Multithreading in Java

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

2] Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

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

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

Advantage of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at 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 exceptions occur in a single thread.

Multitasking

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

1] Process-based Multitasking (Multiprocessing)

2] Thread-based Multitasking (Multithreading)

1] Process-based Multitasking (Multiprocessing)

- Each process has its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2] Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

> ***Note: At least one process is required for each thread.***

Eg:

In MS-Word (i.e Process) there is auto spelling checking, line count, auto-saving etc are threads running in one process.

Thread in java

A thread is a lightweight sub process, a 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 shares a common memory area.

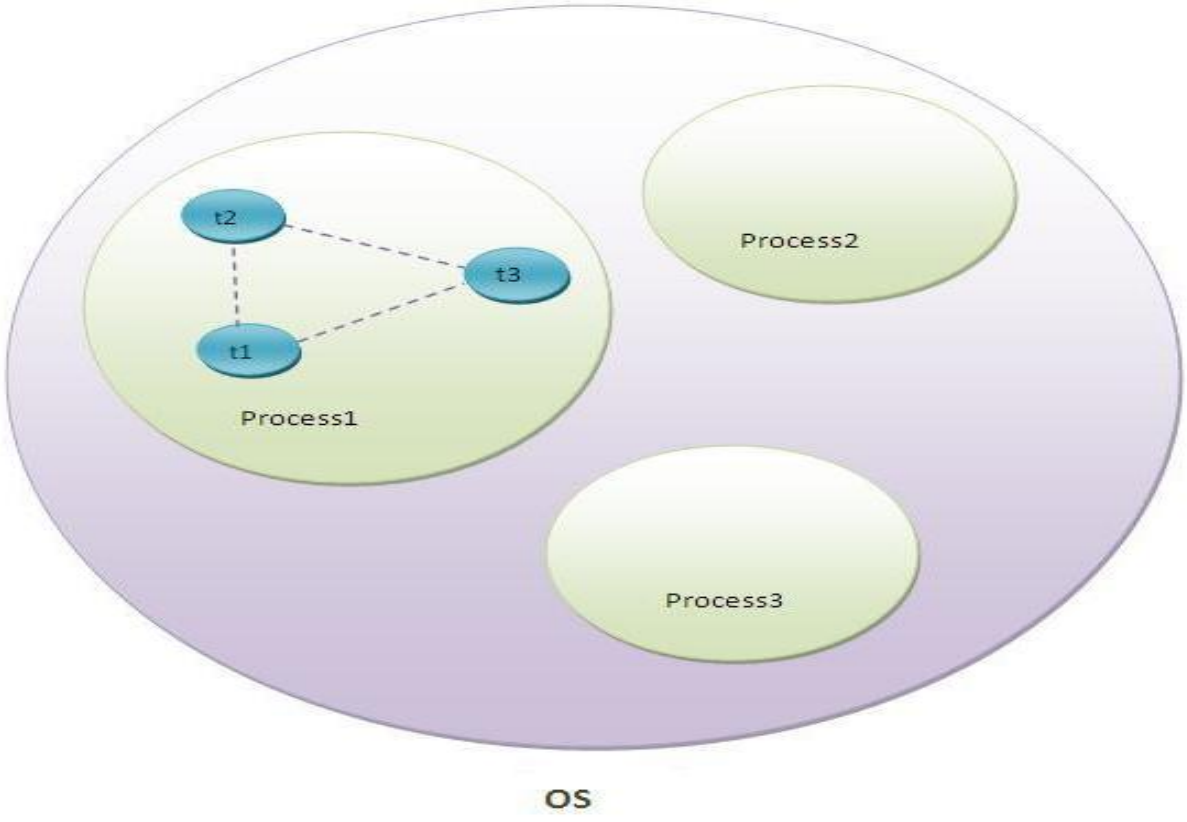


Figure shows, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

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 newborn, runnable, blocked 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:

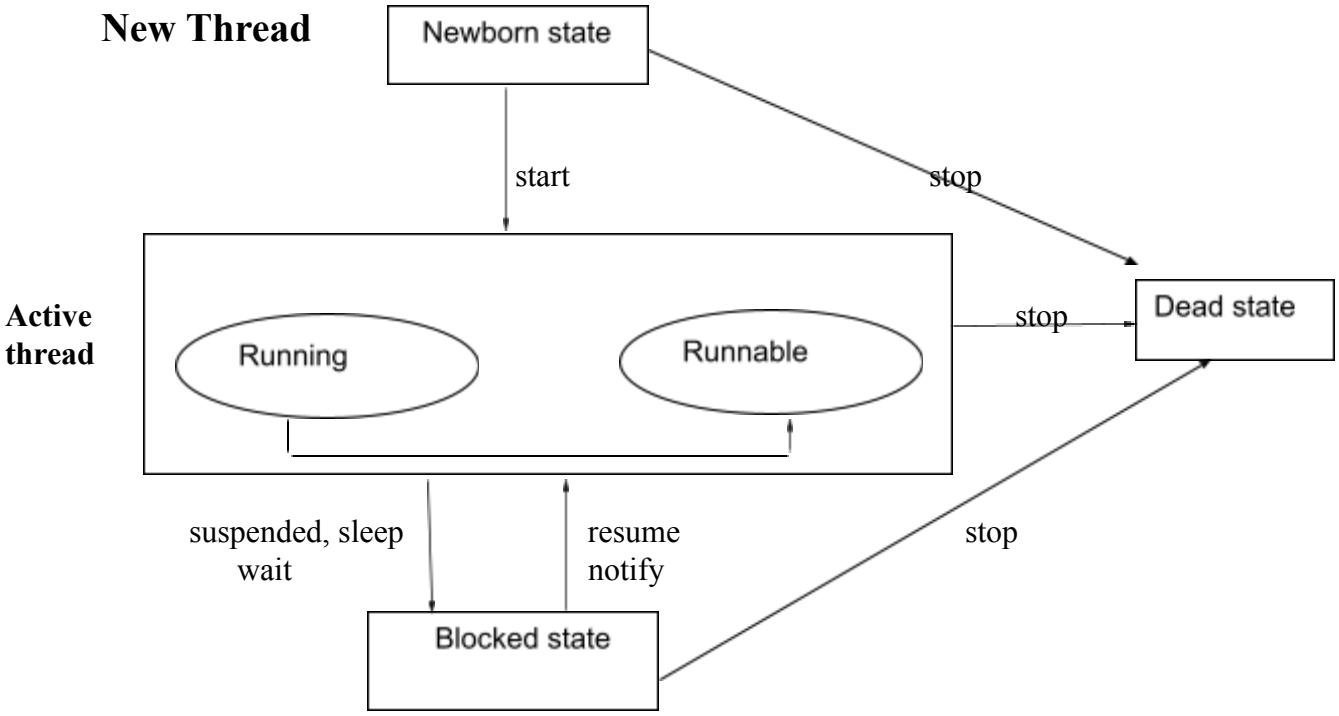
1] Newborn state

2] Runnable state

3] Running state

4] Non-Runnable (Blocked state)

5] Dead state / terminated



Idle thread

Idle thread

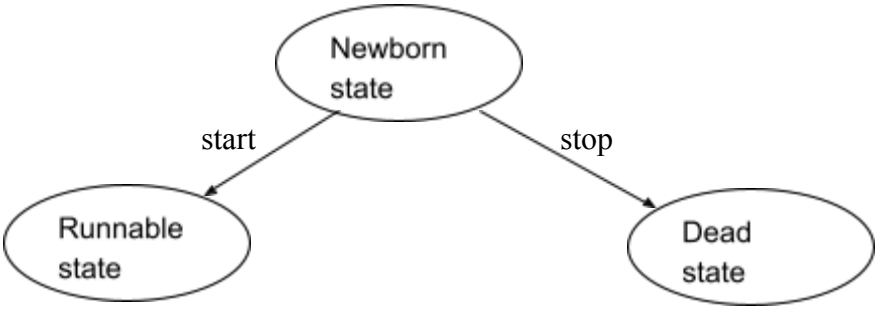
Fig: State transition diagram of a thread

1) Newborn state

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

i. Scheduled it for running using start() method.

ii. Kill it using stop() method.



2) Runnable

It means that the thread is ready for execution and is waiting for the availability of the processor. That is the thread has joined the queue of threads that are waiting for execution. If all the threads have equals priority, then they are given time slots for execution in round robin fashion i.e first come first serve manner. The thread that relinquishes control and join the queue at the end and again waits for its turn. This process of assigning time to threads is known as time-slicing.

However, if we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using the yield() method.

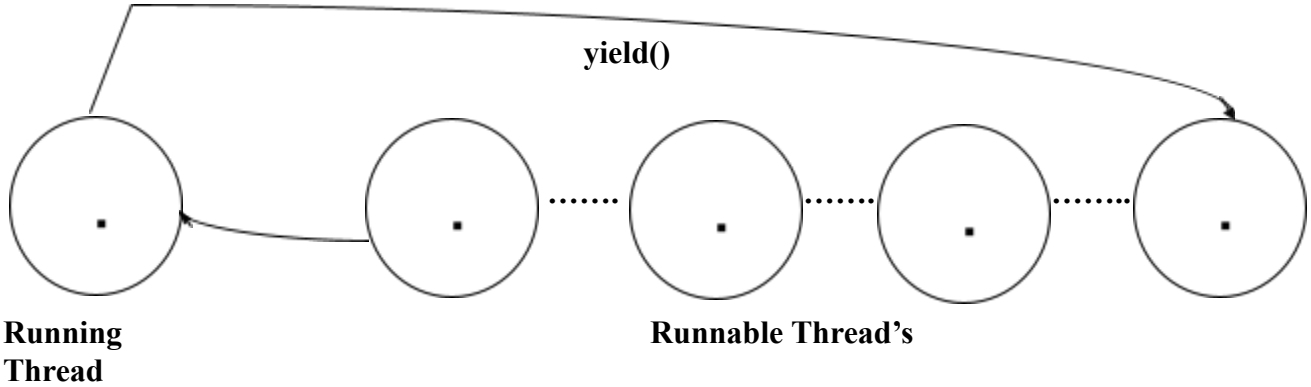


Fig: Relinquishing control using yield() method.

3) Running

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own. A running thread may relinquish its control in one of the following situation:

i. It has been suspended using suspend() method. A suspended thread can be revived by using the resume() method. This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.

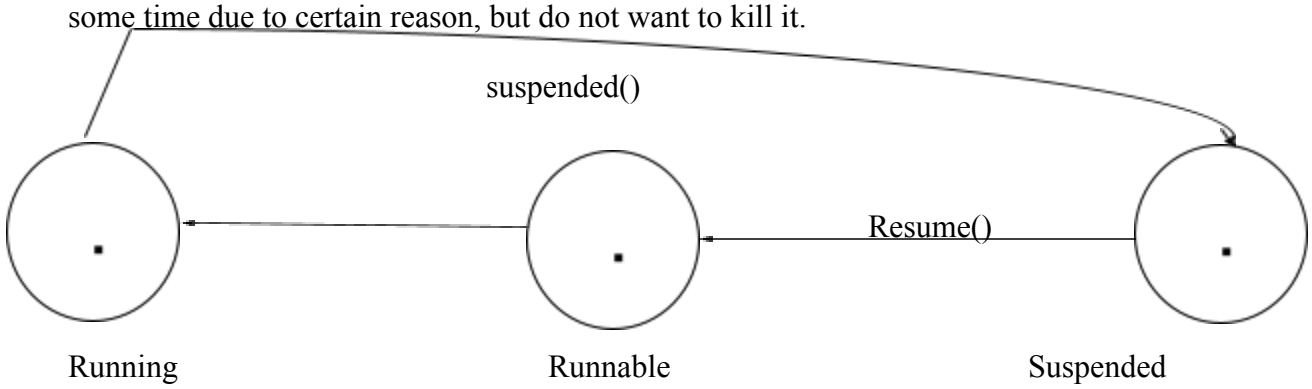


Fig: Relinquishing control using suspend() method.

ii. It has been made to sleep. We can put a thread to sleep for a specified time period using the method sleep(time) where time is in milliseconds. This means that the thread is out queue during this time period. The thread re-enters the runnable state as soon as this time period is finished.

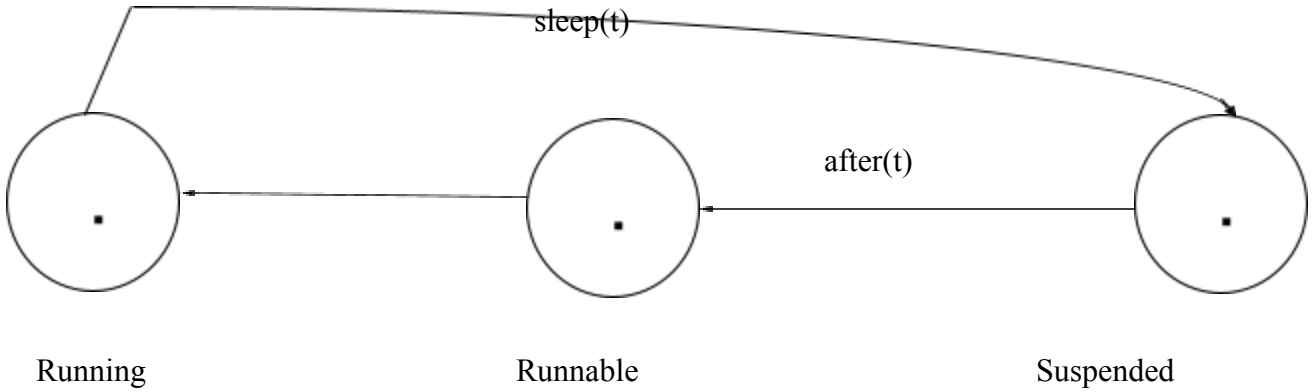


Fig: Relinquishing control using sleep() method.

iii. It has been told to wait until some event occurs. This is done using the wait() method. The thread can be scheduled to run again using the notify() method.

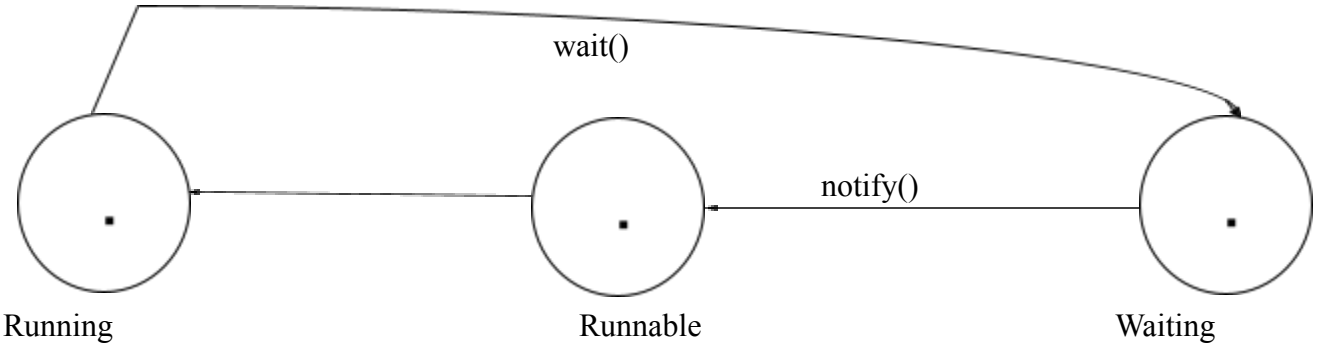


Fig: Relinquishing control using wait() method.

4) Blocked

A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements. A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

5) Dead state / Terminated

A running thread ends its life when it has completed executing its run() method. It is a natural death. However we can kill it by sending stop message using stop() method to it at any state thus causing a premature death. A thread can be killed as soon as it born or while it is running or even when it is in blocked state.

How to create thread

Creating threads in java is simple. Threads are implemented in the form of Objects that contain a method called "run()" method. The run() method is the heart and soul of any thread. It makes up the entire body of a thread in which thr thread's behavior can be implemented. A run() method would appear as follows:

Syntax:

public void run()

{

}

The run() method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called "start()" method.

There are two ways to create a thread:

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

By extending Thread class:

1. Declaring the class as extending the "Thread" class.
2. Implementing the "run()" method.
3. Creating a thread object and call the start() method to initiate the thread execution.

Example:

```
class DemoThread extends Thread
{

public void run()
{
        int i;
        for(i=1;i<=5;i++)
        {
                try{
                        Thread.sleep(1000);
                }catch(Exception ex){ex.printStackTrace();}

                System.out.println(i);
        }
}
```

```
public static void main(String args[])
{
  DemoThread d1=new DemoThread();
  DemoThread d2=new DemoThread();

        d1.start();
        d2.start();
}

}
```

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.

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 Newborn state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax

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 DemoSleep 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[]){
  DemoSleep t1=new DemoSleep ();
```

```
 DemoSleep t2=new DemoSleep ();

  t1.start();    t2.start();  }  }
```

Note: As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

Can we start a thread twice?

No. After starting a thread, it can never be started again. If you do so, an IllegalThreadStateException is thrown. In such case, thread will run once but for second time, it will throw exception.

Example

```
public class TestThreadTwice extends Thread{

 public void run(){

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

 }

 public static void main(String args[]){

  TestThreadTwice t1=new TestThreadTwice();

  t1.start();

  t1.start();

 }

}
```

Output: running

Exception in thread "main" java.lang.IllegalThreadStateException

What if we call run() method directly instead start() method?

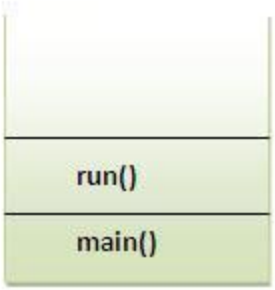
- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
class TestCallRun extends Thread{

 public void run(){

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

```
}
 public static void main(String args[]){
  TestCallRun t1=new TestCallRun();
  t1.run();//fine, but does not start a separate call stack
} }
```



Stack
(main thread)

```
class TestCallRun 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[]){
  TestCallRun t1=new TestCallRun();
  TestCallRun t2=new TestCallRun();
```

```
  t1.run();
  t2.run();
 }
}
```

Output:1

2

3

4

5

1

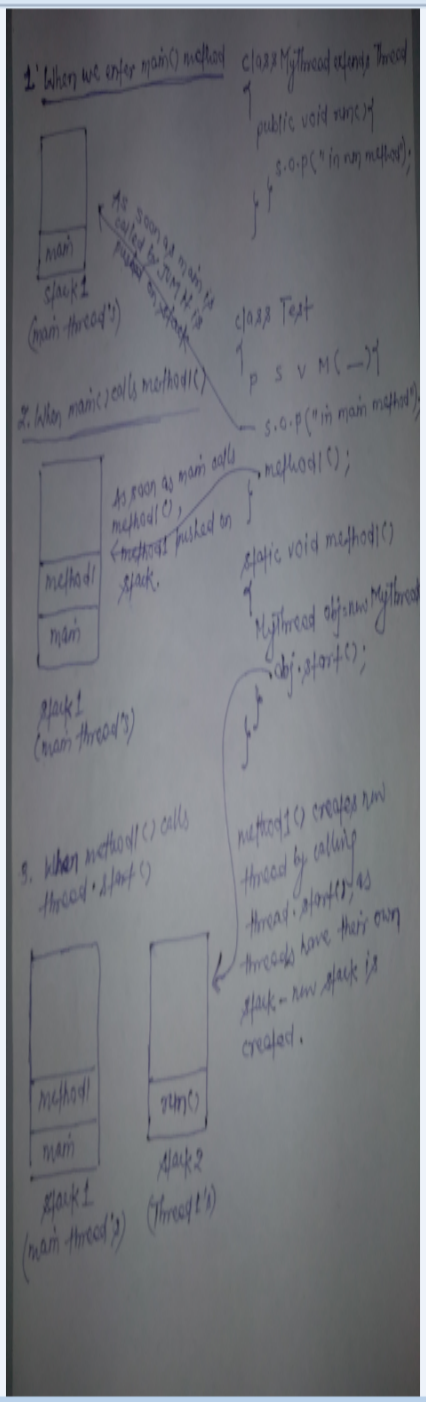
2

3

4

5

As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.


1. When we enter main() method
class MyThread extends Thread
public void run()
s.o.p("in run method");
main
stack1
(main thread)
As soon as main is called by JVM it is pushed to stack
class Test
p s v m(--)
s.o.p("in main method");
methodA();
2. When main() calls methodA()
As soon as main calls methodA(), methodA pushed to stack.
methodA
main
stack1
(main thread)
static void methodA()
MyThread obj=new MyThread();
obj.start();
methodA() creates new thread by calling thread.start(), as threads have their own stack - new stack is created.
3. When methodA() calls thread.start()
methodA
main
stack1
(main thread)
run()
stack2
(Thread)

The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

- public void join()throws InterruptedException
- public void join(long milliseconds)throws InterruptedException

Example of join()

```
class TestJoinMethod 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[]){
 TestJoinMethod t1=new TestJoinMethod();
 TestJoinMethod t2=new TestJoinMethod();
 TestJoinMethod t3=new TestJoinMethod();
 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}
   t2.start();
 t3.start();
```

```
 }
}
```

In the above example, when t1 completes its task then t2 and t3 starts executing.

Example of join(long miliseconds) method

```
class TestJoinMethod 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[]){
 TestJoinMethod t1=new TestJoinMethod();
 TestJoinMethod t2=new TestJoinMethod();
 TestJoinMethod t3=new TestJoinMethod();
 t1.start();
 try{
  t1.join(1500);
 }catch(Exception e){System.out.println(e);}
   t2.start();
 t3.start();
 }
}
```

In the above example, when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing.

getName(),setName(String) and getId() method:

```
class demo extends Thread{
  public void run(){
   System.out.println("running...");
  }
 public static void main(String args[]){
  demo t1=new demo ();
  demo t2=new demo ();
  System.out.println("Name of t1:"+t1.getName());
  System.out.println("Name of t2:"+t2.getName());
  System.out.println("id of t1:"+t1.getId());

  t1.start();
  t2.start();

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

The currentThread() method:

The currentThread() method returns a reference to the currently executing thread object.

Example

```
class Demo extends Thread{
 public void run(){
  System.out.println(Thread.currentThread().getName());
 }
 }
 public static void main(String args[]){
  Demo t1=new Demo();
  Demo t2=new Demo();

  t1.start();
  t2.start();
 }
}
```