Date: / / /

12. Concurrency.

* Thread :

→ A thread is a lightweight process & the smallest unit of processing.

→ A thread exists within a process & single ~~thr~~ process have multiple thread.

→ When multiple threads are running concurrently is called multithreading, which is similar to multitasking.

→ In multithreading sys^m, multi^l threads [sub processes] run at the same time.

Example :

- Downloading an audio while playing it at the same time.

Date: / /

- Browsing something, while scrolling web page & downloading something, printing a page in background etc.

→ In Multithreading, thread use a shared memory area, it means separate memory area does not required. The thread saves memory.

* Multithreading in Java :

→ In single-threaded appli^n, only one thread can be executed at a time..

→ Multithreading is a feature of Java that allows concurrent exce^n of 2 or more threads.

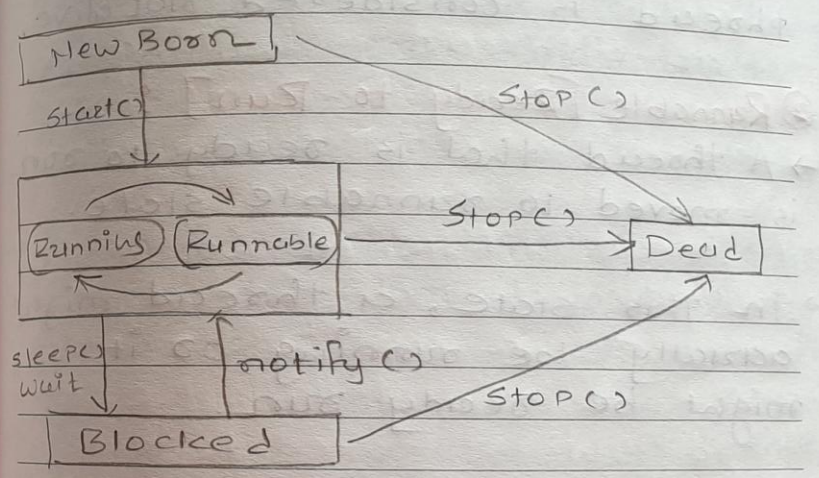
→ Threads can be exe^d by 2 methods in Java :

Date: / /

iff = interface.

① Extending the thread class.
② Imple^g the Runnable i/f.

* Life cycle of Java Thread :



① New :

→ A new thread begins its life cycle in the new state. It remains in this state until prog^m starts the thread. It is also referred as a born thread.

Date: / / /

→ After the creation of thread instance the thread is in this state, but b4 the start() method invocation. At this point, the thread is considered not alive.

② Runnable [Ready-to-Run] :

→ A thread that is ready to run is moved to runnable state.

→ In this state, a thread might actually be running or it might be ready run.

→ The thread 1st enters in runnable state after invoking of start(), but thread can return to this state after either running, waiting, sleeping or coming back from blocked state.

Date: / / /

→ In multi-threaded prog^m, each & every thread runs for a short while & then pauses, so other threads can get a chance to run because of a limited amount of time allocated to each individual thread.

③ Running State :

→ A thread is in running state that means the thread is currently executing.

→ There are several ways to enter into Runnable state, but there is only one way to enter in Running state, the thread scheduler will select a thread from the runnable pool to run.

④ Blocked :

→ When a thread is temporarily inactive, then it might be in blocked or waiting state.

→ When a thread is waiting for I/O resources to complete the task, it lies in the blocked state

→ A thread will lie in waiting state when it waits for another thread on a condi^n. When the condi^n will be fulfilled, the waiting thread will be moved to runnable state.

⑤ Dead State [Terminate] :

→ Thread terminates, either exist normally or there are some unusual erroneous events are occuring.

Date: / / /

* Thread creation & execution in Java :

→ In multithreading, thread can run concurrently, either asynchronously or synchronously

→ Thread can be imple^d in the form of obj^{ts} that contain a method called run ().

```
Public void run ()
{
    // stmnt to perf$^m$ action for thread
}
```

→ There are 2 ways to create thread in Java :

① Imple^t the Runnable i/f. [Java.lang. Runnable].

② By extending the thread class [Java.lang. Thread].

① Imple^{n} the Runnable I/f :

→ The runnable i/f should be imple^{d} by any class whose instances are created to execute the thread.

→ In this, run () contains the logic of the thread. The runnable i/f declares the run () that require to imple^{t} the thread.

→ The runnable i/f has only one method named run ().

Example:

```
public interface Runnable
{
    void run ();
}
```

→ The procedure for creating thread based on the Runnable i/f is as follows:

① Create a class that impl^{t} the Runnable i/f. An obj^{t} of this class is a Runnable obj^{t}.

② Imple^{t} the run () that will be executed by the thread.

③ Create thread using obj^{t} of Runnable class.

④ Call start () to run the thread.
- start () : a new thread will start.
- Thread will move from new state to the runnable state.
- When the thread gets a chance to execute, its target run () will run.

Date: / / /

Example:

```
class RunnableThread implements
Runnable
{
  public void run()
  {
     for(int i=0; i<=5; i++)
     {
        System.out.println("Thread:"
        +i);
     }
     System.out.println("End of
     Thread");
  }
}
public class RunnableExample
{
  public static void main(String[]
  args)
  {
     RunnableThread t = new
     RunnableThread();
```

Date: / / /

```
     Thread thread1 = new
     Thread(t);
     thread1.start();
     System.out.println("End of
     Main Thread");
  }
}
```

Output:

End of Main Thread
Thread: 1
Thread: 2
Thread: 3
Thread: 4
Thread: 5
End of Thread.

② Extending the thread class:

→ The procedure for creating threads based on extending the thread class is as follows:

```
class ABC extends Thread
{
  . . .
}
```

Date: / /

① Declare a class extending the thread class.

② Implet the run() for executing the sequence of code that the thread will execute.

③ Create a thread objt & call start() to initiate the thread execution.

Example :

```
class A extends Thread
{
    public void run()
    {
        for (int i=1 ; i<5 ; i++)
        {
            Sysm.out.println("Thread A: " + i);
        }
    }
}
```

Date: / /

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

Output :

Thread : 1
Thread : 2
Thread : 3
Thread : 4
Thread : 5

*Methods of Java Thread :

① Thread.sleep() : This static method ask the sysm to put the current thread to sleep for [approx.] specific time period.

→ A thread can be interrupted from its sleep.

② Interrupt ( ) : This is used to interrupt the corresponding thread if it is sleeping or waiting.

③ setPriority ( )/ getPriority( ) : It is used to get & set the priority of the given thread.

④ Join ( ) : This method is used to wait for the other thread to complete.

⑤ setName ( ) & getName ( ) : It is used to get & set name of a thread.

⑥ isAlive ( ) : Used to check thread is still running or not.

① start ( ) : Start a thread by calling its run method.

* Thread Priority :

→ Every java thread has a priority & it helps the OS to determine the order in which threads are scheduled & get executed.

→ Threads with higher priority are more important to a prog^m & should be allocated the processor by lower-priority threads.

→ Thread priority cannot guarantee the order in which thread execute. It would be platform dependent.

→ We can modify priority of thread using setPriority ( ) & getPriority ( ).

Date: / / /

→ Integer constant of thread class for priority:

- MIN_PRIORITY (0): Lowest Priority
- NORM_PRIORITY (5): Default Priority
- MAX_PRIORITY (10): Highest Priority

→ If 2 threads of the same priority are waiting for the CPU, the thread scheduler chooses one of them to run in round-robin fashion.

→ The chosen thread will run until one of the following condⁿ is true:

- A higher priority thread becomes Runnable. [Premptive Scheduling].
- Its run() method exist.
- On symⁿˢ that support time-slicing, its time allotment has expired.

Date: / / /

Example:

```
class A extends Thread
{
  public void run()
  {
    for (int i=1; i<5; i++)
    {
      Sysᵐ.out.println("Thread A:" +i);
    }
    Sysᵐ.out.Println("Exit from
    Thread A");
  }
}
Class B extends Thread
{
  public void run()
  {
    for (int j=1; j<5; j++)
    {
      Sysᵐ.out.Println("Thread
      B:" +j);
    }
```

```
        System.out.println("Exit from Thread B");
    }
}
public class ThreadPriority
{
    public static void main(String[] args)
    {
        A threadA = new A();
        B threadB = new B();
        threadA.setPriority(Thread.MIN_PRIORITY);
        threadB.setPriority(threadA.getPriority() +1);
        threadA.start();
        threadB.start();
    }
}
```

Output :

```
Thread B : 1
Thread B : 2
Thread B :
Thread B :
Thread B :
Exit from Thread B
Thread A : 1
Thread A : 2
Thread A : 3
Thread A : 4
Thread A : 5
Exit from Thread A.
```

* Thread Executor Framework :

0 Executor Framework :

→ The java.util.concurrent.Executors provide methods which are used to create ThreadPools of worker threads.

→ To use the Executor Framework we need to create one such thread pool & submit the task to it for execution.

→ The tasks submitted to the executor are executed by the n threads.

③ Cached Thread Pool :

ExecutorService executorService = Executors.newCachedThreadPool();

→ This pool is used mostly where there are lots of short-lived parallel task to be executed.

→ Unlike the fixed thread pool, the no. of threads of this executor pool is not bounded.

④ Scheduled Executor :

ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);

Date: / /

→ This executor is used when we have a task that needs to be run at regular intervals or if we wish to delay a certain task.

→ There are 2 method in scheduled executor are :

- AtFixedRate :

scheduledExecutorService.scheduleAtFixedRate(Runnable command, long initialDelay, long Period, TimeUnit unit)

- ~~Sch~~ scheduleWithFixedDelay :

scheduledExecutorService.scheduleWithFixedDelay(Runnable command, long period, TimeUnit unit).

* Thread Synchronization :

- Synchronization :

→ Synchronization is a way to ensure that the resource will be

Date: / / /

→ It is the job of the executor framework to schedule & execute the submitted task & return the results from the thread pool.

∘ Problems with thread creation:

→ Creating a new thread for a new task leads to overhead of thread creation & managing this thread life-cycle significantly increase the exec^n time.

→ Threads occupy memory & cause wastage of resources & the CPU starts to spend too much time switching contexts when each thread is swapped out & another thread comes in for exec^n.

→ ~~Ab~~ This factors reduce the throughput of the sys^m. Thread pool overcom this issue by keeping the threads.

Date: / / /

∘ Types of Executors:

① Single Thread Executor:

```
ExecutorService executorService =
Executors.newSingleThreadExecutor();
```

→ This thread pool exec^n has only a single thread.

→ If the thread dies due to an exception while exec^g a task, a new thread is created to replace the old thread & the subsequent tasks are executed in the new one.

② FixedThreadPool(n):

```
ExecutorService executorService =
Executors.newFixedThreadPool(4);
```

→ This is a thread pool of a fixed number of threads.

used by only one thread at a time.

→ It is the process of controlling the access of shared resources by the multiple thread in such a manner that only one thread can access a particular resource at a time.

○ Java Thread Synchronization:

→ There are 2 way to synchronize multithreaded appli^n:

① ~~Usi~~ Using Synchronized Method:

```
Synchronized void update()
{
    // synchronized code
}
```

→ The method that will read info^n from a file & the method that can update the file (same file) by declaring a method as Synchronized.

② Using Synchronized Statement:

```
Synchronized (lock-object)
{
    // Synchronized code
}
```

○ Inter-Thread Communication:

→ ITC [Inter-Thread Comm^n] is very impot where 2 or more threads exchange some info^n.

→ ITC is achieved using 3 methods:

○ wait(): It tells the calling thread to give up the monitor & to go to sleep until some other thread enters the same monitor & calls notify().

Date: / /

② notify () : Wakes up a thread that called wait () on the same obj^t.

③ notifyAll () : Wakes up all the threads that called wait () on the same obj^t. One of the thread will get access.

○ How ITC will be handled?

① The thread will enter to acquire the lock.

② The lock will be acquired by the thread.

③ The thread will go to waiting state if you call wait () method on the obj^t, else it will release the lock & exits.

Date: / /

④ If you call notify () or notifyAll () method, the thread will move to the notified state [runnable state].

⑤ The thread is available to acquire lock.

⑥ After compl^n of the task, thread will release the lock & exits the monitor state of the obj^t.

Example :

```
class Customer
{
    int amount = 10000;
    synchronized void withdraw
    (int amount)
    {
        Sys$^m$.out.Println ("going to
        withdraw");
        if(this.amount < amount)
```

```
{
        Sysm.out.Println("Less balance; waiting for deposit");
        wait();
    }
    this.amount = amount;
    Sysm.out.Println("Withdrow completed");
}
synchronized void deposit(int amount)
{
    Sysm.out.Println("going to deposit");
    this.amount += amount;
    Sysm.out.Println("deposit Completed");
    notify();
}
}
```

Class Demo

Date: / / /

```
{
    public static void main (String args[])
    {
        final Customer c = new Customer();
        new Thread()
        {
            public void run()
            {
                c.withdraw(15000);
            }
        }
        .start()
        new Thread()
        {
            public void run()
            {
                c.deposit(10000);
            }
        }
        .start();
}
```

Output :

going to withdraw
Less balance ; waiting for deposit
going to deposit
deposit completed
withdraw completed.