizedOutput {  // if the 'synchronized' keyword is removed, the message  // is displayed in random fashion  public static synchronized void displayList(String name, String list[]) {  for (int i = 0; i < list.length; i++) {  SyncMethodsExample t = (SyncMethodsExample) Thread  .currentThread();  t.randomWait();  System.out.println(name + list[i]);  }  }  } |

Output

thread1: Beginner  
thread1: java  
thread1: tutorial,  
thread1: .,  
thread1: com  
thread1: is  
thread1: the  
thread1: best  
t1 is dead.  
thread2: Beginner  
thread2: java  
thread2: tutorial,  
thread2: .,  
thread2: com  
thread2: is  
thread2: the  
thread2: best  
t2 is dead.

[**Download**](http://www.javabeginner.com/synchronized-block.zip)Synchronized Methods Thread Program Example

Class Locks

Page2)

**Synchronized Blocks**

Static methods synchronize on the class lock. Acquiring and relinquishing a class lock by a thread in order to execute a static synchronized method, proceeds analogous to that of an object lock for a synchronized instance method. A thread acquires the class lock before it can proceed with the execution of any static synchronized method in the class, blocking other threads wishing to execute any such methods in the same class. This, of course, does not apply to static, non-synchronized methods, which can be invoked at any time. Synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class. A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass.The synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object.  
The general form of the synchronized block is as follows:

synchronized (<object reference expression>) {  
<code block>  
}

A compile-time error occurs if the expression produces a value of any primitive type. If execution of the block completes normally, then the lock is released. If execution of the block completes abruptly, then the lock is released.  
A thread can hold more than one lock at a time. Synchronized statements can be nested. Synchronizedstatements with identical expressions can be nested. The expression must evaluate to a non-null referencevalue, otherwise, a NullPointerException is thrown.

The code block is usually related to the object on which the synchronization is being done. This is the case with synchronized methods, where the execution of the method is synchronized on the lock of the current object:

public Object method() {  
synchronized (this) { // Synchronized block on current object  
// method block  
}  
}

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is relinquished. This happens when the execution of the code block completes normally or an uncaught exception is thrown.

Object specification in the synchronized statement is mandatory. A class can choose to synchronize the execution of a part of a method, by using the this reference and putting the relevant part of the method in the synchronized block. The braces of the block cannot be left out, even if the code block has just one statement.

**class** SmartClient {

BankAccount account;

// …

**public** **void** updateTransaction() {

**synchronized** (account) { // (1) synchronized block

account.update(); // (2)

}

}

}

In the previous example, the code at (2) in the synchronized block at (1) is synchronized on the BankAccount object. If several threads were to concurrently execute the method updateTransaction() on an object of SmartClient, the statement at (2) would be executed by one thread at a time, only after synchronizing on the BankAccount object associated with this particular instance of SmartClient.

Inner classes can access data in their enclosing context. An inner object might need to synchronize on its associated outer object, in order to ensure integrity of data in the latter. This is illustrated in the following code where the synchronized block at (5) uses the special form of the this reference to synchronize on the outer object associated with an object of the inner class. This setup ensures that a thread executing the method setPi() in an inner object can only access the private double field myPi at (2) in the synchronized block at (5), by first acquiring the lock on the associated outer object. If another thread has the lock of the associated outer object, the thread in the inner object has to wait for the lock to be relinquished before it can proceed with the execution of the synchronized block at (5). However, synchronizing on an inner object and on its associated outer object are independent of each other, unless enforced explicitly, as in the following code:

**class** Outer { // (1) Top-level Class

**private** **double** myPi; // (2)

**protected** **class** Inner { // (3) Non-static member Class

**public** **void** setPi() { // (4)

**synchronized** (Outer.**this**) { // (5) Synchronized block on outer

// object

myPi = Math.*PI*; // (6)

}

}

}

}

Below example shows how synchronized block and object locks are used to coordinate access to shared objects by multiple threads.

|  |
| --- |
| public class SyncBlockExample extends Thread {  static String[] msg = { "Beginner", "java", "tutorial,", ".,", "com",  "is", "the", "best" };  public SyncBlockExample(String id) {  super(id);  }  public static void main(String[] args) {  SyncBlockExample thread1 = new SyncBlockExample("thread1: ");  SyncBlockExample thread2 = new SyncBlockExample("thread2: ");  thread1.start();  thread2.start();  boolean t1IsAlive = true;  boolean t2IsAlive = true;  do {  if (t1IsAlive && !thread1.isAlive()) {  t1IsAlive = false;  System.out.println("t1 is dead.");  }  if (t2IsAlive && !thread2.isAlive()) {  t2IsAlive = false;  System.out.println("t2 is dead.");  }  } while (t1IsAlive || t2IsAlive);  }  void randomWait() {  try {  Thread.currentThread().sleep((long) (3000 \* Math.random()));  } catch (InterruptedException e) {  System.out.println("Interrupted!");  }  }  public void run() {  synchronized (System.out) {  for (int i = 0; i < msg.length; i++) {  randomWait();  System.out.println(getName() + msg[i]);  }  }  }  } |

Output

thread1: Beginner  
thread1: java  
thread1: tutorial,  
thread1: .,  
thread1: com  
thread1: is  
thread1: the  
thread1: best  
t1 is dead.  
thread2: Beginner  
thread2: java  
thread2: tutorial,  
thread2: .,  
thread2: com  
thread2: is  
thread2: the  
thread2: best  
t2 is dead.

Synchronized blocks can also be specified on a class lock:

synchronized (<class name>.class) { <code block> }

The block synchronizes on the lock of the object denoted by the reference <class name>.class. A static synchronized method  
classAction() in class A is equivalent to the following declaration:

static void classAction() {

synchronized (A.class) { // Synchronized block on class A

// …

}

}

In summary, a thread can hold a lock on an object

* by executing a synchronized instance method of the object
* by executing the body of a synchronized block that synchronizes on the object
* by executing a synchronized static method of a class

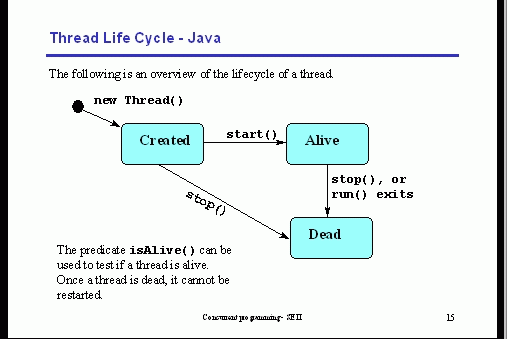
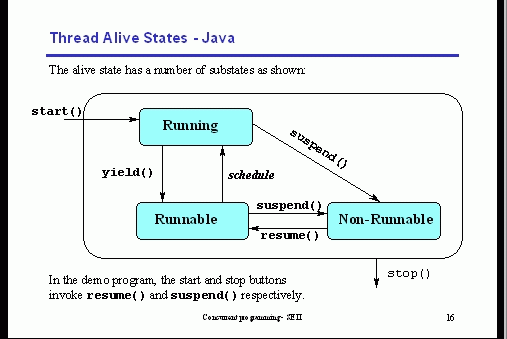
page 3)

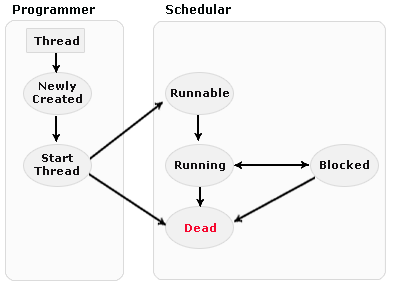
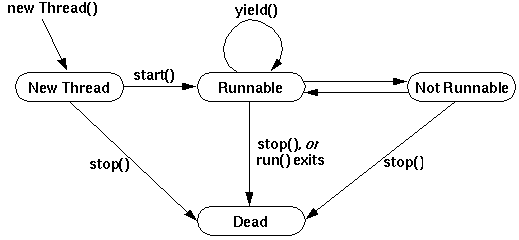
**Thread States**

A Java thread is always in one of several states which could be running, sleeping, dead, etc.

A thread can be in any of the following states:

* New Thread state (Ready-to-run state)
* Runnable state (Running state)
* Not Runnable state
* Dead state

**New Thread**

A thread is in this state when the instantiation of a **Thread**object creates a new thread but does not start it running.

A thread starts life in the Ready-to-run state. You can call only the **start()** or **stop()** methods when the thread is in this state. Calling any method besides **start()** or **stop()** causes an IllegalThreadStateException.

**Runnable**

When the **start()** method is invoked on a New Thread() it gets to the runnable state or running state by calling the run() method.

A Runnable thread may actually be running, or may be awaiting its turn to run.

**Not Runnable**

A thread becomes Not Runnable when one of the following four events occurs:

* When **sleep()** method is invoked and it sleeps for a specified amount of time
* When **suspend()** method is invoked
* When **the wait()**method is invoked and the thread waits for notification of a free resource or waits for the completion of another thread or waits to acquire a lock of an object.
* The thread is blocking on I/O and waits for its completion

**Example:** Thread.currentThread().sleep(1000);

**Note:** Thread.currentThread() may return an output like Thread[threadA,5,main]

The output shown in bold describes

* the name of the thread,
* the priority of the thread, and
* the name of the group to which it belongs.

Here, the **run()** method put itself to sleep for one second and becomes Not Runnable during that period.  
A thread can be awakened abruptly by invoking the **interrupt()** method on the sleeping thread object or at the end of the period of time for sleep is over. Whether or not it will actually start running depends on its priority and the availability of the CPU.

Hence I hereby list the scenarios below to describe how a thread switches form a non runnable to a runnable state:

* If a thread has been put to sleep, then the specified number of milliseconds must elapse (or it must be interrupted).
* If a thread has been suspended, then its **resume()** method must be invoked
* If a thread is waiting on a condition variable, whatever object owns the variable must relinquish it by calling  
  either **notify()** or **notifyAll()**.
* If a thread is blocked on I/O, then the I/O must complete.

**Dead State**

A thread enters this state when the **run()** method has finished executing or when the **stop()** method is invoked. Once in this state, the thread cannot ever run again.

Page4)

**Thread Priority**

In Java we can specify the priority of each thread relative to other threads. Those threads having higher priority get greater access to available resources then lower priority threads.

A Java thread inherits its priority from the thread that created it.

Heavy reliance on thread priorities for the behaviour of a program can make the  
program non portable across platforms, as thread scheduling is host platform–dependent.

You can modify a thread’s priority at any time after its creation using the **setPriority()** method and retrieve  
the thread priority value using **getPriority()** method.

The following static final integer constants are defined in the **Thread** class:

* MIN\_PRIORITY (0) Lowest Priority
* NORM\_PRIORITY (5) Default Priority
* MAX\_PRIORITY (10) Highest Priority

The priority of an individual thread can be set to any integer value between and including the above defined constants.

When two or more threads are ready to be executed and system resource becomes available to execute athread, the runtime system (the thread scheduler)  chooses  the  *Runnable*  thread with the highest priority for execution.

“If two threads of the same priority are waiting for the CPU, the thread scheduler chooses one of them to run in a > *round-robin fashion.* The chosen thread will run until one of the following conditions is true:

* a higher priority thread becomes*Runnable*. (Pre-emptive scheduling)
* it*yields*, or its **run()** method exits
* on systems that support *time-slicing*, its time allotment has expired

**Thread Scheduler**

Schedulers in JVM implementations usually employ one of the two following strategies:

**Preemptive scheduling**

If a thread with a higher priority than all other Runnable threads becomes Runnable, the scheduler will preempt the running thread (is moved to the runnable state) and choose the new higher priority thread for execution.

**Time-Slicing or Round-Robin scheduling**

A running thread is allowed to execute for a fixed length of time (a time slot it’s assigned to), after which it moves to the Ready-to-run state (runnable) to await its turn to run again.

A thread scheduler is implementation and platform-dependent; therefore, how threads will be scheduled is unpredictable across different platforms.

**Yielding [running thread->runnable state]**

A call to the static method yield(), defined in the Thread class, will cause the current thread in the Running state to move to the Runnable state, thus relinquishing the CPU. The thread is then at the mercy of the threadscheduler as to when it will run again. If there are no threads waiting in the Ready-to-run state, this thread continues execution. If there are other threads in the Ready-to-run state, their priorities determine which thread gets to execute. The **yield()** method gives other threads of the same priority a chance to run. If there are no equal priority threads in the “Runnable” state, then the yield is ignored.

**Sleeping and Waking Up[sleep(ms)**- pause but don’t free the resource it has  **,**  after milli-sec sleep **waking up** will resumes the run with the resource it has before sleep**]**

The thread class contains a static method named sleep() that causes the currently running thread to pause its execution and transit to the Sleeping state. The method does not relinquish any lock that the thread might have. The thread will sleep for at least the time specified in its argument, before entering the runnable state where it takes its turn to run again. If a thread is interrupted while sleeping, it will throw an **InterruptedException** when it awakes and gets to execute. The Thread class has several overloaded versions of the **sleep()** method.

**Waiting and Notifying**

Waiting and notifying provide means of thread inter-communication that synchronizes on the same object. The threads execute wait() and notify() (or notifyAll()) methods on the shared object for this purpose. The notifyAll(), notify() and wait() are methods of the Object class. These methods can be invoked only from within a synchronized context (synchronized method or synchronized block), otherwise, the call will result in an **IllegalMonitorStateException**. The notifyAll() method wakes up all the threads waiting on the resource. In this situation, the awakened threads compete for the resource. One thread gets the resource and the others go back to waiting.

wait() method signatures

final void wait(long timeout) throws InterruptedException  
final void wait(long timeout, int nanos) throws InterruptedException  
final void wait() throws InterruptedException

The wait() call can specify the time the thread should wait before being timed out. An another thread can invoke an interrupt() method on a waiting thread resulting in an InterruptedException. This is a checked exception and hence the code with the wait() method must be enclosed within a try catch block.

notify() method signatures

final void notify()  
final void notifyAll()

A thread usually calls the wait() method on the object whose lock it holds because a condition for its continuedexecution was not met. The thread leaves the Running state and transits to the Waiting-for-notification state. There it waits for this condition to occur. The thread relinquishes ownership of the object lock. The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object after acquiring its lock.  
The wait() method causes the current thread to wait until another thread notifies it of a condition change.

A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:

1. Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.  
2. The waiting thread times out.  
3. Another thread interrupts the waiting thread.

Notify

Invoking the notify() method on an object wakes up a single thread that is waiting on the lock of this object.  
A call to the notify() method has no consequences if there are no threads in the wait set of the object.  
The notifyAll() method wakes up all threads in the wait set of the shared object.

Below program shows three threads, manipulating the same stack. Two of them are pushing elements on the stack, while the third one is popping elements off the stack. This example illustrates how a thread waiting as a result of calling the wait() method on an object, is notified by another thread calling the notify() method on the same object

|  |
| --- |
| class StackClass {  private Object[] stackArray;  private volatile int topOfStack;  StackClass(int capacity) {  stackArray = new Object[capacity];  topOfStack = -1;  }  public synchronized Object pop() {  System.out.println(Thread.currentThread() + ": popping");  while (isEmpty()) {  try {  System.out.println(Thread.currentThread()  + ": waiting to pop");  wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  Object obj = stackArray[topOfStack];  stackArray[topOfStack--] = null;  System.out.println(Thread.currentThread()  + ": notifying after pop");  notify();  return obj;  }  public synchronized void push(Object element) {  System.out.println(Thread.currentThread() + ": pushing");  while (isFull()) {  try {  System.out.println(Thread.currentThread()  + ": waiting to push");  wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  stackArray[++topOfStack] = element;  System.out.println(Thread.currentThread()  + ": notifying after push");  notify();  }  public boolean isFull() {  return topOfStack >= stackArray.length - 1;  }  public boolean isEmpty() {  return topOfStack < 0;  }  }  abstract class StackUser extends Thread {  protected StackClass stack;  StackUser(String threadName, StackClass stack) {  super(threadName);  this.stack = stack;  System.out.println(this);  setDaemon(true);  start();  }  }  class StackPopper extends StackUser { // Stack Popper  StackPopper(String threadName, StackClass stack) {  super(threadName, stack);  }  public void run() {  while (true) {  stack.pop();  }  }  }  class StackPusher extends StackUser { // Stack Pusher  StackPusher(String threadName, StackClass stack) {  super(threadName, stack);  }  public void run() {  while (true) {  stack.push(new Integer(1));  }  }  }  public class WaitAndNotifyExample {  public static void main(String[] args) {  StackClass stack = new StackClass(5);  new StackPusher("One", stack);  new StackPusher("Two", stack);  new StackPopper("Three", stack);  System.out.println("Main Thread sleeping.");  try {  Thread.sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println("Exit from Main Thread.");  }  } |

[**Download**](http://www.javabeginner.com/java-threads-wait-notify.zip) Wait Notify methods Thread Program Example

The field topOfStack in class StackClass is declared volatile, so that read and write operations on this variable will access the master value of this variable, and not any copies, during runtime.  
Since the threads manipulate the same stack object and the push() and pop() methods in the class StackClass are synchronized, it means that the threads synchronize on the same object.

How the program uses wait() and notify() for inter thread communication.

(1) The synchronized pop() method – When a thread executing this method on the StackClass object finds that the stack is empty, it invokes the wait() method in order to wait for some other thread to fill the stack by using the synchronized push. Once an other thread makes a push, it invokes the notify method.  
(2)The synchronized push() method – When a thread executing this method on the StackClass object finds that the stack is full, i t invokes the wait() method to await some other thread to remove an element to provide space for the newly to be pushed element.  
Once an other thread makes a pop, it invokes the notify method.

Page 5)

**Joining**

A thread invokes the join() method on another thread in order to wait for the other thread to complete its execution.  
Consider a thread t1 invokes the method join() on a thread t2. The join() call has no effect if thread t2 has already completed. If thread t2 is still alive, then thread t1 transits to the Blocked-for-join-completion state.

Below is a program showing how threads invoke the overloaded thread join method.

|  |
| --- |
| public class ThreadJoinDemo {  public static void main(String[] args) {  Thread t1 = new Thread("T1");  Thread t2 = new Thread("T2");  try {  System.out.println("Wait for the child threads to finish.");  t1.join();  if (!t1.isAlive())  System.out.println("Thread T1 is not alive.");  t2.join();  if (!t2.isAlive())  System.out.println("Thread T2 is not alive.");  } catch (InterruptedException e) {  System.out.println("Main Thread interrupted.");  }  System.out.println("Exit from Main Thread.");  }  } |

[**Download**](http://www.javabeginner.com/threads-join-method.zip) Java Thread Join Method Program Example

Output

Wait for the child threads to finish.  
Thread T1 is not alive.  
Thread T2 is not alive.  
Exit from Main Thread.

**Deadlock**

There are situations when programs become deadlocked when each thread is waiting on a resource that cannot become available. The simplest form of deadlock is when two threads are each waiting on a resource that is locked by the other thread. Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked-for-lock-acquisition state. The threads are said to be deadlocked.

Thread t1 at tries to synchronize first on string o1 and then on string o2. The thread t2 does the opposite. It synchronizes first on string o2 then on string o1. Hence a deadlock can occur as explained above.

Below is a program that illustrates deadlocks in multithreading applications

|  |
| --- |
| public class DeadLockExample {  String o1 = "Lock ";  String o2 = "Step ";  Thread t1 = (new Thread("Printer1") {  public void run() {  while (true) {  synchronized (o1) {  synchronized (o2) {  System.out.println(o1 + o2);  }  }  }  }  });  Thread t2 = (new Thread("Printer2") {  public void run() {  while (true) {  synchronized (o2) {  synchronized (o1) {  System.out.println(o2 + o1);  }  }  }  }  });  public static void main(String[] args) {  DeadLockExample dLock = new DeadLockExample();  dLock.t1.start();  dLock.t2.start();  }  } |

[**Download**](http://www.javabeginner.com/java-thread-deadlock.zip)Java Thread deadlock Program Example

**Note:**The following methods namely **join, sleep** and **wait** name the InterruptedException in its throws clause and can have a timeout argument as a parameter. The following methods namely **wait, notify** and **notifyAll** should only be called by a thread that holds the lock of the instance on which the method is invoked. The Thread.start method causes a new thread to get ready to run at the discretion of the thread scheduler. The Runnable interface declares the run method. The Thread class implements the Runnable interface. Some implementations of the Thread.yield method will not yield to a thread of lower priority. A program will terminate only when all user threads stop running. A thread inherits its daemon status from the thread that created it.

|  |
| --- |
| **Threads Interview Questions** |
| **Q1) What is a Thread?**  Ans) In Java, "thread" means two different things:   * An instance of class java.lang.Thread. * A thread of execution.   An instance of Thread is just…an object. Like any other object in Java, it has variables and methods, and lives and dies on the heap. But a thread of execution is an individual process (a "lightweight" process) that has its own call stack. In Java, there is one thread per call stack—or, to think of it in reverse, one call stack per thread. Even if you don't create any new threads in your program, threads are back there running.  The main() method, that starts the whole ball rolling, runs in one thread, called (surprisingly) the main thread. If you looked at the main call stack (and you can, any time you get a stack trace from something that happens after main begins, but not within another thread), you'd see that main() is the first method on the stack— the method at the bottom. But as soon as you create a new thread, a new stack materializes and methods called from that thread run in a call stack that's separate from the main() call stack. |
| **Q2) What is difference between thread and process?**  Ans) **Differences between threads and processes are:-** 1. Threads share the address space of the process that  created it; processes have their own address.  2. Threads have direct access to the data segment of its process; processes have their own copy of the data segment of the parent process.  3. Threads can directly communicate with other threads of its process; processes must use interprocess communication to communicate with sibling processes.  4. Threads have almost no overhead; processes have considerable overhead.  5. New threads are easily created; new processes require duplication of the parent process.  6. Threads can exercise considerable control over threads of the same process; processes can only exercise control over child processes.  7. Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process; changes to the parent process do not affect child processes. |
| **Q3) What are the advantages or usage of threads?**  Ans)  **Threads support concurrent operations. For example,**  • Multiple requests by a client on a server can be handled as an individual client thread.  • Long computations or high-latency disk and network operations can be handled in the background without disturbing foreground computations or screen updates.   **Threads often result in simpler programs.** • In sequential programming, updating multiple displays normally requires a big while-loop that performs small parts of each display update. Unfortunately, this loop basically simulates an operating system scheduler. In Java, each view can be assigned a thread to provide continuous updates. • Programs that need to respond to user-initiated events can set up service routines to handle the events without having to insert code in the main routine to look for these events.  **Threads provide a high degree of control.** • Imagine launching a complex computation that occasionally takes longer than is satisfactory. A "watchdog" thread can be activated that will "kill" the computation if it becomes costly, perhaps in favor of an alternate, approximate solution. Note that sequential programs must muddy the computation with termination code, whereas, a Java program can use thread control to non-intrusively supervise any operation.   **Threaded applications exploit parallelism.** • A computer with multiple CPUs can literally execute multiple threads on different functional units without having to simulating multi-tasking ("time sharing"). • On some computers, one CPU handles the display while another handles computations or database accesses, thus, providing extremely fast user interface response times. |
| **Q4)What are the two ways of creating thread?**  Ans) There are two ways to create a new thread.  1)**Extend the Thread class** and override the run() method in your class. Create an instance of the subclass and invoke the start() method on it, which will create a new thread of execution. e.g.  public class NewThread extends Thread{  public void run(){  // the code that has to be executed in a separate new thread goes here }  public static void main(String [] args){  NewThread c = new NewThread();  c.start();  }  }  2)**Implements the Runnable interface.**The class will have to implement the run() method in the Runnable interface. Create an instance of this class. Pass the reference of this instance to the Thread constructor a new thread of execution will be created. e.g. class  public class NewThread implements Runnable{  public void run(){  // the code that has to be executed in a separate new thread goes here }  public static void main(String [] args){  NewThread c = new NewThread();  Thread t = new Thread(c); t.start(); }  } |
| **Q5) What are the different states of a thread's lifecycle?**  Ans) The different states of threads are as follows:  1) **New** – When a thread is instantiated it is in New state until the start() method is called on the thread instance. In this state the thread is not considered to be alive.  2) **Runnable** – The thread enters into this state after the start method is called in the thread instance. The thread may enter into the Runnable state from Running state. In this state the thread is considered to be alive.  3) **Running** – When the thread scheduler picks up the thread from the Runnable thread’s pool, the thread starts running and the thread is said to be in Running state.  4)**Waiting/Blocked/Sleeping** – In these states the thread is said to be alive but not runnable. The thread switches to this state because of reasons like wait method called or sleep method has been called on the running thread or thread might be waiting for some i/o resource so blocked. 5)      Dead – When the thread finishes its execution i.e. the run() method execution completes, it is said to be in dead state. A dead state can not be started again. If a start() method is invoked on a dead thread a runtime exception will occur. |
| **Q6) What is use of synchronized keyword?**  Ans) synchronized keyword can be applied to static/non-static methods or a block of code. Only one thread at a time can access synchronized methods and if there are multiple threads trying to access the same method then other threads have to wait for the execution of method by one thread. Synchronized keyword provides a lock on the object and thus prevents race condition. E.g.  public void synchronized method(){}   public void synchronized staticmethod(){} public void myMethod(){              synchronized (this){             // synchronized keyword on block of  code             }  } |
| **Q7) What is the difference when the synchronized keyword is applied to a static method or to a non static method?**  Ans) When a synch non static method is called a lock is obtained on the object. When a synch static method is called a lock is obtained on the class and not on the object. The lock on the object and the lock on the class donâ€™t interfere with each other. It means, a thread accessing a synch non static method, then the other thread can access the synch static method at the same time but canâ€™t access the synch non static method. |
| **Q8) What is a volatile keyword?**  Ans) In general each thread has its own copy of variable, such that one thread is not concerned with the value of same variable in the other thread. But sometime this may not be the case. Consider a scenario in which the count variable is holding the number of times a method is called for a given class irrespective of any thread calling, in this case irrespective of thread access the count has to be increased so the count variable is declared as volatile. The copy of volatile variable is stored in the main memory, so every time a thread access the variable even for reading purpose the local copy is updated each time from the main memory. The volatile variable also have performance issues. |
| **Q9) What is the difference between yield() and sleep()?**  Ans)  yield() allows the current the thread to release its lock from the object and scheduler gives the lock of the object to the other thread with same priority.          sleep() allows the thread to go to sleep state for x milliseconds. When a thread goes into sleep state it doesn’t release the lock. |
| **Q10) What is the difference between wait() and sleep()?**  Ans)  1) wait() is a method of Object class. sleep() is a method of Object class.  2) sleep() allows the thread to go to sleep state for x milliseconds. When a thread goes into sleep state it doesn’t release the lock. wait() allows thread to release the lock and goes to suspended state. The thread is only active when a notify() or notifAll() method is called for the same object. |
| **Q11) What is difference between notify() and notfiyAll()?**  Ans) notify( ) wakes up the first thread that called wait( ) on the same object.  notifyAll( ) wakes up all the threads that called wait( ) on the same object. The  highest priority thread will run first. |
| **Q12) What happens if a start method is not invoked and the run method is directly invoked?**  Ans) If a thread has been instantiated but not started its is said to be in new state. Unless until a start() method is invoked on the instance of the thread, it will not said to be alive. If you do not call a start() method on the newly created thread instance thread is not considered to be alive. If the start() method is not invoked and the run() method is directly called on the Thread instance, the code inside the run() method will not run in a separate new thread but it will start running in the existing thread. |
| **Q13) What happens when start() is called?**  Ans) A new thread of execution with a new call stack starts. The state of thread changes from new to runnable. When the thread gets chance to execute its target run() method starts to run. |
| **Q14) If code running is a thread creates a new thread what will be the initial priority of the newly created thread?**  Ans) When a code running in a thread creates a new thread object , the priority of the new thread is set equal to the priority of the thread which has created it. |
| **Q15) When jvm starts up, which thread will be started up first?**  Ans) When jvm starts up the thread executing main method is started. |
| **Q16) What are the daemon threads?**  Ans) Daemon thread are service provider threads run in the background,these not used to run the application code generally.When all user threads(non-daemon threads) complete their execution the jvm exit the application whatever may be the state of the daemon threads. Jvm does not wait for the daemon threads to complete their execution if all user threads have completed their execution.  To create Daemon thread set the daemon value of Thread using setDaemon(boolean value) method. By default all the threads created by user are user thread. To check whether a thread is a Daemon thread or a user thread use isDaemon() method.  Example of the Daemon thread is the Garbage Collector run by jvm to reclaim the unused memory by the application. The Garbage collector code runs in a Daemon thread which terminates as all the user threads are done with their execution. |
| **Q17) What all constructors are present in the Thread class?**  Ans) Thread()  Thread(Runnable target) Thread(Runnable target, String name) Thread(String name) |
| **Q18) Can the variables or classes be Synchronized?**  Ans) No. Only methods can be synchronized. |
| **Q19) How many locks does an object have?**  Ans) Each object has only one lock. |
| **Q20) Can a class have both Synchronized and non-synchronized methods?**  Ans) Yes a class can have both synchronized and non-synchronized methods. |