Life Cycle of Thread in Java

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.

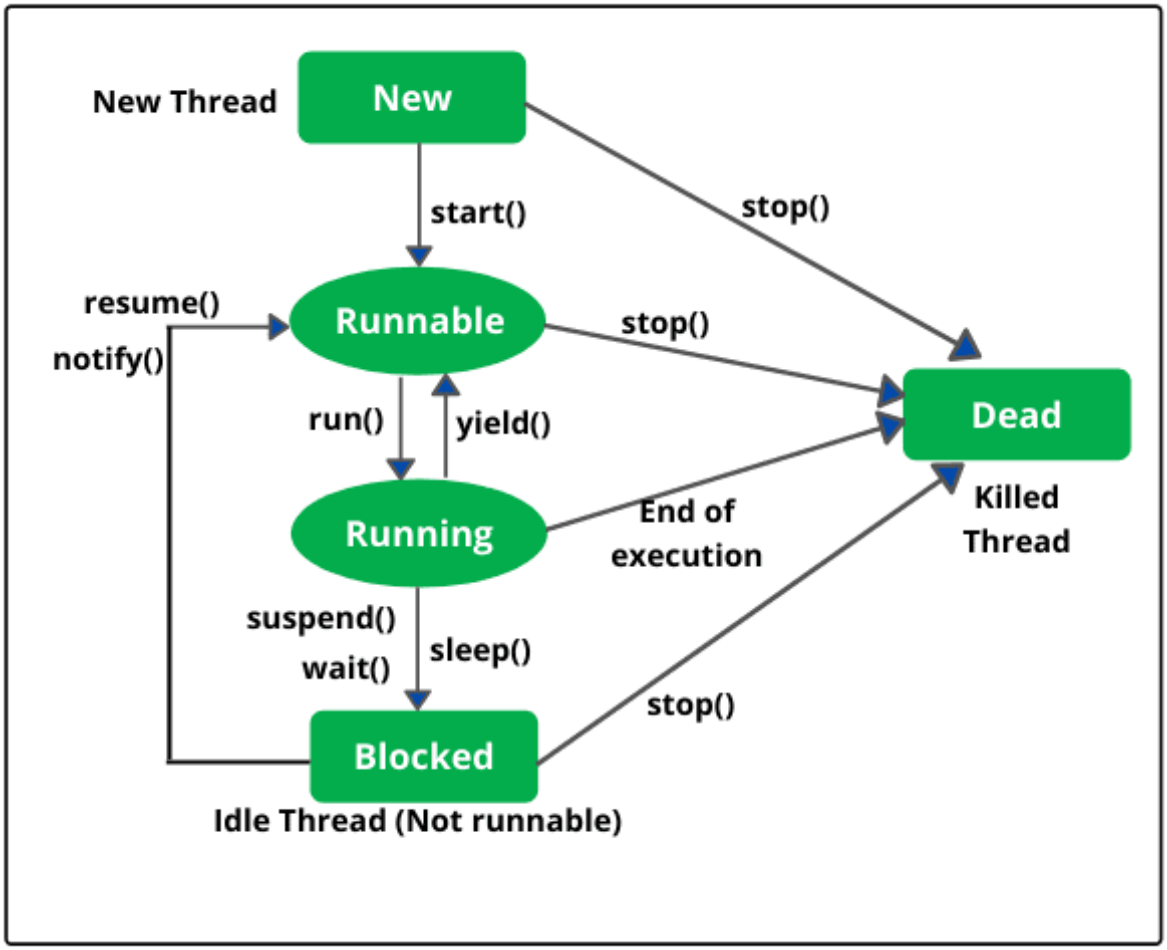
When sleep() or wait() method is called by Thread class, the thread enters into non-runnable state.

From non-runnable state, thread comes back to runnable state and continues execution of statements.

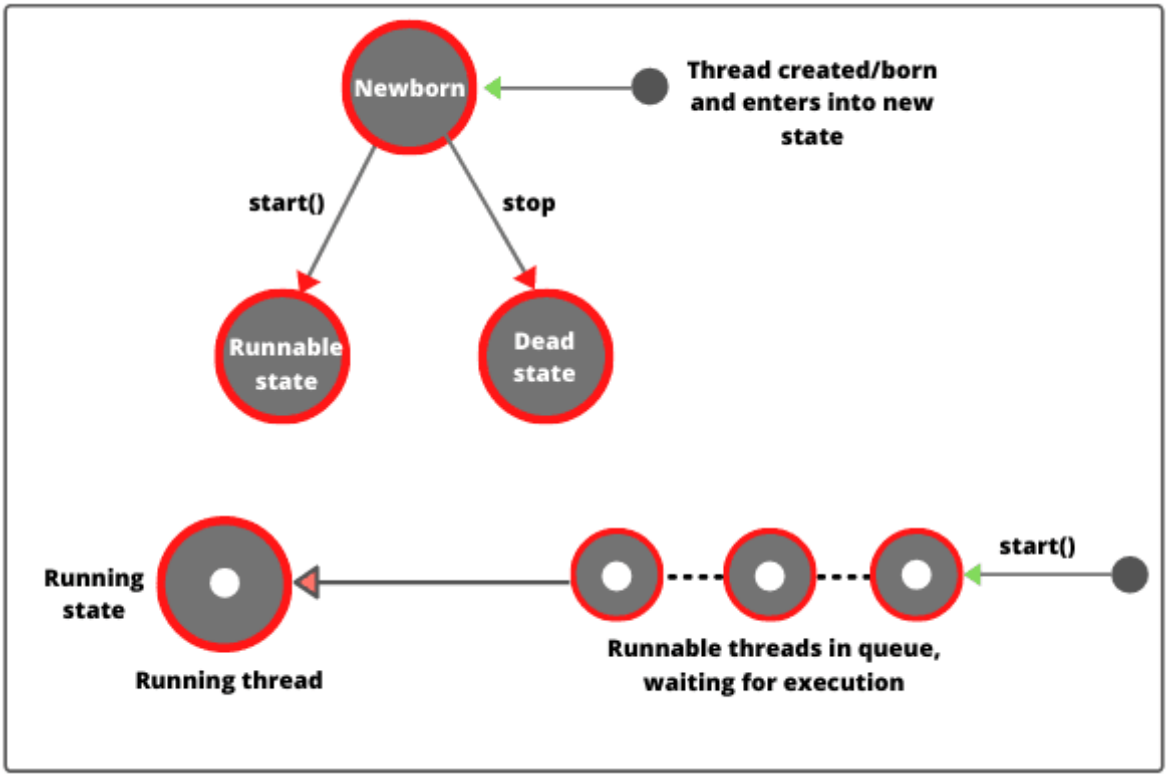
When the thread comes out of run() method, it dies. These state transitions of a thread are called **Thread life cycle in Java**.

Thread States in Java

1. New
2. Runnable
3. Running
4. Blocked (Non-runnable state)
5. Dead



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,

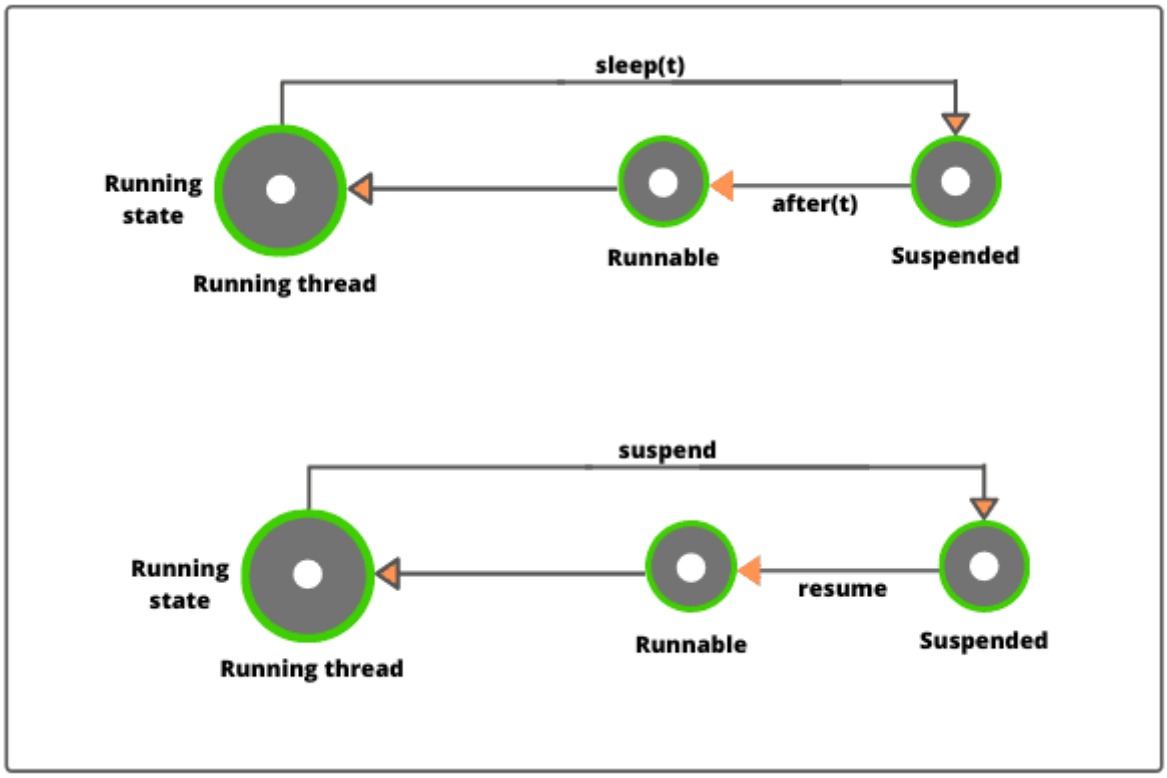
In running state, processor gives its time to the thread for execution and executes its run method. This is the state where thread performs its actual functions. A thread can come into running state only from runnable state.

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

a) 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.

b) 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.

c) 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.

Two threads performing two tasks at a time

```
// Two threads performing two tasks at a time.
public class MyThread extends Thread
{
// Declare a String variable to represent task.
  String task;

MyThread(String task)
{
  this.task = task;
}
public void run()
{
 for(int i = 1; i <= 5; i++)
 {
  System.out.println(task+ " : " +i);
  try
 {
    Thread.sleep(5000); // Pause the thread execution for 1000 milliseconds.
 }
 catch(InterruptedException ie) {
   System.out.println(ie.getMessage());
  }
 } // end of for loop.
} // end of run() method.




public static void main(String[] args)
{
// Create two objects to represent two tasks.
  MyThread th1 = new MyThread("Cut the ticket");

  MyThread th2 = new MyThread("Show your seat number");

// Create two objects of Thread class and pass two objects as parameter
//to constructor of Thread class.
  Thread t1 = new Thread(th1);

  Thread t2 = new Thread(th2);


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

Multiple Threads acting on Single object

```
public class MultipleThread implements Runnable
{
 String task;
 MultipleThread(String task)
 {
  this.task = task;
 }
 public void run()
 {
  for(int i = 1; i <= 5; i++)
  {
    System.out.println(task+ ":" +i);
  try {
     Thread.sleep(1000);
  } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
 }
public static void main(String[] args)
{
 Thread nThread = Thread.currentThread();
  System.out.println("Name of thread: " +nThread);

// Multiple child threads acting on single object.
   MultipleThread mt = new MultipleThread("Hello Java");
   Thread t1 = new Thread(mt);
   Thread t2 = new Thread(mt);
   Thread t3 = new Thread(mt);
    t1.start();
    t2.start();
    t3.start();

 int count = Thread.activeCount();
 System.out.println("No of active threads: " +count);
  }
}
```

Two different tasks by two different threads

```
public class MyThread1 implements Runnable
{
public void run() // Entry point of Thread1
{
 for(int i = 0; i < 5; i++)
 {
    System.out.println("First Child Thread: " +i);
  }
 System.out.println("\t First child existed");
 }
}
public class MyThread2 implements Runnable
{
```

```
public void run() // Entry point of Thread2
{
 for(int i = 0; i < 5; i++)
 {
    System.out.println("Second Child Thread: " +i);
 }
System.out.println("Second child existed");
 }
}
public class MyClass {
public static void main(String[] args)
{
 MyThread1 th1 = new MyThread1();
 Thread t1 = new Thread(th1);
  t1.start(); // Execution of first thread is started.

MyThread2 th2 = new MyThread2();
Thread t2 = new Thread(th2);
 t2.start(); // Execution of second thread is started.

int j = 0;
while(j < 4)
{
 System.out.println("Main Thread: " +j);
 j = j + 1;
}
System.out.println("\t Main thread existing");
  }
}
```

Thread scheduler in Java

Thread scheduler in Java is the component of JVM that determines the execution order of multiple threads on a single processor (CPU).

It decides the order in which threads should run. This process is called **thread scheduling in Java**.

When a system with a single processor executes a program having multiple threads, CPU executes only a single thread at a particular time.

Other threads in the program wait in Runnable state for getting the chance of execution on CPU because at a time only one thread can get the chance to access the CPU.

Java runtime system mainly uses one of the following two strategies:

1. **Preemptive scheduling**
2. **Time-sliced scheduling**

Preemptive Scheduling

This scheduling is based on priority. Therefore, this scheduling is known as priority-based
scheduling. In the priority-based scheduling algorithm, Thread scheduler uses the priority to decide which thread should be run.

If a thread with a higher priority exists in Runnable state (ready state), it will be scheduled to run immediately.

In case more than two threads have the same priority then CPU allocates time slots for thread execution on the basis of first-come, first-serve manner.

Time-Sliced Scheduling

The process of allocating time to threads is known as **time slicing in Java**. Time-slicing is based on non-priority scheduling. Under this scheduling, every running thread is executed for a fixed time period.

A currently running thread goes to the Runnable state when its time slice is elapsed and another thread gets time slots by CPU for execution.

With time-slicing, threads having lower priorities or higher priorities gets the same time slots for execution.

Thread Priority in Java

Thread priority in Java is a number assigned to a thread that is used by *Thread scheduler* to decide which thread should be allowed to execute.

In Java, each thread is assigned a different priority that will decide the order (preference) in which it is scheduled for running.

Thread priorities are represented by a number from 1 to 10 that

The default priority of a thread is 5. *Thread class in Java* also provides several priority constants to define the priority of a thread. These are:

1. MIN_PRIORITY = 1

2. NORM_PRIORITY = 5

3. MAX_PRIORTY = 10

```
public class A implements Runnable
{
public void run()
{
  System.out.println(Thread.currentThread()); // This method is static.
}
public static void main(String[] args)
{
 A a = new A();
 Thread t = new Thread(a, "NewThread");

 System.out.println("Priority of Thread: " +t.getPriority());
 System.out.println("Name of Thread: " +t.getName());
 t.start();
  }
}
```

How to set Priority of Thread in Java

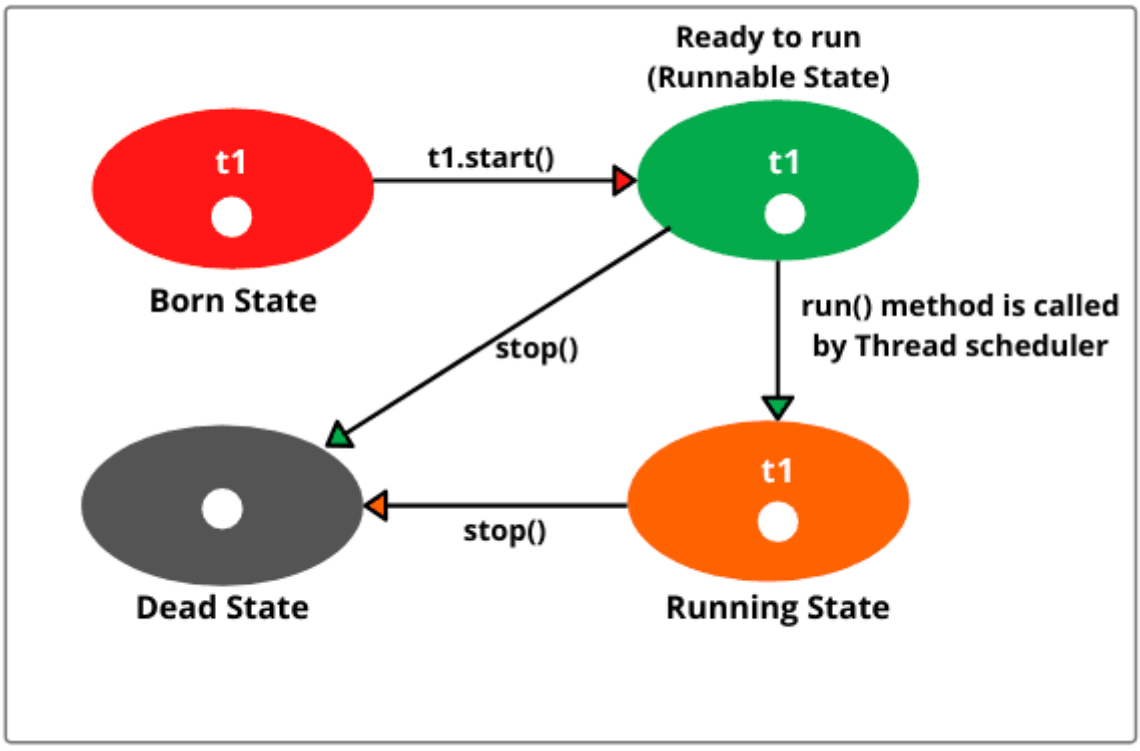
```
public class A implements Runnable
{
public void run()
{
  System.out.println(Thread.currentThread()); // This method is static.
}
public static void main(String[] args)
{
 A a = new A();
 Thread t = new Thread(a, "NewThread");
 t.setPriority(2); // Setting the priority of thread.

 System.out.println("Priority of Thread: " +t.getPriority());
 System.out.println("Name of Thread: " +t.getName());



 Thread t1 = new Thread(a, "First Thread");
 Thread t2 = new Thread(a, "Second Thread");
 Thread t3 = new Thread(a, "Third Thread");

 t1.setPriority(4); // Setting the priority of first thread.
 t2.setPriority(3); // Setting the priority of second thread.
 t3.setPriority(8); // Setting the priority of third thread.

 t.start();
 t1.start();
 t2.start();
 t3.start();

  }
}
```

```
public class X implements Runnable
{
public void run()
{
 System.out.println("Thread X started");
 for(int i = 1; i<=4; i++)
 {
   System.out.println("Thread X: " +i);
 }
System.out.println("Exit from X");
 }
}
public class Y implements Runnable
{
public void run()
{
 System.out.println("Thread Y started");
 for(int j = 0; j <= 4; j++)
 {
  System.out.println("Thread Y: " +j);
 }
 System.out.println("Exit from Y");
}
}
public class Z implements Runnable
{
public void run()
{
 System.out.println("Thread Z started");
 for(int k = 0; k <= 4; k++)
 {
  System.out.println("Thread Z: " +k);
 }
 System.out.println("Exit from Z");
 }
}
public class ThreadPriority {
public static void main(String[] args)
{
 X x = new X();
 Y y = new Y();
 Z z = new Z();

Thread t1 = new Thread(x);
Thread t2 = new Thread(y);
Thread t3 = new Thread(z);

t1.setPriority(Thread.MAX_PRIORITY);
t2.setPriority(t2.getPriority() + 4);
t3.setPriority(Thread.MIN_PRIORITY);

t1.start();
t2.start();
t3.start();
 }
}
```

How to Stop Thread

A thread in Java program will terminate (or move to dead state) automatically when it comes out of run() method.

But if we want to stop a thread from running or runnable state, we will need to calling stop() method of Thread class. The stop() method is generally used when we desire **premature death** of a thread.



How to stop Running Thread

```
public class Kill extends Thread
{
// Declare a static variable to of type Thread.
   static Thread t;

public void run()
{
 System.out.println("Thread is running");
 t.stop(); // Calling stop() method on Kill Thread.
 System.out.println("Hi i am Alive");
}
```

```
public static void main(String[] args)
{
 Kill k = new Kill();
  t = new Thread(k);
  t.start(); // Calling start() method.
 }
}
```

How to stop Runnable Thread

```
public class Thread1 implements Runnable
{
public void run()
{
 System.out.println("First child thread");
 }
}
public class Thread2 implements Runnable
{
static Thread t2;
public void run()
{
for(int i = 0; i <= 10; i ++)
{
 System.out.println("Second child thread: " +i);
 if(i==5)
 {
  t2.stop(); // Calling stop() method to kill running thread.
 }
 }
 }}
public class MyThreadClass
{
public static void main(String[] args)
{
 Thread1 th1 = new Thread1();
 Thread2 th2 = new Thread2();

Thread t1 = new Thread(th1);
Thread t2 = new Thread(th2);

 t1.start();
 t1.stop(); // Calling stop() method to kill runnable thread.
 t2.start();
 }
}
```

Can a thread is again alive when it goes into dead state

```
public class Thread1 implements Runnable
{
static Thread t1;
```

```
public void run()
{
 System.out.println("Thread is running");
 int i = 0;
while(i < 10)
{
 System.out.println("i: " +i);
 if(i == 5)
  t1.stop();
  i = i + 1;
  }
}
public static void main(String[] args)
{
 Thread1 th1 = new Thread1();
 Thread t1 = new Thread(th1);
  t1.start();
  t1.start(); // Calling the start() method again to alive a dead thread.
 }
}
```

Stopping a thread by using boolean variable

```
public class Thread1 extends Thread
{
boolean stop = false;
 public void run()
 {
  System.out.println("Thread is running");
  int i = 0;
while(i < 10)
{
 System.out.println("i: " +i);
 if(i == 5)
 if(stop == true) // Come out of run() method.
   return;
   i = i + 1;
 }
}
public static void main(String[] args)
{
 Thread1 th1 = new Thread1();
 Thread t1 = new Thread(th1);
  t1.start();
  th1.stop = true;
 }
}
```

Stopping a thread by using isInterrupted()

```
public class Thread1 extends Thread
{
```

```
 public void run()
 {
  System.out.println("Thread is running");
  int i = 0;
while(i < 10)
{
 System.out.println("i: " +i);
if(i == 5)
if(!Thread.currentThread().isInterrupted()) // Come out of run() method.
{
 System.out.println("Status of thread: " +!Thread.currentThread().isInterrupted());
 return;
}
 i = i + 1;
 }
}
public static void main(String[] args)
{
 Thread1 th1 = new Thread1();
 Thread t1 = new Thread(th1);
   t1.start();
 }
}
```

Java Thread Sleep

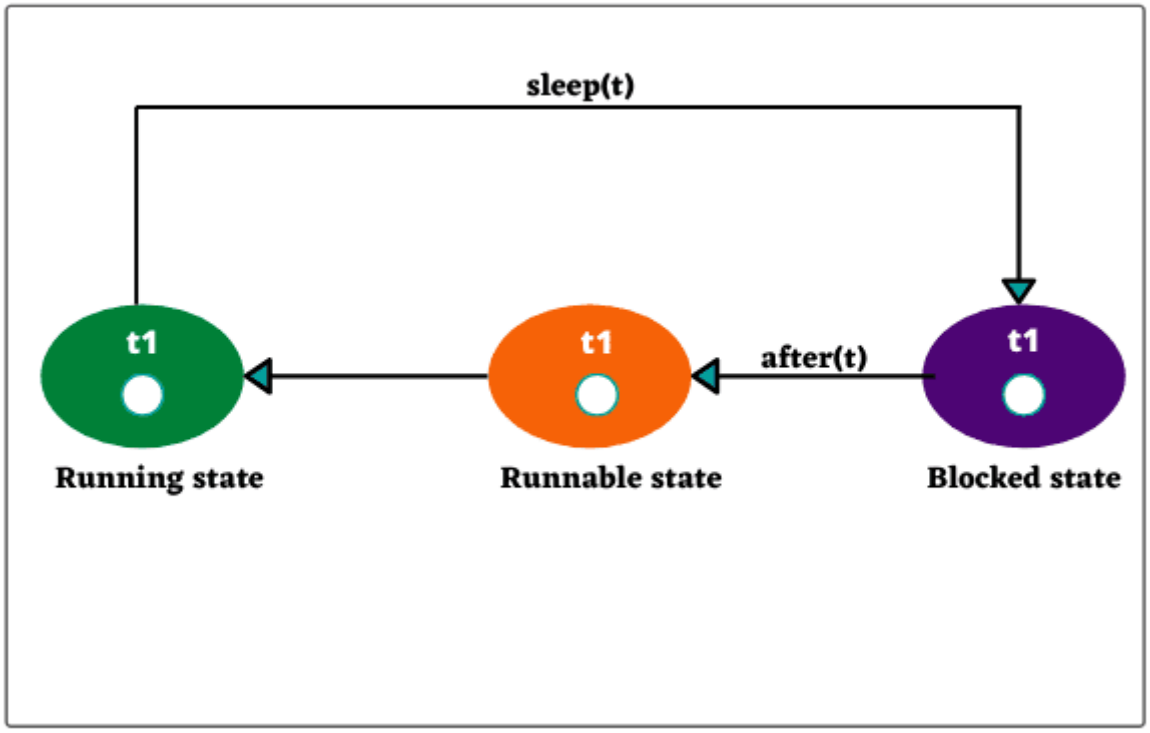
Sometimes we need to make a thread sleep for a particular period of time. For this, we use sleep() method in Java program.

The sleep() method is a static method provided by the *Thread class* so it can be called by its class name.

It is used to sleep a thread for a specified amount of time. It always pauses the current thread for a given period of time.

When the sleep() method is called on Thread object, the thread is become out of the queue and enters into blocked (or non-runnable state) for a specified amount of time.

When the specified amount of time is elapsed, the thread does not go into running state (execution state) immediately. It goes into the runnable state (ready state) until it is called by *Thread scheduler* of JVM.




```
public class A implements Runnable
{
public void run()
{
  System.out.println("Hello");
try
{
 Thread.sleep(2000); // Pausing running thread for 2 sec.
}
catch(InterruptedException ie){
 System.out.println(ie.getMessage());
 }


 System.out.println("Java");
 System.out.println(Thread.currentThread());
 }
void m1()
{
 System.out.println("m1 method");
}
public static void main(String[] args)
{
A a = new A();
Thread t = new Thread(a, "Child Thread");
 t.start();
 System.out.println("Number of active threads: " +Thread.activeCount());
 a.m1();
 }
}
```

interrupt sleep method

```
public class A implements Runnable
{
public void run()
{
try
{
 Thread.sleep(2000);
}
catch (InterruptedException ie)
{
 ie.printStackTrace();
}
System.out.println("Hello Java");
System.out.println(Thread.currentThread());
 }
public static void main(String[] args)
{
A a = new A();
Thread t = new Thread(a, "Child Thread");
 t.start();
 t.interrupt();
 }
}
```

create two threads

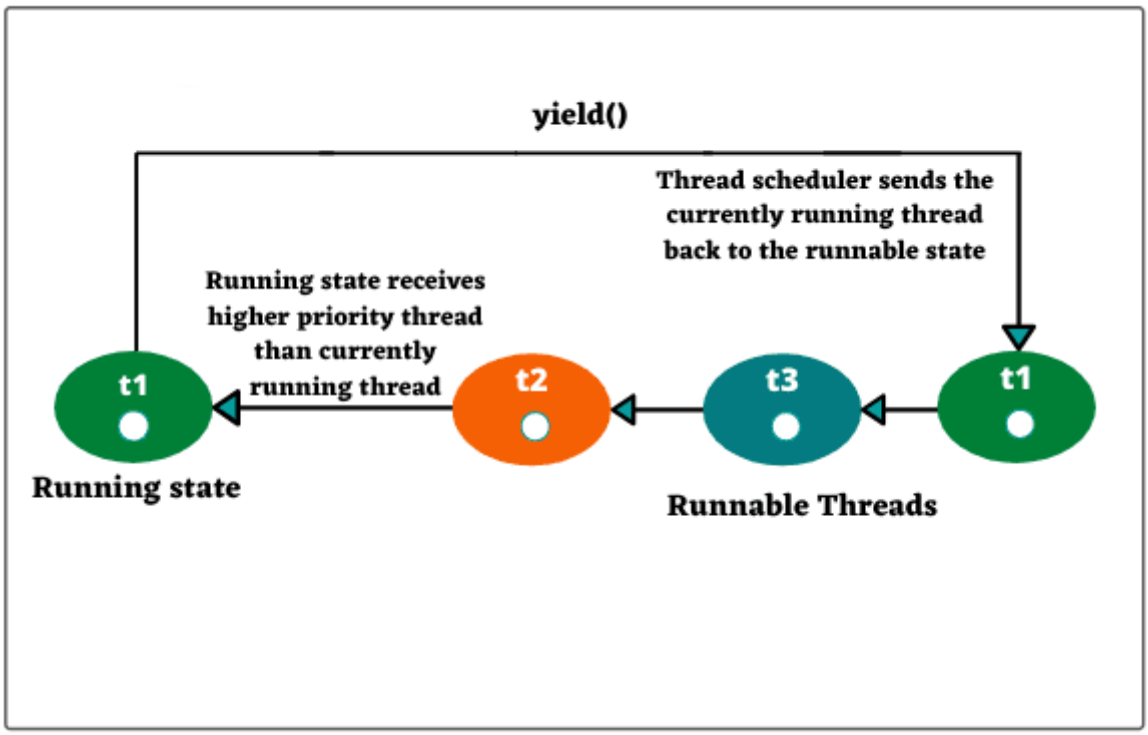
```
public class A implements Runnable
{
public void run()
{
for(int i = 1; i <= 3; i++)
{
 try
 {
  Thread.sleep(500);
 }
catch (InterruptedException ie)
{
  ie.printStackTrace();
 }
System.out.println(Thread.currentThread() + " I: " +i);
 }
}
public static void main(String[] args)
{
A a1 = new A();
Thread t1 = new Thread(a1, "First Child Thread");

A a2 = new A();
```

```
Thread t2 = new Thread(a2, "Second Child Thread");

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

Yield Method in Java

When a currently executing *thread* goes to the runnable state from running state, this process is called **yield in Java**.

When running state receives higher priority thread than the thread that is currently running, *thread scheduler*
 sends the currently running thread back to the runnable state and
selects another thread of equal or higher priority to start its
execution.




```
public class A implements Runnable
{
public void run()
{
System.out.println(Thread.currentThread());
```

```
Thread.yield(); // Calling yield() method on current thread to move back into the runn
able state from running state.

System.out.println(Thread.currentThread());
 }
public static void main(String[] args)
{
A a1 = new A();
Thread t1 = new Thread(a1, "First Child Thread");

A a2 = new A();
Thread t2 = new Thread(a2, "Second Child Thread");

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

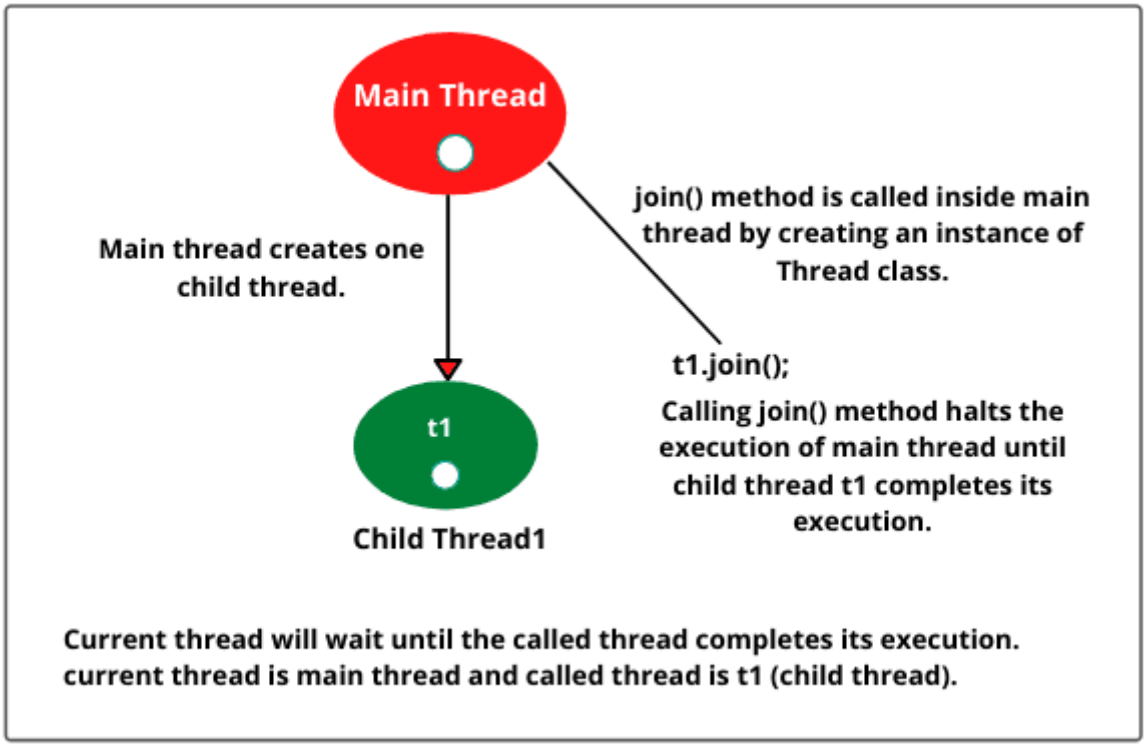
set the priority and then we will call yield()

```
public class A implements Runnable
{
public void run()
{
 System.out.println(Thread.currentThread());
 Thread.yield(); // Calling yield() method on current thread to move back into the run
nable state from running state.
 System.out.println(Thread.currentThread());
}
public static void main(String[] args)
{
 A a1 = new A();
 Thread t1 = new Thread(a1, "First Child Thread");

 A a2 = new A();
 Thread t2 = new Thread(a2, "Second Child Thread");

  t1.setPriority(4); // Setting the priority of thread
  t2.setPriority(8);

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

Java Thread Join

The join() method in Java is used when a thread needs to wait for another thread to finish its execution.

In other words, this method is used to wait the current thread until the thread that has called the join() method completes its execution.




```
public class X implements Runnable
{
public void run()
{
 System.out.println("Child thread is running");
 for(int i = 1; i <= 4; i++)
 {
  System.out.println("I: " +i);
 }
System.out.println("Child thread is ending");
}
public static void main(String[] args)
{
 X x = new X();
 Thread t = new Thread(x);
  t.start(); // thread t is ready to run.

// join() method is called inside the main thread (current thread) through Thread t.
try
{
  t.join(); // Wait for thread t to end.
}
catch(InterruptedException ie)
{
  ie.printStackTrace();
```

```
}
  System.out.println("Main Thread is ending");
 }
}
```

```
public class X implements Runnable
{
public void run()
{
 System.out.println("Child thread is running");
 for(int i = 1; i <= 4; i++)
 {
   System.out.println("I: " +i);
   try
   {
    Thread.sleep(2000); // Pauses the execution of child thread for 2 sec.
} catch (InterruptedException e) {
    e.printStackTrace();
}
 }
 System.out.println("Child thread is ending");
}
public static void main(String[] args)
{
X x = new X();
Thread t = new Thread(x);
 t.start(); // thread t is ready to run.

// join() method is called inside the main thread (current thread) through Thread t.
try
{
  t.join(1000); // Wait for thread t to end till 1 sec.
}
catch(InterruptedException ie)
{
  ie.printStackTrace();
}
System.out.println("Main Thread is ending");
  }
}
```

```
public class A implements Runnable
{
public void run()
{
 System.out.println("Child thread1 starts running");
 for(int i = 1; i <= 3; i++)
 {
  System.out.println("I: " +i);
 }
 System.out.println("Child thread1 is ending");
 }
}
```

```
public class B implements Runnable
{
public void run()
{
 System.out.println("Child thread2 starts running");
 for(int j = 1; j <= 3; j++)
 {
  System.out.println("J: " +j);
 }
 System.out.println("Child thread2 is ending");
 }
}
public class Joining
{
public static void main(String[] args) throws InterruptedException
{
 A a = new A();
 Thread t1 = new Thread(a);

 B b = new B();
 Thread t2 = new Thread(b);

 t1.start(); // thread t1 is ready to start.
 t1.join(); // wait for child thread t1 to complete execution.

 t2.start(); // thread t2 is ready to start.
 t2.join(); // wait for child thread t2 to complete execution.

 System.out.println("Main thread is ending");
 }
}
```