Multithreading in Java

<https://www.geeksforgeeks.org/multithreading-in-java/>

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.  
  
Threads can be created by using two mechanisms :  
1. Extending the Thread class  
2. Implementing the Runnable Interface

**Thread creation by extending the Thread class**  
  
We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

// Java code for thread creation by extending

// the Thread class

class MultithreadingDemo extends Thread

{

    public void run()

    {

        try

        {

            // Displaying the thread that is running

            System.out.println ("Thread " +

                  Thread.currentThread().getId() +

                  " is running");

        }

        catch (Exception e)

        {

            // Throwing an exception

            System.out.println ("Exception is caught");

        }

    }

}

// Main Class

public class Multithread

{

    public static void main(String[] args)

    {

        int n = 8; // Number of threads

        for (int i=0; i<n; i++)

        {

            MultithreadingDemo object = new MultithreadingDemo();

            object.start();

        }

    }

}

**Thread creation by implementing the Runnable Interface**  
  
We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

// Java code for thread creation by implementing

// the Runnable Interface

class MultithreadingDemo implements Runnable

{

    public void run()

    {

        try

        {

            // Displaying the thread that is running

            System.out.println ("Thread " +

                                Thread.currentThread().getId() +

                                " is running");

        }

        catch (Exception e)

        {

            // Throwing an exception

            System.out.println ("Exception is caught");

        }

    }

}

// Main Class

class Multithread

{

    public static void main(String[] args)

    {

        int n = 8; // Number of threads

        for (int i=0; i<n; i++)

        {

            Thread object = new Thread(new MultithreadingDemo());

            object.start();

        }

    }

}

**Thread Class vs Runnable Interface**  
1. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.  
  
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

# Difference between notify() and notifyAll() in Java

<https://www.geeksforgeeks.org/difference-notify-notifyall-java/>notify() and notifyAll() methods with wait() method are used to for communication between the threads. A thread which goes into waiting state by calling wait() method will be in waiting state until any other thread calls either notify() or notifyAll() method on the same object.

Now the question is both notify() and notifyAll() method is used to give notification to the waiting thread, then what is the difference between them or where we should use notify() method and where we should go for notifyAll() method?

**Differences between notify() and notifyAll()**

1. **Notification to number of threads :** We can use notify() method to give the notification **for only one thread** which is waiting for a particular object whereas by the help of notifyAll() methods we can give the notification to **all waiting threads** of a particular object.
2. **Notifying a thread by JVM :**If multiple threads are waiting for the notification and we use notify() method then only one thread get the notification and the remaining thread have to wait for further notification. Which thread will get the notification we can’t expect because it totally depends upon the JVM. But when we use notifyAll() method then multiple threads got the notification but execution of threads will be performed one by one because thread requires lock and only one lock is available for one object.
3. **Interchangeability of threads :** We should go for notify() if all your waiting threads are interchangeable (the order they wake up doesn’t matter). A common example is a thread pool. But we should use notifyAll() for other cases where the waiting threads may have different purposes and should be able to run concurrently. An example is a maintenance operation on a shared resource, where multiple threads are waiting for the operation to complete before accessing the resource.

# Differences between wait() and join() methods in Java

<https://www.geeksforgeeks.org/differences-between-wait-and-join-methods-in-java/>

The wait() and join() methods are used to pause the current thread. The wait() is used in with notify() and notifyAll() methods, but join() is used in Java to wait until one thread finishes its execution.  
wait() is mainly used for shared resources, a thread notifies other waiting thread when a resource becomes free. On the other hand join() is used for waiting a thread to die.

**Similarities between wait() and join()**

* The method wait() and join() both are used to pause the current thread in Java.
* Both wait() and join() can be interrupted by calling interrupt() method in Java.
* Both wait() and join() are a non-static method.
* Both wait() and join() are overloaded in Java. wait() and join() which without timeout as well as accepts a timeout parameter.

**Difference between wait() and join() method**

* Most obvious difference, both are present different packages, the wait() method is declared in java.lang.Object class while join() is declared in java.lang.Thread class.
* The wait() is used for inter-thread communication while the join() is used for adding sequencing between multiple threads, one thread starts execution after first thread execution finished.
* We can start a waiting thread (went into this state by calling wait()) by using notify() and notifyAll() method but we can not break the waiting imposed by join without unless or interruption the thread on which join is called has execution finished.
* One most important difference between wait() and join() that is wait() must be called from synchronized context i.e. synchronized block or method otherwise it will throw IllegalMonitorStateException but On the other hand, we can call join() method with and without synchronized context in Java.

# Inter-thread Communication in Java

<https://www.geeksforgeeks.org/inter-thread-communication-java/>

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.

# Callable and Future in Java

<https://www.geeksforgeeks.org/callable-future-java/?ref=lbp>

**The need for Callable**

There are two ways of creating threads – one by extending the Thread class and other by creating a thread with a Runnable. However, one feature lacking in  Runnable is that we cannot make a thread return result when it terminates, i.e. when run() completes. For supporting this feature, the Callable interface is present in Java.

**Callable vs Runnable**

* For implementing Runnable, the run() method needs to be implemented which does not return anything, while for a Callable, the call() method needs to be implemented which returns a result on completion. Note that a thread can’t be created with a Callable, it can only be created with a Runnable.
* Another difference is that the call() method can throw an exception whereas run() cannot.

**Future**

When the call() method completes, answer must be stored in an object known to the main thread, so that the main thread can know about the result that the thread returned. How will the program store and obtain this result later? For this, a Future object can be used. Think of a Future as an object that holds the result – it may not hold it right now, but it will do so in the future (once the Callable returns). Thus, a Future is basically one way the main thread can keep track of the progress and result from other threads. To implement this interface, 5 methods have to be overridden, but as the example below uses a concrete implementation from the library, only the important methods are listed here.

Observe that Callable and Future do two different things – Callable is similar to Runnable, in that it encapsulates a task that is meant to run on another thread, whereas a Future is used to store a result obtained from a different thread. In fact, the Future can be made to work with Runnable as well, which is something that will become clear when Executors come into the picture.

* **public boolean cancel(boolean mayInterrupt):** Used to stop the task. It stops the task if it has not started. If it has started, it interrupts the task only if mayInterrupt is true.
* **public Object get() throws InterruptedException, ExecutionException:**Used to get the result of the task. If the task is complete, it returns the result immediately, otherwise it waits till the task is complete and then returns the result.
* **public boolean isDone():** Returns true if the task is complete and false otherwise

To create the thread, a Runnable is required. To obtain the result, a Future is required.

The Java library has the concrete type FutureTask, which implements Runnable and Future, combining both functionality conveniently.  
A FutureTask can be created by providing its constructor with a Callable. Then the FutureTask object is provided to the constructor of Thread to create the Thread object. Thus, indirectly, the thread is created with a Callable. For further emphasis, note that there is no way to create the thread directly with a Callable.

# Java.util.concurrent.ExecutorService interface with Examples

<https://www.geeksforgeeks.org/java-util-concurrent-executorservice-interface-with-examples/>

The **ExecutorService** interface extends Executor by adding methods that help manage and control the execution of threads. It is defined in [**java.util.concurrent package**](https://www.geeksforgeeks.org/tag/java-concurrent-package/). It defines methods that execute the threads that return results, a set of threads and that determine the shutdown status.

The **ExecutorService interface** is implemented in a utility class called **Executors**. It defines methods that provide an implementation of the ExecutorService interface and many other interfaces, with some default settings.

**Methods in Executor interface:**

1. **shutdown()**– When invoked the function causes all the currently executing tasks to terminate after completion in the order in which they were started and rejects any new incoming tasks.
2. **shutdownNow()**– On invocation the function forcefully terminates all the tasks, regardless of their current state i.e running, waiting or ready. The lists of tasks that were in ready state in returned.
3. **isShutdown()**– The function tells whether the invoking executor is shutdown or not. Returns true if shutdown, otherwise returns false.
4. **isTerminated()**– The function checks if all the tasks have been completed post shutdown. Return true if completed, otherwise returns false.
5. **awaitTermination()**– The function waits for all the tasks to complete their execution after a shutdown request is found. It waits for the time specified by the timelimit argument.
6. **submit()**– The function adds a task that returns a result to the list of executing tasks for execution. It returns a Future object which returns the result of the task after completion.

# Thread Pools in Java

<https://www.geeksforgeeks.org/thread-pools-java/>

**A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.** Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

* Java provides the Executor framework which is centered around the Executor interface, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
* They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
* To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.

**Executor Thread Pool Methods**

* **Method** **Description**
* newFixedThreadPool(int) Creates a fixed size thread pool.
* newCachedThreadPool() Creates a thread pool that creates new
* threads as needed, but will reuse previously
* constructed threads when they are available
* newSingleThreadExecutor() Creates a single thread.

In case of a fixed thread pool, if all threads are being currently run by the executor then the pending tasks are placed in a queue and are executed when a thread becomes idle.

**Risks in using Thread Pools**

1. [**Deadlock**](https://www.geeksforgeeks.org/deadlock-in-java-multithreading/)**:**While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.
2. **Thread Leakage :**Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.
3. **Resource Thrashing :**If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.

**Important Points**

* + 1. Don’t queue tasks that concurrently wait for results from other tasks. This can lead to a situation of deadlock as described above.
    2. Be careful while using threads for a long lived operation. It might result in the thread waiting forever and would eventually lead to resource leakage.
    3. The Thread Pool has to be ended explicitly at the end. If this is not done, then the program goes on executing and never ends. Call shutdown() on the pool to end the executor. If you try to send another task to the executor after shutdown, it will throw a RejectedExecutionException.
    4. One needs to understand the tasks to effectively tune the thread pool. If the tasks are very contrasting then it makes sense to use different thread pools for different types of tasks so as to tune them properly.
    5. You can restrict maximum number of threads that can run in JVM, reducing chances of JVM running out of memory.
    6. If you need to implement your loop to create new threads for processing, using ThreadPool will help to process faster, as ThreadPool does not create new Threads after it reached it’s max limit.
    7. After completion of Thread Processing, ThreadPool can use the same Thread to do another process(so saving the time and resources to create another Thread.)

Daemon thread in Java

<https://www.geeksforgeeks.org/daemon-thread-java/>

Daemon thread is a low priority thread that runs in background to perform tasks such as garbage collection.

**Properties:**

* They can not prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution
* If JVM finds running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether Daemon thread is running or not.
* It is an utmost low priority thread.

**void setDaemon(boolean status):** This method is used to mark the current thread as daemon thread or user thread. For example if I have a user thread tU then tU.setDaemon(true) would make it Daemon thread. On the other hand if I have a Daemon thread tD then by calling tD.setDaemon(false) would make it user thread.

**boolean isDaemon():**  
This method is used to check that current is daemon. It returns true if the thread is Daemon else it returns false.

// Java program to demonstrate the usage of

// setDaemon() and isDaemon() method.

public class DaemonThread extends Thread

{

    public DaemonThread(String name){

        super(name);

    }

    public void run()

    {

        // Checking whether the thread is Daemon or not

        if(Thread.currentThread().isDaemon())

        {

            System.out.println(getName() + " is Daemon thread");

        }

        else

        {

            System.out.println(getName() + " is User thread");

        }

    }

    public static void main(String[] args)

    {

        DaemonThread t1 = new DaemonThread("t1");

        DaemonThread t2 = new DaemonThread("t2");

        DaemonThread t3 = new DaemonThread("t3");

        // Setting user thread t1 to Daemon

        t1.setDaemon(true);

        // starting first 2 threads

        t1.start();

        t2.start();

        // Setting user thread t3 to Daemon

        t3.setDaemon(true);

        t3.start();

    }

}

Output:

t1 is Daemon thread

t3 is Daemon thread

t2 is User thread

# Difference Between Daemon Threads and User Threads In Java

<https://www.geeksforgeeks.org/difference-between-daemon-threads-and-user-threads-in-java/>

**The Major Difference between User and Daemon Threads:**

|  |  |
| --- | --- |
| **USER THREAD** | **DAEMON THREAD** |
|  |  |
| JVM wait until user threads to finish their work. It never exit until all user threads finish their work. | The JVM will’t wait for daemon threads to finish their work. The JVM will exit as soon as all user threads finish their work. |
| JVM will not force to user threads for terminating, so JVM will wait for user threads to terminate themselves. | If all user threads have finished their work JVM will force the daemon threads to terminate |
| User threads are created by the application. | Mostly Daemon threads created by the JVM. |
| Mainly user threads are designed to do some specific task. | Daemon threads are design as to support the user threads. |
| User threads are foreground threads. | Daemon threads are background threads. |
| User threads are high priority threads. | Daemon threads are low priority threads. |
| Its life independent. | Its life depends on user threads. |

**Find whether a thread is of Daemon nature**

Yes , we can find that a thread is of Daemon nature :-

**Thread.currentThread( ).isDaemon( ) ;**  // statement 1

If statement 1 returns true then it a Daemon Thread otherwise if it returns false then it is a Non-Daemon Thread .

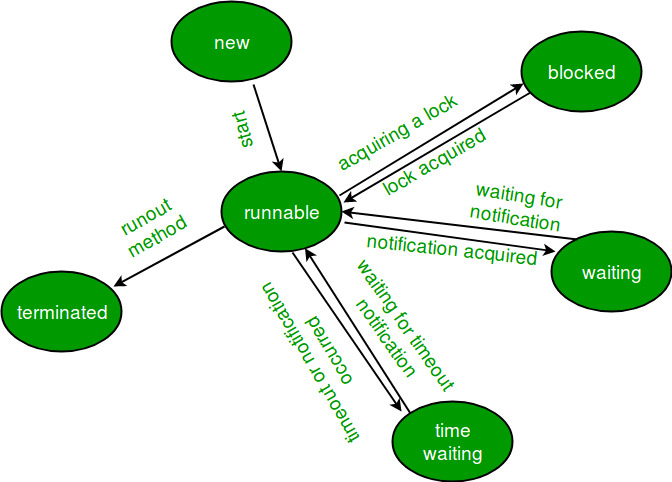
# Lifecycle and States of a Thread in Java

<https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-java/>

A [thread](http://www.geeksforgeeks.org/multithreading-in-java/) in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated

The diagram shown below represent various states of a thread at any instant of time.

[](https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-java/contribute.geeksforgeeks.org/wp-content/uploads/threadLifeCycle.jpg)

**Life Cycle of a thread**

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread lies in the new state, it’s code is yet to be run and hasn’t started to execute.
2. **Runnable State:** A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.  
   A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.
3. **Blocked/Waiting state:**When a thread is temporarily inactive, then it’s in one of the following states:
   * Blocked
   * Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It’s the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states does not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

1. **Timed Waiting:** A thread lies in timed waiting state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.
2. **Terminated State:** A thread terminates because of either of the following reasons:
   * Because it exists normally. This happens when the code of thread has entirely executed by the program.
   * Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

A thread that lies in a terminated state does no longer consumes any cycles of CPU.

**Implementing Thread States in Java**

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** class that defines the ENUM constants for the state of a thread, as summary of which is given below:

1. **Constant type:**New

Declaration: public static final Thread.State NEW

**Description:**Thread state for a thread which has not yet started.

1. **Constant type:**Runnable

Declaration: public static final Thread.State RUNNABLE

**Description:**Thread state for a runnable thread. A thread in the runnable state is executing in the Java virtual machine but it may be waiting for other resources from the operating system such as processor.

1. **Constant type:**Blocked

Declaration: public static final Thread.State BLOCKED

**Description:**Thread state for a thread blocked waiting for a monitor lock. A thread in the blocked state is waiting for a monitor lock to enter a synchronized block/method or reenter a synchronized block/method after calling Object.wait().

1. **Constant type:**Waiting

Declaration: public static final Thread.State WAITING

**Description:**Thread state for a waiting thread. Thread state for a waiting thread. A thread is in the waiting state due to calling one of the following methods:

* + Object.wait with no timeout
  + [Thread.join](http://www.geeksforgeeks.org/joining-threads-in-java/) with no timeout
  + LockSupport.park

A thread in the waiting state is waiting for another thread to perform a particular action.

1. **Constant type:**Timed Waiting

Declaration: public static final Thread.State TIMED\_WAITING

**Description:**Thread state for a waiting thread with a specified waiting time. A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

* 1. Thread.sleep
  2. Object.wait with timeout
  3. Thread.join with timeout
  4. LockSupport.parkNanos
  5. LockSupport.parkUntil

1. **Constant type:**Terminated

Declaration: public static final Thread.State TERMINATED

# You have thread T1, T2, and T3, how will you ensure that thread T2 run after T1 and thread T3 run after T2?

<https://www.java67.com/2015/07/how-to-join-two-threads-in-java-example.html>

# java-multithreading-concurrency-interview-questions-answers

<https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers>

# How to print even and odd numbers using threads in java

<https://java2blog.com/print-even-odd-numbers-threads-java/>

<https://java2blog.com/difference-between-sleep-and-wait-in/>

# Object level locking vs Class level locking in java

<https://java2blog.com/object-level-locking-vs-class-level-locking-java/>

# Why wait(), notify() And notifyAll() methods are in Object Class

<https://java2blog.com/why-wait-notify-notifyall-methods-object-class/>

1. **You have thread T1, T2, and T3, how will you ensure that thread T2 run after T1 and thread T3 run after T2?**  
   This thread interview question is mostly asked in the first round or phone screening round of interview and purpose of this multi-threading question is to check whether the candidate is familiar with the concept of "join" method or not. The answer to this multi-threading questions is simple it can be achieved by using the join method of Thread class. If you are not familiar with join method, please see my post [how to join threads in Java](http://www.java67.com/2015/07/how-to-join-two-threads-in-java-example.html).
2. **What are differences between wait and sleep method in Java?**  
   Another frequently asked thread interview question in Java mostly appear in a phone interview. The only major difference is that wait releases the lock or monitor while sleep doesn't release any lock or monitor while waiting. The wait is used for inter-thread communication while sleep is used to introduce pause on execution See my post [wait vs sleep in Java](http://javarevisited.blogspot.sg/2011/12/difference-between-wait-sleep-yield.html) for more differences
3. **4) Write code to implement blocking queue in Java?**  
   This is relatively tough Java multi-threading interview question which serves many purposes, it checks whether a candidate can actually write Java code using [thread](http://javarevisited.blogspot.sg/2011/02/how-to-implement-thread-in-java.html) or not, it sees how good candidate is on understanding concurrent scenarios and you can ask a lot of follow-up question based upon his code. If he uses [wait() and notify() method](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html) to implement blocking queue, Once candidate successfully writes it you can ask him to write it again using new Java 5 concurrent classes etc.
4. **5) Write code to solve the Produce consumer problem in Java?**([solution](http://javarevisited.blogspot.sg/2015/07/how-to-use-wait-notify-and-notifyall-in.html))  
   Similar to above questions on the thread but more classic in nature, some time interviewer ask follow up questions  How do you solve producer consumer problem in Java, well it can be solved in a multiple ways, I have shared one way to solve producer consumer problem using [BlockingQueue](http://www.java67.com/2015/12/producer-consumer-solution-using-blocking-queue-java.html) in Java, so be prepared for surprises. Sometimes they even ask to implement a solution of dining philosopher problem as well.

**12) How will you awake a blocked thread in Java?**  
This is a tricky question on threading, blocking can result in many ways, if thread is blocked on IO then I don't think there is a way to interrupt the thread, let me know if there is any, on the other hand, if thread is blocked due to result of calling wait(), sleep(), or join() method you can interrupt the thread and it will awake by throwing InterruptedException. See my post [How to deal with blocking methods in Java](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html) for more information on handling blocked thread.