

# Multithreading in java with examples

BY CHAITANYA SINGH | FILED UNDER: [MULTITHREADING](#)

Before we talk about **multithreading**, let's discuss threads. A thread is a lightweight 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](#)

## 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 `synchronized` 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.