MultiThreading

MultiTasking:

Executing several tasks simultaneously is the concept of multitasking. There two types of multitasking

1. Processes based multitasking:
   Executing several tasks simultaneously, where each task is a separate independent program, is called processed based multitasking.
2. Thread based multitasking:
   Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread based multitasking and each independent part is called a Thread.

Whether it is processed based or thread based the main objective of multitasking is to reduce response time of the system and to improve performance.

The main important application areas of multithreading are

1. To develop multimedia graphics
2. To develop animations
3. To develop video games

Defining a Thread:

We can define a thread in the following two ways,

1. By extending Thread class:

```
Class MyThread extends Thread
{
        Public void run(){
                For(int i=0;i<10;i++){
                        Sop("child thread");
                }
        }
}

Whatever written inside run is called Job of Thread
Class ThreadDemo
{
        Psvm(String args[]){
                MyThread t=new MyThread();
                t.start();
                for(int i=0;i<10;i++){
                        sop("main method");
                }
        }
}
```

Case1 : Thread scheduler

- It is the part of JVM it is responsible to schedule Threads i.e. if multiple threads are waiting to the chance of execution then in which order threads will be executed is decided by thread scheduler.
- We cannot expect exact algorithm followed by thread scheduler. It is varied from JVM to JVM hence we cannot expect Thread execution order and exact output.
- Hence whenever situation comes to multithreading there is no guarantee for exact output but we can provide several possible outputs.

Case2: Difference between t.start() and t.run()

- In case of t.start() a new thread will be created which is responsible for execution of run() method, but in case of t.run() a new thread won't be created and run run() method will be executed just like a normal method call by main thread.
- Hence if we replace t.start() with t.run() then output can be predicated easily.

Case3: Importance of Thread class start method:

- It is responsible to register the thread with Thread scheduler and all other mandatory activaties hence without executing thread class start() method there is no chance of starting a new thread in java.
- Due to this Thread class start method is considered as heart of multithreading.

```
Start(){
    Register with scheduler
    Perform mandatory activities
    Invoke run method
```

Case4: Overloading of run() method

- Overloading of run() is always possible but Thread class start() method can invoke no-args run() method, the other overloaded method we have to call explicitly like a normal method call.

Case5: if we are not overrinding run() method

- If we are not overriding run() method then Thread class run() method will be executed which has empty implementation, hence we won't get any output.
- It is highly recommended to override run() method otherwise don't go for multithreading concept.

Case6: overriding of start() method

- If we override start() method then our start() method will be executed just like a normal method call and new Thread won't be created.
- It is not recommended to override Start() method, otherwise don't go for multithreading concept.

Case7: if overridding start() method contains super.start()

Executes as a multi-threaded program

Case8: Thread lifecycle

MyThread t=new MyThread();(new/born)→t.start()→(Ready/Runnable)→if T.S allocates processor→(Running)→if run() method completes→(Dead)

Case9:
After starting a thread if we are trying to restart the same thread then will get RE saying "IllegalThreadStateException".

2. By implementing Runnable Interface:

- We can define a Thread by implementing Runnable interface
- Runnable interface is present in java.lang package and contains only one method Run method.

```
Class MyRunnable implements Runnable{
        Public void run(){
                For(int i=0;i<10;i++){
                        Sop("child thread");
                }
        }
}
Class ThreadDemo
{
        Psvm(String args[]){
                MyRunnable r= new MyRunnable();
                Thread t= new Thread(r); // r is called target runnable
                t.start();
                for(int i=0;i<10;i++){
                        sop("main");
                }
        }
}
```

We will get mixed output and we cannot tell exact output.

Case study:

```
MyRunnable r=new MyRunnable();
Thread t1=new Thread();
Thread t2=new Thread(r);
```

Case1: t1.start()
A new thread will be created which is responsible for the execution of Thread class run method, which has empty implementation.

Case2:t1.run()
No new Thread will be created and Thread class run() method will be executed just like a normal method call.

Case3: t2.start()
A new thread will be created which is responsible for the execution of myRunnable class run method.

Case4: t2.run()

A new thread will not be created and MyRunnable run method will be executed just like a normal method call

Case5: r.start()

We will get compile time error saying MyRunnable class doesn't have start capability.
"cannot find symbol method start() location: class MyRunnable"

Case6: r.run()

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

Thread Class Constructors

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

Getting and Setting Name of a Thread:

Every thread in java has some name, it may be default name Generated by JVM or Customized name provided by programmer.
We can get and set name of a thread by using the following two methods of thread class
Public final string getName()
Public final void setName(String name);

Default name: Thread-0, Thread-1…

Thread Priorities:

- Every Thread in Java has some priority it may be default priority generated by JVM or customized priority provided by programmer.
- The valid range of Thread priorities is 1 to 10, where 1 is MIN_Priority and 10 is MAX_Priority.
- Thread class defines the following constants to represent some standard priorities
  Thread.MIN_PRIORIY→1
  Thread.NORM_PRIORITY→5
  Thread.MAX_PRIORITY→10
- Thread scheduler will use the priorities while allocating processor. The Thread having highest priority will get chance first.
- If two Threads have same priority then we cannot expect exact execution order, it depends on Thread scheduler
- Thread class defines the following methods to get and set priority of a Thread

Public final int getPriority()
Public final void setPriority(int p);
Allowed values range for p 1to 10, otherwise RE “IllegalArgumentExeption”

- The default priority only for the main thread is 5, but for all remaining threads default priority will be inherited from parent to child i.e. whatever priority parent Thread has the same priority will be there for child thread.
- Some platforms will not provide proper support for Thread priorities.

Preventing Thread from execution:

We can prevent a Thread Execution by using the following methods

1. Yield()
    - Yield() method causes to pause the current executing Thread to give the chance for waiting Threads of same priority. If there is no waiting thread or all waiting threads have low priority then same thread can continue its execution.
    - If multiple threads are waiting with same priority then which waiting thread will get the chance we cannot expect, it depends on thread scheduler.
    - The thread which is yielded, when it will get the chance once again it depends onThread scheduler and we cannot expect exactly.
    - Public static native void yield()
    - **Whenever yield() method is called the Thread moves from Running state to Read/Runnadble state.**

2. Join()
    - If a thread wants to wait until completing some other thread then we should go for join() method.
      Eg: if a Thread t1 wants to wait until thread t2 completes then ***t1 has to call t2.join().***
    - If t1 executes t2.join() then immediately t1 will be entered in to waiting state until t2 completes. Once t2 completes then t1 can continue its execution
    - Public final void join() throws InterruptedException
      Public final void join(long ms) throws InterruptedException
      Public final void join(long ms,int ns) throws InterruptedException
    - Every join method throws InterruptedException which is checked exception
    - Whenever join() method is called the thread enters a intermediate waiting state.
      It comes out of this waiting state if t2 execution is complete, waiting time expires or get interrupted.
      After coming out of waiting state the Thread enters Ready/Runnable state.

3. Sleep()
    - Public static native void sleep(long ms) throws InterrruptedException
    - Public static void sleep (long ms , int ns) throws InterruptedException
    - Every sleep method throws Interrupted Exception, which is checked Exception hence whenever we are using sleep method this Exception needs to be handled.
    - A running thread will go into sleeping state when a call is made to sleep method, from sleeping state it will enter Ready/Runnable whenever sleeptime expires or sleep is interrupted

- How a Thread can interrupt another Thread:
  A thread can interrupt a sleeping thread or waiting thread by using interrupt method of thread class.
  Public void interrupt()

  *Note: whenever we are calling interrupt method if the target thread is not in sleeping state or waiting state then there is no impact of interrupt call immediately. Interrupt call will wait until target thread entered into sleeping or waiting state. If the target thread entered in to sleeping or waiting state then immediately interrupt call will interrupt target thread.

Comparison:

property	Yield()	Join()	Sleep()
purpose	Pause execution	Wants to wait until completion of other thread	Don't want to perform activity for certain amount of time.
Overloaded	No	Yes	yes
Final	No	Yes	no
Throws IE	No	yes	yes
Native	Yes	No	Thread.sleep(long ms) →yes

Synchronization:

- Synchronized is the modifier applicable only for methods and blocks but not for classes and variables.
- If multiple threads are trying to operate simultaneously on the same java object then there may be a chance of data inconsistency problem.
- To overcome this problem we should go for synchronized keyword
- If a method or block is declared as synchronized then, at a time only one thread is allowed to execute that method or block on the given object, so that data inconsistency problem will be resolved.
- The main advantage of synchronized keyword is we can resolve data inconsistency problems. But the main disadvantage of synchronized keyword is it increases waiting time of threads and creates performance problems.
- Synchronization is implemented using lock. Every object in java has a unique lock whenever we are using synchronized keyword then only lock concept come into the picture.
- If a thread wants to execute synchronized method on the given object first it has to get lock of that object once thread got the lock then, it is allowed to execute any synchronized method on that object.
- Once method execution completes automatically thread releases the lock. Acquiring and releasing lock in internally taken care by JVM.
- While a thread executes synchronized method on the given object, the remaining threads are not allowed to execute any synchronized method simultaneously on the same object but remaining threads are allowed to execute non-synchronized methods simultaneously.

- Every class in java has a unique lock which nothing but class level lock.
  Thread needs a class level lock to execute static synchronized method
- While a thread executing static synchronized method the remaining threads are not allowed to execute static synchronized methods of that class simultaneously but ,remaining threads are allowed to execute following methods simultaneously
  - Norma static methods
  - Synchronized instance methods
  - Normal instance methods

Synchronized block:

- If very few lines of the code require synchronization then it is not recommended to declare entire method as synchronized, we have to enclose those few lines of the code by using synchronized block.
- The main advantage of synchronized block over synchronized method is it reduces waiting time of threads and improves performance of the system.
- We can declare synchronized block as follows:

To get lock of current object	To get lock of particular object	To get lock of class

Synchronized(this)
{
}
	synchronized(b)
	{
	}
		synchronized(Display.class)
		{
		}

If thread got lock of current Object then only it is allowed To execute this area	if a thread got lock of particular object b then only it is allowed to execute this area	if thread get lock of class then only it is allowed to execute this area.

- What Is synchronized keyword where we can apply
- Advantage of synchronized keyword→data inconsistency
- Disadvantage of synchronized keyword.
- What is race condition→ data inconsistency due multiple threads
- What is object lock and when is it required
- What is class level lock and when it is required
- What is the difference between class level and object level lock.
- While a thread is executing synchronized method on the given object , are remaining threads are allowed to execute any other synchronized method simultaneously on the same object→ no
- What is synchronized block
- How to declare synchronized block
- Advantage of synchronized block over synchronized method.
- Is a thread can acquire multiple locks simultaneously→yes
- What is synchronized statement→statements present inside synchronized method/block

Inter thread communication:

- Two threads can communicate with each other by using wait() notify() and notifyAll() methods.
- The Thread which is expecting update is responsible to call wait() method then immediately thread will enter in to waiting state.
- The Thread which is responsible to perform update, after performing update it is responsible to call notify() method then waiting thread will get that notification and continue its execution, with those updated items.
- Wait() notify() notifyAll() methods present in object class but not in thread class because thread can call these on any java objects.
- To call wait() notify() or notifyALL() methods on any object ,thread should be owner of that object i.e. the thread should have lock of that object i.e the thread should be inside synchronized area.
- Hence we can call wait () notify() and notifyALL() methods only from synchronized area otherwise we will get runtime exception saying "IllegalMonitorStateException"
- If thread calls wait() method on any object it immediately releases the lock of that particular object and enters into waiting state.
- If thread calls notify() method on any object it releases the lock of that object but nay not immediately. Except wait() notify() and notifyAll() no other method where thread releases the lock.
- Which of the following is valid
    1. If thread calls wait() immediately it will enter into waiting state, without releasing any lock.
       →invalid
    2. If a thread calls wait() method it releases the lock of that object but may not immediately.
       →invalid
    3. If a thread calls wait() method on any object it releases all locks acquired by that thread and immediately enter into waiting state.
       →invalid
    4. If a thread calls wait() method on any object it immediately releases the lock of that particular object and enter into waiting state
       →valid
    5. If a thread calls notify() method on any object it immediately releases the lock of that particular object.
       →invalid
    6. If a thread calls notify() method on any object it releases the lock of that object but may not immediately
       →valid

Public final void wait() throws InterruptedException
Public final native void wait(long ms) throws InterruptedException
Public final void wait(long ms,int ns)throws InterruptedException

Public final native void notify()
Public final native void notifyAll()

Note: every wait method throws interrupted exception which is checked exception hence whenever we are using wait() compulsory we should handle interrupted exception either by try catch or throws keyword otherwise will we get compile time error.

Calling wait() on Running thread puts it into waiting state, upon receiving notification the thread will enter another waiting state where it waits for acquiring lock.

Producer consumer problem:
Producer thread is responsible to produce items to the Queue and consumer thread is responsible to consume items from Queue. If queue is empty then consumer thread will call wait() method and enter into waiting state after producing items to the queue, producer thread is responsible to call notify() method then waiting consumer will get that notification and continue its execution with updated values.

Difference between notify() and notifyAll()
Notify() is used for sending notification to only one thread, notifyAll() is used for notifying multiple waiting threads.

Deadlock:

If two threads are waiting for each other forever, such type of infinite waiting is called deadlock.

Daemon Thread:

Threads which are executing in the background are called Daemon threads eg: Garbage collector, signal dispatcher, attached listener etc.
The main objective of Daemon threads is to provide support for non-Daemon threads(main thread)
Usually Daemon threads have low priority but based on our requirement it can run with high priority also
Public boolean isDaemon()
Public void setDaemon(boolean b) //should be before starting a thread

Green Thread:

Threads that are managed by JVM without taking underlying OS support are called Green Thread.

Stop():

We can stop a thread execution by using stop() method of thread class
Public void stop().
If we call stop() then immediately thread will enter into dead state anyway stop method is deprecated and not recommended to use.

Suspend() and Resum():

Suspend()→ suspended state