Multithreading in Java

- Java provides built-in support for multithreaded programming.
- A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is aspecialized form of multitasking.
- Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system.
- One important way multithreading achieves this is by keeping idle time to a minimum. This is especially important for the interactive, networked environment in which Java operates because idle time is common.

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1. **Extending the Thread class**
2. **Implementing the Runnable Interface**

What is the Use of Multi-Thread in Java?

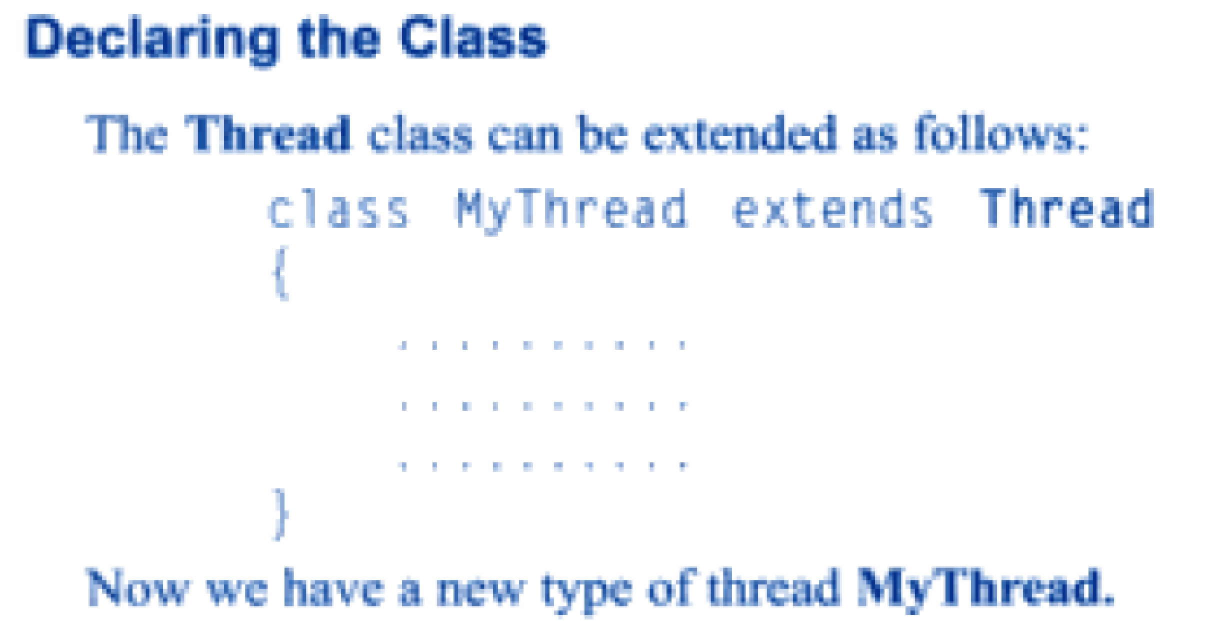
- Because each Thread is managed individually and several operations can be carried out at once, the user is not blocked.
- It is used to save time as multiple operations are performed concurrently.
- Since threads are independent, other threads don't get affected even if an exception occurs in a single thread.

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class.

- This class overrides the **run()** method available in the Thread class.
- A thread **begins its life inside run() method**.
- We create an object of our new class and **call start()** method to start the execution of a thread.

- Start() invokes the run() method on the Thread object.

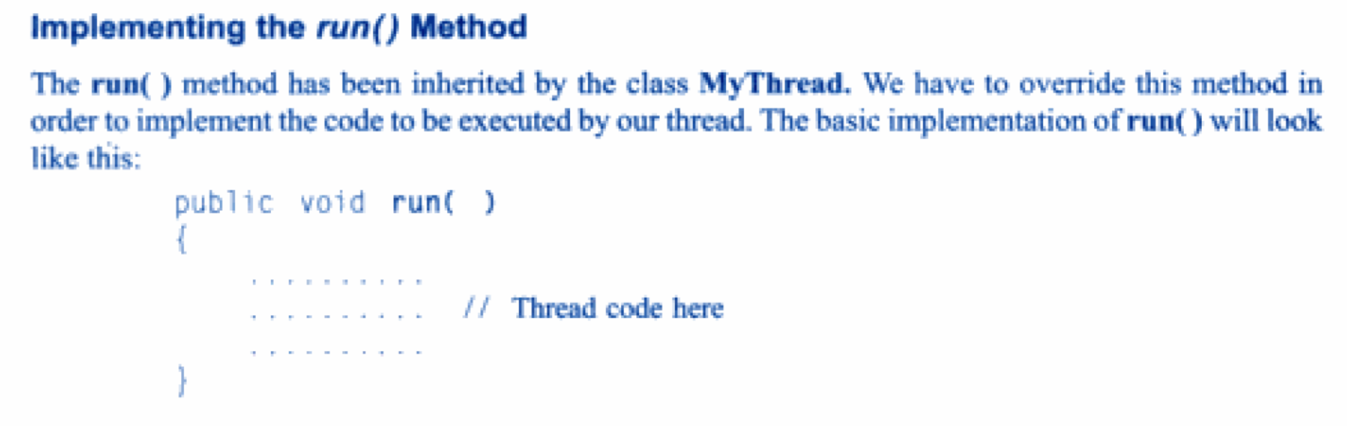


Declaring the Class

The **Thread** class can be extended as follows:

```
class MyThread extends Thread
{
        . . . . . . . . . .
        . . . . . . . . . .
        . . . . . . . . . .
}
```

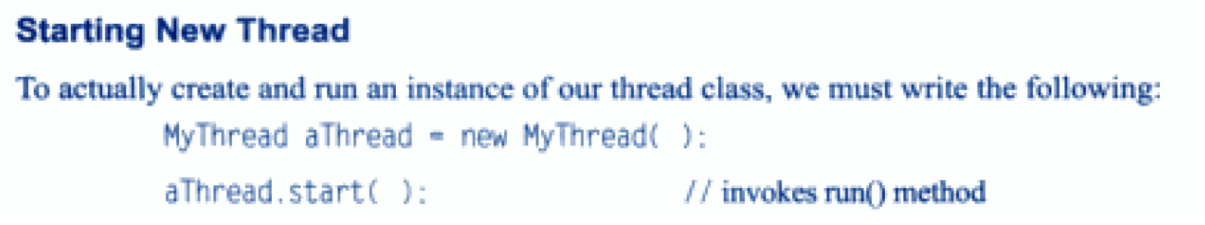
Now we have a new type of thread **MyThread.**



Implementing the *run()* Method

The **run( )** method has been inherited by the class **MyThread.** We have to override this method in order to implement the code to be executed by our thread. The basic implementation of **run( )** will look like this:

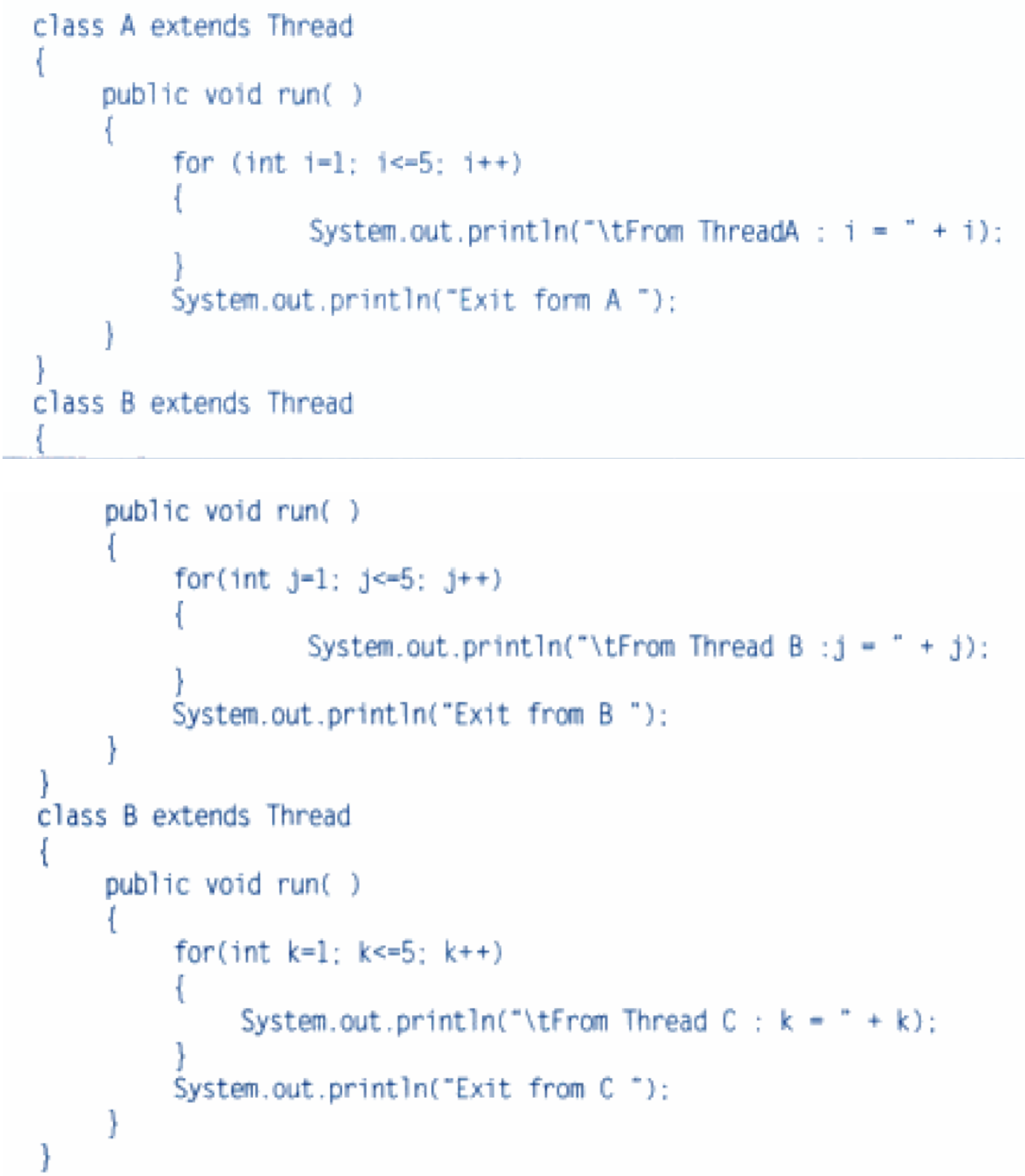
```
public void run( )
{
     . . . . . . . . . . .
     . . . . . . . . . . .    // Thread code here
     . . . . . . . . . . .
}
```



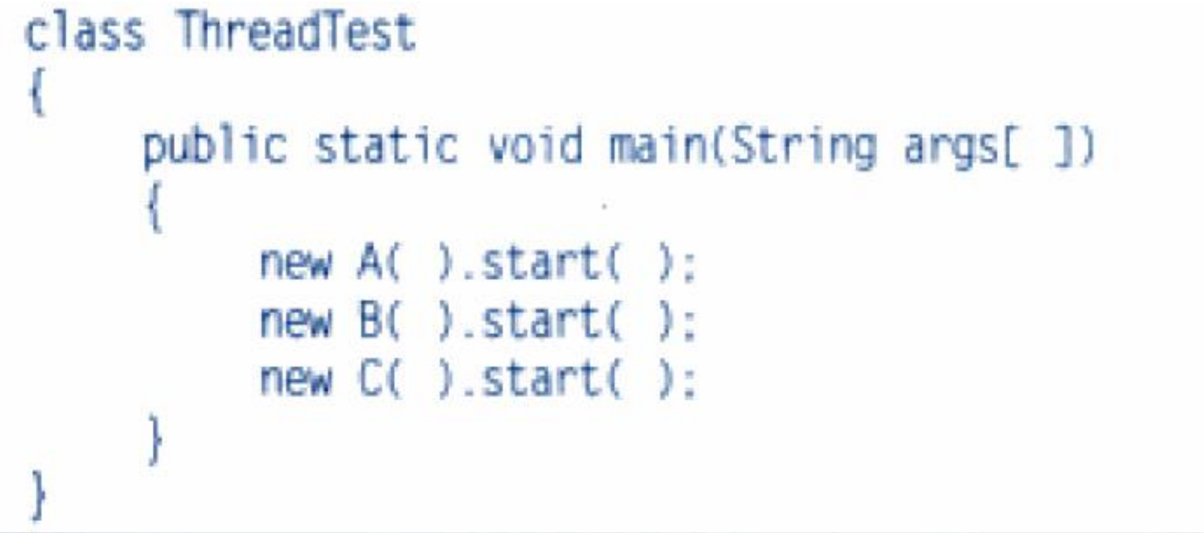
Starting New Thread

To actually create and run an instance of our thread class, we must write the following:

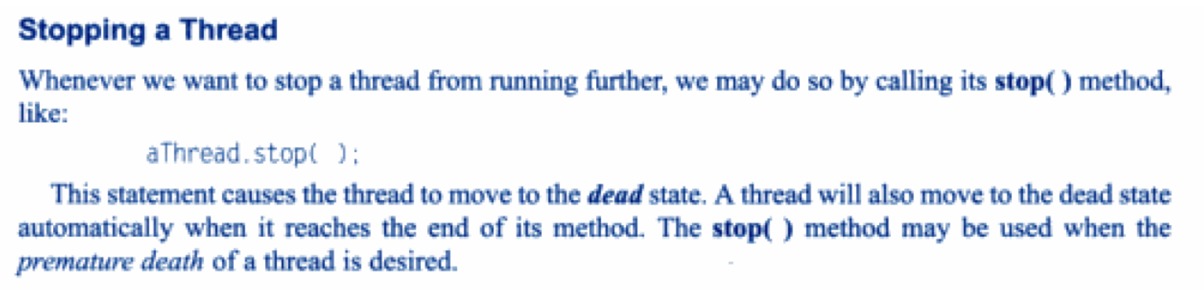
```
MyThread aThread = new MyThread( );
aThread.start( );                  // invokes run() method
```



```
class A extends Thread
{
    public void run( )
    {
        for (int i=1; i<=5; i++)
        {
                System.out.println("\tFrom ThreadA : i = " + i);
        }
        System.out.println("Exit form A ");
    }
}
class B extends Thread
{
    public void run( )
    {
        for(int j=1; j<=5; j++)
        {
                System.out.println("\tFrom Thread B :j = " + j);
        }
        System.out.println("Exit from B ");
    }
}
class B extends Thread
{
    public void run( )
    {
        for(int k=1; k<=5; k++)
        {
            System.out.println("\tFrom Thread C : k = " + k);
        }
        System.out.println("Exit from C ");
    }
}
```



```
class ThreadTest
{
      public static void main(String args[ ])
      {
            new A( ).start( );
            new B( ).start( );
            new C( ).start( );
      }
}
```

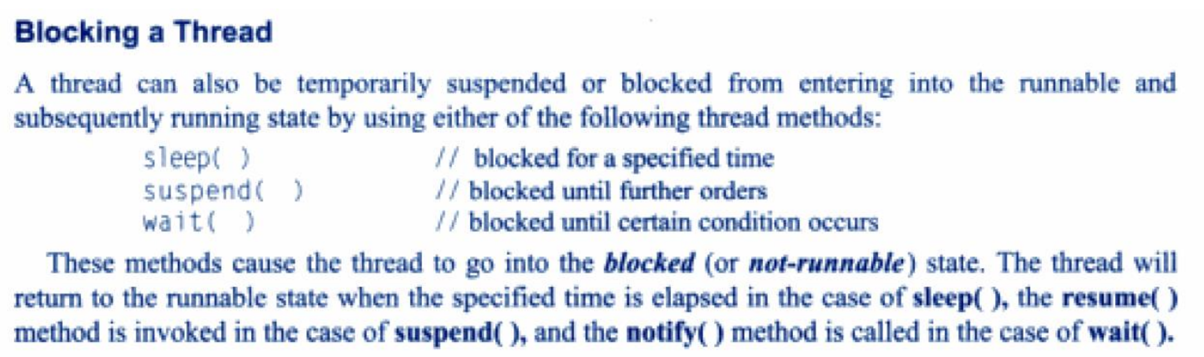


Stopping a Thread

Whenever we want to stop a thread from running further, we may do so by calling its **stop( )** method, like:

```
aThread.stop( );
```

This statement causes the thread to move to the ***dead*** state. A thread will also move to the dead state automatically when it reaches the end of its method. The **stop( )** method may be used when the *premature death* of a thread is desired.



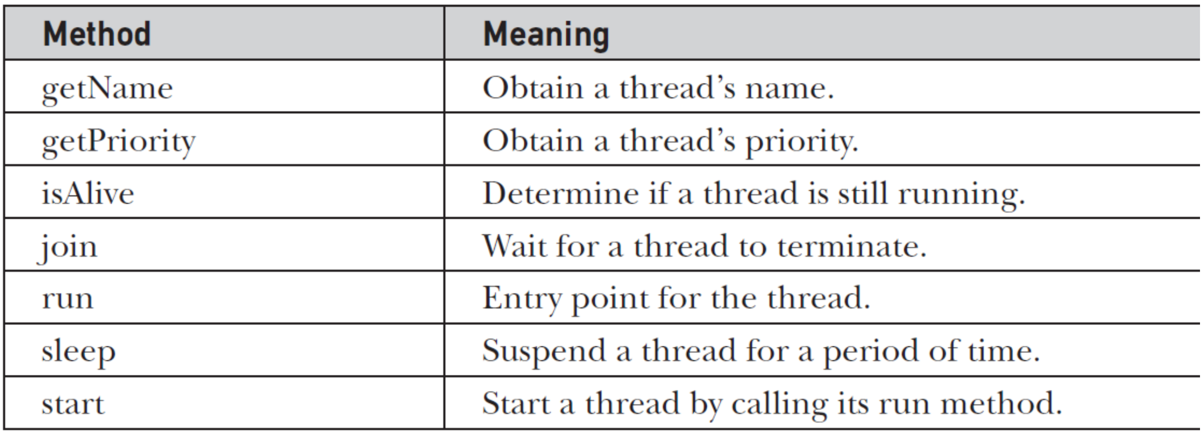
Blocking a Thread

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

```
sleep( )          // blocked for a specified time
suspend( )        // blocked until further orders
wait( )           // blocked until certain condition occurs
```

These methods cause the thread to go into the ***blocked*** (or ***not-runnable***) state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep( )**, the **resume( )** method is invoked in the case of **suspend( )**, and the **notify( )** method is called in the case of **wait( )**.

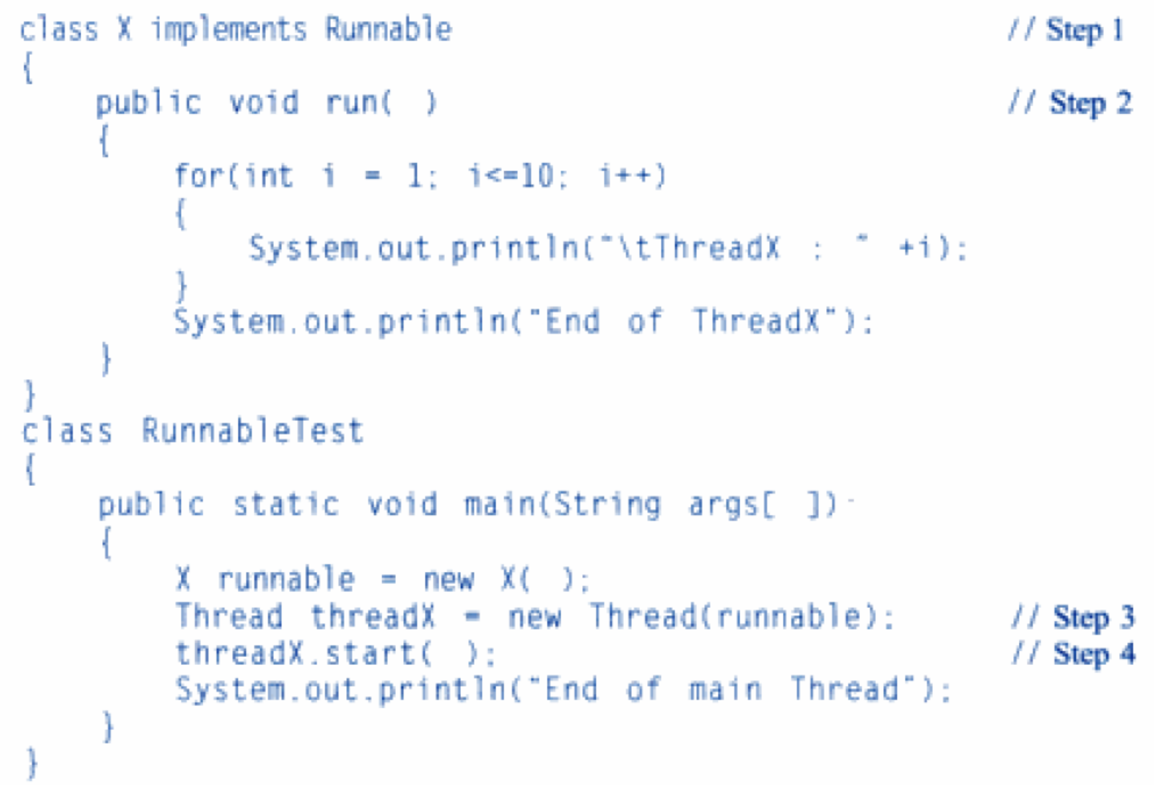
The **Thread** class defines several methods that help manage threads. Several of those used in this chapter are shown here:



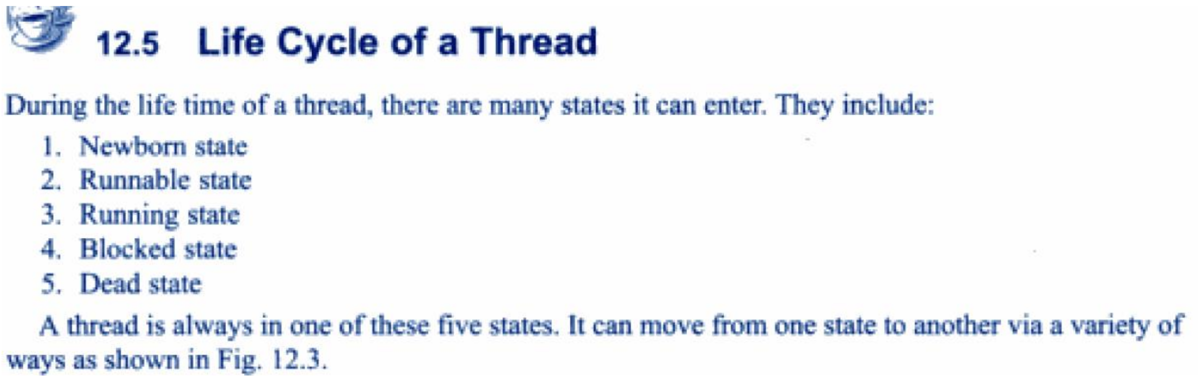
Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Steps to create a new thread using Runnable

- Create a Runnable implementer and implement the run() method.
- Instantiate the Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instances.
- Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start() creates a new Thread that executes the code written in run().



```
class X implements Runnable                                   // Step 1
{
    public void run( )                                        // Step 2
    {
        for(int i = 1; i<=10; i++)
        {
            System.out.println("\tThreadX : " +i);
        }
        System.out.println("End of ThreadX");
    }
}
class RunnableTest
{
    public static void main(String args[ ])
    {
        X runnable = new X( );
        Thread threadX = new Thread(runnable);                // Step 3
        threadX.start( );                                     // Step 4
        System.out.println("End of main Thread");
    }
}
```



12.5 Life Cycle of a Thread

During the life time of a thread, there are many states it can enter. They include:

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in Fig. 12.3.

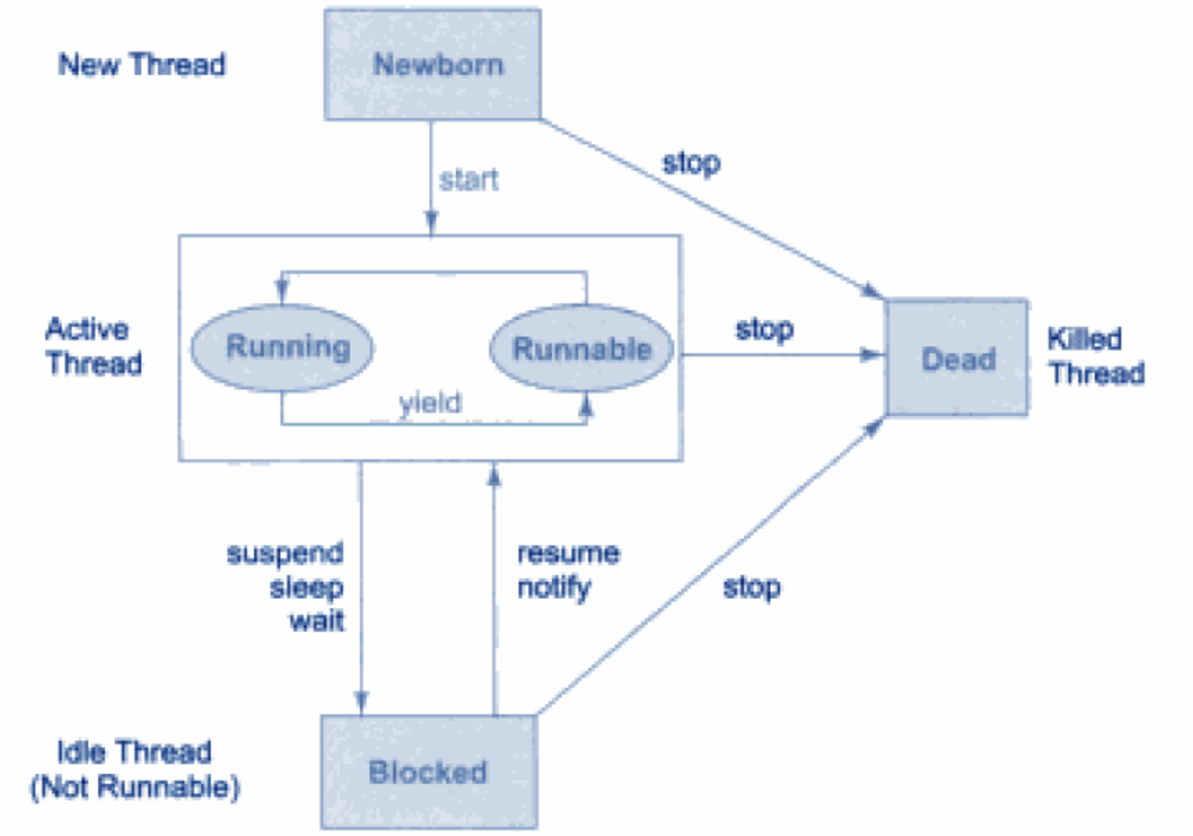


Fig. 12.3 *State transition diagram of a thread*

Following are the stages of the life cycle –

New – A new thread begins its life cycle in the new state.

- It remains in this state until the program starts the thread.
- It is also referred to as a born thread.
- In simple words, a thread has been created, but it has not yet been started. A thread is started by calling its start() method.

Runnable – The thread is in the runnable state after the invocation of the start() method, but the thread scheduler has not selected it to be the running thread.

A thread starts life in the Ready-to-run state by calling the start method and waiting for its turn. The thread scheduler decides which thread runs and for how long.

Running – When the thread starts executing, then the state is changed to a “running” state.

The scheduler selects one thread from the thread pool, and it starts executing in the application.

Dead – This is the state when the thread is terminated.

- The thread is in a running state and as soon as it is completed processing it is in a “dead state”.
- Once a thread is in this state, the thread cannot even run again.

Blocked (Non-runnable state):

- This is the state when the thread is still alive but is currently not eligible to run.
- A thread that is blocked waiting for a monitor lock is in this state.
- A running thread can transit to one of the non-runnable states depending on the situation.
- A thread remains in a non-runnable state until a special transition occurs.
- A thread doesn’t go directly to the running state from a non-runnable state but transits first to the Ready-to-run state.

The non-runnable states can be characterized as follows:

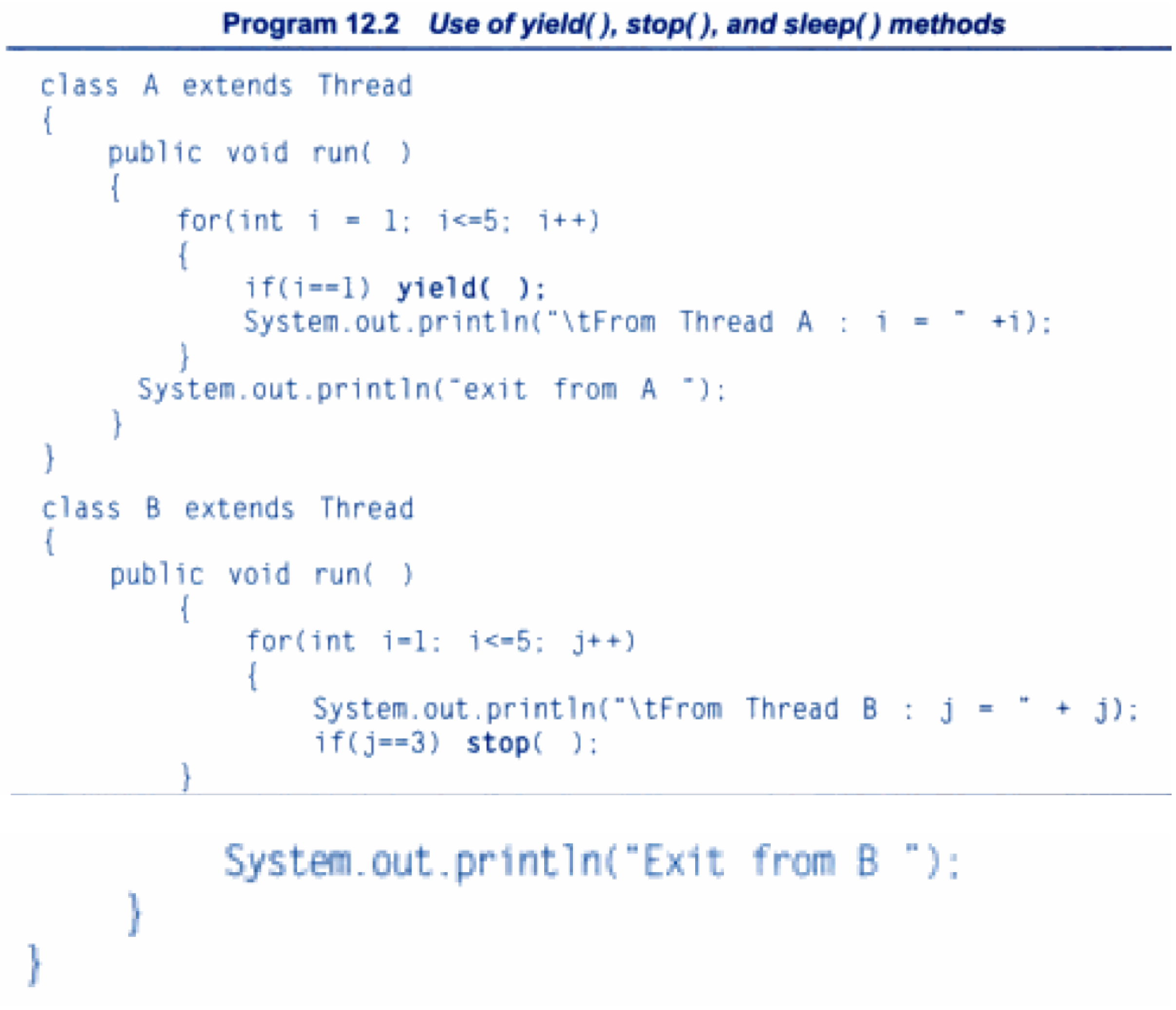
Sleeping:- The thread sleeps for a specified amount of time.

Blocked for I/O:- The thread waits for a blocking operation to complete.

Blocked for Join completion: – The thread awaits completion of another thread.

Waiting for notifications: – The thread awaits a notification from another thread.

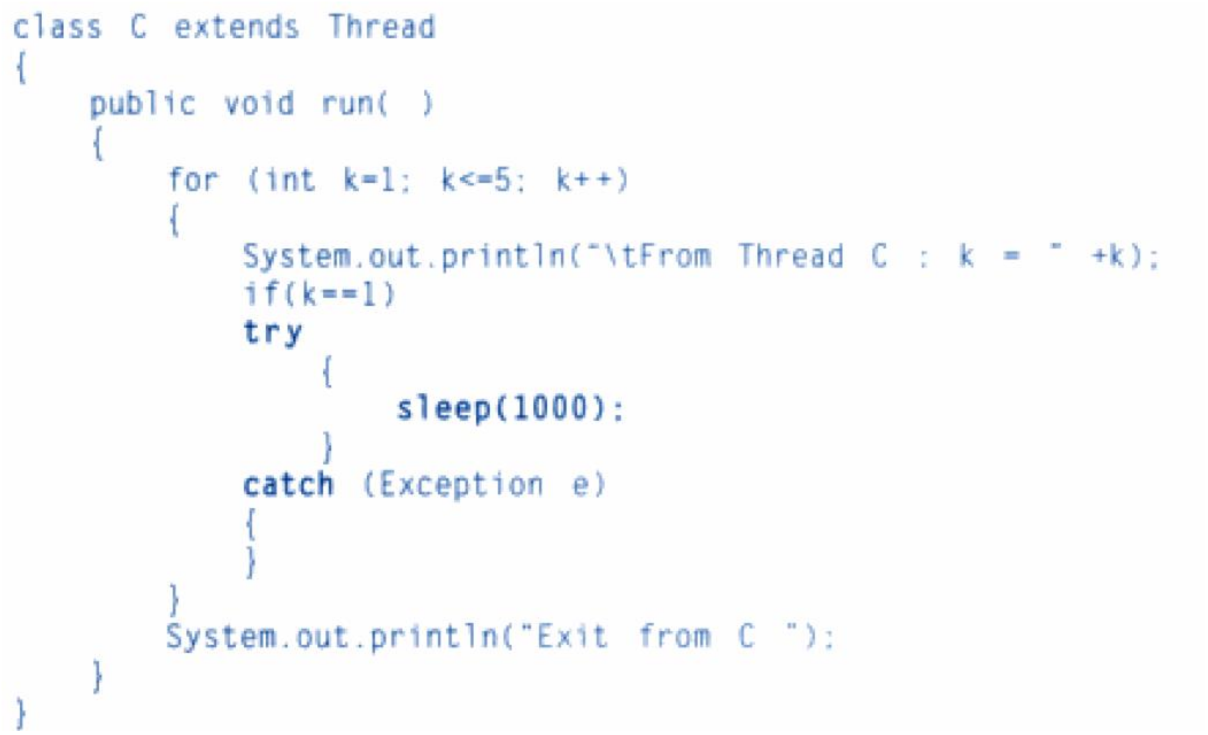
Using Thread Methods-



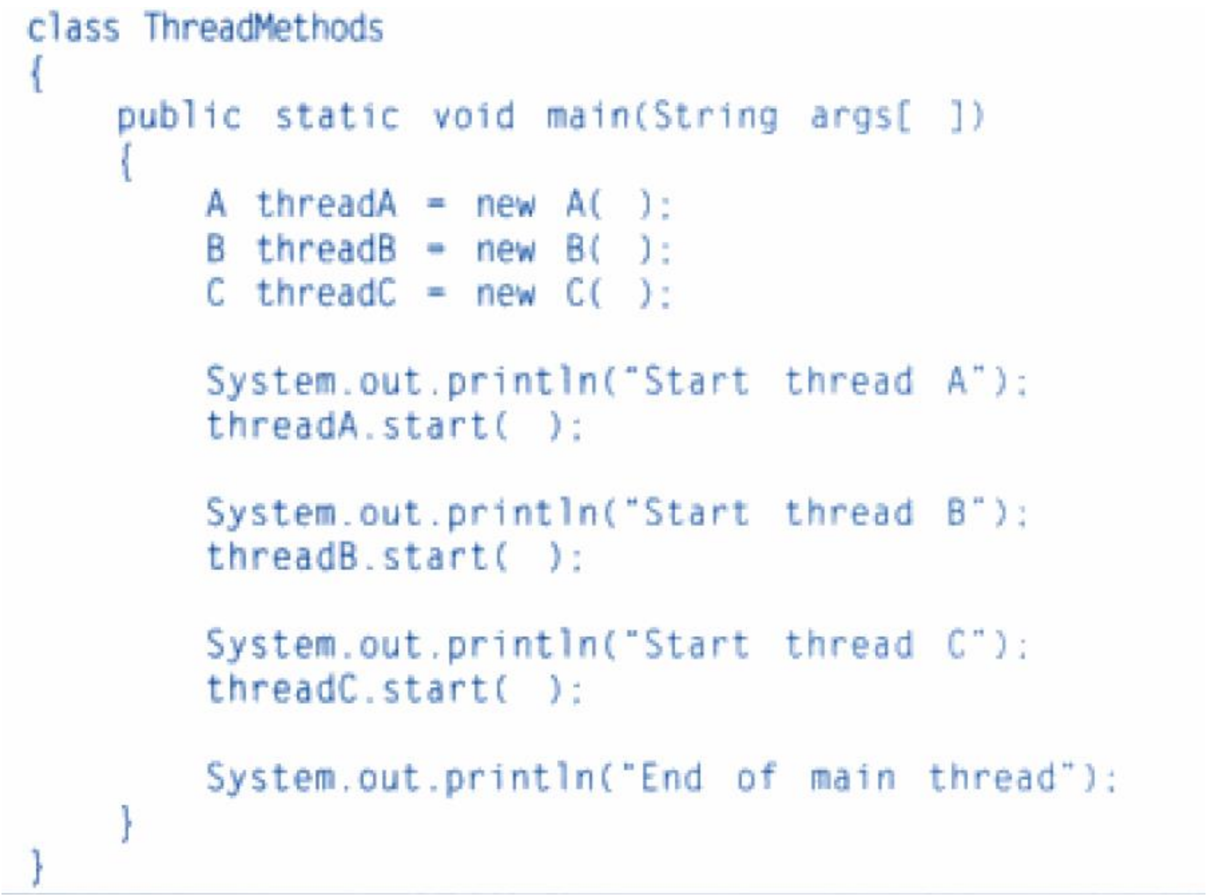
Program 12.2 ***Use of yield( ), stop( ), and sleep( ) methods***

```
class A extends Thread
{
    public void run( )
    {
        for(int i = 1; i<=5; i++)
        {
            if(i==1) yield( );
            System.out.println("\tFrom Thread A : i = " +i);
        }
      System.out.println("exit from A ");
    }
}
class B extends Thread
{
    public void run( )
        {
            for(int i=1; i<=5; j++)
            {
                System.out.println("\tFrom Thread B : j = " + j);
                if(j==3) stop( );
        }

            System.out.println("Exit from B ");
        }
}
```



```
class C extends Thread
{
    public void run( )
    {
        for (int k=1; k<=5; k++)
        {
            System.out.println("\tFrom Thread C : k = " +k);
            if(k==1)
            try
                {
                    sleep(1000);
                }
            catch (Exception e)
            {
            }
        }
        System.out.println("Exit from C ");
    }
}
```



```
class ThreadMethods
{
    public static void main(String args[ ])
    {
        A threadA = new A( );
        B threadB = new B( );
        C threadC = new C( );

        System.out.println("Start thread A");
        threadA.start( );

        System.out.println("Start thread B");
        threadB.start( );

        System.out.println("Start thread C");
        threadC.start( );

        System.out.println("End of main thread");
    }
}
```

Thread priority

- **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 specifies the relative priority of one thread to another.
- The thread with the **highest priority is selected by the scheduler to be executed** first.
- 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

How to get Priority of Current Thread in Java?

- Thread class provides a method named **getPriority()** that is used to determine the priority of a thread.
- It returns the priority of a thread through which it is called.
- These constants are public, final, and static members of the Thread class.

How to set Priority of Thread in Java?

The **setPriority()** of Thread class is used to set the priority of a thread.

This method accepts an integer value as an argument and sets that value as priority of a thread through which it is called.

The syntax to set the priority of a thread is as follows:

Syntax:

ThreadName.setPriority(n);

where, n is an integer value which ranges from 1 to 10.

Example

```
 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();
 }
}
```