CS20004: Object Oriented Programming using Java

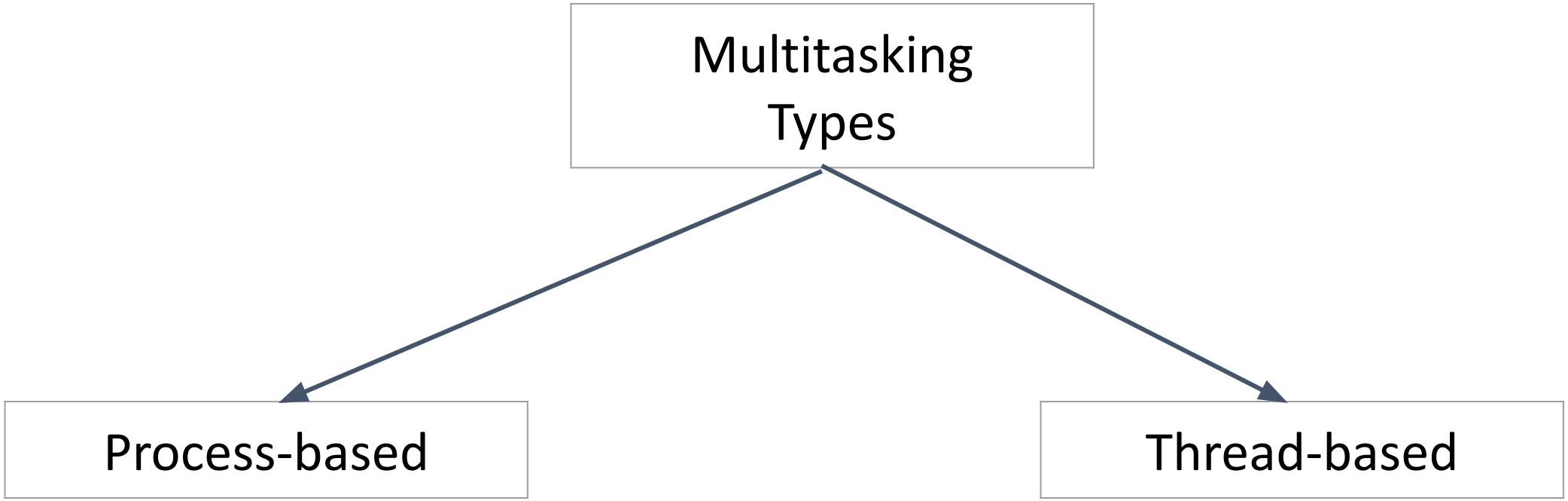
Lec- 23

In this Discussion . . .

- Multithreading
  - Introduction
  - Different ways to define, instantiate and start a new Thread:
    - By extending Thread class
    - By implementing Runnable interface
  - Thread class constructors
  - Thread priority
  - Getting and setting name of a Thread.
  - The methods to prevent(stop) Thread execution:
    - yield()
    - join()
    - sleep()

Introduction

- **Multitasking:** Executing several tasks simultaneously



Multitasking: Process-based

- Executing several tasks simultaneously where each task is a separate independent process
- Example:
  - While typing a java program in the editor, we can also listen to mp3 audio songs, while at the same time we can download a file from the internet.
  - All these tasks are independent of each other and executing simultaneously and hence it is Process-based multitasking.
  - This type of multitasking is best suitable at "os level".

Multitasking: Thread-based

- Executing several tasks simultaneously where each task is a separate independent part of the same program.
- Each independent part is called a "Thread".
- Thrust areas for application of Multithreading:
  - To implement multimedia graphics
  - To develop:
    - animations.
    - video games
    - web and application servers

Whether it is process based or Thread based the main objective of **multitasking is to improve performance of the system by reducing response time.**

Ways to define, instantiate and start a new Thread

- We can define a Thread in the following **two** ways:
  - By **extending Thread class**.
  - By **implementing Runnable interface**.

Defining a Thread by **Extending** the Thread Class

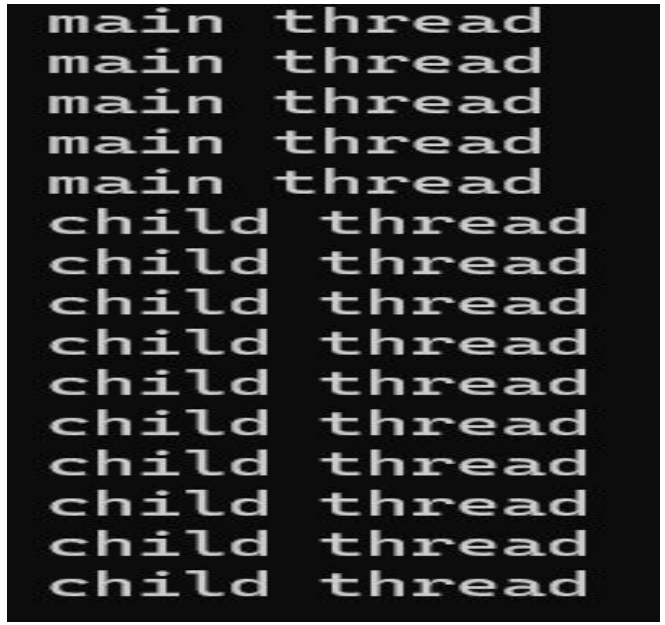
Defining a Thread

```
class MyThread extends Thread
{
        public void run()
        {
                for(int i =0; i<10; i++)
                {
                        System.out.println("child thread");
                }
        }
class ThreadDemo
{
        public static void main(String args[])
        {
                MyThread t = new MyThread(); //Thread Instantiation
                t.start(); // Starting of a Thread
for(int i=0; i<5;i++)
{
        System.out.println("main thread");
                }
        }
}
```

Job of a Thread

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 1: Thread Scheduler***
  - **If multiple Threads** are waiting to execute then which Thread will execute first is decided by "Thread Scheduler" which is part of JVM.
  - Which algorithm or behavior followed by Thread Scheduler we can't expect exactly, as it is the JVM vendor dependent.
  - Hence in multithreading examples we can't expect exact execution order and exact output.



Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 2: Diff. between t.start() and t.run() methods***
  - **In the case of t.start()** **a new Thread will be created** which is responsible for the execution of run() method.
  - **But in the case of t.run()** **no new Thread will be created** and run() method will be executed just like a normal method by the main Thread.
  - For the program two slides earlier if we are replacing t.start() with t.run() the following is the output.
  - Entire output produced by only main Thread.

```
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
```

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 3: importance of Thread class start() method***
  - **For every Thread** the required mandatory activities like registering the Thread with Thread Scheduler will be taken care of by Thread class **start()** method.
  - While the programmer is responsible to just define the job of the Thread inside run() method.
  - That is start() method acts as best assistant to the programmer.

Example:
start()
{
    1. Register Thread with Thread Scheduler
    2. All other mandatory low level activities.
    3. Invoke or calling run() method.
}

- We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java.
- Due to this **start()** is ***considered*** as heart of multithreading.

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 4: What happens If we are not overriding run() method***
  - If we are not overriding run() method in our class, then the default parent Thread class run() method will be executed which has empty implementation and hence we won't get any output.

```
class MyThread1 extends Thread
{
}
class ThreadDemo1
{
      public static void main(String args[])
      {
            MyThread1 t = new MyThread1();
//Thread Instantiation
            t.start(); // Starting of a Thread
      }
}
```

- It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac ThreadDemo1.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo1

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>
```

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 5: Overloading of the run() method.***
  - We can overload run() method but Thread class start() method always invokes no argument run() method
  - The other overload run() methods we have to call explicitly then only it will be executed just like normal method.

```
class MyThread2 extends Thread
{
      public void run()
      {
            System.out.println("no arg
method");
      }
      public void run(int i)
      {
      System.out.println("int arg method");
      }}
```

```
class ThreadDemo2
{
public static void main(String[] args)
     {
      MyThread2 t=new MyThread2();
      t.start();
      }}
```

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac ThreadDemo2.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo2
no arg method
```

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 6: Overriding of the start() method.***
  - If we override start() method then our start() method will be executed just like a normal method call and no new Thread will be started.
  - Entire output produced by only main Thread. It is never recommended to override start() method.

```
class MyThread3 extends Thread
{
    public void start()
    {
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
```

```
class ThreadDemo3
{public static void main(String[] args)
    {
        MyThread3 t=new MyThread3();
            t.start();
        System.out.println("main method");
        }}
```

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac ThreadDemo3.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo3
start method
main method
```

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 7:***

```
class Mythread4 extends Thread
{
	public void start()
	{
		System.out.println("start method");
	}
	public void run()
	{
		System.out.println("run method");
	}
}
```

```
class ThreadDemo4
{
	public static void main(String args[])
	{
		Mythread4 t = new Mythread4();
		t.start();
	System.out.println("main method");
		}
	}
```

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac ThreadDemo4.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo4
start method
main method
```

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 7:*** *Example-2*

```
class Mythread5 extends Thread
{
	public void start()
	{
		super.start();
		System.out.println("start method");
	}
	public void run()
	{
		System.out.println("run method");
	}
}
```

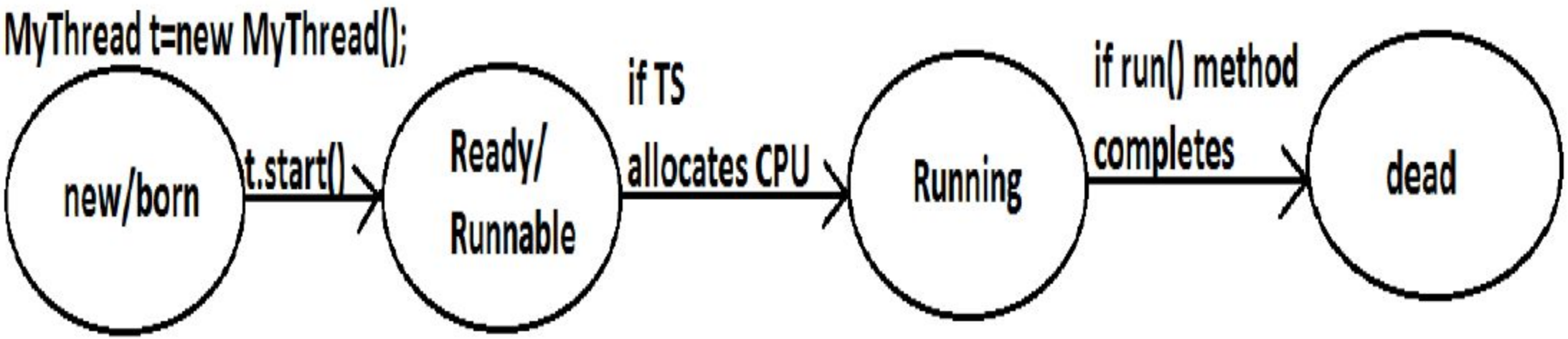
```
class ThreadDemo5
{
	public static void main(String args[])
	{
		Mythread5 t = new Mythread5();
		t.start();
		System.out.println("main method");
		}
	}
```

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac ThreadDemo5.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo5
start method
run method
main method
```

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 8: life cycle of the Thread***



- Once we created a Thread object then the Thread is said to be in new state or born state.
- Once we call start() method then the Thread will be entered into Ready or Runnable state.
- If Thread Scheduler allocates CPU then the Thread will be entered into running state.
- Once run() method completes then the Thread will entered into dead state.

Defining a Thread by **Extending** the Thread Class (Contd.) Cases

- ***Case 9:***

After starting a Thread we are not allowed to restart the same Thread once again otherwise we will get runtime exception saying "**IllegalThreadStateException**".

```
class Mythread6 extends Thread
{
	public void start()
	{
		super.start();
		System.out.println("start method");
	}
	public void run()
	{
		System.out.println("run method");
	}
}
```

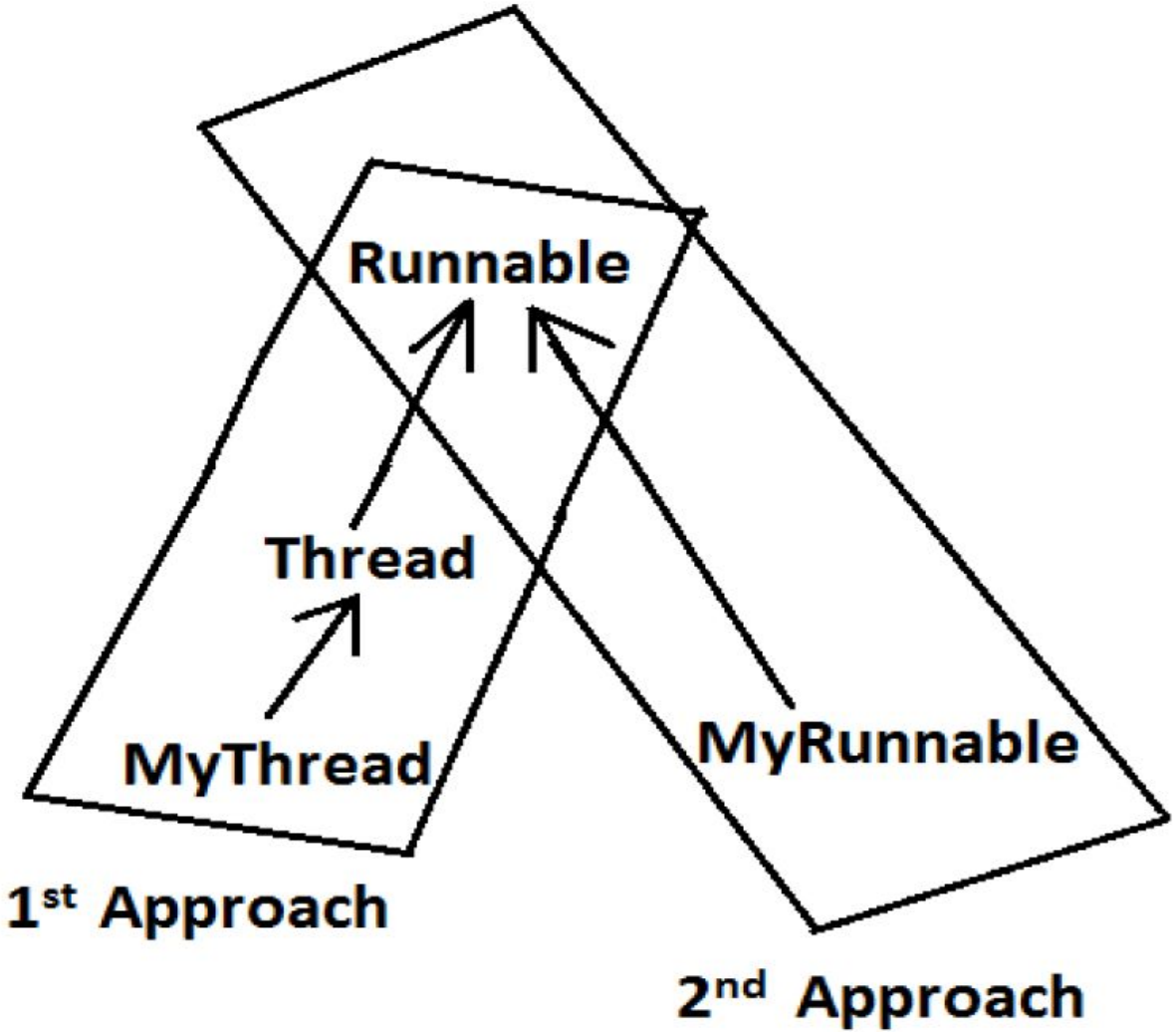
```
class ThreadDemo6
{
	public static void main(String args[])
	{
		Mythread6 t = new Mythread6();
		t.start(); //valid
		.................
		;;;;;;;;;;;;;;;;
		t.start(); //Runtime Exception
	}
}
```

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac ThreadDemo6.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo6
start method
run method
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.base/java.lang.Thread.start(Thread.java:793)
        at Mythread6.start(ThreadDemo6.java:5)
        at ThreadDemo6.main(ThreadDemo6.java:21)
```

Defining a Thread by implementing Runnable interface

- We can define a Thread even by implementing Runnable interface also.
- **Runnable interface present in java.lang package and contains only one method run().**



Defining a Thread by **implementing** Runnable interface

Defining a Thread

```
class MyRunnable implements Runnable
{
      public void run()
      {
            for(int i =0; i<10; i++)
            {
                  System.out.println("child thread");
            }
      }
}
class ThreadDemo7
{
      public static void main(String args[])
      {
            MyRunnable r = new MyRunnable();
           Thread t = new Thread(r); //here “r” is a
target runnable
            t.start(); // Starting of a Thread
```

Job of a Thread

```
for(int i=0; i<5;i++)
{
      System.out.println("main thread");
            }
      }
}
```

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo7
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
```

Defining a Thread by **implementing** Runnable interface

We can't expect exact output as there can be several possible outputs

Defining a Thread

```
class MyRunnable implements Runnable
{
      public void run()
      {
            for(int i =0; i<10; i++)
            {
                  System.out.println("child thread");
            }
      }
}
class ThreadDemo7
{
      public static void main(String args[])
      {
            MyRunnable r = new MyRunnable();
           Thread t = new Thread(r); //here “r” is a
target runnable
            t.start(); // Starting of a Thread
```

Job of a Thread

```
for(int i=0; i<5;i++)
{
      System.out.println("main thread");
            }
      }
}
```

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo7
main thread
child thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
```

Defining a Thread by **implementing** Runnable interface: Cases

- ***Case 1: t.start()***

Defining a Thread

```
class MyRunnable implements Runnable
{
      public void run()
      {
            for(int i =0; i<10; i++)
            {
                  System.out.println("child thread");
            }
      }
class ThreadDemo8
{
      public static void main(String args[])
      {
            MyRunnable r = new MyRunnable();
           Thread t = new Thread();
           Thread t1 = new Thread(r); //
            t.start(); // Starting of a Thread
for(int i=0; i<5;i++)
{
      System.out.println("main thread");
            }
      }
}
```

Job of a Thread

Conclusion: A new Thread will be created which is responsible for the execution of Thread class run()method.

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac ThreadDemo8.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo8
main thread
main thread
main thread
main thread
main thread
```

Defining a Thread by **implementing** Runnable interface: Cases

- ***Case 2: t.run()***

Defining a Thread

```
class MyRunnable implements Runnable
{
      public void run()
      {
            for(int i =0; i<10; i++)
            {
                  System.out.println("child thread");
            }
      }
class ThreadDemo8
{
      public static void main(String args[])
      {
            MyRunnable r = new MyRunnable();
           Thread t = new Thread();
           Thread t1 = new Thread(r); //
            //t.start(); // Starting of a Thread
//t.run();
for(int i=0; i<5;i++)
{
      System.out.println("main thread");
            }
t.run();
      }
}
```

Job of a Thread

Conclusion: No new Thread will be created but Thread class run() method will be executed just like a normal method call.

Defining a Thread by **implementing** Runnable interface: Cases

- ***Case 3: t1.start()***

Defining a Thread

```
class MyRunnable implements Runnable
{
      public void run()
      {
            for(int i =0; i<10; i++)
            {
                  System.out.println("child thread");
            }

}
class ThreadDemo8
{
      public static void main(String args[])
      {
            MyRunnable r = new MyRunnable();
           Thread t = new Thread();
           Thread t1 = new Thread(r); //
            t1.start(); // Starting of a Thread
```

Job of a Thread

```
//t.run();
for(int i=0; i<5;i++)
{
      System.out.println("main thread");
            }


      }
}
```

Conclusion: New Thread will be created which is responsible for the execution of MyRunnable run() method.

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadDemo8
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
```

Defining a Thread by **implementing** Runnable interface: Cases

- ***Case 4: t1.run()***

Defining a Thread

```
class MyRunnable implements Runnable
{
      public void run()
      {
            for(int i =0; i<10; i++)
            {
                  System.out.println("child thread");
            }

}
class ThreadDemo8
{
      public static void main(String args[])
      {
            MyRunnable r = new MyRunnable();
           Thread t = new Thread();
           Thread t1 = new Thread(r); //
            //t1.start(); // Starting of a Thread
```

Job of a Thread

```
t1.run();
for(int i=0; i<5;i++)
{
      System.out.println("main thread");
            }

      }
}
```

Conclusion: No new Thread will be created and MyRunnable run() method will be executed just like a normal method call.

Defining a Thread by **implementing** Runnable interface: Cases

- ***Case 5 & 6: r.start() & r.run();***

Defining a Thread

```
class MyRunnable implements Runnable
{
      public void run()
      {
            for(int i =0; i<10; i++)
            {
                  System.out.println("child thread");
            }

}
class ThreadDemo8
{
      public static void main(String args[])
      {
            MyRunnable r = new MyRunnable();
           Thread t = new Thread();
           Thread t1 = new Thread(r); //
            //t1.start(); // Starting of a Thread
```

Job of a Thread

```
//t1.run();
for(int i=0; i<5;i++)
{
      System.out.println("main thread");
            }

      }
}
```

Conclusion: CHeck what will be the outputs for the statement(s) r.start() and r.run()

Thread class Constructors

1. Thread t=new Thread();
2. Thread t=new Thread(Runnable r);
3. Thread t=new Thread(String name);
4. Thread t=new Thread(Runnable r,String name);
5. Thread t=new Thread(ThreadGroup g,String name);
6. Thread t=new Thread(ThreadGroup g,Runnable r);
7. Thread t=new Thread(ThreadGroup g,Runnable r,String name);
8. Thread t=new Thread(ThreadGroup g,Runnable r,String name,long stackSize);

Getting and Setting name of a Thread

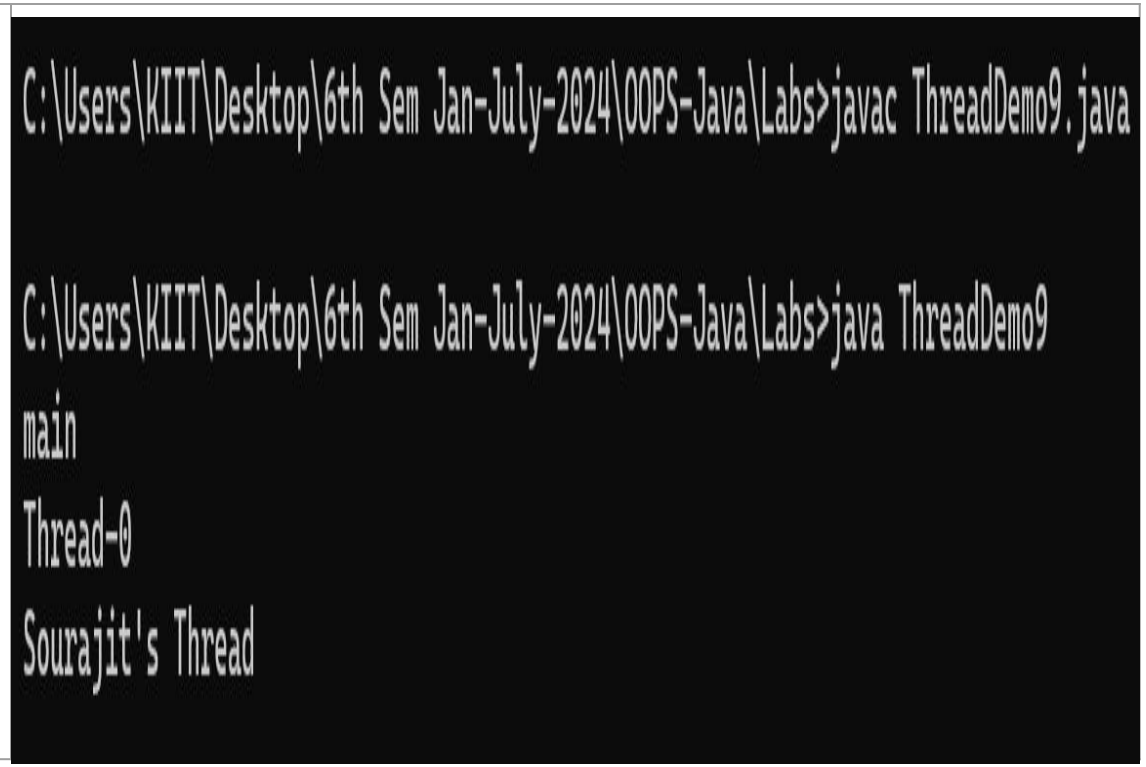
- Every Thread in java has some name. It's name can be provided explicitly by the programmer or automatically generated by JVM.
- Thread class defines the following methods to get and set name of a Thread.

Methods:

1. public final String getName()
2. public final void setName(String name)

Getting and Setting name of a Thread (Contd.)

```
class MyThread9 extends Thread
{}
class ThreadDemo9
{
	public static void main(String[] args)
	{
		System.out.println(Thread.currentThread().getName());//mai
n
		MyThread9 t=new MyThread9();
		System.out.println(t.getName());//Thread-0
		Thread.currentThread().setName("Sourajit's Thread”);
System.out.println(Thread.currentThread().getName());//Sourajit's
Thread
}
}
```



Note: We can get current executing Thread object reference by using Thread.currentThread() method.

Thread Priorities

- Every Thread in java has some priority, which may be a default priority generated by JVM (or) explicitly provided by the programmer.
- The valid range of Thread priorities is 1 to 10[but not 0 to 10] where 1 is the least priority and 10 is highest priority.
- Thread class defines the following constants to represent some standard priorities.

CONSTANTS	PRIORITY
Thread. MIN_PRIORITY	1
Thread. MAX_PRIORITY	10
Thread. NORM_PRIORITY	5

Thread Priorities (Contd.)

- Thread scheduler uses these priorities while allocating CPU.
- The Thread which is having highest priority will get chance for first execution.

If two threads are having the same priority, then we can't expect exact execution order as it depends on Thread scheduler whose behavior is vendor dependent.

Thread Priorities (Contd.)

- We can get and set the priority of a Thread by using the following methods:

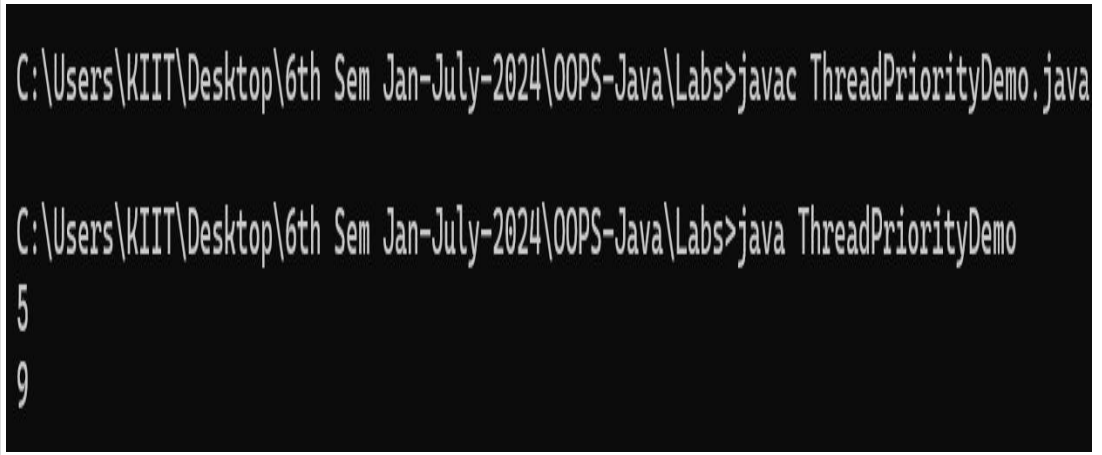
```
1.  public final int getPriority()
2. public final void setPriority(int newPriority);//the allowed values are 1 to 10
```

The allowed values are 1 to 10 otherwise we will get runtime exception saying "IllegalArgumentException".

Default Priority

- The default priority for the main Thread is 5.
- But for all the remaining Threads, the default priority will be inherited from parent to child, i.e., whatever the priority parent has by default the same priority will be for the child also.

```
class MyThread extends Thread
{}
class ThreadPriorityDemo
{
	public static void main(String[] args)
	{
		System.out.println(Thread.currentThread().getPriority
());//5
		Thread.currentThread().setPriority(9);
		MyThread t=new MyThread();
		System.out.println(t.getPriority());//9
	}
}
```



Default Priority

```
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<10;i++)
		{
			System.out.println("child thread");
		}
	}
}
class ThreadPriorityDemo1
{
	public static void main(String[] args)
	{
		MyThread t=new MyThread();
		//t.setPriority(10); //----> Line no. 1
		t.start();
		for(int i=0;i<10;i++)
		{
			System.out.println("main thread");
		}
	}
}
```

- If we are commenting line 1 then both main and child Threads will have the same priority and hence we can't expect exact execution order.

```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>javac ThreadPriorityDemo1.java

C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadPriorityDemo1
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
```

Default Priority

-

```
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<10;i++)
		{
			System.out.println("child thread");
		}
	}
}
class ThreadPriorityDemo1
{
	public static void main(String[] args)
	{
		MyThread t=new MyThread();
		t.setPriority(10); //----> Line no. 1
		t.start();
		for(int i=0;i<10;i++)
		{
			System.out.println("main thread");
		}
	}
}
```

- If we are not commenting line 1 then child Thread has the priority 10 and main Thread has the priority 5. Hence child Thread will get chance for execution and after completing child Thread main Thread will get the chance in this the output:

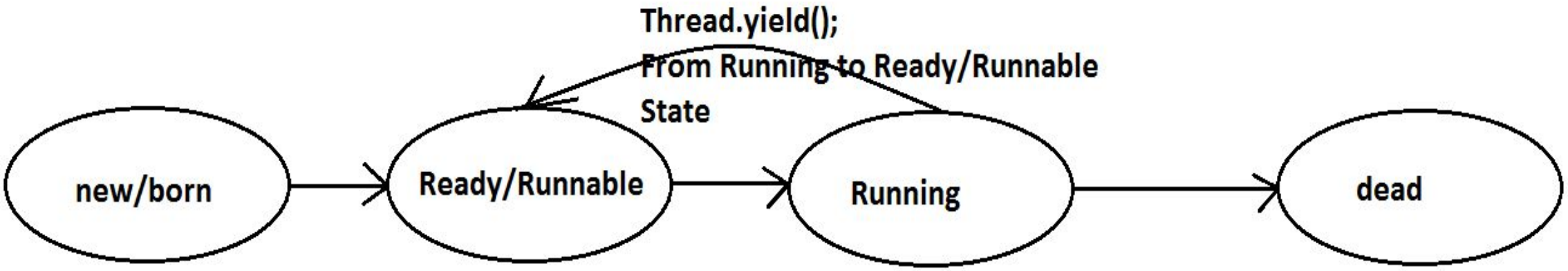
```
C:\Users\KIIT\Desktop\6th Sem Jan-July-2024\OOPS-Java\Labs>java ThreadPriorityDemo1
main thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
```

Methods to Prevent a Thread from Execution

- We can prevent or stop a Thread execution by using the following methods:

1. yield();
2. join();
3. sleep();

Methods to Prevent a Thread from Execution: yield()



- **yield()** method causes "to pause current executing Thread for giving the chance to remaining waiting Threads of same priority".
- If all waiting Threads have the low priority or if there is no waiting Threads then the same Thread will continue its execution.
- If several waiting Threads with same priority available then we can't expect exactly which Thread will get chance for execution.
- Once any thread has yielded, then the next time it gets a chance for execution is dependent on mercy of the Thread scheduler.

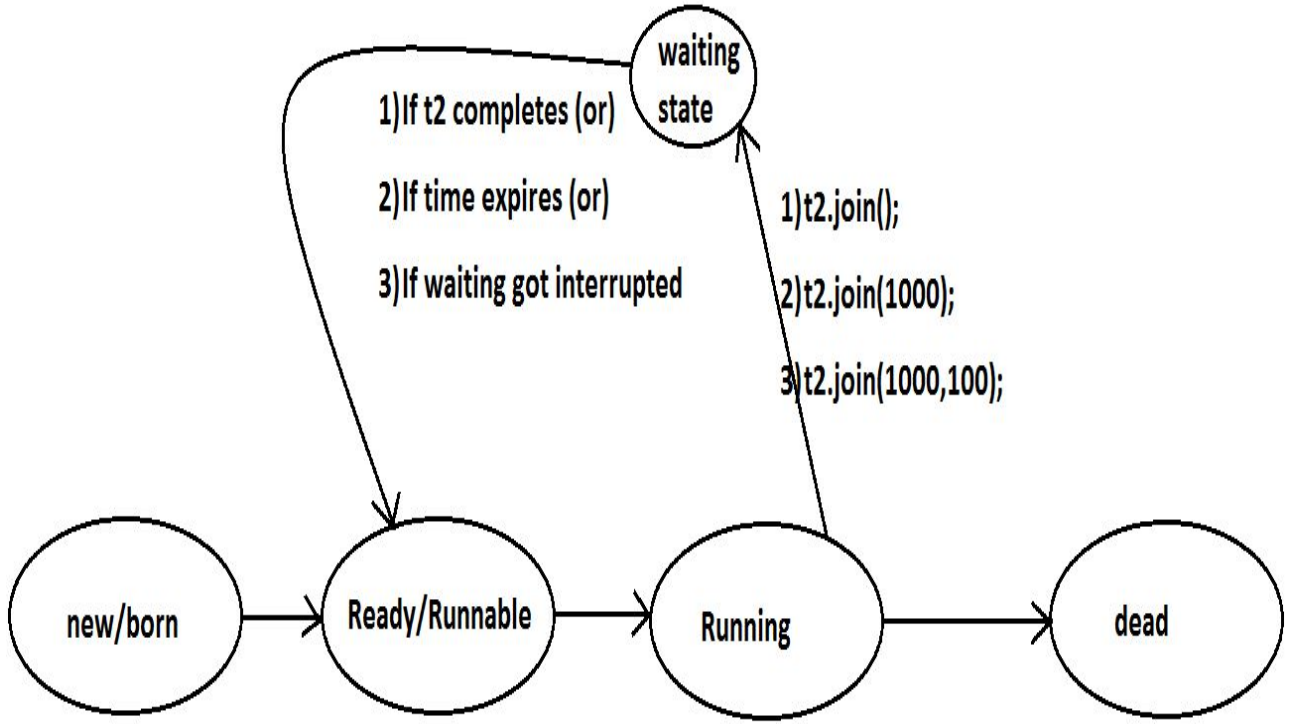
Methods to Prevent a Thread from Execution: yield()

```
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<5;i++)
		{
			Thread.yield();
			System.out.println("child thread");
		}
	}
}
class ThreadYieldDemo
{
	public static void main(String[] args)
	{
		MyThread t=new MyThread();
		t.start();
		for(int i=0;i<5;i++)
		{
			System.out.println("main thread");
		}
	}
}
```

- In the above program child Thread always calls the yield() method
- Hence main Thread will get the chance more number of times for execution.
- Thus, the chance of completing the main Thread first is high.

Methods to Prevent a Thread from Execution: join()

- If a Thread wants to wait until completing some other Thread then we should go for join() method.
- Example: If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.



Methods to Prevent a Thread from Execution: join()

- Every join() method throws InterruptedException, which is checked exception. Hence compulsorily we should either handle by try catch or by using throws keyword, Otherwise we will get compile time error.

```
class MyThread extends Thread
{
	public void run()
	{
		for(int i=0;i<5;i++)
		{
		System.out.println("Behera's Thread");
			try
			{
				Thread.sleep(2000);
			}
			catch (InterruptedException e){}
		}
	}
}
```

```
class ThreadJoinDemo
{
	public static void main(String[] args)throws InterruptedException
	{
		MyThread t=new MyThread();
		t.start();
		//t.join(); //--->Line no. 1
		for(int i=0;i<5;i++)
		{
		System.out.println("Sourajit's Thread");
		}
	}
}
```

Methods to Prevent a Thread from Execution: join()

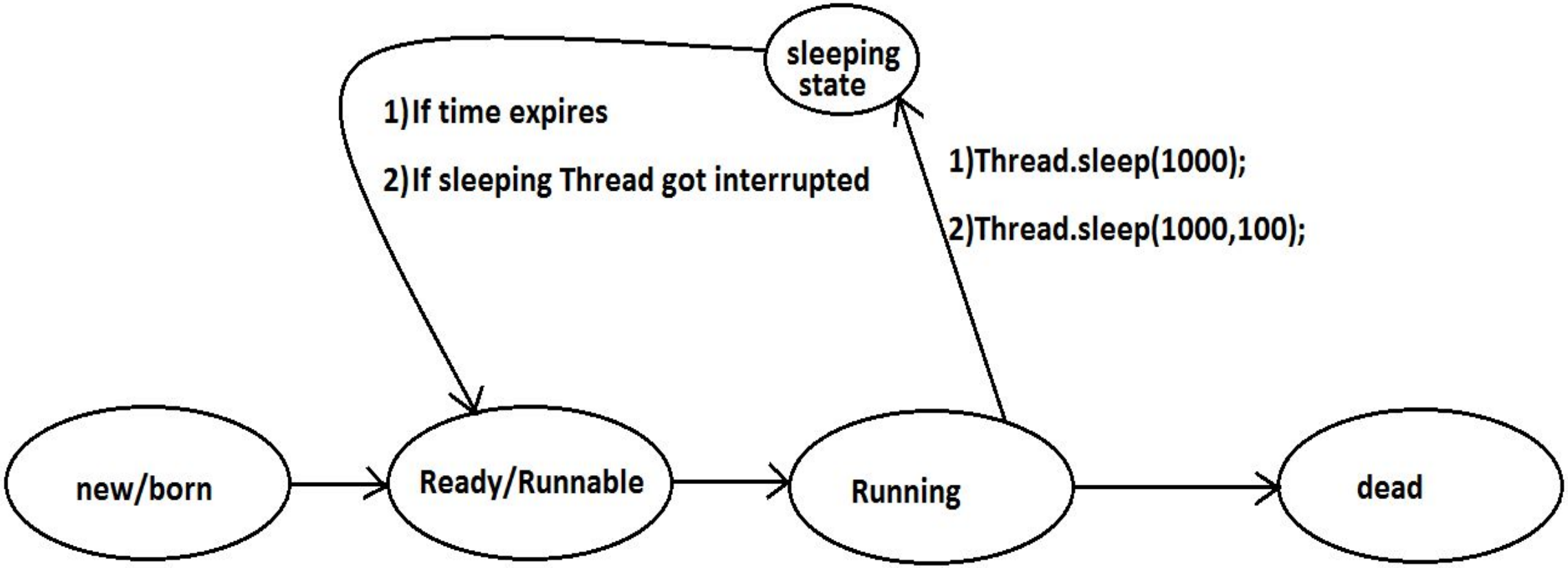
```
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<5;i++)
                {
                System.out.println("Behera's Thread");
                        try
                        {
                                Thread.sleep(2000);
                        }
                        catch (InterruptedException e){}
                }
        }
}
```

```
class ThreadJoinDemo
{
        public static void main(String[] args)throws InterruptedException
        {
                MyThread t=new MyThread();
                t.start();
                //t.join(); //--->Line no. 1
                for(int i=0;i<5;i++)
                {
                System.out.println("Sourajit's Thread");
                }
        }
}
```

- If we are commenting line 1 then both Threads will be executed simultaneously and we can't expect exact execution order.
- If we are not commenting line 1 then main Thread will wait until completing child Thread output which is “Behera’s Thread” 5 times followed by “Sourajit’s” Thread 5 times.

Methods to Prevent a Thread from Execution: sleep()

- If a Thread doesn't want to perform any operation for a particular amount of time then we should go for sleep() method.



Methods to Prevent a Thread from Execution: sleep()

- If a Thread doesn't want to perform any operation for a particular amount of time then we should go for sleep() method.
- Ex- Thread.sleep(3000); //thread sleeps for 3000 ms

Methods:

1. public static native void sleep(long ms) throws InterruptedException
2. public static void sleep(long ms,int ns)throws InterruptedException

Yield() Vs. join() Vs. sleep()

Property	Yield()	join()	sleep()
Purpose	To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.	If a Thread wants to wait until completing some other Thread then we should go for join.	If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
Is it static?	Yes	no	yes
Is it final?	No	yea	no
Can it be overloaded?	no	yes	yes
throws InterruptedException?	no	yes	yes
is it a native method?	yes	no	sleep(long ms) -- >native sleep(long ms,int ns) -->non-native