**Multithreading in java** is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Multithreading gives Java the ability to achieve multiple tasks in parallel. One task does not wait for another to complete. That is, without completing one task, another task can start and also can execute.

###### **Multithreading Realtime Examples**

1. Background jobs like running application servers like Oracle application server, Web servers like Tomcat ctc which will come into action whenever a request comes.
2. Performing some execution while I/O blocked.
3. Gathering information from different web services running in parallel.
4. Typing MS Word document while listening to music.
5. Games are very good examples of threading. You can use multiple objects in games like cars, motor bikes, animals, people etc. All these objects are nothing but just threads that run your game application.
6. Railway ticket reservation system where multiple customers accessing the server.
7. Multiple account holders accessing their accounts simultaneously on the server. When you insert a ATM card, it starts a thread for perform your operations.
8. Servlets are multithreaded.

**Thread t = new Thread()**, this statement will create a Thread Object

**Thread** is the one which executes the code and puts things into action in the application.

Only when you say to start, It starts to execute the code. So, we need to invoke the start method on the thread object.

**run() method**.. is the starting point of thread.

So, Every thread object you create will have run() method.

the run method in the**Thread object** has nothing in it which connects to our own code.

Because this is created by the Java  api developers

So, we need to override this run method to make it to invoke our objects, method etc,

For this, You should create your own thread type and extend it to the Thread class, then override the run method to put your own code to tell the thread to start the code execution from this point

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* Process-based Multitasking (Multiprocessing)
* Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

* Threads share the same address space.
* A thread is lightweight.
* Cost of communication between the thread is low.

## What is Thread in java

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

# **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

### Runnable interface:

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

### Starting a thread:

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts(with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

   
**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<8; i++)

        {

            MultithreadingDemo object = new MultithreadingDemo();

            object.start();

        }

    }

}

Thread 8 is running

Thread 9 is running

Thread 10 is running

Thread 11 is running

Thread 12 is running

Thread 13 is running

Thread 14 is running

Thread 15 is running

**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<8; i++)

        {

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

            object.start();

        }

    }

}

Thread 8 is running

Thread 9 is running

Thread 10 is running

Thread 11 is running

Thread 12 is running

Thread 13 is running

Thread 14 is running

Thread 15 is running

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

**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.

Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

|  |
| --- |
| 1) New  The thread is in new state if you create an instance of Thread class but before the invocation of start() method. |

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

Thread class:

|  |
| --- |
| Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. |

Commonly used Constructors of Thread class:

|  |
| --- |
| * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) |

Commonly used methods of Thread class:

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted.    After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception. |

**public** **class** TestThreadTwice1 **extends** Thread{

**public** **void** run(){

   System.out.println("running...");

 }

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

  TestThreadTwice1 t1=**new** TestThreadTwice1();

  t1.start();

  t1.start();

}

}

Exception in thread "main" java.lang.IllegalThreadStateException

## Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

public static void sleep(long miliseconds)throws InterruptedException

public static void sleep(long miliseconds, int nanos)throws InterruptedException

**class** TestSleepMethod1 **extends** Thread{

**public** **void** run(){

**for**(**int** i=1;i<5;i++){

**try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}

    System.out.println(i);

  }

 }

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

  TestSleepMethod1 t1=**new** TestSleepMethod1();

  TestSleepMethod1 t2=**new** TestSleepMethod1();

  t1.start();

  t2.start();

 }

}

1

1

2

2

3

3

4

4

After starting a thread, it can never be started again. If you does so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

public class TestThreadTwice1 extends Thread{

public void run(){

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

}

public static void main(String args[]){

TestThreadTwice1 t1=new TestThreadTwice1();

t1.start();

t1.start();

}

}

**Compile by: javac TestThreadTwice1.java**

**Run by: java TestThreadTwice1**

Exception in thread "main" java.lang.IllegalThreadStateException  
 at java.base/java.lang.Thread.start(Thread.java:804)  
 at TestThreadTwice1.main(TestThreadTwice1.java:8)

* Each thread starts in a separate call stack.
* Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

class TestCallRun2 extends Thread{

public void run(){

for(int i=1;i<5;i++){

try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}

System.out.println(i);

}

}

public static void main(String args[]){

TestCallRun2 t1=new TestCallRun2();

TestCallRun2 t2=new TestCallRun2();

t1.run();

t2.run();

}

}

1  
2  
3  
4  
1  
2  
3  
4

class TestJoinMethod1 extends Thread{

public void run(){

for(int i=1;i<=5;i++){

try{

Thread.sleep(500);

}catch(Exception e){System.out.println(e);}

System.out.println(i);

}

}

public static void main(String args[]){

TestJoinMethod1 t1=new TestJoinMethod1();

TestJoinMethod1 t2=new TestJoinMethod1();

TestJoinMethod1 t3=new TestJoinMethod1();

t1.start();

try{

t1.join();

}catch(Exception e){System.out.println(e);}

t2.start();

t3.start();

}

}

O/p

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

## Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

**class** TestMultiNaming1 **extends** Thread{

**public** **void** run(){

   System.out.println("running...");

  }

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

  TestMultiNaming1 t1=**new** TestMultiNaming1();

  TestMultiNaming1 t2=**new** TestMultiNaming1();

  System.out.println("Name of t1:"+t1.getName());

  System.out.println("Name of t2:"+t2.getName());

  t1.start();

  t2.start();

  t1.setName(“sJ”);

  System.out.println("After changing name of t1:"+t1.getName());

 }

}

Output:Name of t1:Thread-0

Name of t2:Thread-1

id of t1:8

running...

After changeling name of t1:SJ

running...