Java Threads

A **thread in Java** simply represents a single independent path of execution of a group of statements. It is the flow of execution, from beginning to end, of a task.

When we write a group of statements in a program, these statements are executed by JVM one by one. This execution process is called thread in Java.

There is always at least one thread running internally in every program and this thread is used by JVM to execute statements in the program. Every java program creates at least one thread [main() thread]. Additional threads are created through the Thread constructor or by instantiating classes that extend the Thread class.

When a program contains a single flow of control, it is called single-threaded program. In a single thread program, there is a beginning, a body, and an end, and execute commands sequentially. Look at the below figure.

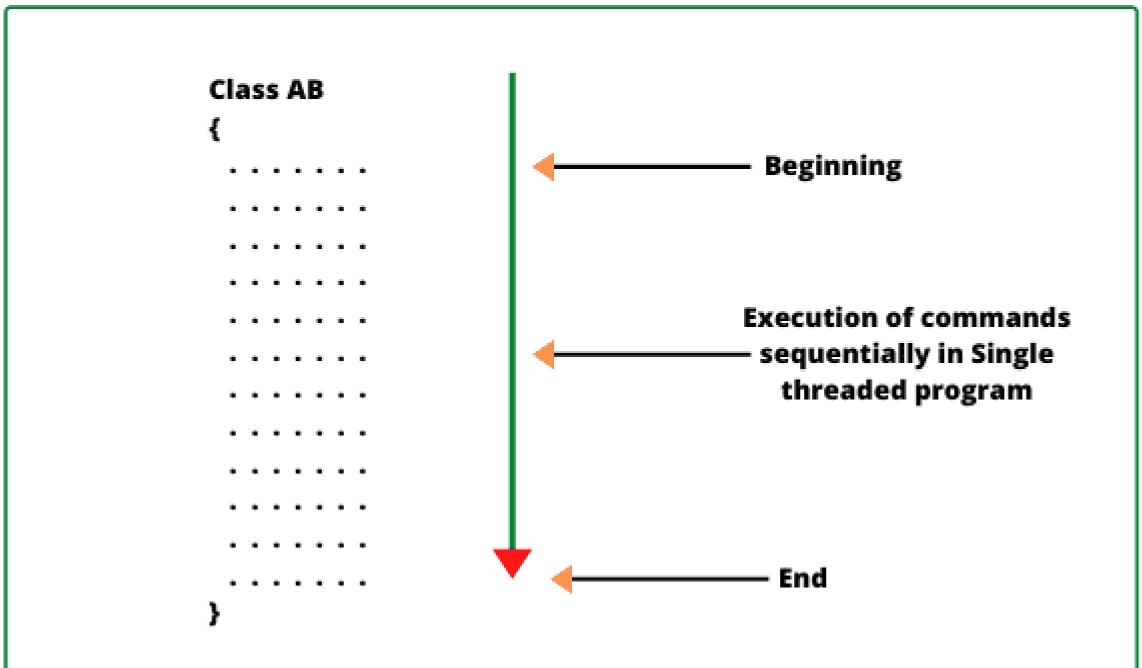

Class AB
{
}
Beginning
Execution of commands sequentially in Single threaded program
End

Fig: Single-threaded program

We can also create more than one execution thread in a program that can be used to perform multiple tasks simultaneously.

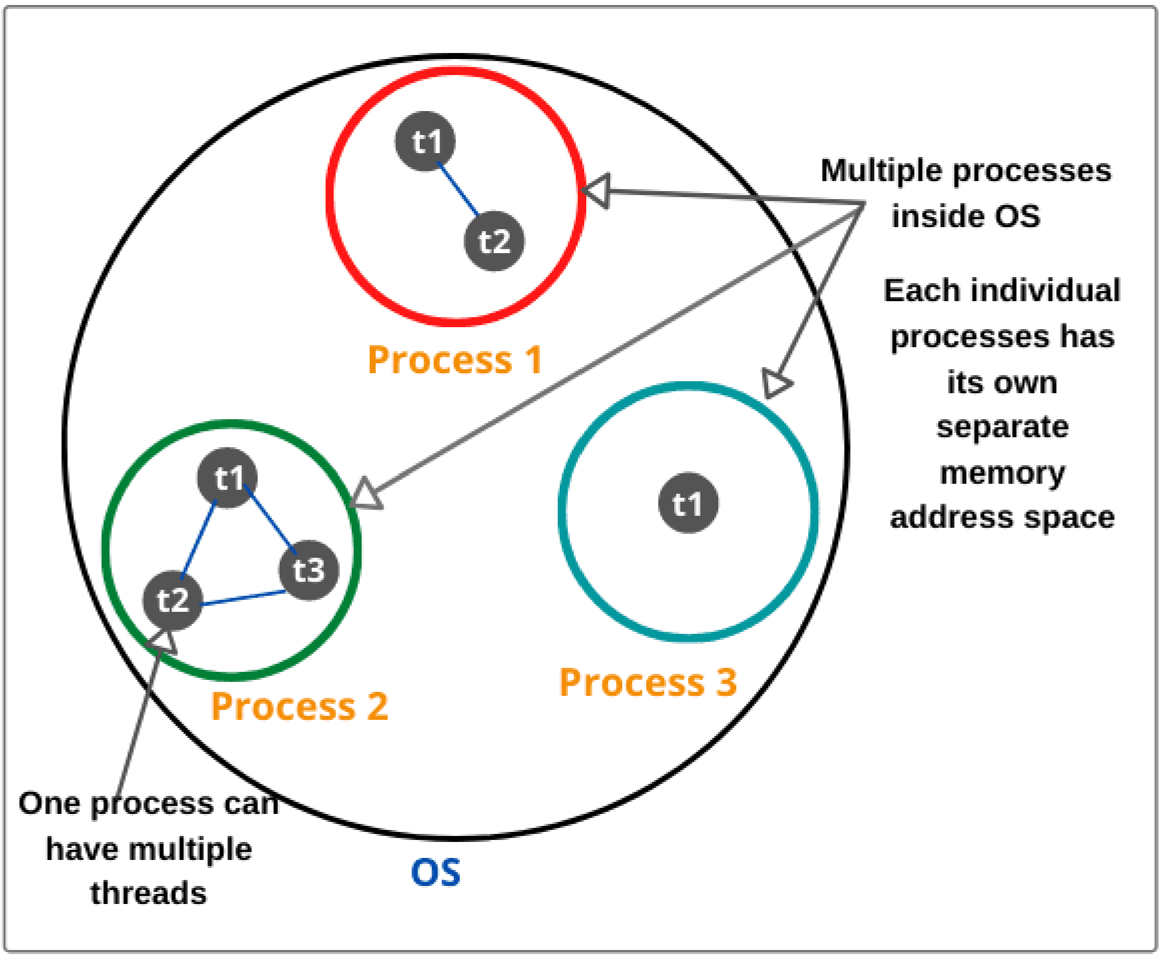
When we create more than one thread in a program, each thread has its own path of execution and all the threads share the same memory address space and data

A Thread in Java is the smallest unit of executable code in a program. It helps to divide a program into multiple parts to speed up the process.

A process is a program that executes as a single thread. In other words, when an executable program is loaded into memory, it is called process.

When we will create a new thread in a program, it shares the same memory address space with other threads in a program whereas every individual process has its own separate memory address space.

Therefore, creating a thread takes fewer resources than creating a new process. Look at the below figure


t1
t2
Process 1
Multiple processes inside OS
Each individual processes has its own separate memory address space
t1
t2
t3
Process 2
t1
Process 3
One process can have multiple threads
OS

In the above figure, the thread is executed inside a process and one process can have also multiple threads. There can be multiple processes inside the operating system.

Multiple Threads are independent of each other. At a time only one thread is executed. If an exception occurs in one thread, it doesn't affect other threads.

What's the need of a thread or why we use Threads?

- To perform asynchronous or background processing
- Increases the responsiveness of GUI applications
- Take advantage of multiprocessor systems
- Simplify program logic when there are multiple independent entities

What happens when a thread is invoked?

When a thread is invoked, there will be two paths of execution. One path will execute the thread and the other path will follow the statement after the thread invocation. There will be a separate stack and memory space for each thread.

Thread creation in Java

Thread implementation in java can be achieved in two ways:

1. Extending the java.lang.Thread class
2. Implementing the java.lang.Runnable Interface

Note: The Thread and Runnable are available in the java.lang.* package

1) By extending thread class

- The class should extend Java Thread class.
- The class should override the run() method.
- The functionality that is expected by the Thread to be executed is written in the run() method.

void start(): Creates a new thread and makes it runnable.
void run(): The new thread begins its life inside this method.

Example:

```
public class MyThread extends Thread {
   public void run(){
    System.out.println("thread is running...");
  }
   public static void main(String[] args) {
     MyThread obj = new MyThread();
     obj.start();
}
```

2) By Implementing Runnable interface

- The class should implement the Runnable interface
- The class should implement the run() method in the Runnable interface
- The functionality that is expected by the Thread to be executed is put in the run() method

Example:

```
public class MyThread implements Runnable {
   public void run(){
     System.out.println("thread is running..");
   }
   public static void main(String[] args) {
     Thread t = new Thread(new MyThread());
     t.start();
}
```

Ending a Thread

A Thread ends due to the following reasons:

- The thread ends when it comes when the run() method finishes its execution.
- When the thread throws an Exception or Error that is not being caught in the program.
- Java program completes or ends.
- Another thread calls stop() methods.

Life Cycle of Thread in Java is basically state transitions of a thread that starts from its birth and ends on its death.

When an instance of a thread is created and is executed by calling start() method of *Thread class*, the thread goes into runnable state.

When sleep() or wait() method is called by Thread class, the thread enters into non-runnable state.

From non-runnable state, thread comes back to runnable state and continues execution of statements. When the thread comes out of run() method, it dies. These state transitions of a thread are called **Thread life cycle in Java**.

To work with threads in a program, it is important to identify thread state. So. let's understand how to identify thread states in Java thread life cycle.

Thread States in Java

A thread is a path of execution in a program that enters in any one of the following five states during its life cycle. The five states are as follows:

1. New

2. Runnable

3. Running

4. Blocked (Non-runnable state)

5. Dead

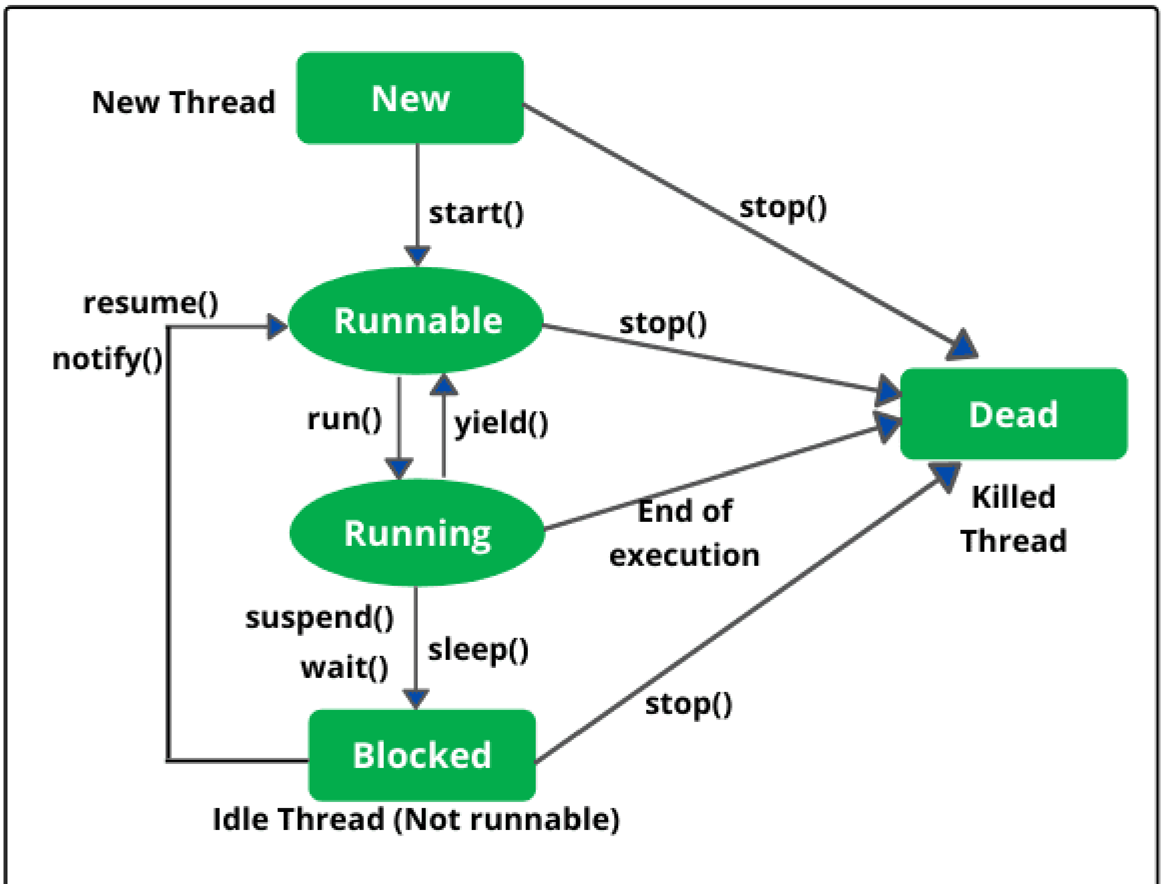


Fig: State Transition Diagram of a Thread

1. New (Newborn State): When we create a thread object using Thread class, thread is born and is known to be in Newborn state. That is, when a thread is born, it enters into new state but the start() method has not been called yet on the instance.

In other words, Thread object exists but it cannot execute any statement because it is not an execution of thread. Only start() method can be called on a new thread; otherwise, an **IllegalThreadStateException** will be thrown.

2. Runnable state: Runnable state means a thread is ready for execution. When the start() method is called on a new thread, thread enters into a runnable state.

In runnable state, thread is ready for execution and is waiting for availability of the processor (CPU time). That is, thread has joined queue (line) of threads that are waiting for execution.

If all threads have equal priority, CPU allocates time slots for thread execution on the basis of first-come, first-serve manner. The process of allocating time to threads is known as **time slicing**. A thread can come into runnable state from running, waiting, or new states.

3. Running state: Running means Processor (CPU) has allocated time slot to thread for its execution. When thread scheduler selects a thread from the runnable state for execution, it goes into running state. Look at the above figure.

In running state, processor gives its time to the thread for execution and executes its run method. This is the state where thread performs its actual functions. A thread can come into running state only from runnable state.

A running thread may give up its control in any one of the following situations and can enter into the blocked state.

1. When sleep() method is invoked on a thread to sleep for specified time period, the thread is out of queue during this time period. The thread again reenters into the runnable state as soon as this time period is elapsed.

2. When a thread is suspended using suspend() method for some time in order to satisfy some conditions. A suspended thread can be revived by using resume() method.

3. When wait() method is called on a thread to wait for some time. The thread in wait state can be run again using notify() or notifyAll() method.

4. Blocked state: A thread is considered to be in the blocked state when it is suspended, sleeping, or waiting for some time in order to satisfy some condition.

5. Dead state: A thread dies or moves into dead state automatically when its run() method completes the execution of statements. That is, a thread is terminated or dead when a thread comes out of run() method. A thread can also be dead when the stop() method is called.

During the life cycle of thread in Java, a thread moves from one state to another state in a variety of ways. This is because in multithreading environment, when multiple threads are executing, only one thread can use CPU at a time.

All other threads live in some other states, either waiting for their turn on CPU or waiting for satisfying some conditions. Therefore, a thread is always in any of the five states.