Multithreading in java

By:Avinash Srivastava

Multithreading

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.

Java Multithreading is mostly used in games, animation, etc.

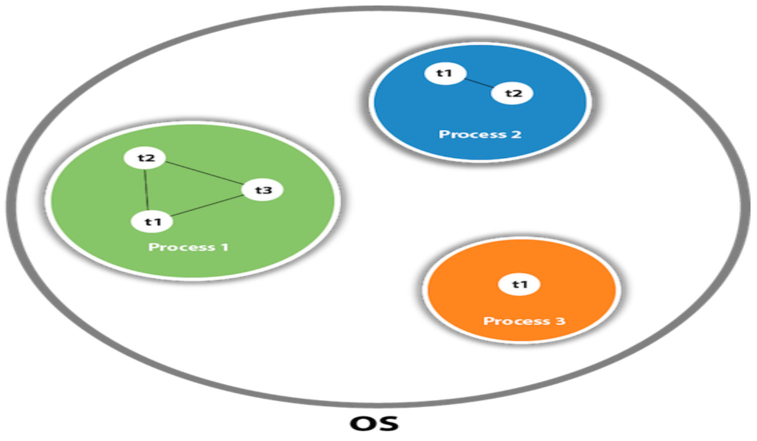
Advantages of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

2) You can perform many operations together, so it saves time.

3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Thread: 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.



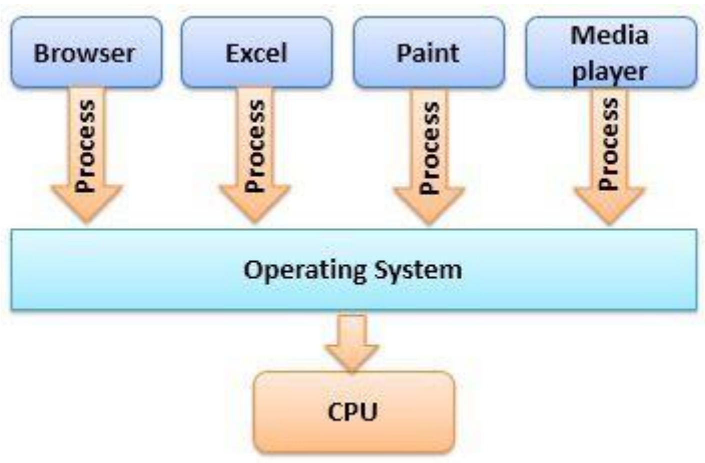
As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

Note: At a time one thread is executed only.

Difference Between Multitasking and Multithreading

Multitasking:

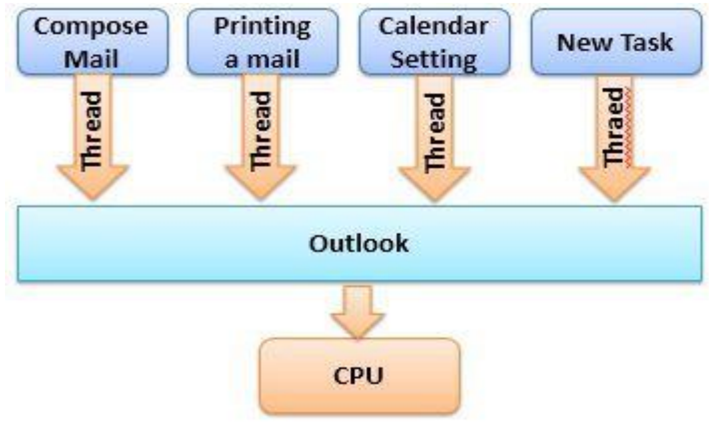
Multitasking is when a CPU is provided to execute multiple tasks at a time. Multitasking involves often CPU switching between the tasks, so that users can collaborate with each program together. Unlike multithreading, In multitasking, the processes share separate memory and resources. As multitasking involves CPU switching between the tasks rapidly, So the little time is needed in order to switch from the one user to next.



Multitasking

Multithreading:

Multithreading is a system in which many threads are created from a process through which the computer power is increased. In multithreading, CPU is provided in order to execute many threads from a process at a time, and in multithreading, process creation is performed according to cost. Unlike multitasking, multithreading provides the same memory and resources to the processes for execution.



Multithreading

The difference between multitasking and multithreading:

S.NO	MULTITASKING	MULTITHREADING
1.	In multitasking, users are allowed to perform many tasks by CPU.	While in multithreading, many threads are created from a process through which computer power is increased.
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated same memory.
4.	Multitasking component involves multiprocessing.	While multithreading component does not involve multiprocessing.
5.	In multitasking, CPU is provided in order to execute many tasks at a time.	While in multithreading also, CPU is provided in order to execute many threads from a process at a time.
6.	In multitasking, processes don’t share same resources, each process is allocated separate resources.	While in multithreading, each process share same resources.

7.	Multitasking is slow compared to multithreading.	While multithreading is faster.
8.	In multitasking, termination of process takes more time.	While in multithreading, termination of thread takes less time.

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.

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

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

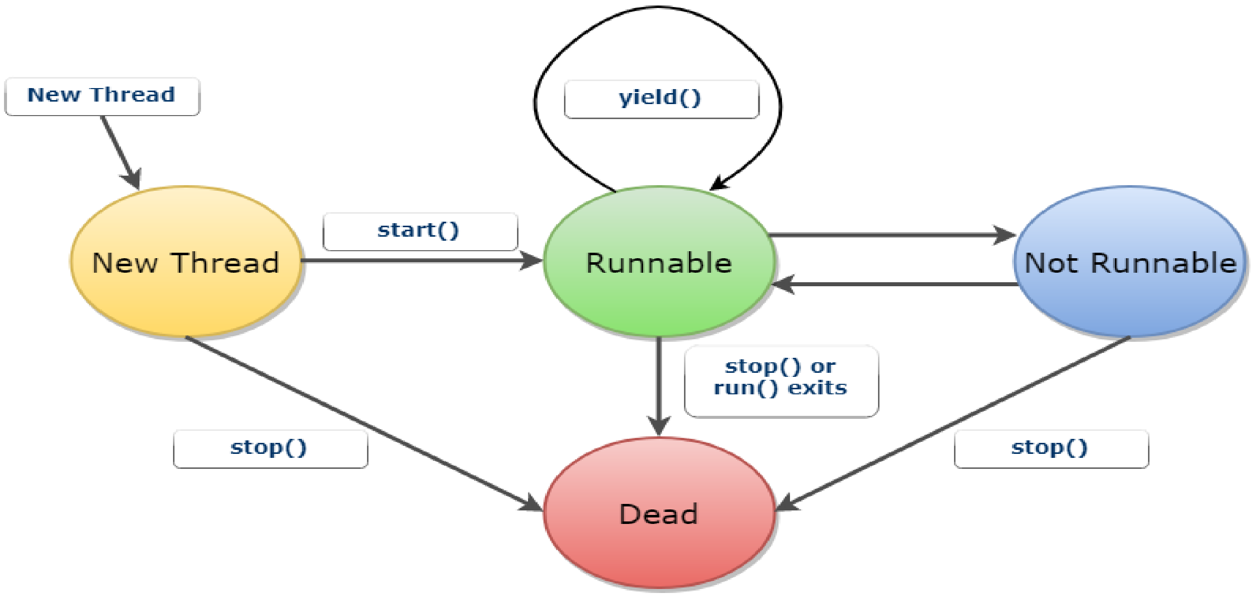


Fig: Life Cycle of a Thread in Java

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.

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

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)

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().

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.

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{

public void run(){

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

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

 }

}
```

Output:thread is running...

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable

{

public void run()

{

System.out.println("thread is running...");
}
public static void main(String args[])
{
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output:thread 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 Priority in Multithreading

In a Multi threading environment, thread scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly .
Accepted value of priority for a thread is in range of 1 to 10. There are 3 static variables defined in Thread class for priority.

public static int MIN_PRIORITY: This is minimum priority that a thread can have. Value for this is 1.
public static int NORM_PRIORITY: This is default priority of a thread if do not explicitly define it. Value for this is 5.
public static int MAX_PRIORITY: This is maximum priority of a thread. Value for this is 10.

Get and Set Thread Priority:

1. **public final int getPriority():** java.lang.Thread.getPriority() method returns priority of given thread.
2. **public final void setPriority(int newPriority):** java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

Examples of getPriority() and set

```
// Java program to demonstrate getPriority() and setPriority()
```

```
import java.lang.*;

class ThreadDemo extends Thread

{

        public void run()

        {

                System.out.println("Inside run method");

        }

        public static void main(String[]args)

        {

                ThreadDemo t1 = new ThreadDemo();

                ThreadDemo t2 = new ThreadDemo();

                ThreadDemo t3 = new ThreadDemo();


                System.out.println("t1 thread priority : " +

                                                        t1.getPriority()); // Default 5

                System.out.println("t2 thread priority : " +

                                                        t2.getPriority()); // Default 5

                System.out.println("t3 thread priority : " +

                                                        t3.getPriority()); // Default 5


                t1.setPriority(2);

                t2.setPriority(5);

                t3.setPriority(8);


                // t3.setPriority(21); will throw IllegalArgumentException

                System.out.println("t1 thread priority : " +

                                                        t1.getPriority()); //2

                System.out.println("t2 thread priority : " +
                                        t2.getPriority()); //5
                System.out.println("t3 thread priority : " +
                                        t3.getPriority());//8

                // Main thread
                System.out.print(Thread.currentThread().getName());
                System.out.println("Main thread priority : "
                                + Thread.currentThread().getPriority());

                // Main thread priority is set to 10
                Thread.currentThread().setPriority(10);
                System.out.println("Main thread priority : "
                                + Thread.currentThread().getPriority());
        }
}
```

Output:

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Main thread priority : 5
Main thread priority : 10
```

Note:

- Thread with highest priority will get execution chance prior to other threads. Suppose there are 3 threads t1, t2 and t3 with priorities 4, 6 and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.

- Default priority for main thread is always 5, it can be changed later. Default priority for all other threads depends on the priority of parent thread.

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

}
```

Here, the objectidentifier is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples, where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

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.
```