MODULE IV
ADVANCED FEATURES OF JAVA

Syllabus

- Multithreaded Programming - The Java Thread Model, The Main Thread, Creating Thread, Creating Multiple Threads, Synchronization, Suspending, Resuming and Stopping Threads.
- Event handling - Event Handling Mechanisms, Delegation Event Model, Event Classes, Sources of Events, Event Listener Interfaces, Using the Delegation Model.

THREAD

- JAVA is a multi-threaded programming language which means we can develop multi-threaded program using Java.
- A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
- Each part of such program is called a thread. So, threads are lightweight processes within a process.
- It is a separate path of execution.
- They are independent, if there occurs exception in one thread, it doesn't affect other threads.

- **Two types of multitasking:**
  - **Process-based(Multiprocessing)**
    - A ***process*** is a program that is executing.
    - Process-based multitasking allows our computer to run two or more programs concurrently
    - Example: can use a Java compiler and a text editor simultaneously.
  - **Thread-based(Multithreading)**
    - A single program can perform two or more tasks simultaneously
    - Example : A text editor can format text at the same time that it is printing
- Thus 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..

- **Advantages of multithreaded programs :**
  - Threads are lightweight
  - They share the same address space
  - Interthread communication is inexpensive
  - Context switching from one thread to the next is low cost.
- **Disadvantages of multitasking processes :**
  - More overhead than Multitasking threads.
  - Processes are heavyweight tasks
  - Processes requires their own separate address spaces.
  - Interprocess communication is expensive and limited.
  - Context switching from one process to another is also costly.

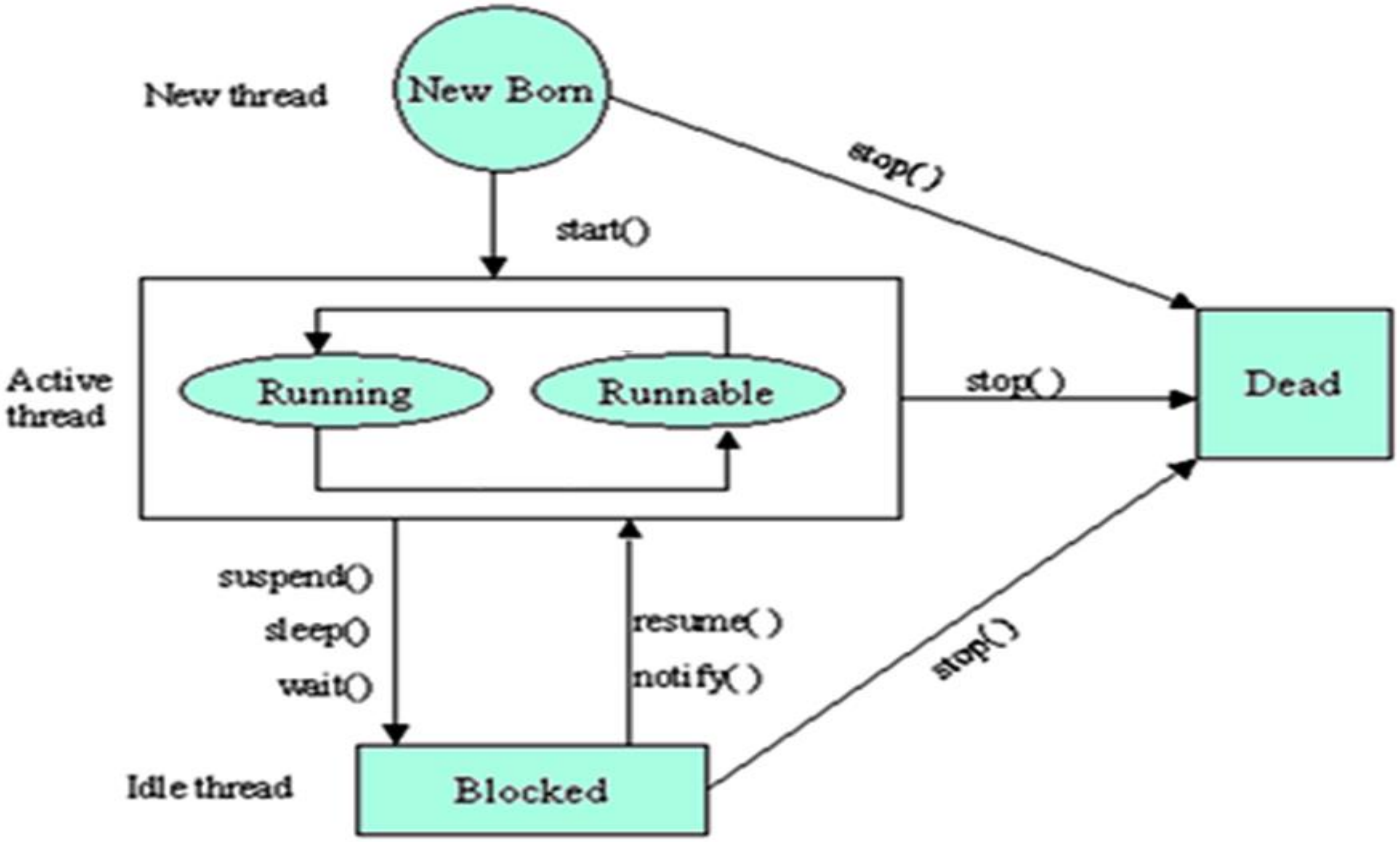
- Java supports **multithreaded multitasking**.
- Every thread in Java is created and controlled by the **java.lang.Thread** class

LIFE CYCLE OF THREAD

A thread can be in one of the **five states**.

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated

Thread Model (or Thread Life Cycle)



- **Newborn state :**
  - When we create a thread object, the thread is born and is said to be in newborn state.
  - This thread is not yet scheduled for running.
  - If we attempt to use any other method at this state, an exception will be thrown.
  - The following things can be done at this state:
    - Schedule it for running using **start()** : it moves to the runnable state
    - Kill it using **stop()**

- **Runnable State**
  - The thread is ready for execution and is waiting for the availability of the processor.
  - The thread has joined the queue of threads that are waiting for execution.
  - If all threads have equal priority, then they are given time slots for execution in round robin fashion.
  - The thread that gives up control joins the queue at the end and again waits for its turn.

- <u>**Running State**</u>
  - The processor has given its time to the thread for its execution.
  - The thread runs until it gives up control on its own or it is preempted by a higher priority thread.
  - A running thread relinquish its control in one of the following situations
  - It has been suspended using **suspend().** A suspend thread can be revived by using the **resume().**
  - We can put a thread to sleep for a specified time period using the method **sleep(time)** where time is in milliseconds. The thread re-enters the runnable state as soon as this time period is elapsed.
  - It has been told to wait until some event occurs. This is done using the **wait().** The thread can be scheduled to run again using the **notify().**

- **Blocked State**
  - A thread is prevented from entering into the runnable state and subsequently the running state
  - This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements.

- **Dead State**
  - Natural death : A running thread ends its life when it has completed executing its run().
  - Premature death : We can kill a thread by sending the **stop()** message to it at any state.
    - A thread can be killed as soon as it is born, or while it is running, or even when it is in blocked condition.
  - At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

CREATING THREAD

- There are two ways to create a thread:
  - By extending Thread class
  - By implementing Runnable interface.

Extending Thread class:

- Thread class provide constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Thread Methods - Following is the list of important methods available in the 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 void join() : waits for a thread to die.
- 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 Thread currentThread() : returns the reference of currently executing thread.
- public int getId() : returns the id of the thread.
- public Thread.State getState() : returns the state of the thread.
- public boolean isAlive() : tests if the thread is alive.
- public void suspend() : is used to suspend the thread(depricated).
- public void resume() : is used to resume the suspended thread
- public void stop() : is used to stop the thread(depricated).
- public boolean isDaemon() : tests if the thread is a daemon thread.

Thread.start() & Thread.run()

- In Java’s multi-threading concept, start() and run() are the two most important methods.

- When a program calls the start() method, a new thread is created and then the run() method is executed.

- But if we directly call the run() method then no new thread will be created and run() method will be executed as a normal method call on the current calling thread itself and no multi-threading will take place.

➢ Let us understand it with an example:

```
class MyThread extends Thread {
    public void run()
    {
        System.out.println("Current thread name: "
                        + Thread.currentThread().getName());
        System.out.println("run() method called");
    }   }
class  Sample {
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    } }
```

Output:

```
Current thread name:Thread-0
 run() method called
```

start ()

- when we call the start() method of our thread class instance, a new thread is created with default name Thread-0 and then run() method is called and everything inside it is executed on the newly created thread.

run ()

- when we call the run() method of our MyThread class, no new thread is created and the run() method executes on the current thread i.e. main thread. Hence, no multi-threading take place. The run() method is called as a normal function call.

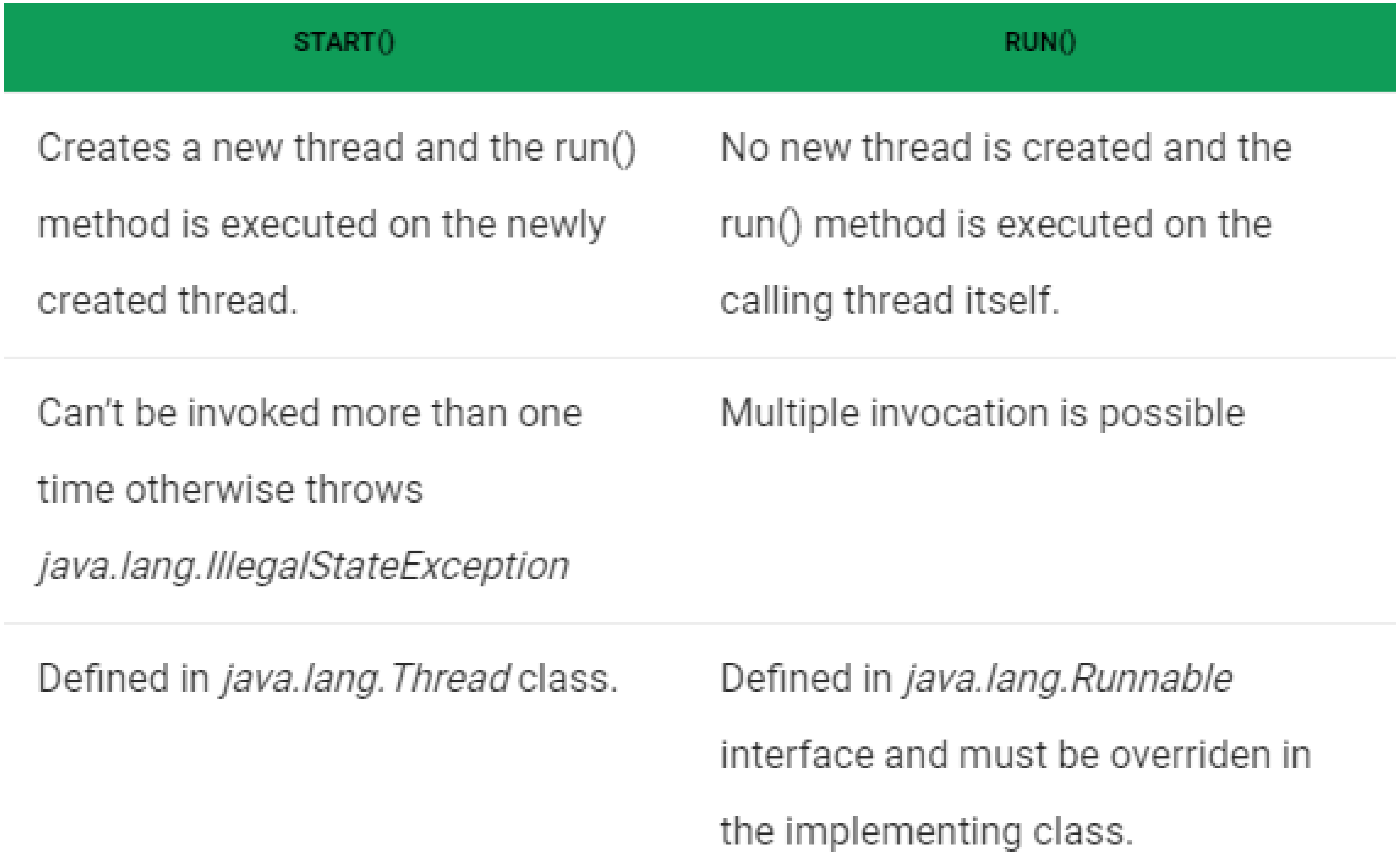
- **Let us try to call run() method directly instead of start() method**

```
class MyThread extends Thread {
   public void run()    {
      System.out.println("Current thread name: "
                    +Thread.currentThread().getName());
      System.out.println("run() method called");
   }   }
class Sample1 {
   public static void main(String[] args)  {
      MyThread t = new MyThread();
      t.run();
   } }
```

Output

```
Current thread name: main
 run() method called
```

start() Vs run()



START()	RUN()
Creates a new thread and the run() method is executed on the newly created thread.	No new thread is created and the run() method is executed on the calling thread itself.
Can't be invoked more than one time otherwise throws *java.lang.IllegalStateException*	Multiple invocation is possible
Defined in *java.lang.Thread* class.	Defined in *java.lang.Runnable* interface and must be overriden in the implementing class.

Implementing Runnable interface:

- The **Runnable interface** should be implemented by any class whose instances are intended to be executed by a thread.
- **Runnable interface** have only one method named **run()**.

  public void run(): is used to perform action for a thread.
- Steps to create a new Thread using Runnable :
  - Create a Runnable implementer and implement run() method.
  - Instantiate Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instance.
  - Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start(), creates a new Thread which executes the code written in run().

Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output: thread is running...

MAIN THREAD

- Every java program has a main method. The main method is the entry point to execute the program.
- So, when the JVM starts the execution of a program, it creates a thread to run it and that thread is known as the main thread.
- Each program must contain at least one thread whether we are creating any thread or not.
- The JVM provides a default thread in each program.
- A program can't run without a thread, so it requires at least one thread, and that thread is known as the main thread.

- If you ever tried to run a Java program with compilation errors you would have seen the mentioning of main thread. Here is a simple Java program that tries to call the non-existent getValue() method.

Example

```
public class TestThread {
public static void main(String[] args) {
TestThread t = new TestThread();
 t.getValue();
} }
```

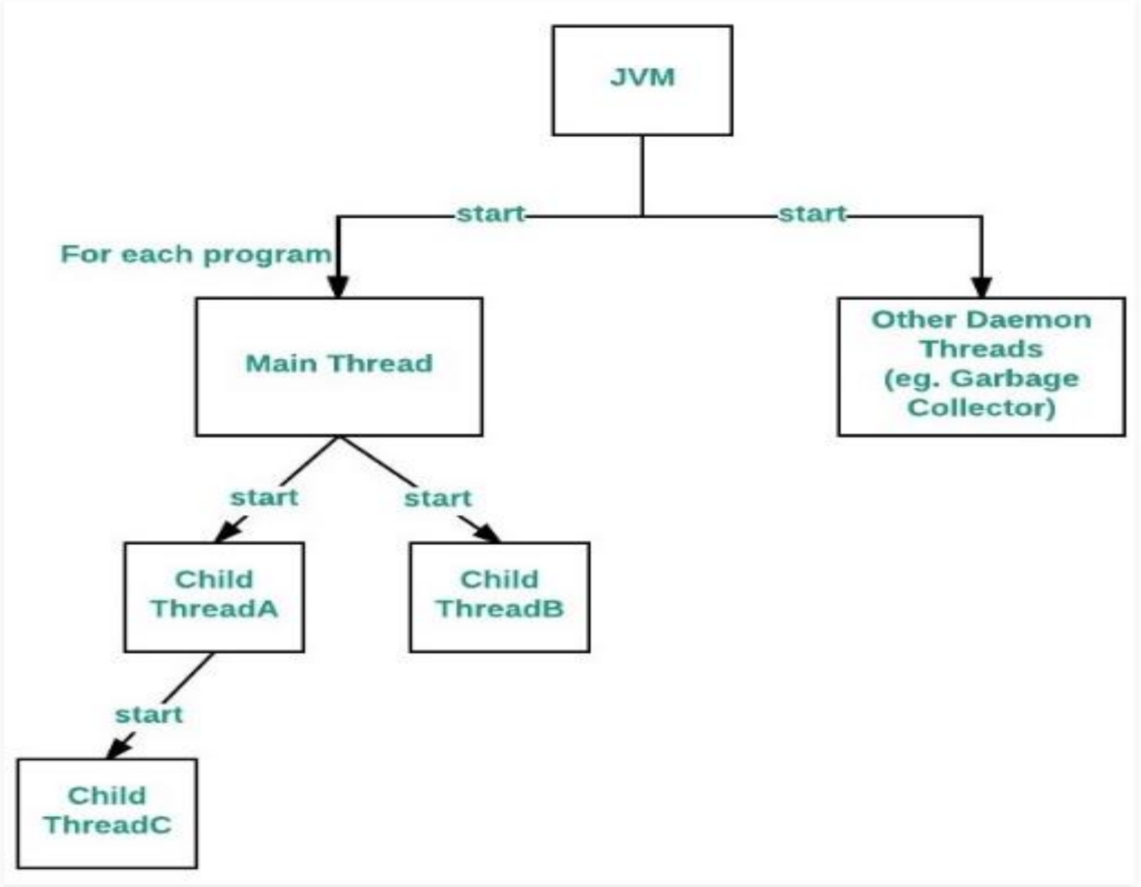
Output

```
Exception in thread "main" java.lang.Error: Unresolved compilation
problem: The method getValue() is undefined for the type TestThread
```

- As you can see in the error when the program is executed, main thread starts running and that has encountered a compilation problem.

Properties of Main thread:

- 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.
- **Flow diagram**



How to control Main thread

- The main thread is created automatically when our program is started.
- To control it we must obtain a reference to it.
- This can be done by calling the method currentThread( ) which is present in Thread class.
- This method returns a reference to the thread on which it is called.
- The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

```
class CurrentThreadDemo
{
   public static void main(String args[])
   {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        t.setName("S3 CS");
        System.out.println("After name change: " + t);
         System.out.println("This thread prints first 10
                                                          numbers");
        try
       {
               for(int n = 1; n<=10; n++)
               {              System.out.print(n);
                              Thread.sleep(1000);
               }
       }
      catch (InterruptedException e)
      {   System.out.println("Main thread interrupted");
       }
  }   }
```

Output :

```
Current thread:
Thread[main,5,main]
After name change: Thread[S3 CS
                              ,5,main]
This thread prints first 10 numbers
12345678910
```

Here
[main,5,main] :

- The 1st main is the name of the thread
- 5 is the default priority
- 2nd main is the name of the group of threads to which this thread belongs.

Program Explanation

- The program first creates a Thread object called 't' and assigns the reference of current thread (main thread) to it. So now main thread can be accessed via Thread object 't'.
- This is done with the help of currentThread() method of Thread class which return a reference to the current running thread.
- The Thread object 't' is then printed as a result of which you see the output Current Thread : Thread [main,5,main].
- The first value in the square brackets of this output indicates the name of the thread, second value is the priority of the thread and the third value is the name of the group to which the thread belongs.

- The program then prints the name of the thread with the help of getName() method.
- The name of the thread is changed with the help of setName() method.
- The thread and thread name is then again printed.
- Then the thread performs the operation of printing first 10 numbers.
- When you run the program you will see that the system wait for sometime after printing each number.
- This is caused by the statement Thread.sleep (1000).

Thread Priorities

- Each thread is assigned a priority.
- Thread priorities are integer values.
- Thread priority determines how that thread should be treated with respect to the others.
- A thread's priority is used for *context switch.*
- Rules for context switch :
  - A thread can voluntarily relinquish control
    - All other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
  - A thread can be preempted by a higher-priority thread.
    - As soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

- Higher-priority threads get more CPU time than lower priority threads. Threads of equal priority should get equal access to the CPU.

- **setPriority( ) :** To set a thread's priority, which is a member of **Thread**. General form: final void setPriority(int *level*)

- There are three constants defined in Thread class associated with priority.

  1. public static int MIN_PRIORITY
  2. public static int NORM_PRIORITY
  3. public static int MAX_PRIORITY

- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

- **getPriority( )** : obtain the current priority of a Thread
  - General Form : final int getPriority( )

```java
public class PriorityEg 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) {
        PriorityEg m1=new PriorityEg();
        PriorityEg m2=new PriorityEg();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m2.start();
        m1.start();
    }

}
```

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

Eg: Creating Multiple Threads using extends Thread

```
class NewThread extends Thread
{     NewThread()
      {
          System.out.println("Child thread: " + this);
          start();
      }
      public void run() {
         try  {
              for(int i = 5; i > 0; i--)
              {
              System.out.println("Child Thread: " + i);
              Thread.sleep(500);
               }
              }
              catch (InterruptedException e)
              {
              System.out.println("Child interrupted.");
              }
              System.out.println("Exiting child
                                                  thread.");
           }
      }
```

```
class ExtendThread
{       public static void main(String args[])
        {   NewThread obj1=new NewThread();
            try
            {  for(int i = 5; i > 0; i--)
               {
                   System.out.println("Main  Thread:"  +
        i);
                   Thread.sleep(1000);
               }
             }
             catch (InterruptedException e)
             {
                  System.out.println("Main        Thread
        interrupted");
               }
             System.out.println("Main             Thread
        exiting.");
        }
}
```

Output :

```
Child thread: Thread[Thread-0,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Eg: Creating Multiple Threads using Runnable interface

```
class NewThread implements Runnable
{       String name;
        Thread t;
        NewThread(String threadname)
        {       name = threadname;
                t = new Thread(this, name);
                System.out.println("New thread: " + t);
                t.start();
        }
        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.");
        }   }
```

```
class MultiThreadDemo
{       public static void main(String args[])
        {     NewThread a=new NewThread("One");
              NewThread b= new NewThread("Two");
             NewThread c= new NewThread("Three");
              try
              {       Thread.sleep(10000);
              }
              catch (InterruptedException e)
              {
              System.out.println("Mainthread
                                        Interrupted");
              }
              System.out.println("Mainthread
              exiting.");
        }
}
```

Output :

```
New thread:Thread[One,5,main]
New thread:Thread[Two,5,main]
New thread:Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

THREAD SYNCHRONIZATION

- When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues.
- For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time.

- Synchronization can be achieved in two ways
  - Using synchronized method
  - Using synchronized block

- Understanding the problem without Synchronization
  - In the following example, we are not using synchronization and creating multiple threads that are accessing display method and produce the random output.

```
class First{
 public void display(String msg)  {
  System.out.print ("["+msg);
  try
  {
    Thread.sleep(1000);
   }
  catch(InterruptedException e){
   e.printStackTrace();
   }
  System.out.println ("]");
 }      }
class Second extends Thread{
 String msg;
 First fobj;
  Second (First fp,String str)
  {
  fobj = fp;
  msg = str;
  start();
   }
public void run()
         {
    fobj.display(msg);
          }
  }
public class Syncro
{
 public static void main (String[] args)
   {
    First fnew = new First();
    Second ss = new Second(fnew, "welcome");
  Second ss1= new Second(fnew,"new");
Second ss2 =new Second(fnew,"programmer”);
    }
  }
```

Output

```
[welcome [ new [ programmer]
]
]
```

Synchronized Keyword

- To synchronize above program, we must synchronize access to the shared display() method, making it available to only one thread at a time. This is done by using keyword synchronized with display() method.
- With a synchronized method, the lock is obtained for the duration of the entire method.
- So if you want to lock the whole object, use a synchronized method.

  synchronized void display (String msg)
- In the following Example we will see the implementation of **synchronized** method

```
class First{
 synchronized public void display(String
                              msg)  {
   System.out.print ("["+msg);
   try {
     Thread.sleep(1000);
     }
   catch(InterruptedException e){
     e.printStackTrace();
     }
   System.out.println ("]");
 }       } //close class First
class Second extends Thread{
 String msg;
 First fobj;
  Second (First fp,String str)
  {
   fobj = fp;
   msg = str;
   start();
   }
```

```
public void run()
  {
      fobj.display(msg);
   }
  } //close class Second
public class Syncro
{
 public static void main (String[] args)    {
  First fnew = new First();
 Second ss = new Second(fnew, "welcome");
 Second ss1= new Second(fnew,"new");
 Second ss2 =new Second(fnew."
                        Second(fnew,"programmer");
    }
  }
```

Output

[welcome]

[new]

[programmer]

Using Synchronized block

- If we want to synchronize access to an object of a class or only a part of a method to be synchronized then we can use synchronized block for it.
- It is capable to make any part of the object and method synchronized.
- With synchronized blocks we can specify exactly when the lock is needed. If you want to keep other parts of the object accessible to other threads, use synchronized block.
- Following is the general form of the synchronized statement
- **Syntax**

```
        synchronized(object identifier) {
                // Access shared variables and other shared resources
                                                  }
```

Example

- In this example, we are using synchronized block that will make the display method available for single thread at a time.

```
class First{
 public void display(String   msg)  {
  System.out.print ("["+msg);
  try {
    Thread.sleep(1000);
    }
  catch(InterruptedException e){
    e.printStackTrace();
    }
  System.out.println ("]");
 }      }
class Second extends Thread{
 String msg;
 First fobj;
  Second (First fp,String str)
  {
   fobj = fp;
   msg = str;
   start();
   }
```

```
public void run()
  {
       synchronized(fobj) //Synchronized block
        {
         fobj.display(msg);
         }
     }
}//close class Second
public class Syncro
{
 public static void main (String[] args)  {
  First fnew = new First();
  Second ss = new Second(fnew, "welcome");
  Second ss1= new Second(fnew,"new");
   Second ss2 =new
   Second(fnew,"programmer”);
    }
  }
```

Output

```
[welcome]
[ new]
[ programmer]
```

Which is more preferred - Synchronized method or Synchronized block?

- In Java, synchronized keyword causes a performance cost.
- A synchronized method in Java is very slow and can degrade performance.
- So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

Thread suspend() method

- The suspend() method of thread class puts the thread from running to waiting state.
- This method is used if you want to stop the thread execution and start it again when a certain event occurs.
- This method allows a thread to temporarily cease execution.
- The suspended thread can be resumed using the resume() method.
- **Syntax**

```
final void suspend()
```

```
public class JavaSuspendExp extends
Thread
 {
  public void run()
   {
    for(int i=1; i<5; i++)   {
     try
         {
      sleep(500);
     System.out.println(Thread
       .currentThread().getName());
         }
    catch(InterruptedException e)
      {
       System.out.println(e);
      }
        System.out.println(i);
      }
   }
```

```
public static void main(String args[])
{
       // creating three threads
 JavaSuspendExp t1=new JavaSuspendExp ();

JavaSuspendExp t2=new JavaSuspendExp ();

 JavaSuspendExp t3=new JavaSuspendExp ();

 // call run() method
    t1.start();
    t2.start();
 // suspend t2 thread
    t2.suspend();
 // call run() method
    t3.start();
   }
}
```

Output:

```
Thread-0
1
Thread-2
1
Thread-0
2
Thread-2
 2
Thread-0
3
Thread-2
3
Thread-0
4
Thread-2
4
```

Thread resume() method

- The resume() method of thread class is only used with suspend() method.
- This method is used to resume a thread which was suspended using suspend() method.
- This method allows the suspended thread to start again.
- **Syntax**

```
public final void resume()
public class JavaResumeExp extends
Thread
 {
   public void run()
   {
    for(int i=1; i<5; i++)  {
     try
      {
       sleep(500);
       System.out.println(Thread.
             currentThread().getName());
       }
      catch(InterruptedException e)
       {
        System.out.println(e);
       }
          System.out.println(i);
       }
   }
```

```
 public static void main(String args[])
 {
  // creating three threads
  JavaResumeExp t1=new JavaResumeExp ()
;
 JavaResumeExp t2=new JavaResumeExp ();

 JavaResumeExp t3=new JavaResumeExp ();
 // call run() method
   t1.start();
   t2.start();
// suspend t2 thread
   t2.suspend()
 // call run() method
    t3.start();
// resume t2 thread
  t2.resume();
    }
}
```

Output:

```
Thread-0
 1
Thread-2
 1
Thread-1
 1
Thread-0
 2
Thread-2
 2
Thread-1
 2
Thread-0
 3
Thread-2
 3
Thread-1
 3
```

```
 Thread-0
4
Thread-2
4
Thread-1
4
```

Thread stop() method

- The stop() method of thread class terminates the thread execution.
- Once a thread is stopped, it cannot be restarted by start() method.
- **Syntax**

```
public final void stop()
 public final void stop(Throwable obj)
public class JavaStopExp extends
                                        Thread
{
  public void run()
  {
   for(int i=1; i<5; i++)  {
     try
      {
       sleep(500);
       System.out.println(Thread.
           currentThread().getName());
       }
      catch(InterruptedException e)
     {
        System.out.println(e);
      }
         System.out.println(i);
      }
   }
```

```
public static void main(String args[])
    {
     // creating three threads
     JavaStopExp t1=new JavaStopExp ();
     JavaStopExp t2=new JavaStopExp ();
     JavaStopExp t3=new JavaStopExp ();
     // call run() method
     t1.start();
     t2.start();
     // stop t3 thread
    t3.stop();
  System.out.println("Thread t3 is stopped");


    }
}
```

Deadlock

- Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in a different order. A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, associated with the specified object.
- In order to avoid deadlock, one should ensure that when you acquire multiple locks, you always acquire the locks in the same order in all threads.