**Thread**

# **Multithreading in java with examples**

**Before we talk about** multithreading**, let’s discuss threads. A thread is a light-weight smallest part of a process that can run concurrently with the other parts(other threads) of the same process. Threads are independent because they all have separate path of execution that’s the reason if an exception occurs in one thread, it doesn’t affect the execution of other threads. All threads of a process share the common memory.** The process of executing multiple threads simultaneously is known as multithreading.

Let’s summarize the discussion in points:  
1. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.

2. Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.

3. A thread can be in one of the following states:  
NEW – A thread that has not yet started is in this state.  
RUNNABLE – A thread executing in the Java virtual machine is in this state.  
BLOCKED – A thread that is blocked waiting for a monitor lock is in this state.  
WAITING – A thread that is waiting indefinitely for another thread to perform a particular action is in this state.  
TIMED\_WAITING – A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.  
TERMINATED – A thread that has exited is in this state.  
A thread can be in only one state at a given point in time.

Read more about thread states at this link: [Life cycle of threads](https://beginnersbook.com/2013/03/thread-life-cycle-in-java/)

## Multitasking vs Multithreading vs Multiprocessing vs parallel processing

If you are new to java you may get confused among these terms as they are used quite frequently when we discuss multithreading. Let’s talk about them in brief.

Multitasking: Ability to execute more than one task at the same time is known as multitasking.

Multithreading: We already discussed about it. It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.

Multiprocessing: It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.

Parallel Processing: It refers to the utilization of multiple CPUs in a single computer system.

## Creating a thread in Java

There are two ways to create a thread in Java:  
1) By extending Thread class.  
2) By implementing Runnable interface.

Before we begin with the programs(code) of creating threads, let’s have a look at these methods of Thread class. We have used few of these methods in the example below.

* getName(): It is used for Obtaining a thread’s name
* getPriority(): Obtain a 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 1: Thread creation by extending Thread class

Example 1:

class MultithreadingDemo extends Thread{

public void run(){

System.out.println("My thread is in running state.");

}

public static void main(String args[]){

MultithreadingDemo obj=new MultithreadingDemo();

obj.start();

}

}

Output:

My thread is in running state.

Example 2:

class Count extends Thread

{

Count()

{

super("my extending thread");

System.out.println("my thread created" + this);

start();

}

public void run()

{

try

{

for (int i=0 ;i<10;i++)

{

System.out.println("Printing the count " + i);

Thread.sleep(1000);

}

}

catch(InterruptedException e)

{

System.out.println("my thread interrupted");

}

System.out.println("My thread run is over" );

}

}

class ExtendingExample

{

public static void main(String args[])

{

Count cnt = new Count();

try

{

while(cnt.isAlive())

{

System.out.println("Main thread will be alive till the child thread is live");

Thread.sleep(1500);

}

}

catch(InterruptedException e)

{

System.out.println("Main thread interrupted");

}

System.out.println("Main thread's run is over" );

}

}

Output:

my thread createdThread[my runnable thread,5,main]

Main thread will be alive till the child thread is live

Printing the count 0

Printing the count 1

Main thread will be alive till the child thread is live

Printing the count 2

Main thread will be alive till the child thread is live

Printing the count 3

Printing the count 4

Main thread will be alive till the child thread is live

Printing the count 5

Main thread will be alive till the child thread is live

Printing the count 6

Printing the count 7

Main thread will be alive till the child thread is live

Printing the count 8

Main thread will be alive till the child thread is live

Printing the count 9

mythread run is over

Main thread run is over

### Method 2: Thread creation by implementing Runnable Interface

**A Simple Example**

class MultithreadingDemo implements Runnable{

public void run(){

System.out.println("My thread is in running state.");

}

public static void main(String args[]){

MultithreadingDemo obj=new MultithreadingDemo();

Thread tobj =new Thread(obj);

tobj.start();

}

}

Output:

My thread is in running state.

**Example Program 2:**  
Observe the output of this program and try to understand what is happening in this program. If you have understood the usage of each thread method then you should not face any issue, understanding this example.

class Count implements Runnable

{

Thread mythread ;

Count()

{

mythread = new Thread(this, "my runnable thread");

System.out.println("my thread created" + mythread);

mythread.start();

}

public void run()

{

try

{

for (int i=0 ;i<10;i++)

{

System.out.println("Printing the count " + i);

Thread.sleep(1000);

}

}

catch(InterruptedException e)

{

System.out.println("my thread interrupted");

}

System.out.println("mythread run is over" );

}

}

class RunnableExample

{

public static void main(String args[])

{

Count cnt = new Count();

try

{

while(cnt.mythread.isAlive())

{

System.out.println("Main thread will be alive till the child thread is live");

Thread.sleep(1500);

}

}

catch(InterruptedException e)

{

System.out.println("Main thread interrupted");

}

System.out.println("Main thread run is over" );

}

}

Output:

my thread createdThread[my runnable thread,5,main]

Main thread will be alive till the child thread is live

Printing the count 0

Printing the count 1

Main thread will be alive till the child thread is live

Printing the count 2

Main thread will be alive till the child thread is live

Printing the count 3

Printing the count 4

Main thread will be alive till the child thread is live

Printing the count 5

Main thread will be alive till the child thread is live

Printing the count 6

Printing the count 7

Main thread will be alive till the child thread is live

Printing the count 8

Main thread will be alive till the child thread is live

Printing the count 9

mythread run is over

Main thread run is over

## Thread priorities

* Thread priorities are the integers which decide how one thread should be treated with respect to the others.
* Thread priority decides when to switch from one running thread to another, process is called context switching
* A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
* A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
* To set the priority of the thread setPriority() method is used which is a method of the class Thread Class.
* In place of defining the priority in integers, we can use MIN\_PRIORITY, NORM\_PRIORITY or MAX\_PRIORITY.

## Methods: isAlive() and join()

* In all the practical situations main thread should finish last else other threads which have spawned from the main thread will also finish.
* To know whether the thread has finished we can call isAlive() on the thread which returns true if the thread is not finished.
* Another way to achieve this by using join() method, this method when called from the parent thread makes parent thread wait till child thread terminates.
* These methods are defined in the Thread class.
* We have used isAlive() method in the above examples too.

## Synchronization

* Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.
* When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.
* To implement the synchronous behavior java has synchronous method. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. All the other threads then wait until the first thread come out of the synchronized block.
* When we want to synchronize access to objects of a class which was not designed for the multithreaded access and the code of the method which needs to be accessed synchronously is not available with us, in this case we cannot add the synchronized to the appropriate methods. In java we have the solution for this, put the calls to the methods (which needs to be synchronized) defined by this class inside a synchronized block in following manner.

Synchronized(object)

{

// statement to be synchronized

}

## Inter-thread Communication

We have few methods through which java threads can communicate with each other. These methods are wait(), notify(), notifyAll(). All these methods can only be called from within a synchronized method.  
1) To understand synchronization java has a concept of monitor. Monitor can be thought of as a box which can hold only one thread. Once a thread enters the monitor all the other threads have to wait until that thread exits the monitor.  
2) wait()  tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().  
3) 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.

## What are Java Threads?

A thread is a:

* Facility to allow multiple activities within a single process
* Referred as lightweight process
* A thread is a series of executed statements
* Each thread has its own program counter, stack and local variables
* A thread is a nested sequence of method calls
* Its shares memory, files and per-process state

#### Whats the need of a thread or why we use Threads?

* To perform asynchronous or background processing
* Increases the responsiveness of GUI applications
* Take advantage of multiprocessor systems
* Simplify program logic when there are multiple independent entities

#### What happens when a thread is invoked?

When a thread is invoked, there will be two paths of execution. One path will execute the thread and the other path will follow the statement after the thread invocation. There will be a separate stack and memory space for each thread.

#### Risk Factor

* Proper co-ordination is required between threads accessing common variables [use of synchronized and volatile] for consistence view of data
* overuse of java threads can be hazardous to program’s performance and its maintainability.

### Threads in Java

Java threads facility and API is deceptively simple:  
Every java program creates at least one thread [ main() thread ]. Additional threads are created through the Thread constructor or by instantiating classes that extend the Thread class.

#### Thread creation in Java

Thread implementation in java can be achieved in two ways:

1. Extending the java.lang.Thread class
2. Implementing the java.lang.Runnable Interface

Note: The Thread and Runnable are available in the   java.lang.\* package

#### 1) By extending thread class

* The class should extend Java Thread class.
* The class should override the run() method.
* The functionality that is expected by the Thread to be executed is written in the run() method.

void start(): Creates a new thread and makes it runnable.  
void run(): The new thread begins its life inside this method.

Example:

public class MyThread extends Thread {

public void run(){

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

}

public static void main(String[] args) {

MyThread obj = new MyThread();

obj.start();

}

#### 2) By Implementing Runnable interface

* The class should implement the Runnable interface
* The class should implement the run() method in the Runnable interface
* The functionality that is expected by the Thread to be executed is put in the run() method

Example:

public class MyThread implements Runnable {

public void run(){

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

}

public static void main(String[] args) {

Thread t = new Thread(new MyThread());

t.start();

}

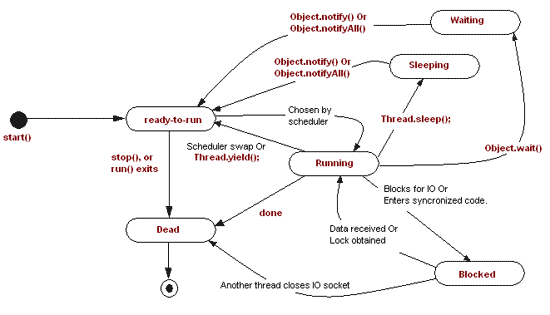
### Extends Thread class vs Implements Runnable Interface?

* Extending the Thread class will make your class unable to extend other classes, because of the single inheritance feature in  JAVA. However, this will give you a simpler code structure. If you implement Runnable, you can gain better object-oriented design and consistency and also avoid the single inheritance problems.
* If you just want to achieve basic functionality of a thread you can simply implement Runnable interface and override run() method. But if you want to do something serious with thread object as it has other methods like suspend(), resume(), ..etc which are not available in Runnable interface then you may prefer to extend the Thread class.

## Thread Life cycle in Java

* The start method creates the system resources, necessary to run the thread, schedules the thread to run, and calls the thread’s run method.
* A thread becomes “Not Runnable” when one of these events occurs:
  + If sleep method is invoked.
  + The thread calls the wait method.
  + The thread is blocking on I/O.
* A thread dies naturally when the run method exits.

Below diagram clearly depicts the various phases of thread life cycle in java.



### 2.2 Thread Priority

* When a Java thread is created, it inherits its priority from the thread that created it.
* You can modify a thread’s priority at any time after its creation using the setPriority method.
* Thread priorities are integers ranging between MIN\_PRIORITY (1) and MAX\_PRIORITY (10) . The higher the integer, the higher the priority.Normally the thread priority will be 5.

### 2.3 isAlive() and join() methods

* isAlive() method is used to determine if a thread is still alive. It is the best way to determine if a thread has been started but has not yet completed its run() method. final boolean isAlive();
* The nonstatic join() method of class Thread lets one thread “join onto the end” of another thread. This method waits until the thread on which it is called terminates. final void join();

## Thread vs Process

1) A program in execution is often referred as process. A thread is a subset(part) of the process.

2) A process consists of multiple threads. A thread is a smallest part of the process that can execute concurrently with other parts(threads) of the process.

3) A process is sometime referred as task. A thread is often referred as lightweight process.

4) A process has its own address space. A thread uses the process’s address space and share it with the other threads of that process.

5)

Per process items | Per thread items

------------------------------|-----------------

Address space | Program counter

Global variables | Registers

Open files | Stack

Child processes | State

Pending alarms |

Signals and signal handlers |

Accounting information |

6) A thread can communicate with other thread (of the same process) directly by using methods like wait(), notify(), notifyAll(). A process can communicate with other process by using inter-process communication.

7) New threads are easily created. However the creation of new processes require duplication of the parent process.

8) Threads have control over the other threads of the same process. A process does not have control over the sibling process, it has control over its child processes only.

**Daemon Thread**

**Daemon thread is a low priority thread (in context of JVM) that runs in background to perform tasks such as garbage collection (gc) etc., they do not prevent the JVM from exiting (even if the daemon thread itself is running) when all the user threads (non-daemon threads) finish their execution. JVM terminates itself when all user threads (non-daemon threads) finish their execution, JVM does not care whether Daemon thread is running or not, if JVM finds running daemon thread (upon completion of user threads), it terminates the thread and after that shutdown itself.**

## Properties of Daemon threads:

1. A newly created thread inherits the daemon status of its parent. That’s the reason all threads created inside main method (child threads of main thread) are non-daemon by default, because main thread is non-daemon. However you can make a user thread to Daemon by using setDaemon() method of thread class.  
   Just a quick note on main thread: When the JVM starts, it creates a thread called “Main”. Your program will run on this thread, unless you create additional threads yourself. The first thing the “Main” thread does is to look for your static void main (String args[]) method and invoke it. That is the entry-point to your program. If you create additional threads in the main method those threads would be the child threads of main thread.
2. Methods of Thread class that are related to Daemon threads:  
   public void setDaemon(boolean status): This method is used for making a user thread to Daemon thread or vice versa. For example if I have a user thread t then t.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 normal thread(user thread/non-daemon thread).  
   public boolean isDaemon(): This method is used for checking the status of a thread. It returns true if the thread is Daemon else it returns false.
3. setDaemon() method can only be called before starting the thread. This method would throw IllegalThreadStateException if you call this method after Thread.start() method. (refer the example)

Example 1: DaemonThreadExample1.java **This example is to demonstrate the usage of setDaemon() and isDaemon() method.**

public class DaemonThreadExample1 extends Thread{

public void run(){

// Checking whether the thread is Daemon or not

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

System.out.println("Daemon thread executing");

}

else{

System.out.println("user(normal) thread executing");

}

}

public static void main(String[] args){

/\* Creating two threads: by default they are

\* user threads (non-daemon threads)

\*/

DaemonThreadExample1 t1=new DaemonThreadExample1();

DaemonThreadExample1 t2=new DaemonThreadExample1();

//Making user thread t1 to Daemon

t1.setDaemon(true);

//starting both the threads

t1.start();

t2.start();

}

}

Output:

Daemon thread executing

user(normal) thread executing

## **Difference between Daemon threads and non-Daemon thread (user thread)**

The main difference between Daemon thread and user threads is that the JVM does not wait for Daemon thread before exiting while it waits for user threads, it does not exit until unless all the user threads finish their execution.

# **Thread join() method in Java with example**

**The join() method is used to hold the execution of currently running thread until the specified thread is dead(finished execution). In this tutorial we will discuss the purpose and use of join() method with examples.**

## Why we use join() method?

In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads.  
For example lets have a look at the following code:

## Without using join()

Here we have three threads th1, th2 and th3. Even though we have started the threads in a sequential manner the thread scheduler does not start and end them in the specified order. Everytime you run this code, you may get a different result each time. So the question is: How can we make sure that the threads executes in a particular order. The Answer is: By using join() method appropriately.

public class JoinExample2 {

public static void main(String[] args) {

Thread th1 = new Thread(new MyClass2(), "th1");

Thread th2 = new Thread(new MyClass2(), "th2");

Thread th3 = new Thread(new MyClass2(), "th3");

th1.start();

th2.start();

th3.start();

}

}

class MyClass2 implements Runnable{

@Override

public void run() {

Thread t = Thread.currentThread();

System.out.println("Thread started: "+t.getName());

try {

Thread.sleep(4000);

} catch (InterruptedException ie) {

ie.printStackTrace();

}

System.out.println("Thread ended: "+t.getName());

}

}

Output:

Thread started: th1

Thread started: th3

Thread started: th2

Thread ended: th1

Thread ended: th3

Thread ended: th2

Lets have a look at the another code where we are using the join() method.

## The same example with join()

Lets say our requirement is to execute them in the order of first, second and third. We can do so by using join() method appropriately.

public class JoinExample {

public static void main(String[] args) {

Thread th1 = new Thread(new MyClass(), "th1");

Thread th2 = new Thread(new MyClass(), "th2");

Thread th3 = new Thread(new MyClass(), "th3");

// Start first thread immediately

th1.start();

/\* Start second thread(th2) once first thread(th1)

\* is dead

\*/

try {

th1.join();

} catch (InterruptedException ie) {

ie.printStackTrace();

}

th2.start();

/\* Start third thread(th3) once second thread(th2)

\* is dead

\*/

try {

th2.join();

} catch (InterruptedException ie) {

ie.printStackTrace();

}

th3.start();

// Displaying a message once third thread is dead

try {

th3.join();

} catch (InterruptedException ie) {

ie.printStackTrace();

}

System.out.println("All three threads have finished execution");

}

}

class MyClass implements Runnable{

@Override

public void run() {

Thread t = Thread.currentThread();

System.out.println("Thread started: "+t.getName());

try {

Thread.sleep(4000);

} catch (InterruptedException ie) {

ie.printStackTrace();

}

System.out.println("Thread ended: "+t.getName());

}

}

Output:

Thread started: th1

Thread ended: th1

Thread started: th2

Thread ended: th2

Thread started: th3

Thread ended: th3

All three threads have finished execution

In this example we have used the join() method in such a way that our threads execute in the specified order.

**References**

* The Complete Reference Java 2 by Herbert Schildt