Thread

Dr. Firoz Ahmed

Professor

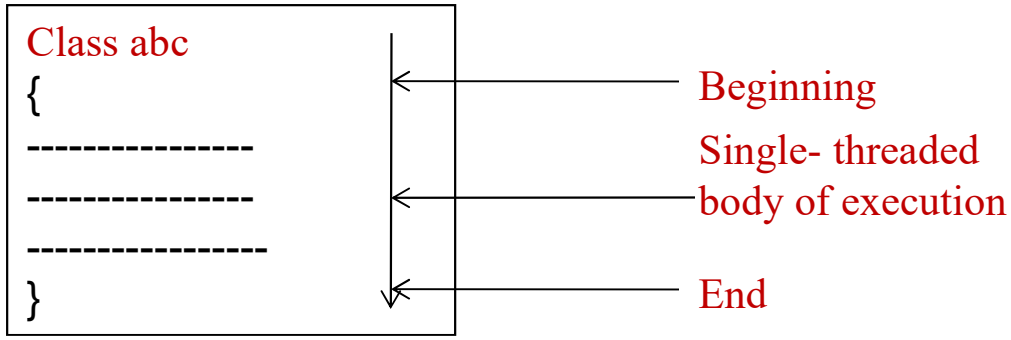
Department of ICE, RU

Outlines of Presentation

- Thread
- Multithread
- Synchronization
- Deadlock
- Thread scheduling

Thread

- A ***Thread*** is similar to a program that has a single flow of control
- On the other hand, ***threads*** are independent processes that can be run simultaneously
- It has
  - A beginning
  - A body and an end
  - Executes command sequentially



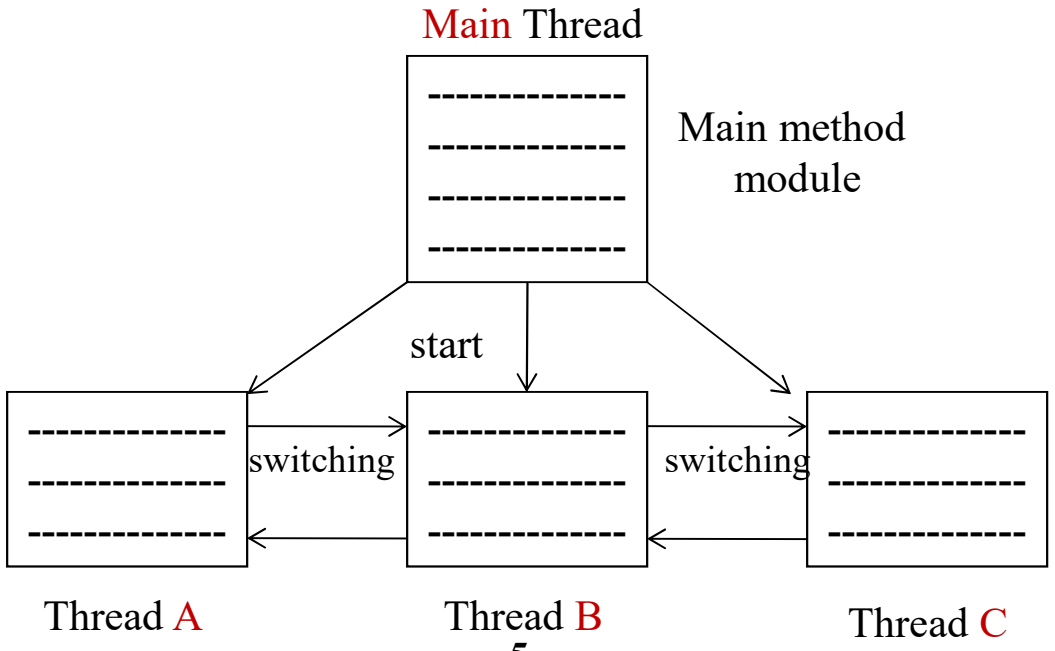
Single Threaded Program

Multithread programming

- The modern operating systems may recognize that they can execute several programs simultaneously
- This ability is known as multitasking
- In system's technology, it is called multithreading
- A unique property of Java is its support for multithreading
- Multithreading is a conceptual programming paradigm where a program is divided into two or more subprograms
- And it can be implemented at the same time in parallel
- Example, one subprogram can display animation on screen while another may build the next animation to be displayed

Multithread programming

- Java enable us to use Multiple flows of control in developing programs
- Each flow of control may be thought of as separate tiny program known as a *thread* that runs parallel to others
- A program that contains multiple flows of control is known a *multiplethreaded program*



Multithread programming

- The main thread is actually the main method module, which is designed to create and start the other three methods A, B, C
- Once initiated by the main thread, the thread A, B and C run concurrently and share the resource jointly
- It is like people living in joint families and sharing certain resources among all of them
- The ability of a language to support multithreads is referred to as concurrency
- Since the threads in Java are subprograms of a main application program and share the same memory space, they are known as lightweight thread

Multithread programming

- It is important to remember that 'threads running in parallel' does not really mean that they actually run at the same time
- Since all the threads are running o a single processor, the flow of execution is shared between the threads
- The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently
- Multithreading is useful in a number of ways
- It enables programmer to do multiple things at one time
- They can divide a long program into threads and executes them in parallel
- This approach considerably improve the speed of or program

Creating Threads

- Creating threads in Java is simple
- Threads are implemented in the form of objects that contain a method called run()
- The run() method is the heart and soul of any thread
- It makes up the entire body of a thread and is the only method in which the thread's behavior can be implemented
- The typical run() method would appear as follows

```
Public void run()
{
-----------
------------ (statement for implementing thread)
-----------
}
```

Creating Threads

- 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()
- A new thread can be created in two ways
  - By extending Thread class (By creating a thread class)
    - Define a class that extend Thread class and override its run() method with the code required by the thread
  - By implementing Runnable interface (By converting a class to a thread)
    - Define a class that implements Runnable interface
    - The runnable interface has only one method, run(), that is to be defined in the method with the code to be executed by the thread

Extending the Thread Class

- It can make the class runnable as a thread by extending the class java.lang.Thread
- It includes the following steps
  - Declare the class as extending the thread class
  - Implement the run() method that is responsible for the sequence of code that the thread will execute
  - Create a thread object and call the start() method to initiate the thread execution

Extending the Thread Class

- Declare the class
  - The **Thread** class can be extended as follows

```
Class MyThread extends Thread
{
----------------
----------------
----------------
}
```

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

Extending the Thread Class

- 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 () methos is:

```
Public void run()
{
----------------
----------------      //Thread code here
----------------
}
```

  - When we start the new thread, Java calls the thread's call() method, so it the run() where all the action takes place

Extending the Thread Class

- 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();                    //invoke run() method
```

  - The first line instantiates a new object of class MyThread
  - This statement just creates the object
  - The thread that will run this object in not yet running
  - The thread is in a newborn state
  - The line calls the start() method causing the thread to move into the runnable state
  - Then, the Java runtime will schedule the thread to run by invoking its run () method
  - Now, the thread is said to be in the running state

Example of Extending the Thread Class

```
class A extends Thread
{
public void run()
{
for(int i=1; i<=5; i++)
{
System.out.println ("\t from thread A: i="+i);
}
System.out.println("exit from A");
}
}

class B extends Thread
{
public void run()
{
for(int j=1; j<=5; j++)
{
System.out.println ("\t from thread B: j="+j);
}
System.out.println("exit from B");
}
}
class C extends Thread
{
public void run()
{
for(int k=1; k<=5; k++)
{
System.out.println ("\t from thread C:ki="+k);
}
System.out.println("exit from C");
}
}

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

Example of Extending the Thread Class

```
First run
   From   Thread      A     :     i     =     1
   From   Thread      A     :     i     =     2
   From   Thread      B     :     j     =     1
   From   Thread      B     :     j     =     2
   From   Thread      C     :     k     =     1
   From   Thread      C     :     k     =     2
   From   Thread      A     :     i     =     3
   From   Thread      A     :     i     =     4
   From   Thread      B     :     j     =     3
   From   Thread      B     :     j     =     4
   From   Thread      C     :     k     =     3
   From   Thread      C     :     k     =     4
    From    Thread      A     :     i     =     5
  Exit from A
    From    Thread      B     :     j     =     5
  Exit from B
    From    Thread      C     :     k     =     5
  Exit from C
```

```
Second run
   From   Thread      A     :     i     =     1
   From   Thread      A     :     i     =     2
   From   Thread      C     :     k     =     1
   From   Thread      C     :     k     =     2
   From   Thread      A     :     i     =     3
   From   Thread      A     :     i     =     4
   From   Thread      B     :     j     =     1
   From   Thread      B     :     j     =     2
   From   Thread      C     :     k     =     3
   From   Thread      C     :     k     =     4
   From   Thread      A     :     i     =     5
  Exit from A
   From   Thread      B     :     k     =     4
   From   Thread      B     :     j     =     5
   From   Thread      C     :     k     =     5
  Exit from C
   From   Thread      B     :     j     =     5
  Exit from B
```

Stopping and Blocking a Thread

- Stopping a thread
  - If we want to stop a thread from running further, we may do so calling its stop() method

```
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 reches the end of its method
  - The stop() method may be used when the premature death of a thread is desired

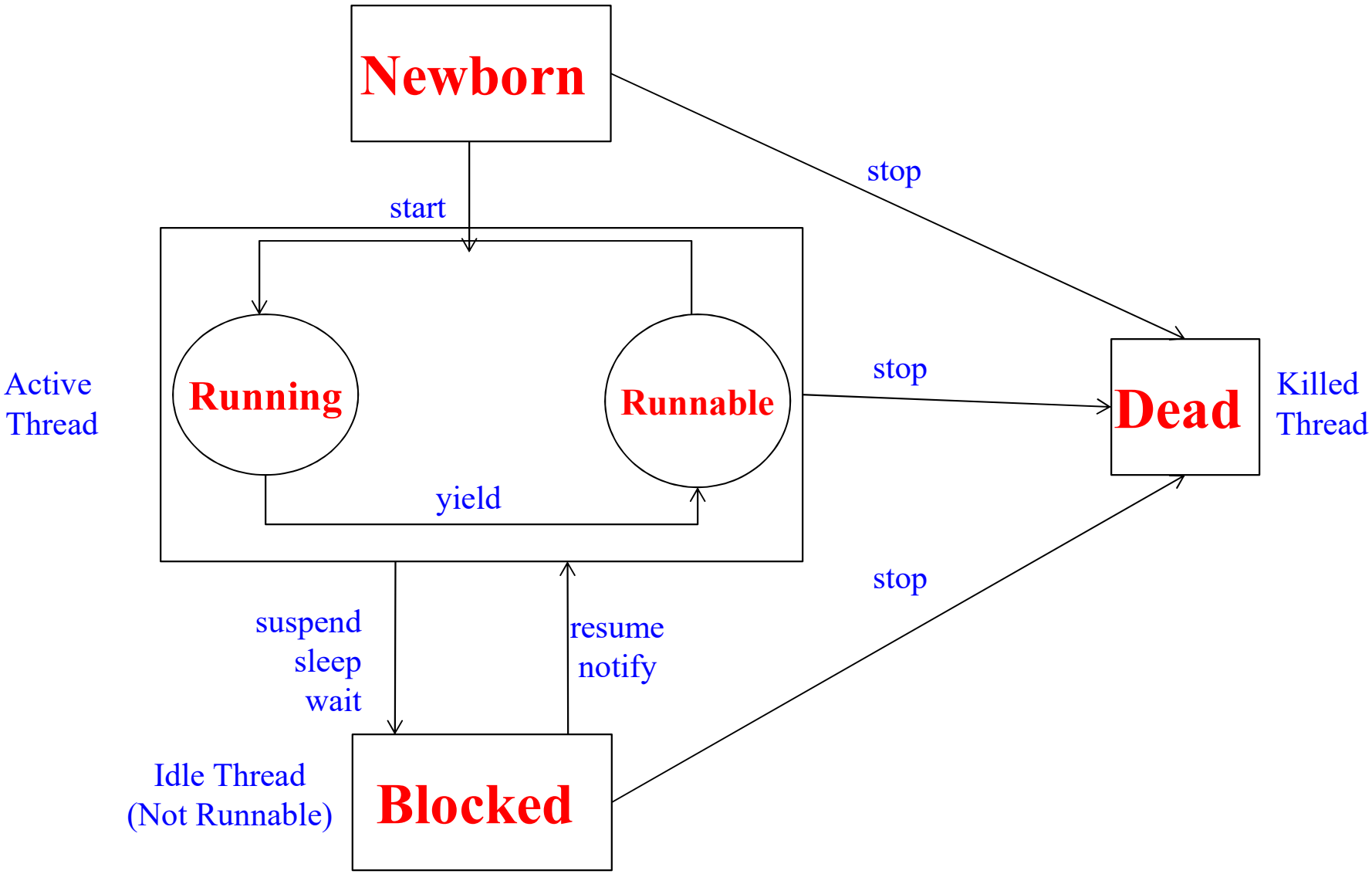
Stopping and Blocking a Thread

- Blocking a thread
  - A thread can also be temporarily suspended or blocked from entering into the runnable and subsequentially running state by using the following thread method
    - Sleep() //block from the specified time
    - Suspended() //block until further order specified time
    - Wait() //block until certain condition occurs
  - These methods caused 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()

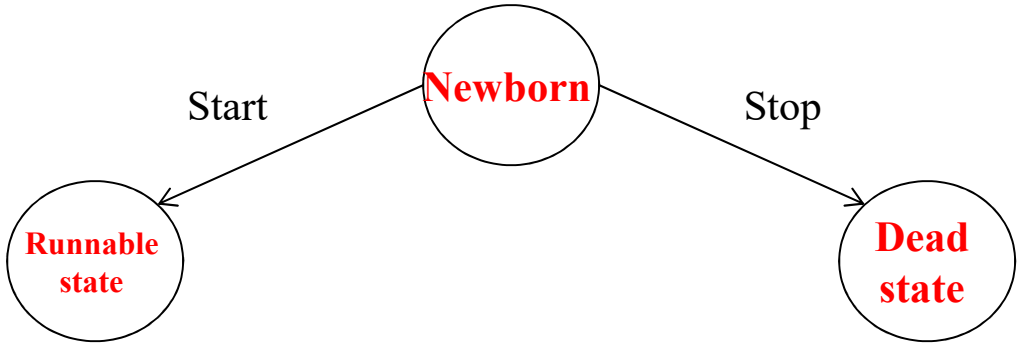
Life Cycle of a Thread

- During the life time of a thread, there are many states it can enter. They include
  - Newborn state
  - Runnable state
  - Running state
  - Blocked state
  - 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

Life Cycle of a Thread

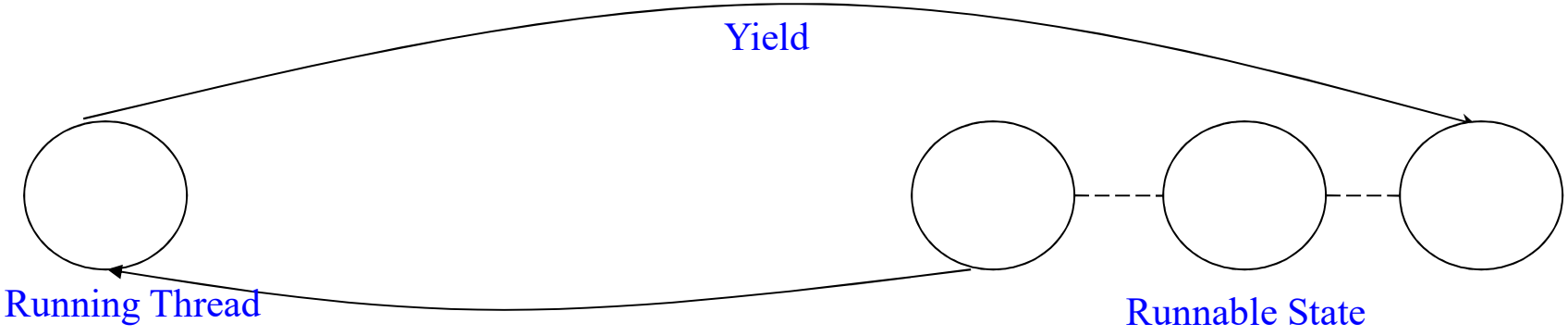


Newborn State



- 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 one of the following
  - Schedule it for running using start() method
  - Kill it using stop() method
- If schedule, it moves to the runnable state

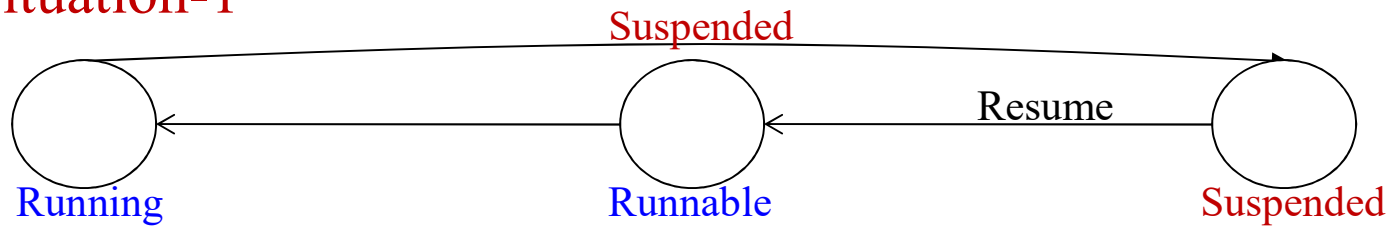
Runnable State



- The runnable state means that the thread is ready for execution and is waiting for the availability of the processor
- If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e, first come first serve manner
- This process of assigning time to thread is known as time slicing
- However, if we want a thread to relinquish control to another thread of equal priority before its turn comes, we can solve by using the yield() method

Running State

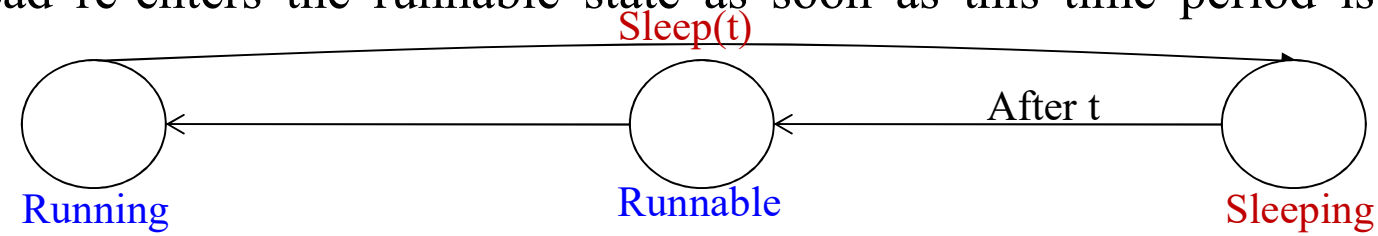
- Running means that the processor has given its time to the thread for its execution
- The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread
- A running thread may relinquish its control in one of the following situation
  - Situation-1



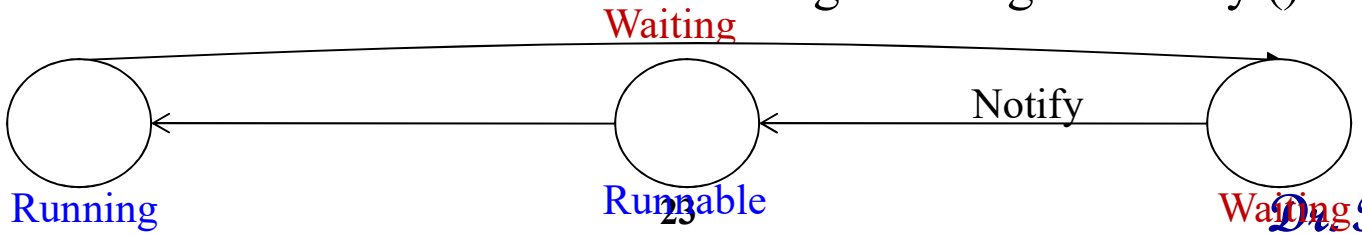
    - It has been suspended using suspend() method
    - A suspended thread can be revived by using the resume() method
    - This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it

Running State

- Situation-2
  - 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 of the queue during this time period
  - The thread re-enters the runnable state as soon as this time period is elapsed



- Situation-3
  - It has been told to until some event occurs
  - This is done using wait() method
  - The thread can be scheduled to run again using the notify () method



Blocked and Dead State

- Blocked state
  - A thread is said to be blocked when it is prevent 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 block thread is considered “not runnable” but not dead and therefore fully qualified to run again
- Dead state
  - 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 the stop() message to it at any state thus causing a premature death to it
  - A thread can be killed as soon it is born, or while it is running, or even when it is in “not runnable” condition

Example

```
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 j=1; j<=5; j++){
System.out.println ("\tFrom Thread B : j="+j);
if (j==3) stop ();
}
System.out.println ("Exit from C");
} }
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 lifecycle{
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");
}
}
```

Example

Here is the output of Program

```
Start thread A
Start thread B
Start thread C
    From Thread B : j = 1
    From Thread B : j = 2
    From Thread A : i = 1
    From Thread A : i = 2
End of main thread
    From Thread C : k = 1
    From Thread B : j = 3
    From Thread A : i = 3
    From Thread A : i = 4
    From Thread A : i = 5
Exit from A
    From Thread C : k = 2
    From Thread C : k = 3
    From Thread C : k = 4
    From Thread C : k = 5
Exit from C
```

Synchronization

- So far, we have seen threads that use their own data and methods provided inside their run() methods
- What happened when they try to use data and methods outside themselves?
- On such occasions, they may compete for the same resources and may lead to serious problems
- For example, one thread may try to read a record from a file while another is still writing to the same file
- Depending on this situation, we may get strange results
- Java enables us to overcome this problem using a technique known as synchronization

Synchronization

- *Synchronization* control the access the multiple threads to a shared resources
- *Synchronization* is the process of allowing threads to execute one after another
- Without synchronization of threads…
  - One thread can modify a shared variable while another thread can update the same shared variable, which leads to significant errors
- Synchronization avoids memory consistence errors caused due to inconsistent view of shared memory

Synchronization

- When we declare a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time
- As long as the thread holds the monitor, no other thread can enter the synchronized section of code
- A monitor is like a key and the thread that holds the key can only open a the look

```
synchronized(lock-object)
{
  ------------------------
  ------------------------    // statements to be synchronized
  ------------------------
}
```

- Whenever a thread has completed its work of using synchronized method it will hands over the monitor to the next thread that is ready to use the same resources

Example

```
// File Name : Callme.java
class Callme {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      System.out.println("Interrupted");  }
    System.out.println("]");
  }  }

// File Name : Caller.java
class Caller implements Runnable {
  String msg;  Callme target;  Thread t;
  public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();  }
   // synchronize calls to call()
  public void run() {
```

```
//synchronized(target) { // synchronized block
      target.call(msg);
    // }
}  }
// File Name : Synch.java
class sync {
  public static void main(String args[]) {
    Callme target = new Callme();
    Caller ob1 = new Caller(target, "Hello");
    Caller ob2 = new Caller(target,
"Synchronized");
    Caller ob3 = new Caller(target, "World");

    // wait for threads to end
    try {
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }   }  }
```

Deadlock

- An interesting situation may occur when two or more threads are waiting to gain control of a resource
- Due to some reason, the condition on which the waiting threads rely on to gain control does not happen
- This result in what is know as deadlock
- For example assume that the thread A must access Method1 before it can release Method2, but the thread B cannot release Method1 until it gets hold of Method2
- Because these are mutually exclusive conditions, deadlock occurs

Deadlock

Thread A

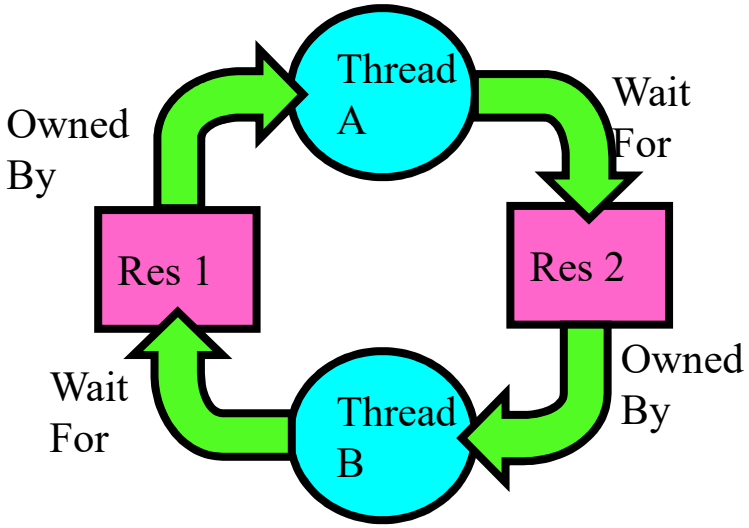
```
synchronized method2 ()
{
synchronized method1 ()
  ----------------------
  ----------------------
}
}
```

Thread B

```
synchronized method1 ()
{
synchronized method2 ()
  ----------------------
  ----------------------
}
}
```

Deadlock

- Deadlock: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
  - Thread B owns Res 2 and is waiting for Res 1



- Consider mutexes ‘x’ and ‘y’:

Thread A	Thread B
x.P();	y.P();
y.P();	x.P();

Example

```
class A {
  synchronized void foo(B b) {
    String name =
Thread.currentThread().getName();
    System.out.println(name + " entered A.foo");
    try {
      Thread.sleep(1000);
    } catch(Exception e) {
      System.out.println("A Interrupted");
    }
    System.out.println(name + " trying to call
B.last()");
    b.last();
  }
  synchronized void last() {
    System.out.println("Inside A.last");
  }} class B {
  synchronized void bar(A a) {
    String name =
Thread.currentThread().getName();
    System.out.println(name + " entered B.bar");
    try {
      Thread.sleep(1000); }
 catch(Exception e) {
System.out.println("B Interrupted");
    } System.out.println(name + " trying to
callA.last()");
    a.last();
  }
synchronized void last() {
    System.out.println("Inside A.last");
  }}
class Deadlock implements Runnable {
  A a = new A();
  B b = new B();
  Deadlock() {
  Thread.currentThread().setName("MainThread");
    Thread t = new Thread(this, "RacingThread");
    t.start();
    a.foo(b); // get lock on a in this thread.
    System.out.println("Back in main thread");}
  public void run() {
    b.bar(a); // get lock on b in other thread.
    System.out.println("Back in other thread");}
  public static void main(String args[]) {
    new Deadlock();
  }
}
```

Implementing the runnable interface

- The runnable interface declares the run() method that is required for implementing threads in programs
- To do this, we must perform the steps listed below
- Declare the class as implementing the runnable interface
- Implement the run() method
- Create a thread by defending an object that is instantiated from this “runnable” class as the target of the thread
- Call the thread’s start() method to run the thread

Implementing the runnable interface

Program *Using Runnable interface*

```
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");
    }
}
```

Implementing the runnable interface

Output of Program

```
End of main Thread
        ThreadX : 1
        ThreadX : 2
        ThreadX : 3
        ThreadX : 4
        ThreadX : 5
        ThreadX : 6
        ThreadX : 7
        ThreadX : 8
        ThreadX : 9
        ThreadX : 10
End of ThreadX
```