CHAPTER 11

MULTI THREADING

INTRODUCTION

Those who are familiar with the modern operating systems such as Linux, Windows-8, Window-10 may recognize that they can execute several program simultaneously. For example you can play a song using Media Player while typing a Word document. This ability is known as multitasking. In system's terminology, it is called multithreading.

Multithreading is a conceptual programming technique where a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel.

For example, one subprograms (processes) can display an animation on the screen while another may build the next animation to be displayed. This is something similar to dividing a task into subtask and assigning them to different people for execution independently and simultaneously.

A thread is similar to a program that has a single flow of control. It has a beginning, a body, and a end, and executes commands sequentially. In fact, all main programs in our earlier example can be called *single-threaded* programs. Every program will have at least one thread.

A unique property of java is its support for multithreading. That is java enable multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny program known as *thread* that runs in parallel to other threads. A program that contains multiple flows of control is known as *multithreading* program.

> **Threads are Java's way of making a single Java Virtual Machine look like many machines, all running at the same time. This effect, usually, is an illusion: There is only one JVM and most often only one CPU; but the CPU switches among the JVM's various subtask to give the impression of multiple CPUs.**

The ability of a language to support multithreads is referred to as **concurrency**. There are system threads that work behind the scenes on your behalf, listening for user input and managing garbage collection. The best way to cooperate with these facilities is to understand what threads really are.

> **TIP:**
>
> **The terms *parallel* and *concurrent* occur frequently in computer literature, and the difference between them can be confusing. When two threads run in parallel, they are both being executed at the same time on different CPUs. However, two concurrent threads are both in progress, or trying to get some CPU time for execution at the same time, but are not necessarily being executed simultaneously on different CPUs.**

Hence **Multithreading in java** is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

Differences between multi threading and multitasking:

MULTI THREADING	MULTI TASKING
1) More than one thread running simultaneously	1). More than one process running simultaneously
2) Its part of a program	2) Its a program.
3) It is a light-weight process.	3) It is a heavy-weight process.
4) Threads are divided into sub threads	4) Process is divided into threads.
5) Within the process threads are communicated.	5) Inter process communication is difficulty
6) Context switching between threads is cheaper.	6). Context switching between process is costly
7) It is controlled by Java(JVM)	7). It is controlled by operating System.
8) It is a specialized form of multi tasking	8) It is a generalized form of multi threading.
9) Example: java's automatic garbage collector.	8) Program compilation at command prompt window and preparing documentation at MS-Office.

Advantage of Java Multithreading

- ➢ You can perform many operations together so it saves time.
- ➢ Threads are independent so it doesn't affect other threads if exception occur in a single thread.
- ➢ Threads are used in designing serverside programs to handle multiple clients at a time.
- ➢ Threads are used in games and animations.
- ➢ We can reduce the idle time of processor.
- ➢ Performance of processor is improved.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins.

The main thread is important for two reasons:
• It is the thread from which other "child" threads will be spawned.
• Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.

Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun microsystem, there are only 4 states in thread life cycle in java **new**, **runnable**, **non-runnable** and **terminated**. There is no **running state**.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable

3. Running
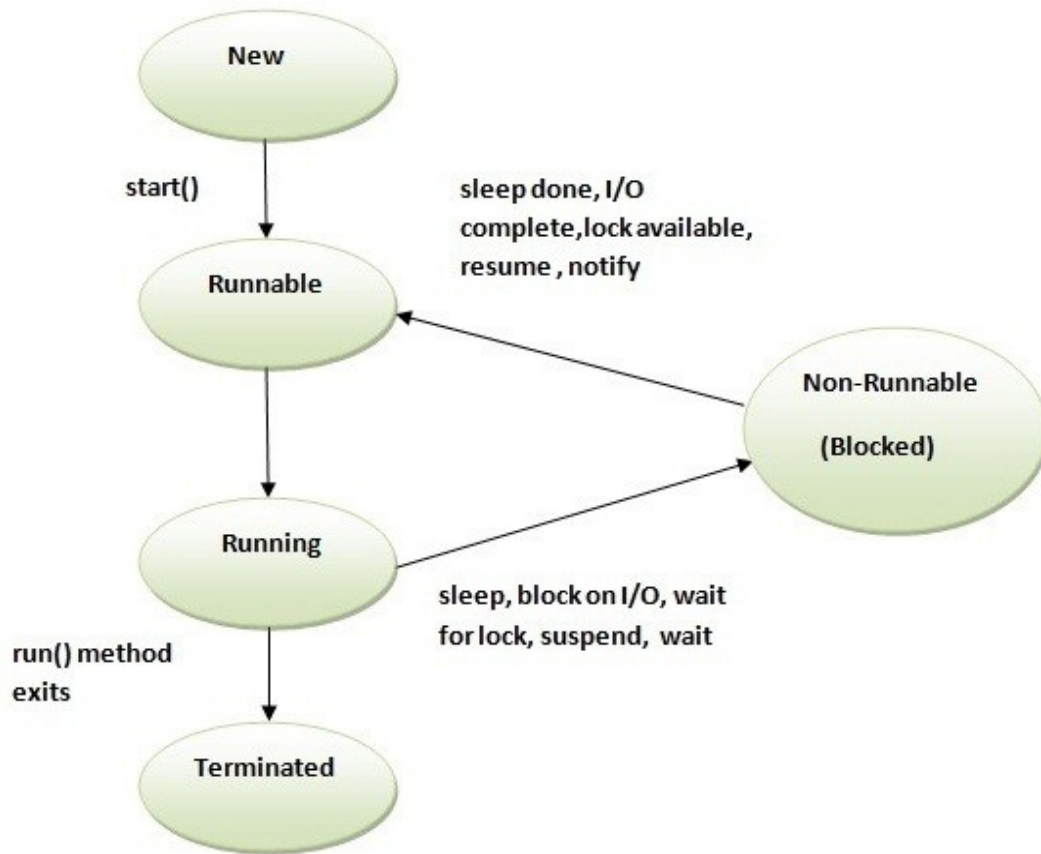4. Non-Runnable (Blocked)
5. Terminated

Figure 12.1

1) New State

When we create a thread object, the thread is born and said to be in newborn state. The thread is not yet scheduled for running. At this state we can do only one of the following things with it.

➢ Schedule it for running using **start()** method.
➢ Kill it using **stop()** method.
➢ If scheduled, it moves to the runnable state. If we attempt to use any

other method at this stage, an exception will be thrown.

2) Runnable State

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of the threads that are waiting for the execution. If all the threads have equal priority, then they are given time slots for execution in round robin fashion i.e. first-come, first-serve manner. Thread that relinquishes control joins the queue at the end and again waits for its turns. This process of assigning time to threads is known as time-slicing.

However, if we want a thread to relinquish control to another thread to equal priority its turn comes, we can do so by using the **yield()** method.
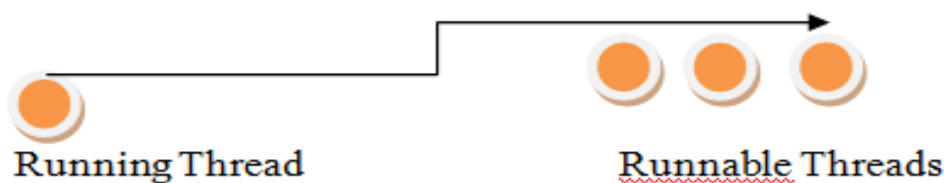

Running Thread   Runnable Threads

Figure 12.2 Relinquishing control using yield( ) method

3) Running

The thread is in running state if the thread scheduler has selected it i.e. the processor has given its Time to the thread for its execution. Thread runs until it relinquishes control on its own or it is preempted by higher priority thread. A running thread may relinquishes its control in one the following situation:

1. It has been suspended using **suspend()** method. A suspended thread can be revived by using **resume()** method. This approach is useful

when we want to suspend a thread for some time due to certain reasons, but do not want to kill it.
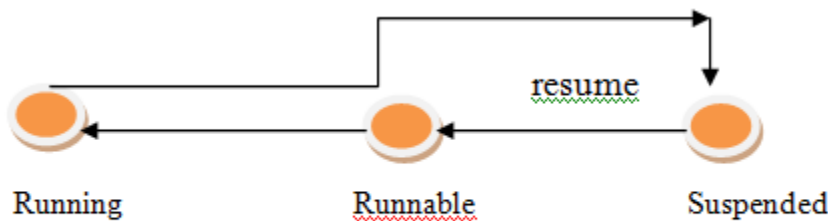

Figure 12.3 Relinquishing control using suspend( ) method

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 time period is elapsed.
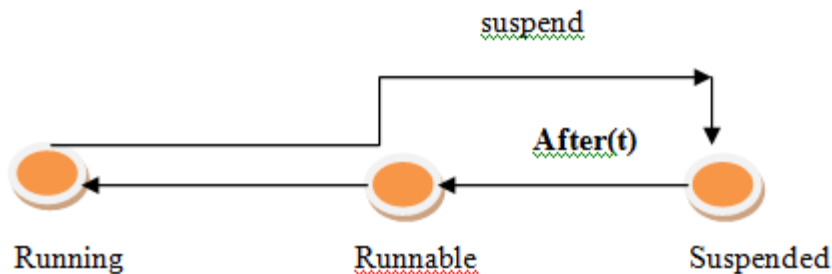

Figure 12.4 Relinquishing control using sleep() method

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

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its **run()** method exits.

How to create thread

Creating threads in Java is simple. Threads are implemented in the form of objects that contain a method **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. A typical run( ) method would appear as follows:

```
public void run( )
{
    ……………….
    ……………….
    ………………
}
```

The **run( )** method should be invoked by an object of the concerned thread. Thus can be achieved by creating the thread and initiating it with the help of another thread method called **start( )**

There are two ways to create a thread:

1. By extending **Thread** class
2. By implementing **Runnable** interface.

1. Extending Thread class:

Define a class that extends **Thread** class and override its **run( )** method with the code required by the thread.

Thread class provide constructors and methods to create and perform operations on a thread.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(object name, String name);

Commonly used methods of Thread class:

public void run(): is used to perform action for a thread.
public void start(): starts the execution of the thread.JVM calls the run() method on the thread.
public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
public int getPriority(): returns the priority of the thread.
public int setPriority(int priority): changes the priority of the thread.
public String getName(): returns the name of the thread.
public void setName(String name): changes the name of the thread.
public Thread currentThread(): returns the reference of currently executing thread.
public boolean isAlive(): tests if the thread is alive.
public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
public void suspend(): is used to suspend the thread(depricated).
public void resume(): is used to resume the suspended thread(depricated).
public void stop(): is used to stop the thread(depricated).
public boolean isDaemon(): tests if the thread is a daemon thread.
public void setDaemon(boolean b): marks the thread as daemon or user thread.

An Example of Using Thread

Example1
```
class A extends Thread
{
    public void run()
    {
        for(int i = 1 ; i<=5;i++)
        {
```

```java
            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 :K = " + k);
        }
        System.out.println("Exit From C");
    }
}
public class ThreadExample1 {

    public static void main(String[] args)
    {
        // TODO Auto-generated method stub
```

```java
        //A3 objA = new A3();
        //objA.start();
     A obA = new A();
      B obB =  new B();
      C obC = new C();
       obA.start();
       obB.start();
       obC.start();

    }

}
```

Output:

```
    From Thread A :i = 1
    From Thread A :i = 2
    From Thread A :i = 3
    From Thread A :i = 4
    From Thread A :i = 5
Exit From A
    From Thread C :K = 1
    From Thread B :j = 1
    From Thread C :K = 2
    From Thread B :j = 2
    From Thread C :K = 3
    From Thread B :j = 3
    From Thread C :K = 4
    From Thread B :j = 4
    From Thread C :K = 5
    From Thread B :j = 5
Exit From C
Exit From B
```

We have simply initiated three new threads and started them. We did not hold on to them any further. They are running concurrently on their own. Note that the output from the threads are not specially sequential. They do not follow

any specific order. They are running independently of one another and each executes whenever it has a chance. Remember, once the threads are started, we cannot decide with certainty the order in which they may execute statements. Note that a second run has a different sequence.

Can We Call the run( ) Method Directly?
Example2

```java
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");
    }
}
```

```java
class C extends Thread
{
    public void run()
    {
        for(int k = 1 ; k<=5;k++)
        {
        System.out.println("\t From Thread C :K = " +
k);
        }
        System.out.println("Exit From C");
    }
}
public class ThreadExample1 {

    public static void main(String[] args)
    {
        // TODO Auto-generated method stub
         //A3 objA = new A3();
         //objA.run();
         new A().run();
         new B().run();
         new C().run();

    }

}
```

Output
```
From Thread A :i = 1
      From Thread A :i = 2
      From Thread A :i = 3
      From Thread A :i = 4
      From Thread A :i = 5
Exit From A
      From Thread B :j = 1
      From Thread B :j = 2
```
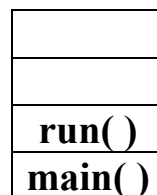
```
      From Thread B :j = 3
      From Thread B :j = 4
      From Thread B :j = 5
Exit From B
      From Thread C :K = 1
      From Thread C :K = 2
      From Thread C :K = 3
      From Thread C :K = 4
      From Thread C :K = 5
Exit From C
```

Compare the above output with example1 you will notice that the modified program does not depict multithreading programming behavior and run sequentially instead. Reason behind is that: each thread run in a separate callstack.

main()	run()	run()
Stack A	Stack B	Stack C
(main thread)	(t1 thread)	(t2 thread)
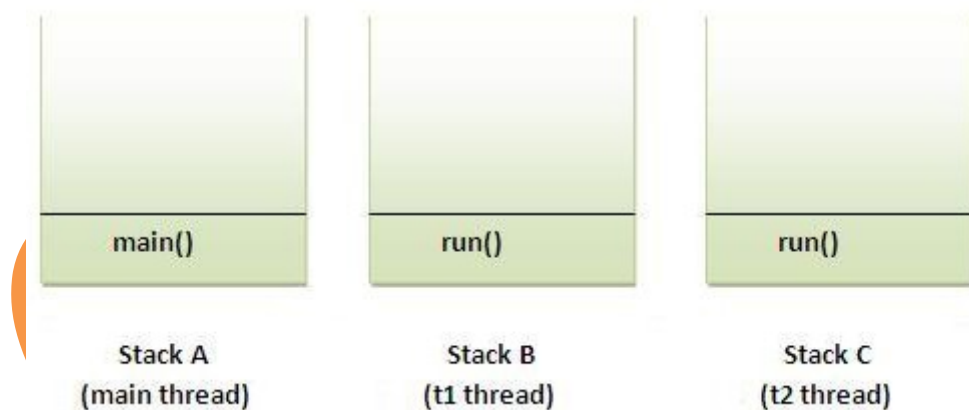
run( ) method invoke by start( ) method

run( )
main( )

Call stack (main thread)
run( ) invoked directly from main( ) method

Since the direct call to the run() method from main does not create a separate call stack, its instructions are executed sequentially and not parallel. Thus calling the run() method directly causes the program to behave like normal single threaded program.

How to perform multiple tasks by multiple threads (multitasking in multithreading)?

If you have to perform multiple tasks by multiple threads, have multiple run() methods. For example:

Example1:

```
class Simple1 extends Thread
{
    public void run()
    {
        System.out.println("task one");
    }
}

class Simple2 extends Thread
{
    public void run()
    {
        System.out.println("task two");
    }
}

class TestMultitasking3
{
```

```
    public static void main(String args[])
    {
        Simple1 t1=new Simple1();
        Simple2 t2=new Simple2();

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

THREAD PRIORITIES

In Java, thread scheduler can use the thread **priorities** in the form of **integer value** to each of its thread to determine the execution schedule of threads . Thread gets the **ready-to-run** state according to their priorities. The **thread scheduler** provides the CPU time to thread of highest priority during ready-to-run state.

Priorities are integer values from 1 (lowest priority given by the constant **Thread.MIN_PRIORITY**) to 10 (highest priority given by the constant **Thread.MAX_PRIORITY**). The default priority is 5(**Thread.NORM_PRIORITY**).

Constant	Description
Thread.MAX_PRIORITY	The maximum priority of any thread (an int value of 10)
Thread.MIN_PRIORITY	The minimum priority of any thread

	(an int value of 1)
Thread.NORM_PRIORITY	The normal priority of any thread (an int value of 5)

The methods that are used to set the priority of thread shown as:

Method	Description
setPriority()	This is method is used to set the priority of thread.
getPriority()	This method is used to get the priority of thread.

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new **higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.

Thread Scheduler

In the implementation of threading scheduler usually applies one of the two following strategies:

- **Preemptive scheduling –** If the new thread has a higher priority then current running thread leaves the runnable state and higher priority

thread enter to the runnable state.

- **Time-Sliced (Round-Robin) Scheduling** – A running thread is allowed to be execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state.

Example 3

```java
class Athread 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 Bthread extends Thread
{
    public void run()
    {
        for(int j = 1 ; j<=5;j++)
        {
            System.out.println("\t From Thread B :j = " +
j);
```

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

class Cthread extends Thread
{
    public void run()
    {
        for(int k = 1 ; k<=5;k++)
        {
            System.out.println("\t From Thread C :K = "
+ k);
        }
        System.out.println("Exit From C");
    }
}
public class ThreadPriorityDemo
{

    public static void main(String[] args)
    {
        // TODO Auto-generated method stub

        Athread objA = new Athread();
        Bthread objB = new Bthread();
        Cthread objC = new Cthread();

        objC.setPriority(Thread.MAX_PRIORITY);
        objB.setPriority(5);
```

```java
        objA.setPriority(1);

        System.out.println("Start Thread A");
        objA.start();
        System.out.println("Start Thread B");
        objB.start();
        System.out.println("Start Thread C");
        objC.start();
        System.out.println("End of Main Thread");


    }

}
```

Output:
```
Start Thread A
Start Thread B
Start Thread C
End of Main Thread
        From Thread B :j = 1
        From Thread B :j = 2
        From Thread C :K = 1
        From Thread C :K = 2
        From Thread C :K = 3
        From Thread C :K = 4
        From Thread C :K = 5
Exit From C
        From Thread B :j = 3
        From Thread B :j = 4
        From Thread B :j = 5
Exit From B
```

```
      From Thread A :i = 1
      From Thread A :i = 2
      From Thread A :i = 3
      From Thread A :i = 4
      From Thread A :i = 5
Exit From A
```

Stopping and Blocking a Thread

Stopping 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( )**.

Thread Exceptions

Note that the call to **sleep( )** method is enclosed in a **try** block and followed by a **catch** block. This is necessary because the sleep() method throws an exception, which should be caught. If we fail to catch the exception, program will not compile.

Java run time system will throw **illegalThreadException** whenever we attempt to invoke a method that a thread cannot handle in the given state. For example, a sleeping thread cannot deal with the resume( ) method because a sleep thread cannot receive any instructions. The same is true with the suspend( ) method when it is used on a blocked (Not Runnable) thread.

Whenever we call a thread method that is likely to throw an exception, we have to supply an appropriate exception handler to catch it. The catch statement may take one of the following forms:

```
Catch(ThreadDeatch e)
{
        …….……..
        ……………          //Killed Thread
}

Catch(InterruptedException e)
{
        …………..
        ……………          //cannot handle it in current state
}

Catch(Exception e)
{
        …………..
        ……………          //any Other
}
```
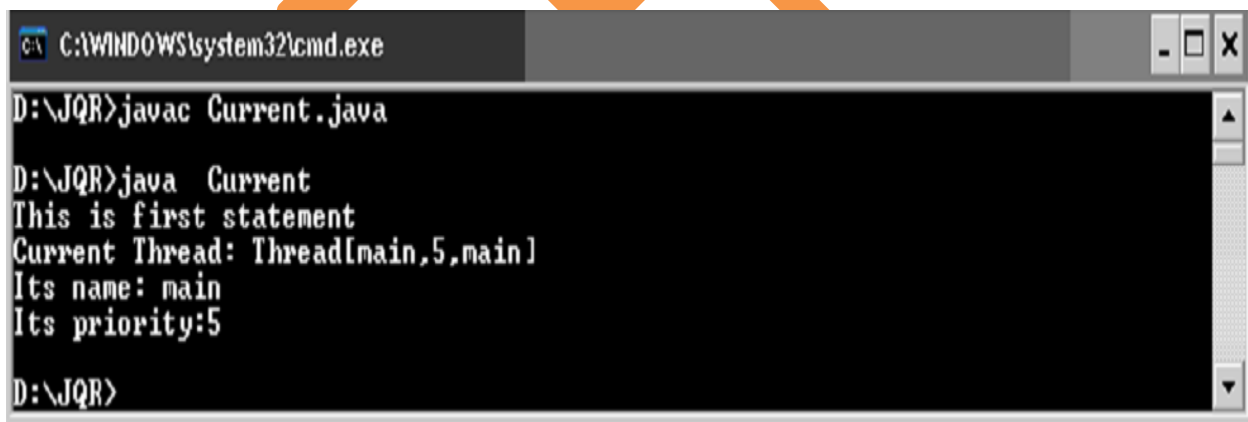
Program : Write a program to know the currently running Thread

```
//Currently running thread
class Current
{
   public static void main(String args[])
   {
       System.out.println ("This is first statement");
       Thread t = Thread.currentThread ();
       System.out.println ("Current Thread: " + t);
       System.out.println ("Its name: " + t.getName ());
       System.out.println ("Its priority:" + t.getPriority ());
   }
}
```

Output:

```
C:\WINDOWS\system32\cmd.exe

D:\JQR>javac Current.java

D:\JQR>java  Current
This is first statement
Current Thread: Thread[main,5,main]
Its name: main
Its priority:5

D:\JQR>
```

2. Implementing Runnable Interface

We stated earlier that we can create threads in two ways: one by using the extended **Thread** class and another by implementing the **Runnable** interface. We have already discussed in detail how the **Thread** class is used for creating and running threads.

The Runnable interface declares the **run( )** method that is required for implementing threads in our programs. To do this, we must perform the following steps listed below:

1. Declare the class as implementing the Runnable interface.
2. Implementing the **run ( )** method.
3. Create a thread by defining an object that is instantiated from the runnable class as the target of the thread.
4. Call the thread's **start( )** method to run the thread.

Example : Using Runnable Interface

Example1:

```java
class X implements Runnable
{
    public void run()
    {
        for(int i = 1; i<=10;i++)
        {
            System.out.println("\t ThreadX " + i);
        }
        System.out.println("End of ThreadX");
    }
}

public class ThreadExample3 {

    public static void main(String[] args)
    {
        // TODO Auto-generated method stub
        X   obj1 = new X();
        X   obj2 = new X();

        Thread threadX = new Thread(obj1, "Sample");
        Thread threadY = new Thread(obj2, "Sample");
```

```
        threadX.start();
        threadY.start();


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

}
```

In the above program we first create an instance of X and then pass this instance as the intial value of the object thread(and object of X class). Whenever, the new thread **threadX** starts up, its run( ) method of the target object supplied to it. Here, the target object is **runnable**.

Output:

```
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
```

Example2:

```java
public class ThreadDemo implements Runnable
{
    public void run()
    {
        for(int counter=1;counter<=100;counter++)
        {
            System.out.println(Thread.currentThread().getName
()+"thread is running..."+counter);
        }
    }
    public static void main(String args[])
    {
        ThreadDemo threadDemo = new ThreadDemo();
        Thread t1 = new Thread(threadDemo,"First");
        Thread t2 = new Thread(threadDemo,"Second");
        t1.start();
        t2.start();
    }
}
```

What is the difference between 'extends thread' and 'implements Runnable' ? which one is advantageous?

'extends thread' and 'implements Runnable'-both are functionally same. But when we write extends Thread, there is no scope to extend another class, as multiple inheritance is not supported in java.

Class MyClass extends Thread,AnotherClass //invalid

If we write implements Runnable, then still there is scope to extend another class.

Class MyClass extends AnotherClass implements Runnable //valid

This is definitely advantageous when the programmer wants to use threads and also wants to access the features of another class.

Single tasking using a thread:

A thread can be employed to execute one task at a time Suppose there are 3 tasks to be executed. We can create a thread and pass the 3 tasks one by one to the thread. For this purpose, we can write all these tasks separately in separate methods; task1(), task2(), task3(). Then these methods should be called from run() method, one by one. Remember, a thread executes only the code inside the run() method. It can never execute other methods unless they are called from run().

Note: public void run() method is executed by the thread by default.

```
//single tasking using a thread
class MyThread implements Runnable
{
       public void run()
       {
              //executes tasks one by one by calling the methods.
              task1();
              task2();
              task3();
       }
       void task1()
       {
              System.out.println("this is task1");
       }
       void task2()
       {
              System.out.println("this is task2");
       }
```

```java
        void task3()
        {
                System.out.println("this is task3");
        }
}
class Sin
{
        public static void main(String args[])
        {
                MyThread obj=new MyThread();
                Thread t1=new Thread(obj);
                t1.start();
        }
}
```

Output: java Sin

This is task1()

This is task2()

This is task3()

In this program, a single thread t1 is used to execute three tasks.

Multi Tasking Using Threads:

In multi tasking, several tasks are executed at a time. For this purpose, we need more than one thread. For example, to perform 2 tasks, we can take 2 threads and attach them to the 2 tasks. Then those tasks are simultaneously executed by the two threads. Using more than one thread is called 'multi threading'.

Program : Write a program to create more than one thread.

```java
//using more than one thread is called Multi Threading
class Theatre extends Thread
{  String str;
  Theatre (String str)
  {  this.str = str;
```

```java
    }
public void run()
{
  for (int i = 1; i <= 10 ; i++)
    {
    System.out.println (str + " : " + i);
    try
    {  Thread.sleep (2000);
    }
    catch (InterruptedException ie)
    {    }

    }
  }
}

class TDemo1
{
    public static void main(String args[])
    {
        Theatre obj1 = new Theatre ("Cut Ticket");
        Theatre obj2 = new Theatre ("Show Chair");
        Thread t1 = new Thread (obj1);
        Thread t2 = new Thread (obj2);
        t1.start ();
        t2.start ();
    }
}
```
**Output:**

```
C:\WINDOWS\system32\command.com                      _ □ ×

D:\JQR>javac TDemo1.java

D:\JQR>java TDemo1
Cut Ticket : 1
Show Chair : 1
Cut Ticket : 2
Show Chair : 2
Cut Ticket : 3
Show Chair : 3
Cut Ticket : 4
Show Chair : 4
Cut Ticket : 5
Show Chair : 5
Cut Ticket : 6
Show Chair : 6
Cut Ticket : 7
Show Chair : 7
Cut Ticket : 8
Show Chair : 8
Cut Ticket : 9
Show Chair : 9
Cut Ticket : 10
Show Chair : 10

D:\JQR>
```

In the preceding example, we have used 2 threads on the 2 objects of TDemo1 class. First we have taken a String variable str in Theatre class. Then we passed two strings- cut ticket and show chair into that variable from TDemo1 class. When t1. start () is executed, it starts execution run () method code showing cut ticket. Note that in run () method, we used: Thread. sleep (2000) is a static method in Thread class, which is used to suspend execution of a thread for some specified milliseconds. Since this method can throw InterruptedException, we caught it in catch block. When Thread t1 is suspended immediately t2. start () will make the thread t2 to execute and when it encounters Thread.sleep(2000), it will suspend for specified time meanwhile t1 will get executed respectively. In this manner, both the threads are simultaneously executed.

Multiple Threads Acting on Single Object:

For example, imagine a Java application where one thread (which let us assume as Producer) writes data to a data structure, while a second thread (consider this as Consumer) reads data from the data structure This example use concurrent threads that share a common resource: a data structure.
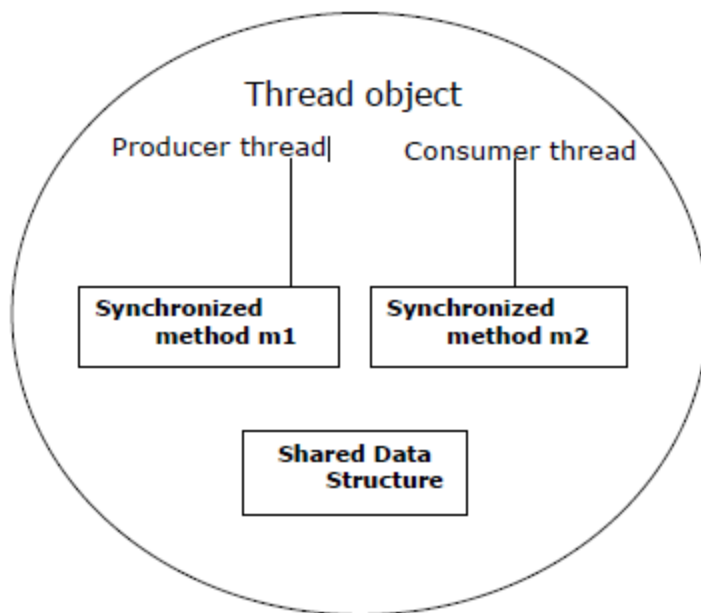
Suppose there is a producer thread in the threaded object that is writing into a data structure within the thread object using a method say M1.

- While the producer thread is in method M1 and in the process writing into the data structure, care must be taken to ensure that while data is in the process of being written to the data structure, no other thread, say a consumer thread must be allowed to read the data through some other method (say M2) of the thread object at the same time while the writing of data is on.

- The consumer thread should wait till the producer thread has finished writing into the data structure, i.e., till the producer thread returns from method M1.

- The moment the producer thread returns from method M1, the consumer thread should be allowed to access the data structure through method M2.

The argument is moving towards a mechanism by which you are trying to ensure that no two threads end up accessing a shared data structure at the same time that might lead to corrupting the data in the data structure, and might also lead to unpredictable results.

What is the solution for this problem?

There is a need for a mechanism to ensure that the shared data will be used by only one thread at a time. This mechanism is called synchronization.

Synchronizing Threads or Thread Synchronization or Thread Safe: When a thread is acting on an object preventing other threads from acting on the same object is called Thread Synchronization or Thread Safe. The object on which the threads are synchronized is called 'synchronized object'.

The Object on which the Threads are synchronized is called synchronized object or Mutex (Mutually Exclusive Lock).Synchronized object is like a locked object, locked on a thread. It is like a room with only one door. A person has entered the room and locked form it from behind. The second person who wants to enter the room should wait till the first person comes out. In this way, a thread also locks the object after entering it. Then the next thread cannot enter it till the first thread comes out. This means the object is locked mutually on threads. So, this object is called 'mutex'.

Implementation of concurrency control mechanism is very simple because every Java object has its own implicit monitor associated with it

If a thread wants to enter an object's monitor, it has to just call the synchronized method of that object

While a thread is executing a synchronized method, all other threads that are trying to invoke that particular synchronized method or any other synchronized method of the same object, will have to wait
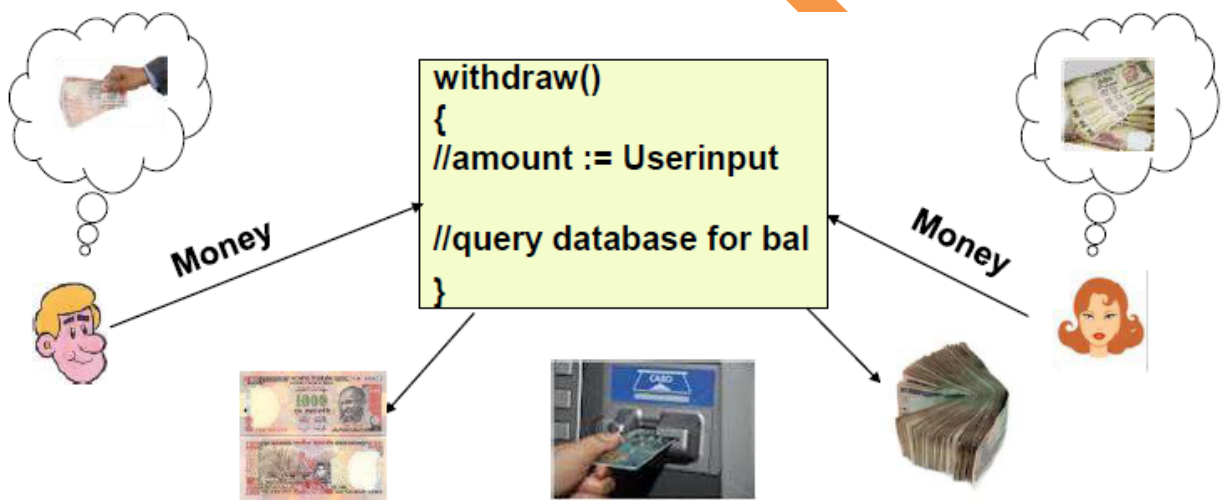
John and Mary are withdrawing money from their joint account. They are withdrawing at the same **instance** from different ATMs.

What will be the concern here?

Each transaction occurs independently through different threads. Since both transactions are unaware of the other, this may lead to inconsistent state. How to avoid this situation?
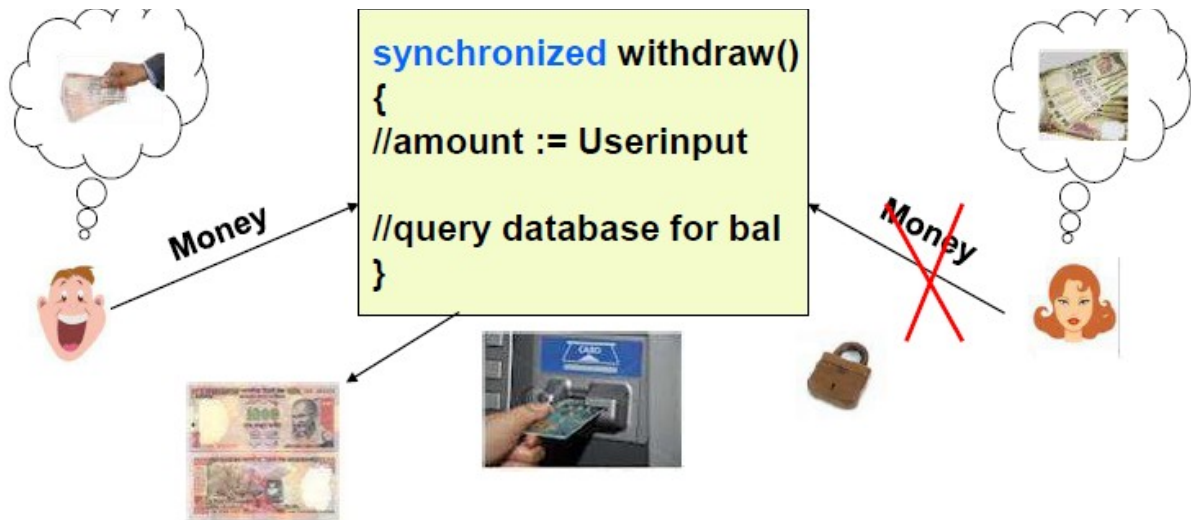
The concern shared in the previous slide can be easily handled using synchronization. When John wants to withdraw cash from his account, the thread that is responsible for handling this transaction gets an exclusive lock(monitor), which ensures that Mary cannot access this method concurrently. This mechanism comes in the form of synchronized method.

Thread synchronization is done in two ways:

➢ Using synchronized block we can synchronize a block of statements.

e.g.: *synchronized (obj)*

  {

  statements;

  }

Here, object represents the object to be locked or synchronized. The statements inside the synchronized block are all available to only one thread at a time. They are not available to more than one thread simultaneously.

➢ To synchronize an entire method code we can use synchronized word before method name

e.g.: synchronized void method ()
        {
            Stmts;
        }

Now the statements inside the method are not available to more than one thread at a time. This method code is synchronized.

Implementing Synchronization-Example

Example1

```java
class Account
{
    int balance;
    public Account()
    {
        balance=5000;
    }
    public synchronized void withdraw(int bal)
    {
        If(bal<=balance)
        {
            System.out.println("Amount Widraw " + bal);
            try
            {
                Thread.sleep(1000);
            }
            catch(Exception ex)
            {
                System.out.println("EXCEPTION
                    OCCURED.."+ex);
            }
        balance= balance-bal;
```

```java
            System.out.println("Balance remaining:::" +
balance);
        }

}

}


class Amtthread implements Runnable
{
        Account obj;
        public Amtthread(Account a)
        {
            obj=a;
        }
        public void run()
        {
            obj.withdraw(500);
        }
}


public class SynchExample
{
        public static void main(String args[])
        {
            Account a1=new Account();
            Amtthread c1=new Amtthread(a1);
```

```
        Thread t1=new Thread(c1, "ABC");
        Thread t2=new Thread(c1,"XTZ");
        t1.start();
        t2.start();
        }
}
```

Example2

Take the case of railway reservation. Every day several people want reservation of a berth for them. The procedure to reserve the berth is same for all the people. So we need some object with same run () method to be executed repeatedly for all the people (threads).

Let us think that only one berth is available in a train and two passengers (threads) are asking for that berth in two different counters. The clerks at different counters sent a request to the server to allot that berth to their passengers. Let us see now to whom that berth is allotted.

Write a program to thread synchronization by using synchronized keyword before the method name

```
class Reserve implements Runnable
{ //available berths are 1
  int available = 1;
  int wanted;
 //accept wanted berths at runtime Reserve (int i)
    Reserve(int i)
     {
        wanted = i;
     }
     public void run()
      {//display available  berths
```

```java
        System.out.println ("Number of berths available: "
+ available);
        //if available berths more  thanwanted  betrhs
            if ( available >= wanted)
            {    //get the name of the passenger
          String name = Thread.currentThread ().getName
();
      System.out.println (wanted + " berths alloted to: "
+ name);
            try
            {
      Thread.sleep (2000); // wait for priniting the
ticket
                available = available - wanted;
                //update the no.of  available berths
          }   catch (InterruptedException ie){ }
          }
          else
          {     System.out.println ("Sorry, no berths
available");
          }
        }
}
class UnSafe
{
 public static void main(String args[])
  {
          Reserve obj = new Reserve (1);
          Thread t1 = new Thread (obj);
          Thread t2 = new Thread (obj);
          t1.setName ("First Person");
          t2.setName ("Second Person");
          t1.start ();
          t2.start ();
      }
}
```

/ Write a Java program that creates three threads. First thread displays "Good Morning" every one second, the second thread displays "Hello" every two seconds and the third thread displays "Welcome" every threeseconds. */**

```java
class A extends Thread
{
synchronized public void run()
{
try
{
while(true)
{
sleep(1000);
System.out.println("good morning");
}
}
catch(Exception e)
{
}
}
}

class B extends Thread
{
synchronized public void run()
{
try
{
while(true)
{
sleep(2000);
System.out.println("hello");
}
}
catch(Exception e)
{       }
```

```java
}
}
class C extends Thread
{
synchronized public void run()
{
try
{
while(true)
{
sleep(3000);
System.out.println("welcome");
}
}
catch(Exception e)
{      }
}
}
class ThreadDemo
{
public static void main(String args[])
{
A t1=new A();
B t2=new B();
C t3=new C();
t1.start();
t2.start();
t3.start();
}
}
```

Output:

Press Cntrl+C to exit

Good morning

Hello

Good morning

Welcome

Good morning

Hello

Good morning

Good morning

Hello

Interthread Communication

Java provide benefit of avoiding thread pooling using interthread communication. The **wait()**, **notify()**, **notifyAll()** of Object class. These method are implemented as **final** in Object. All three method can be called only from within a **synchronized** context.

- **wait()** tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
- **notify()** wakes up a thread that called wait() on same object.
- **notifyAll()** wakes up all the thread that called wait() on same object.

*Difference between `wait()` and `sleep()`*

wait()	sleep()
called from synchronised block	no such requirement
monitor is released	monitor is not released
awake when notify() or notifyAll() method is called.	not awake when notify() or notifyAll() method is called
not a static method	static method
wait() is generaly used on condition	sleep() method is simply used to put

	your thread on sleep.

Thread Pooling

Pooling is usually implemented by loop i.e to check some condition repeatedly. Once condition is true appropriate action is taken. This waste CPU time.

Deadlock
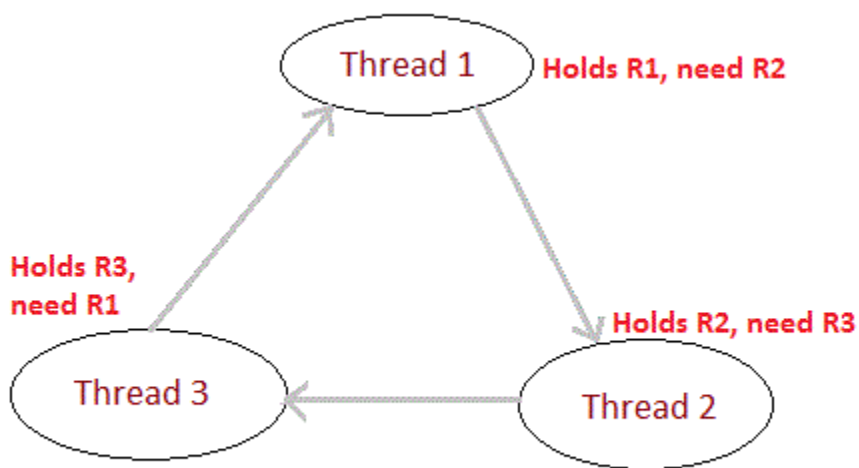
Deadlock Condition

Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.

Things to Remember about Using wait(), notify() and notifyAll() method

1. You can use wait() and notify() method to implement inter-thread communication in Java. Not just one or two threads but multiple threads can communicate to each other by using these methods.

2. Always call `wait()`, `notify()` and `notifyAll()` methods from synchronized method or synchronized block otherwise JVM will throw `IllegalMonitorStateException`.
3. Always call wait and notify method from a loop and never from `if()` block, because loop test waiting condition before and after sleeping and handles notification even if waiting for the condition is not changed.

Example of deadlock

```
class Pen{}
class Paper{}

public class Write {

    public static void main(String[] args)
    {
        final Pen pn =new Pen();
        final Paper pr =new Paper();

        Thread t1 = new Thread()
        {
                public void run()
                {
                    synchronized(pn)
                    {
        System.out.println("Thread1 is holding Pen");
                        try{
                            wait();
                        }catch(InterruptedException e){}
                        synchronized(pr)
    {   System.out.println("Requesting for Paper"); }
                        Notify()
                    }
                }
        };
        Thread t2 = new Thread()
        {
```

```java
            public void run()
            {
                synchronized(pr)
                {
                    System.out.println("Thread2 is
holding Paper");
                    try{
                        wait();
//Thread.sleep(1000);
                    }catch(InterruptedException e){}
                    synchronized(pn)
                    {   System.out.println("requesting
for Pen");
Notify() }

                }
            }
        };

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

}
```

Output :

```
Thread1 is holding Pen
Thread2 is holding Paper
```

/** Write a Java program that correctly implements producer consumer problem using the concept of interthread communication.*/

//java program for producer and consumer--inter thread communication

class Producer implements Runnable{

```java
        Thread t;
        Producer(Thread t)
        {
                this.t=t;
                new Thread(this,"Producer").start();
        }
        public void run()
        {
                int i=0;
                while (true)
                {
                        t.put(i++);
                }
        }
}
class Consumer implements Runnable
{
    Thread t;
        Consumer(Thread 1t)
        {
                this.t=t;
                new Thread(this,"Consumer").start();
        }
        public void run()
        {
                int i=0;
                while (true)
                {
                        t.get();
                }
        }
}
class  ProducerConsumer
{
        public static void main(String[] args)
        {
                Thread1 t=new Thread1();
```

```
                System.out.println("Press Control+c to exit");
                new Producer(t);
                new Consumer(t);


        }
}
```

Output:

Press Control+C to exit

Put:0

Get:0

Put:1

Get1

……………

Deamon Threads:

Daemon threads are sometimes called "service" threads that normally run at a low priority and provide a basic service to a program or programs when activity on a machine is reduced. An example of a daemon thread that is continuously running is the garbage collector thread. This thread, provided by the JVM, will scan programs for variables that will never be accessed again and free up their resources back to the system. A thread can set the daemon flag by passing a true boolean value to the **setDaemon()** method. If a false boolean value is passed, the thread will become a user thread. However, this must occur before the thread has been started.

A deamon thread is a thread that executes continuously. Daemon threads are service providers for other threads or objects. It generally provides a background processing.

➢ To make a thread t as a deamon thread, we can use setDeamon() method as:

   t.setDeamon(true);

➢ To know if a thread is deamon or not, isDeamon is useful.

   boolean x=t.isDaemon().

Write a example program for setting a thread as a daemon thread

```java
public class DaemonDemo extends Thread
{
    public void run()
    {
        for(int i=0;i<5;i++)
        System.out.println(this.getName()+" :"+i);
    }
    public static void main(String args[])
    {
        DaemonDemo d1=new DaemonDemo();
        DaemonDemo d2=new DaemonDemo();
        d1.setName("Daemon thread");
        d2.setName("Normal thread");
        d1.setDaemon(true);
        d1.setPriority(Thread.MIN_PRIORITY);
        d1.start();
        d2.start();
    }
}
```

Output:
Daemon thread:0
Normal thread:0
………….
Daemon thread:4
Normal thread:4

Thread Groups:

Thread Group: A ThreadGroup represents a group of threads. The main advantage of taking several threads as a group is that by using a single method, we will be able to control all the threads in the group.

Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads. For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience.

> Creating a thread group*:   ThreadGroup tg = new ThreadGroup ("groupname");*
> To add a thread to this group (tg):*Thread t1 = new Thread (tg, targetobj, "threadname");*
> To add another thread group to this group (tg):

   ThreadGroup tg1 = new ThreadGroup (tg, "groupname");

> To know the parent of a thread: *tg.getParent ();*
> To know the parent thread group: *t.getThreadGroup ();*

This returns a ThreadGroup object to which the thread t belongs.

> To know the number of threads actively running in a thread group: *t.activeCount ();*
> To change the maximum priority of a thread group tg: *tg.setMaxPriority ();*

Method	Description
int activeCount( )	Returns the number of threads in the group plus any groups for which this thread is a parent.
int activeGroupCount( )	Returns the number of groups for which the invoking thread is a parent.
final void checkAccess( )	Causes the security manager to verify that the invoking thread may access and/or change the group on which **checkAccess( )** is called.
final void destroy( )	Destroys the thread group (and any child groups) on which it is called.
int enumerate(Thread *group*[ ])	The threads that comprise the invoking thread group are put into the *group* array.
int enumerate(Thread *group*[ ], boolean *all*)	The threads that comprise the invoking thread group are put into the *group* array. If *all* is **true**, then threads in all subgroups of the thread are also put into *group*.
int enumerate(ThreadGroup *group*[ ])	The subgroups of the invoking thread group are put into the *group* array.
int enumerate(ThreadGroup *group*[ ], boolean *all*)	The subgroups of the invoking thread group are put into the *group* array. If *all* is **true**, then all subgroups of the subgroups (and so on) are also put into *group*.
final int getMaxPriority( )	Returns the maximum priority setting for the group.
final String getName( )	Returns the name of the group.
final ThreadGroup getParent( )	Returns **null** if the invoking **ThreadGroup** object has no parent. Otherwise, it returns the parent of the invoking object.
final void interrupt( )	Invokes the **interrupt( )** method of all threads in the group.

TABLE 16-18   The Methods Defined by **ThreadGroup**

 Program : Write a program to demonstrate the creation of thread group.

//Using ThreadGroup

import java.io.*;

class WhyTGroups

{ public static void main (String args[]) throws IOException

  { Reservation res = new Reservation ();

   Cancellation can = new Cancellation ();

   //Create a ThreadGroup

   ThreadGroup tg = new ThreadGroup ("Reservation Group");

```java
    //Create 2 threads and add them to thread group

    Thread t1 = new Thread (tg, res, "First Thread");

    Thread t2 = new Thread (tg, res, "Second Thread");

    //Create another thread group as a child to tg

    ThreadGroup tg1 = new ThreadGroup (tg, "Cancellation Group");

    Thread t3 = new Thread (tg1, can, "Third Thread");

    Thread t4 = new Thread (tg1, can, "Fourth Thread");

    //find parent group of tg1

    System.out.println ("Parent of tg1 = " + tg1.getParent ());

    //set maximum priority

    tg1.setMaxPriority (7);

    System.out.println ("Thread group of t1 = " + t1.getThreadGroup ());

    System.out.println ("Thread group of t3 = " + t3.getThreadGroup ());

    t1.start ();

    t2.start ();

    t3.start ();

    t4.start ();

    System.out.println ("Number of threads in this group : " + tg.activeCount
() );

  }

}

class Reservation extends Thread

{ public void run ()

  { System.out.println ("I am Reservation Thread");
```

```
  }

}

class Cancellation extends Thread

{  public void run ()

  {  System.out.println ("I am Cancellation Thread");

  }

}
```

Output:

What is Garbage Collection?

Garbage Collection in computer science is a form of automatic memory management. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. Garbage collection does not traditionally manage limited resources other than memory that typical programs use, such as network sockets, database handles, user interaction windows, and file and device descriptors.

History:

Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp.

Basic principle of Garbage Collection:

The basic principles of garbage collection are:

- Find data objects in a program that cannot be accessed in the future
- Reclaim the resources used by those objects

It is kind of interesting to know how the objects without reference are found. Java normally finds all the objects that have reference and then regards rest of the objects are reference less – which is in fact a very smart way of finding the unreferenced java objects.

Types of Java Garbage Collectors/Garbage Collection Algorithms:
On J2SE 5.0 and above, one can normally find the following types of Java Garbage collectors that the programmers can normally choose to do a garbage collection through JVM Parameters.

The Serial Collector:

- JVM Option Parameter: -XX:+UseSerialGC

The Throughput Collector or The Parallel Collector:

- JVM Option Parameter: -XX:+UseParallelGC
- Young Generation GC done in parallel threads
- Tenured Generation GC done in serial threads.

Parallel Old Generation Collector:

- JVM Option Parameter: -XX:+UseParallelOldGC
- Certain phases of an 'Old Generation' collection can be performed in parallel, speeding up a old generation collection.

The Concurrent Low Pause Collector:

- JVM Option Parameter -Xincgc or -XX:+UseConcMarkSweepGC
- The concurrent collector is used to collect the tenured generation and does most of the collection concurrently with the execution of the application. The application is paused for short periods during the collection.
- A parallel version of the young generation copying collector is used with the concurrent collector.
- The concurrent low pause collector is used if the option -XX:+UseConcMarkSweepGC is passed on the command line.
- -XX:+UseConcMarkSweepGC -XX:+UseParNewGC
  - Selects the Concurrent Mark Sweep collector.
  - This collector may deliver better response time properties for the application (i.e., low application pause time).
  - It is a parallel and mostly-concurrent collector and and can be a good match for the threading ability of an large multi-processor systems.

The incremental (sometimes called train) low pause collector:
JVM Option Parameter: -XX:+UseTrainGC
This collector has not changed since the J2SE Platform version 1.4.2 and is currently not under active development.
It will not be supported in future releases.

Note that -XX:+UseParallelGC should not be used with -XX:+UseConcMarkSweepGC .

The argument parsing in the J2SE Platform starting with version 1.4.2 should only allow legal combination of command line options for garbage collectors, but earlier releases may not detect all illegal combination and the results for illegal combination are unpredictable.

Benefits of Garbage Collection:
Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of bugs are eliminated or substantially reduced:

- Dangling pointer bugs, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is then used. By then the memory may have been re-assigned to another use, with unpredictable results.
- Double free bugs, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again.
- Certain kinds of memory leaks, in which a program fails to free memory occupied by objects that will not be used again, leading, over time, to memory exhaustion.

Disadvantages of Garbage Collection:

- Consumes computing resources in deciding what memory is to be freed, reconstructing facts that may have been known to the programmer often leading to decreased or uneven performance.
- Interaction with memory hierarchy effects can make this overhead intolerable in circumstances that are hard to predict or to detect in routine testing.

- The moment when the garbage is actually collected can be unpredictable, resulting in stalls scattered throughout a session.
- Memory may leak despite the presence of a garbage collector, if references to unused objects are not themselves manually disposed of. This is described as a logical memory leak.The belief that garbage collection eliminates all leaks leads many programmers not to guard against creating such leaks.
- In virtual memory environments, it can be difficult for the garbage collector to notice when collection is needed, resulting in large amounts of accumulated garbage, a long, disruptive collection phase, and other programs' data swapped out.
- Garbage collectors often exhibit poor locality (interacting badly with cache and virtual memory systems), occupy more address space than the program actually uses at any one time, and touch otherwise idle pages.
- Garbage collectors may cause thrashing, in which a program spends more time copying data between various grades of storage than performing useful work.