Multithreaded Programming

Multithreading fundamentals, The Thread Class and Runnable Interface, Creating Thread, Creating Multiple Threads, Determining When a Thread Ends, Thread Priorities, Synchronization, using Synchronization Methods, The Synchronized Statement, Thread Communication using notify(), wait() and notifyAll(), suspending, Resuming and stopping Threads.

⎞ Multithreading fundamentals

- ⎞ *Multitasking*: Running 2 or more programmes simultaneously at the same time.
- ⎞ *Multithreading*: dividing the programs into smaller chunks called threads and running them simultaneously.
- ⎞ *Thread*:is a lightweight process that executes some task.

Life Cycle of thread
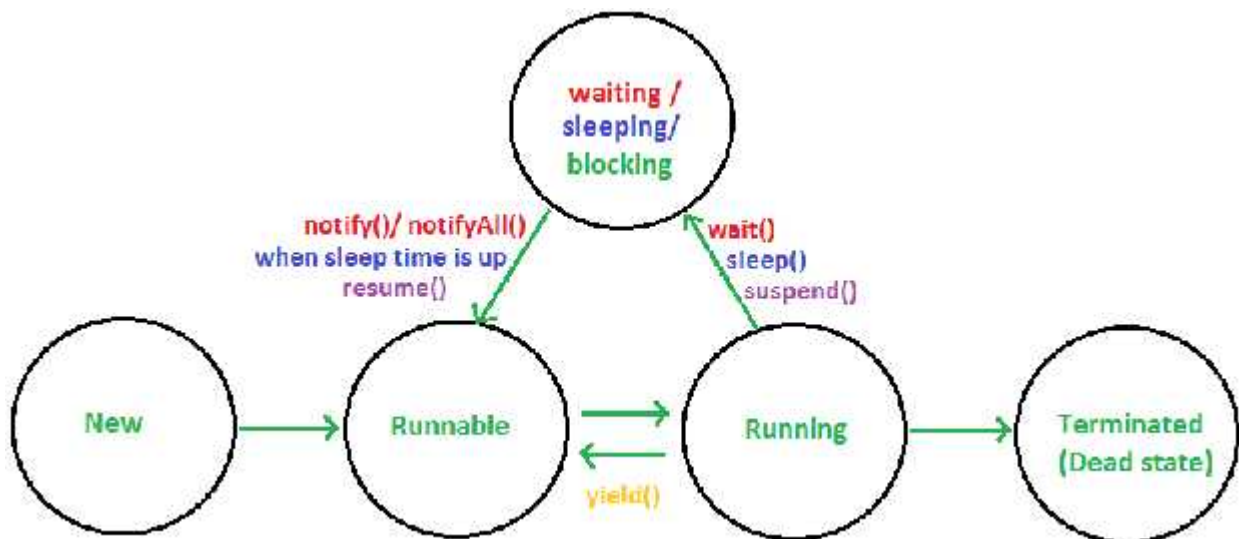
- ⎞ **new:** thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- ⎞ **Runnable:** A thread in this state is considered to be executing its task
- ⎞ **Running:**
- ⎞ **Waiting**: a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- ⎞ **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Advantages of Threading

- Threads are lightweight than processes, it takes less time, less resource to create a thread.
- Threads share their parent process data and code
- Context switching between threads is usually less expensive than between processes.
- Thread intercommunication is relatively easy than process communication
- Better resource utilization.
- Simpler program design in some situations.
- More responsive programs.

Thread class and Runnable Interface

- Multithreading is built on the Thread class and its companion interface Runnable.
- Both are packaged in java.lang.
- All processes have atleast on thread of execution, which is usually called the main thread.
  - **Thread can be created in two ways**
    - implementing Runnable interface
    - extending Thread class

- *implementing Runnable interface*
  - the Runnable interface signature

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

  - Steps to create thread by implementing **Runnable**
    - a class need to only implement a single method called **run( )**
    - thread can be created by any method (like main or constructor) with following steps
      1. Create object of class that implements run method
      2. Create thread by passing object current class that implements run method.
         Thread t = new Thread(Runnable obj)
      3. Call start( ) method which in turn calls run method and starts thread.

⌡ Example:

```
class MyThread implements Runnable {
        public void run()  {
                for(int i=0;i<40;i++)
                System.out.println(i);
            }
    }
public static void main(String[] args)  {
            // first, construct a MyThread object
            MyThread  ta = new MyThread();

            //next, construct a thread from that object.
            Thread t = new Thread(ta);

            //finally, start execution of the thread
            t1.start();
    }
```

⌡ *extending Thread class*

```
Class Test extends Thread
{
        public void run()
        {
                for(int i=0; i<40;i++)
                System.out.println(i);
                System.out.println("end of run");
        }
        Test t =new Test( );
        t.start();
}
```

Note:

⌡ Even though we don't create any thread, java treats the **main** method as the first thread called *main thread*.

⌡ It is the thread from which other "child" threads will be spawned. it must be the last thread to finish execution because it performs various shutdown actions.

⌡ Controlling main thread
  o it can be controlled through a Thread object. To do so, you must obtain a reference to iby calling the method currentThread( ), which is a public static member of Thread.

꜀ **General form**
  o **static Thread currentThread( )**
꜀ This method returns a reference to the thread in which it is called.
꜀ Once you have a reference to the main thread, you can control it just like any other thread.
꜀ **Example: Getting reference to main thread:**

```
Thread th = Thread.currentThread();
th.getName(); //gives thread name
```

```java
public class Demo {
 public static void main(String args[]) {
    Thread th = Thread.currentThread();
    System.out.println("Current thread: " + th.getName());

    th.setName("My Thread"); //change the name of the thread
    System.out.println("After name change: " + th);

    try {  for(int n = 4; n > 0; n--)
    {
     System.out.println(n);
     Thread.sleep(1000);
    }
    }
    catch (InterruptedException e) {
     System.out.println("Main thread interrupted");
    }
  }
}
```
  Output:
  Current thread: **main**
  After name change: Thread[My Thread,5,main]
  4
  3
  2
  1

꜀ **sleep() method**

```
static void sleep (long milliseconds) throws InterruptedException
```

꜀ causes the thread from which it is called to suspend execution for the specified period of milliseconds.

Choosing an approach

⟩ If you want to override many methods of Thread class then extend the thread class.
⟩ If you want to override only run method then better to implement runnable interface.

Creating Multiple Threads

```java
class myThread implements Runnable {

  String name; // name of thread

  public myThread(String threadName)
     {
          name = threadName;
     }

  public void run()
  {
  try
  {   for(int i = 5; i > 0; i--)
     {
      System.out.println( name + ": " + i);
      Thread.sleep(1000);
     }
  }
  catch (InterruptedException e)
  {
   System.out.println(name + "Interrupted");
  }
  System.out.println( name + " exiting.");
 }
}

public class MultiThreadDemo {

 public static void main(String args[]) {

    Thread  t1 = new Thread ( new myThread ("One") );
    Thread  t2 = new Thread ( new myThread ("Two") );

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

Output

⎰ all child threads execute concurrently
⎰ Threads are started in order in which they are created, however this may not always be
the case. Java is free to schedule the execution of threads in its own way, because of
differences in timing or environment, output from the program may differ.

Determining When a Thread Ends

⎰ As main is a parent for all child process, we want main to finish last over all threads.
⎰ How to ensure the main finishes last. Java provides 2 methods for this

i). **isAlive( )** helps to determine whether a thread has finished.
ii). **join ( )** wait for a thread to finish.

Syntax for isAlive( ):

```
final boolean isAlive( )
```

returns **true** if the thread upon which it is called is still running otherwise **false**.

Syntax for join( ):

```
final void join( ) throws InterruptedException
```

This method waits until the thread on which it is called terminates.

```java
class MyThread implements Runnable {

  String name;

  public MyThread ( String threadName) {
  name = threadName;
 }

    public void run()
      {
      try { for ( int i = 3; i > 0; i--)
      {
          System.out.println ( name + ": " + i);
          Thread.sleep(1000);
      }
   }
    catch (InterruptedException e)
      {
        System.out.println ( name + "Interrupted");
      }
  System.out.println ( name + " exiting.");
 }
}

public class DemoJoin {
public static void main(String args[]) throws Exception {
      Thread t1 = new Thread(new ThreadDemo("myThread"));
      t1.start();
      System.out.println("myThread status = " + t1.isAlive());

      t1.join();   //main thread stops and waits for t1 thread to get terminated
      System.out.println ( "\n Main Thread Exiting..\n" );
   }
}
```

Output:

myThread status = true
myThread: 3
myThread: 2
myThread: 1
myThread exiting.

Main Thread Exiting..

Thread Priorities

⟩ All Java threads have a priority in the range 1-10.

⟩ JVM selects to run a highest priority thread

⟩ 3 constants defined in Thread class:

    o Thread.MAX_PRIORITY -10

    o Thread.MIN_PRIORITY -1

    o Thread.NORM_PRIORITY   (Default Priority) 5

⟩ In case two threads have the same priority a FIFO ordering is followed.

⟩ JVM uses a preemptive, priority based scheduling algorithm.

⟩ A thread with a higher priority than the thread currently running enters the Runnable

⟩ state. The lower priority thread is preempted and the higher priority thread is scheduled to run.

⟩ **Context switch:**

    o Context switch is the OS switch from one running thread to the next

⟩ **The rules to determine a context switch:**

1. A thread can voluntarily relinquish control.

    o This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

2. A thread can be preempted by a higher-priority thread

    o As soon as a higher-priority thread wants to run, it pre-empts the lower priority thread. This is called preemptive multitasking**.**

⟩ **Example of priority of a Thread:**

```
class TestMultiPriority extends Thread
 {
public void run( )
   {
       System.out.println("running thread name is:"+Thread.currentThread().getName());
         System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
   }
   public static void main(String args[ ] )
    {
       TestMultiPriority m1=new TestMultiPriority( );
       TestMultiPriority m2=new TestMultiPriority( );
             m1.setPriority(Thread.MIN_PRIORITY);
             m2.setPriority(Thread.MAX_PRIORITY);
       m1.start();
       m2.start();
   }
}
```

Output:

running thread name is:Thread-0
running thread priority is: 1
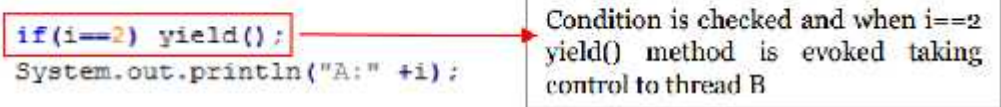running thread name is:Thread-1
running thread priority is: 10

Using yield().

⎰ Causes the currently running thread to yield() to any other threads with the same priority, waiting to be scheduled.

```java
class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            if(i==2) yield();
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
}
```

Condition is checked and when i==2 yield() method is evoked taking control to thread B

Synchronization

⎰ When more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

⎰ **Thread Synchronization**
  o There are two types of thread synchronization *mutual exclusive* and *inter-thread communication.*

⎰ **Mutual Exclusive**
  o Synchronized method.
  o Synchronized block.

⎰ **Cooperation or Inter-thread communication**
  o wait( ), notify( ), notifyAll( )

Synchronization Methods

⎰ If you declare any method as synchronized, it is known as synchronized method.

⎰ Synchronized method is used to lock an object for any shared resource.

⎰ Synchronization in java is the capability *to control the access of multiple threads to any shared resource.*

```
synchronized void print( ) {

        ……..//code

}
```

The Synchronized Statement

) Synchronized block can be used to perform synchronization on any specific resource of the method.

) Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

) Synchronized block is used to lock an object for any shared resource.

) Scope of synchronized block is smaller than the method.

) Synchronized block is used to lock an object for any shared resource.

) General form:

```
synchronized (object reference expression) {
                //code block
        }
```

Example:

```
class Bank
{
        int balance = 0;
        Bank() {  Bank.balance = 100;  }
        public void withdraw (int amount)
        {
                System.out.println (" withdraw Amount is: " + amount );
                synchronized(this)
                {
                        if ( balance - amount < 0 )
                        {
                                System.out.println ("Balance not enough.  Can't
                                withdraw money");
                        }
                        else
                        {
                                balance =  balance - amount;
                        }
                }
        }
}
```

Thread Communication using notify( ),wait( ) and notifyAll( )

⌡ **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

⌡ Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

⌡ Allowing synchronized threads to communicate with each other

⌡ Implemented by following methods of Object class.
   o wait( )
   o notify( )
   o notifyAll( )

⌡ **wait( )**
   o It causes current thread to wait until either another thread invokes
   o It must be called from synchronized context i.e. from block or method. It means before wait() method is called
   o *Syntax*:
      ▪ public final void wait( )throws InterruptedException
      ▪ public final void wait(long timeout)throws InterruptedException

⌡ **notify()**
   o Wakes up a single thread that is waiting on this object's monitor.
   o *Syntax*: public final void notify( )

⌡ **notifyAll()**
   o Wakes up all threads that are waiting on this object's monitor
   o *Syntax*: public final void notifyAll( )

⌡ **Lab Program – Producer and Consumer**

suspending, Resuming and stopping Threads

⌡ a program used **suspend( )** and **resume( )**, and **stop()** which are methods defined by **Thread.**
⌡ **They have the form shown below:**
   o **final void suspend ( )**
   o **final void resume ( )**
   o **final void stop ( )**

Method	Description
suspend()	This method puts a thread in suspended state and *can be resumed* using *resume()* method.
resume()	This method resumes a thread which was suspended using suspend() method.
stop()	This method stops a thread completely. Once a thread has been stopped, it cannot be restarted using *resume( )*.

Program with Synchronization methods and Synchronization blocks

```java
class Table{
      synchronized void printTable(int n){

              for(int i=1;i<=5;i++){
                  System.out.println(n*i);
              try{
                   Thread.sleep(400);
                  }catch(Exception e){System.out.println(e);}
              }

          }
}
class MyThread1 extends Thread{
      Table t;
      MyThread1(Table t){
              this.t=t;
      }
      public void run(){
              t.printTable(5);
      }

}
class MyThread2 extends Thread{
      Table t;
      MyThread2(Table t){
              this.t=t;
      }
      public void run(){
              t.printTable(100);
      }
}

class TestSynchronization1{
      public static void main(String args[]){
      Table obj = new Table();//only one object
      MyThread1 t1=new MyThread1(obj);
      MyThread2 t2=new MyThread2(obj);
      t1.start();
      t2.start();
}
```

```
  }

  // Synchronized Block

class Table{
      void printTable(int n){
         synchronized(this){  //synchronized block
            for(int i=1;i<=5;i++){
                System.out.println(n*i);
                try{
                        Thread.sleep(400);
                }catch(Exception e){System.out.println(e);}
            }
          }
      }
}
```