**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

# Java - Multithreading - Tutorialspoint   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 concurrently 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.

## **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.example : Thermal Power Plant

## 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" );

}

}

Public 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

**import** java.util.Scanner;

**public** **class** First {

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

// object creation for thread classes

Numeric num = **new** Numeric();

Alpha alp = **new** Alpha();

// start the thread

num.start();

alp.start();

// get the name of the thread

System.***out***.println("Number Thread : "+num.getName());

System.***out***.println("Alphabets Thread : "+alp.getName());

// get the priority

System.***out***.println("Number Thread : "+num.getPriority());

System.***out***.println("Alphabets Thread : "+alp.getPriority());

// check the state of the thread

System.***out***.println("Number Thread : "+num.getState());

System.***out***.println("Alphabets Thread : "+alp.getState());

// checking the current thread

System.***out***.println(Thread.*currentThread*().getName());

// set the name to thread

num.setName("Number Thread");

alp.setName("Alphabets Thread");

// check the state of the thread

System.***out***.println("Number Thread : "+num.getState());

System.***out***.println("Alphabets Thread : "+alp.getState());

// checking whether its alive or not

**if** (num.isAlive()) {

System.***out***.println("Number thread is Alive");

} **else** {

System.***out***.println("Num Thread is dead");

}

**if** (alp.isAlive()) {

System.***out***.println("Alphabets thread is Alive");

} **else** {

System.***out***.println("Alphabets Thread is dead");

}

}

}

**class** Numeric **extends** Thread {

**int** a = 0;

**public** **void** run() {

**try** {

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

a++;

System.***out***.println("Numeric Thread : " + a);

}

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

**class** Alpha **extends** Thread {

**char** a = 'A';

**public** **void** run() {

**try** {

**for** (**int** i = 0; i <= 25; i++) {

System.***out***.println("Alpha Thread : " + a);

a++;

}

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

**package** One;

**import** java.io.\*;

**import** java.util.Scanner;

**public** **class** First {

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

// object creation for thread classes

Minute num = **new** Minute();

Seconds alp = **new** Seconds();

// start the thread

num.start();

alp.start();

}

}

**class** Minute **extends** Thread {

**int** a = 0;

**public** **void** run() {

**try** {

**for** (**int** i = 0; i <= 59; i++) {

**try** {

System.***out***.println("Minutes : " + a);

a++;

Thread.*sleep*(60000);

}

**catch**(Exception e) {

System.***out***.println(e);

}

}

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

**class** Seconds **extends** Thread {

**int** a=0;

**public** **void** run() {

**try** {

**for** (**int** i = 0; i <= 59; i++) {

**try** {

System.***out***.println("Seconds :" + a);

a++;

Thread.*sleep*(1000);

}

**catch**(Exception e) {

System.***out***.println("Issues in Minutes Thread class");

}

}

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

**package** One;

**import** java.io.\*;

**import** java.util.Scanner;

**public** **class** First {

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

// object creation for thread classes

Numeric num = **new** Numeric();

Alpha alp = **new** Alpha();

// start the thread

num.start();

**try** {

num.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

alp.start();

}

}

**class** Numeric **extends** Thread {

**int** a = 0;

**public** **void** run() {

**try** {

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

System.***out***.println("Numeric Thread : " + a);

a++;

}

} **catch** (Exception e) {

System.***out***.println(e);

}

}

}

**class** Alpha **extends** Thread {

**char** a = 'A';

**public** **void** run() {

**try** {

**for** (**int** i = 0; i <= 25; i++) {

System.***out***.println("Alpha Thread : " + a);

a++;

}

} **catch** (Exception e) {

System.***out***.println(e);

}

}

### }

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

**A Simple Example**

Public 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" );

}

}

Public 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.

## **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.

# Java - Thread Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement −

### **Syntax**

synchronized(objectidentifier) {

// Access shared variables and other shared resources

}

## **Multithreading Example without Synchronization**

Here is a simple example which may or may not print counter value in sequence and every time we run it, it produces a different result based on CPU availability to a thread.

### **Example**

class PrintDemo {

public void printCount() {

try {

for(int i = 5; i > 0; i--) {

System.out.println("Counter --- " + i );

}

} catch (Exception e) {

System.out.println("Thread interrupted.");

}

}

}

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

PrintDemo PD;

ThreadDemo( String name, PrintDemo pd) {

threadName = name;

PD = pd;

}

public void run() {

PD.printCount();

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

PrintDemo PD = new PrintDemo();

ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );

ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

T1.start();

T2.start();

// wait for threads to end

try {

T1.join();

T2.join();

} catch ( Exception e) {

System.out.println("Interrupted");

}

}

}

This produces a different result every time you run this program −

### **Output**

Starting Thread - 1

Starting Thread - 2

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 5

Counter --- 2

Counter --- 1

Counter --- 4

Thread Thread - 1 exiting.

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread - 2 exiting.

## **Multithreading Example with Synchronization**

Here is the same example which prints counter value in sequence and every time we run it, it produces the same result.

### **Example**

class PrintDemo {

public void printCount() {

try {

for(int i = 5; i > 0; i--) {

System.out.println("Counter --- " + i );

}

} catch (Exception e) {

System.out.println("Thread interrupted.");

}

}

}

class ThreadDemo extends Thread {

private Thread t;

private String threadName;

PrintDemo PD;

ThreadDemo( String name, PrintDemo pd) {

threadName = name;

PD = pd;

}

public void run() {

synchronized(PD) {

PD.printCount();

}

System.out.println("Thread " + threadName + " exiting.");

}

public void start () {

System.out.println("Starting " + threadName );

if (t == null) {

t = new Thread (this, threadName);

t.start ();

}

}

}

public class TestThread {

public static void main(String args[]) {

PrintDemo PD = new PrintDemo();

ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );

ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

T1.start();

T2.start();

// wait for threads to end

try {

T1.join();

T2.join();

} catch ( Exception e) {

System.out.println("Interrupted");

}

}

}

This produces the same result every time you run this program −

### **Output**

Starting Thread - 1

Starting Thread - 2

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread - 1 exiting.

Counter --- 5

Counter --- 4

Counter --- 3

Counter --- 2

Counter --- 1

Thread Thread - 2 exiting.