OOPS Through JAVA
Chapter - III
Threads

What is Multitasking in Java?

Multitasking is a process that allows the user to perform multiple tasks at a time.
There are two ways to enable multitasking in Java.
1. Process-based multitasking: The process of executing several independent applications simultaneously within the same time is called process-based multitasking. For example, a user can use a text editor, video player, web browser and USB drive at a time in a PC. It is used to improve system performance in the OS level.
2. Thread-based multitasking: The process of executing several independent tasks concurrently within a single program is known as thread based multitasking. It is used to complete the entire program within the less time. Hence thread based multitasking helps to improve performance in program level.

JAVA THREADS

What is a Thread in Java?

- A thread is a facility to allow multiple activities within a single process.
- A thread is an independent path of execution within a single process.
- A thread is a series of executable statements.
- A thread is referred as lightweight process.
- A thread is a sequential flow of control within a program. Every program has at least one thread that is called main thread or primary thread. Additional threads are created through the constructor of Thread class or by instantiating classes that extend the Thread class.
- Threads are used to perform several tasks concurrently.

Single-Threaded and Multi-Threaded Applications: - A process that is made up of only one thread is said to be single-threaded. A single-threaded application can perform only one task at a time. A process having more than one thread is said to be multithreaded. Multiple threads in a process run at the same time, perform different tasks, and interact with each other. Java has built-in support for threads. A major portion of the java architecture is multithreaded. In java programs, the most common use of a thread is to allow an applet to accept input from a user and at the same time, display animated information in another part of the screen. Any application that requires two things to be done at the same time is probably a great candidate for multithreading.

Why we use Threads?

- To perform background processing
- To increases the responsiveness of GUI applications
- To take advantage of multiprocessor systems
- To simplify program logic when there are multiple independent entities
- To Minimize the load on system resources. Threads impose minimal impact on system resources.

How to create thread

Java.lang package provides Thread class and Runnable interface to create customized (user defined) threads. So there are two ways to create a thread.

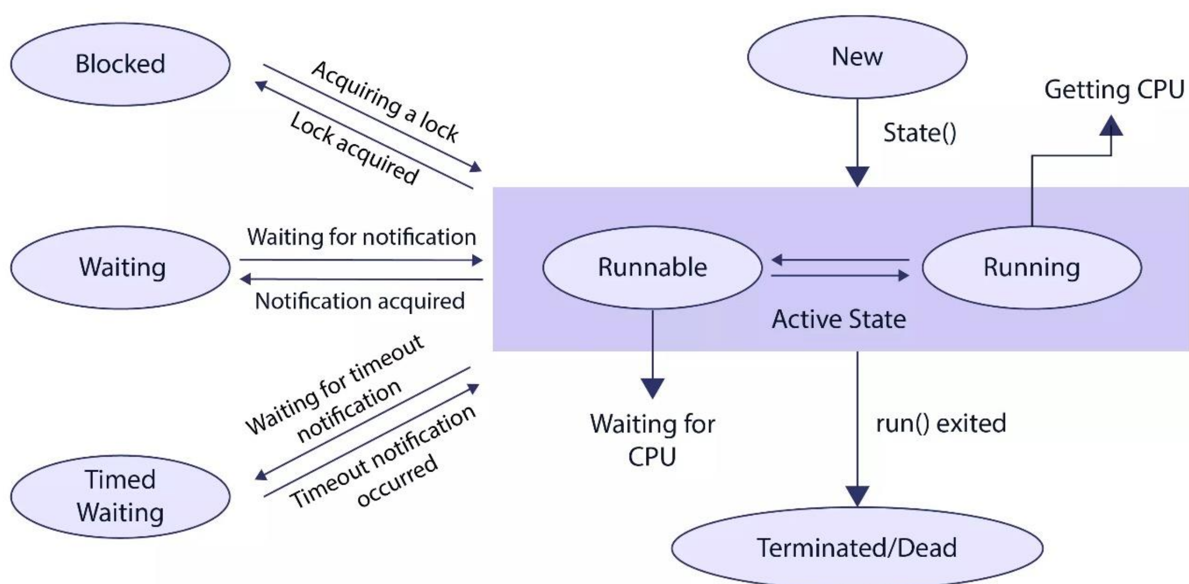
1. By extending Thread class
2. By implementing Runnable interface.

To write a multithreaded program we should inherit the properties of either Thread class or Runnable interface to our local class. Thread class provides several methods to perform multithreading in java. If you extend Thread class, then the following methods are accessible.

Methods available in Thread class: -

1. getName() : It is used to get name of a thread. The default names are Thread-0, Thread-1, Thread-2, …
2. setName() : It is used to assign a name to a thread.
3. getId() : It is used to get the ID of a thread. Thread ID is a unique number which was generated at the time of thread creation. The thread ID remains unchanged during its lifetime.
4. getPriority() : It is used to check the priority of a thread. The priority is a number between 1 to 10. Minimum priority is 1, Normal priority is 5 and Maximum priority is 10.
5. setPriority() : It is used to change the priority of a thread.
6. start() : It is used to start the execution of a thread.
7. run() : It is used to define the actual task which will be performed by the thread.
8. stop() : It is used to stop the execution of a thread permanently.
9. isAlive() : It returns true if the thread is alive. If thread completes its execution, then it returns false.
10. yield() : It is used change the state of current thread from running state to runnable state. It is static method in Thread class and it can stop the execution of current thread and give a chance to other waiting threads with the same priority. If there are no waiting threads or if all the waiting threads have low priority then the same thread will continue its execution.
11. sleep() : It is used to stop the execution of a thread temporarily for a short time which is specified in milliseconds. It is a static method. It causes runtime error. So we should use **Thread.sleep(number);** with in the try block with **Interrupted Exception**.
12. join() : It permits one thread to complete its execution entirely without any interrupt. All the remaining threads were being in the waiting state during the execution of join() thread.
13. suspend() : It is used to stop the execution of a thread until it will be resumed.
14. resume() : It used to restart the execution of a thread which has been suspended previously.
15. wait() : It is used to stop the execution of a thread until it will be notified.
16. notify() : It is used to restart the execution of a thread which has been stopped by the wait() method.
17. notifyAll() : It is used to wake up all threads that are waiting for a specific object's monitor.
18. currentThread() : It is used to get the reference of the of the currently executing thread object.

Explain Thread Lifecycle in Java



Life Cycle of a Thread

The thread goes through various stages in its lifecycle. For example, a thread is first born (created), then it gets started, and goes through various stages until it dies. This is nothing but lifecycle of a thread. Here is detailed explanation about thread life cycle.

New Born State: - Creating a new thread is known as its new born state.

Runnable State: - If we use start() method, the thread enters into the process queue and waiting for CPU. This state is called runnable state.

Running State or Active Stage: - When the thread scheduler allocates time slot, the threaded starts its execution. This state is known as running state.

Waiting State: - In this state the execution of a thread is temporarily stopped because of some methods like yield(), wait(), suspend(), etc. Whenever the thread got notification its execution will be restarted. Sometimes a thread changed into waiting state due to sleep() method. Then it is called Timed Waiting State. In this case, the thread automatically restarts its execution after expiring the waiting time.

Blocked state: In this state the execution of a thread is temporarily stopped because of some interrupts or lack of resources or waiting for the key of a specific object. When the system resources are available, the thread again starts its execution. Then it will be changed to running state.

Terminated State or Dead State: - If the execution of a thread completed entirely then it moved to terminated stated. Sometimes a thread forced to terminated state due to stop() method. Then the thread stops its execution immediately. A terminated thread means that it is dead and is no longer available for use.

Creating Threads in JAVA

I) Steps to create a thread using Thread class: -

1. Create a sub-class to Thread class.
2. Override **public void run()**method in the sub-class to define a task in the thread.
3. Create an object for the sub-class in the main() method. The object is considered as a thread.
4. Start the execution of run() method using the statement **threadobject.start();**

Example: -

Write java program to create a thread by extending the Thread class.

```
Import java.lang.Thread;
class MyThread extends Thread
{
  public void run()
  {
    System.out.println("Welcome to java");
  }
}
class ThreadDemo
{
  public static void main(String args[])
  {
     MyThread t1 = new MyThread();
     t1.start();
  }
}
```

Output: - Welcome to java

II) Steps to create a thread using Runnable interface: -

1. Create a sub-class to Runnable interface.
2. Override **public void run()** method in the sub-class to define a task in the thread.

3. Create an object for the sub-class in the main() method.
4. Also create an object of Thread class and submit the sub-class object to its target.
5. Finally execute the run() method by using **threadobject.start();**

Example: -

Write java program to create a thread by implementing Runnable Interface.

```
Import java.lang.Runable;
class Message implements Runnable
{
  public void run()
  {
    System.out.println("Welcome to java");
  }
}
class RunnableDemo
{
  public static void main(String args[])
  {
      Message x = new Message();
      Thread t1 = new Thread(x);
      t1.start();
  }
}
```

Output: - Welcome to java

Naming a thread:

The Thread class provides methods to change and get the name of a thread.

1. **public String getName():** It is used to return the name of a thread.
2. **public void setName(String name):** It is used to change the name of a thread.
3. **public int getId():** It returns the thread ID.

A program for naming a thread: -

```
Import java.lang.Thread;
class Mythread extends Thread
{
    public void run()
    {
     System.out.println("Welcone to Java");
    }
}
class NameTesting
{
    public static void main(String args[])
    {
      Mythread  t1 = new Mythread ();
      System.out.println("Name of thread = "+t1.getName());
      System.out.println("Thread ID = "+t1.getId());
      t1.start();
      t1.setName("Amrutha");
      System.out.println("New name = "+t1.getName());
    }
}
```

Output: -
Name of thread = Thread-0
Thread ID = 8
The Thread is Running...
New name = Amrutha

A program to perform single task several times by using multiple threads

```
Import java.lang.Thread;
class Mythread extends Thread
{
   public void run()
   {
     System.out.println("Welcome to Java");
   }
}
class  ThreadDemo
{
    public static void main(String args[])
   {
     MyThread t1 = new MyThread();
     MyThread t2 = new MyThread();
     MyThread t3 = new MyThread();
     t1.start();
     t2.start();
     t3.start();
   }
}
```

Output: -
Welcome to Java
Welcome to Java
Welcome to Java

Can we start a thread twice?

No. After starting a thread, it can never be started again. If you do so, an **IllegalThreadStateException** is thrown.
If you start a thread twice, the thread will run once but for second time, it will throw exception.
Let's understand this by using the following example.

```
class SimpleThread2 extends Thread
{
    public void run()
    {
     System.out.println("The Thread is Running...");
    }
    public static void main(String args[])
    {
     SimpleThread2 t1 = new SimpleThread2();
     t1.start();
     t1.start();
    }
}
```

Output: -

```
The Thread is Running...
Exception in thread "main" java.lang.IllegalThreadStateException
```

A program to perform single task several times by single thread

```
class SimpleThread3 implements Runnable
{
  public void run()
  {
    System.out.println("Welcome to java  world");
  }
  public static void main(String args[])
  {
      SimpleThread3  t1 = new SimpleThread3();
      t1.run();
      t1.run();
      t1.run();
  }
}
```

Which approach is recommended extends Thread or implements Runnable?

In the first approach, our class always extends Thread class. There is no chance of extending any other class. Hence we are missing Inheritance benefits. In the second approach, while implementing Runnable interface we can extends any other class. Hence we are able to use the benefits of Inheritance. Because of the above reasons, implementing Runnable interface approach is recommended than extending Thread class.

The significant differences between extending Thread class and implementing Runnable interface:

- When we extend Thread class, we can't extend any other class even we require and When we implement Runnable, we can save a space for our class to extend any other class in future or now.
- When we extend Thread class, each of our thread creates unique object and associate with it. When we implement Runnable, it shares the same object to multiple threads.

By using the following two programs we can clearly understand about extends Thread vs implements Runnable

```
import  java.lang.Runnable;
class implementsrunnable  implements Runnable
{
   private int  c=0;
   public void run()
   {
     c++;
     System.out.println("implementsrunnable  Counter = " + c);
   }
}
class  Test1
{
    public static void main(String[] args)
    {
     implementsrunnable  r = new  implementsrunnable  ();
     Thread t1= new Thread(r);
     t1.start();
     try
     {
```

```
        Thread.sleep(1000);
      }
      catch(Exception  e) {}
      Thread t2= new Thread(r);
      t2.start();
      try
      {
        Thread.sleep(1000);
      }
      catch(Exception  e) {}
      Thread t3= new Thread(r);
      t3.start();
    }
}
```

Output: -
implementsrunnable Counter = 1
implementsrunnable Counter = 2
implementsrunnable Counter = 3

```
import  java.lang.Thread;
class extendsthread  extends  Thread
{
   private int  c=0;
   public void run()
   {
      c++;
      System.out.println("extendsthread  Counter = " + c);
   }
}
class  Test2
{
    public static void main(String[] args)
    {
      extendsthread  t1= new extendsthread();
      t1.start();
      try
      {
        Thread.sleep(1000);
      }
      catch(Exception  e) {}
      extendsthread  t2= new extendsthread();
      t2.start();
      try
      {
        Thread.sleep(1000);
      }
      catch(Exception  e) {}
      extendsthread  t3= new extendsthread();
```

```
      t3.start();
   }
}
```

Output: -

extendsthread Counter = 1
extendsthread Counter = 1
extendsthread Counter = 1

Thread Synchronization

In java the keyword "synchronized" is used to perform synchronization. The synchronized keyword is used with methods and blocks. But not used with variables and classes.

Definition: - The process of allowing only one thread at a time to complete its task entirely is known as thread synchronization. It is used to avoid data inconsistency problem. But there is a major disadvantage with synchronization. It increases waiting time of threads and creates performance problems. Obliviously it provides poor or less performance. Hence if there is no specific requirement, it is not recommended to use synchronized keyword. **"java.util.concurrent"** package provides solution for synchronized keyword.

Internal Mechanism: - synchronization concept internally implemented by the JVM on the top of locking mechanism. If a thread wants to execute synchronized method on a given object, first it acquires lock of the object. Then only it is allowed to execute any synchronized method on that object. Once method execution is completed it releases the lock. Acquiring and releasing the lock is an activity automatically done by the JVM.

Main theme of Synchronization

While a thread executing synchronized method on a given object, the remaining threads are not allowed to execute any synchronized method on the same object. But the remaining threads are allowed to execute non-synchronized methods on the same object simultaneously.

Purpose of Synchronization

Synchronization is used to overcome the problem with Race condition which will be faced by non-synchronized methods. The synchronized keyword helps in writing concurrent parts of java applications. It also protects shared resources within the block.

Why we use Synchronization

Synchronization helps in preventing thread interference.
Synchronization helps to prevent concurrency problems.
Synchronization helps to handle racing of threads.

Explain the with a real time example? [5 Marks]

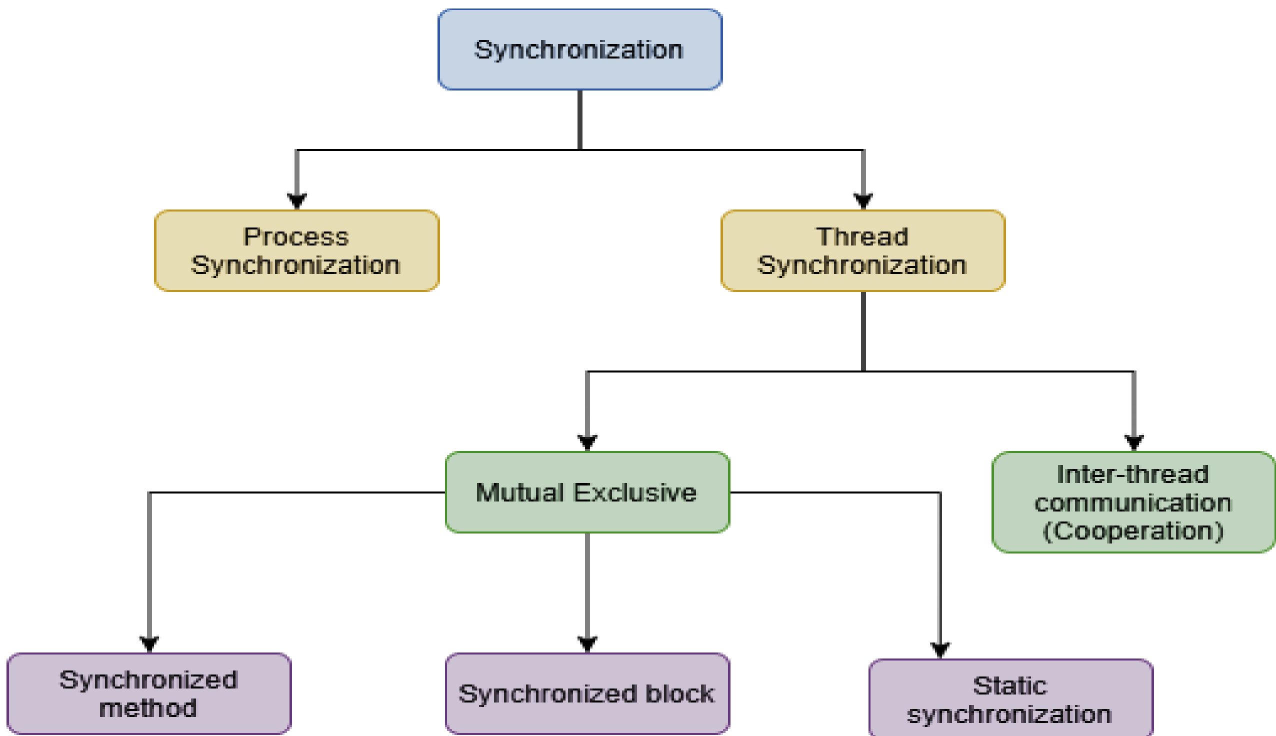
Let’s take a scenario of railway reservation system where two passengers are trying to book seats from Dhanbad to Delhi in Rajdhani Express. Both passengers are trying to book tickets from different locations.
Now suppose that both passengers start their reservation process at 11 am and observe that only two seats are available. First passenger books two seats and simultaneously the second passenger books one seat.

Since the available number of seats is only two but booked seats are three. This problem happened due to asynchronous access to the railway reservation system.

In this real-time scenario, both passengers can be considered as threads, and the reservation system can be considered as a single object, which is modified by these two threads asynchronously.

This asynchronous problem is known as **race condition** in which multiple threads access the same object and modify the state of object inconsistently.
The solution to this problem can be solved by a synchronization mechanism in which when one thread is accessing the state of object, another thread will wait to access the same object at a time until their come turn.

Classification of Synchronization



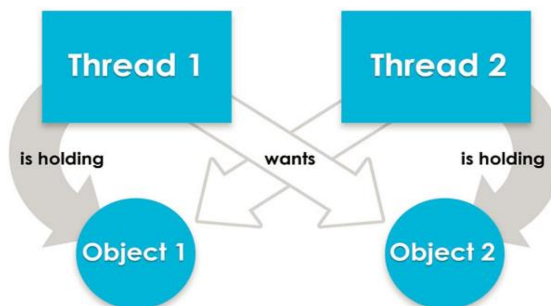
/* Write java program to implement thread synchronization in multi-threading */

```
import java.lang.*;
class College
{
  public synchronized  void classRoom(String fn)
   {
        for(int i=1 ; i<10 ; i++)
        {
           System.out.println(i + " class taken by " + fn);
           try
           {
              Thread.sleep(1000);
           }
           catch(InterruptedException  e){}
        }
        System.out.println(fn + " task completed\n");
   }
}
```

```
class MyThread extends  Thread
{
     College  c;
     String faculty;
     MyThread(College obj, String name)
     {
          c=obj;
          faculty=name;
     }
     public void run()
     {
        c.classRoom(faculty);
     }
}
class SyncDemo
{
   public static void main(String args[])
   {
       College  x = new College();
       MyThread t1 = new MyThread(x,"Babuji Sir");
       MyThread t2 = new MyThread(x,"Kumar Sir");
       MyThread t3 = new MyThread(x,"Pradeep Sir");
       t1.start();
       t2.start();
       t3.start();
   }
}
```

Deadlock in Multi-Threading

Definition: - If two threads are waiting for each other forever such type of situation is called deadlock.



Note: -

1. There is no resolution technique for deadlock but several prevention techniques are available in programming.
2. The synchronized keyword is the only cause for deadlock. Hence wherever we are using synchronized keyword we have to take very special care. Otherwise there is a possibility for deadlock.
3. If we remove at least one synchronized keyword, then we won't get deadlock.

/* Write java program to perform deadlocak activity */
/* Write java program to face deadlock situation */

```
import java.lang.*;
class A
```

```
{
   public synchronized void eamcet(B  b)
   {
       System.out.println("Thread-1 starts eamcet process using
a.eamcet()");
       try
       {
           Thread.sleep(1000);
       }
       catch(InterruptedException  e){}
       System.out.println("Thread-1 trying to announce results using
b.reasult()");
       b.result();
   }
   public synchronized void admission()
   {
       System.out.println("Admission Process Successfully Completed");
   }
}
class B
{
   public synchronized  void counselling(A  a)
   {
       System.out.println("Thread-2 starts counselling process using
b.counselling()");
       try
       {
          Thread.sleep(1000);
       }
       catch(InterruptedException  e){}
       System.out.println("Thread-2 trying to perform seat allotment
using a.admission()");
       a.admission();
   }
   public synchronized void result()
   {
       System.out.println("eamcet result announcement successfully
completed.");
   }
}
class DeadlockDemo implements Runnable
{
  A a = new A();
  B b = new B();

  DeadlockDemo()
  {
      Thread t1 = new Thread(this);
      t1.start();
      a.eamcet(b);
  }
  public void run()
  {
      b.counselling(a);
```

```
    }
    public static void main(String args[])
    {
        new  DeadlockDemo();
    }
}
```

What is Daemon thread, Explain?

Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically. In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in its life other than to serve user threads.

Example of Daemon Thread in Java:- Garbage collection in Java (gc), finalizer, etc.

Note:- We have two Methods for Daemon Thread as follows.

1. void setDaemon (boolean status):

This method marks the current thread as a daemon thread or user thread.
For example, if I have a user thread tU.
Then tU.setDaemon(true) would make it as a Daemon thread.
On the other hand, if I have a Daemon thread tD then
tD.setDaemon(false) would make it as a user thread.

Syntax:- public final void setDaemon (boolean on)

Parameters:

- **on:** If true, marks the thread as a daemon thread.
- **on:** If false, marks the thread as a user thread.

2. boolean isDaemon():

This method is used to check whether the current thread is a daemon thread or user thread. It returns true if the thread is Daemon. otherwise, it returns false.

```
/* Write Java program to demonstrate setDaemon() and isDaemon()
methods */
public class DaemonThread extends Thread
{
    public DaemonThread(String name)
    {
        super(name);
    }
    public void run()
    {
        if(Thread.currentThread().isDaemon())
        {
            System.out.println(getName() + " is Daemon thread");
        }
        else
        {
            System.out.println(getName() + " is User thread");
        }
```

```
    }
    public static void main(String[] args)
    {

        DaemonThread t1 = new DaemonThread("t1");
        DaemonThread t2 = new DaemonThread("t2");
        DaemonThread t3 = new DaemonThread("t3");

        t1.setDaemon(true);
        t1.start();
        t2.start();
        t3.setDaemon(true);
        t3.start();
    }
}
```

Output:

```
t1 is Daemon thread
t3 is Daemon thread
t2 is User thread
```

ThreadGroup in Java

A ThreadGroup is a collection of several individual threads. Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

In java a thread group is created by extending *java.lang.ThreadGroup* class.

A thread is allowed to access information about its own thread group.

A thread is not allowed to access the information about its parent's thread group or any other thread groups.

The method **ThreadGroup(String name)** is a constructor in **ThreadGroup** class creates a thread group with given name.

Methods of ThreadGroup Class:-

A list of methods present in the ThreadGroup class are explained below.

S.No	Method	Description
1)	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	activeGroupCount()	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	destroy()	This method destroys the thread group and all of its subgroups.
5)	isDestroyed()	This method tests if this thread group has been destroyed.
6)	getMaxPriority()	This method returns the maximum priority of the thread group.

7)	setMaxPriority(int pri)	This method sets the maximum priority of the group.
8)	getName()	This method returns the name of the thread group.
9)	getParent()	This method returns the parent of the thread group.
10)	interrupt()	This method interrupts all threads in the thread group.
11)	setDaemon(boolean daemon)	This method changes the daemon status of the thread group.
12)	isDaemon()	This method tests if the thread group is a daemon thread group.
13)	suspend()	This method is used to suspend all threads in the thread group.
14)	resume()	This method is used to resume all threads in the thread group which was suspended using suspend() method.
15)	stop()	This method is used to stop all threads in the thread group.

Benefits of ThreadGroup in java: -

ThreadGroup provides the following functionality.

1. Logical organization of your threads for diagnostic purposes.
2. You can interrupt() all the threads in the group.
3. You can set the maximum priority of the threads in the group.
4. Sets the ThreadGroup as a daemon so that all new threads added to it will be daemon threads.
5. It allows you to override its uncaughtExceptionHandler.
6. It provides you some extra tools such as getting the list of threads, how many active ones you have etc.

Case Study

Write java programs to observe the accessibility of class members with respect to the access specifiers.

Program1: - Creating a public class in a package.

```
/* within the class all members are accessible */
package com.aditya.mca.acet;
public class Abcd
{
  private int a;
  int b; //default
  protected int c;
  public int d;
  public Abcd()
  {
    a=10;
    b=20;
    c=30;
    d=40;
  }
  public void print()
  {
    System.out.println("a="+a+"\nb="+b+"\nc="+c+"\nd="+d);
```

```
  }
}
```

javac -d . Abcd.java

Program2: - Creating (Scinsp) sub class in same package.

```
/* private member is not accessible in a subclasses of same package */
package com.aditya.mca.acet;
import com.aditya.mca.acet.Abcd;
public class Scinsp extends Abcd
{
   public void printit()
   {
      System.out.println("a="+a); //error private member not accessible
      System.out.println("\nb="+b+"\nc="+c+"\nd="+d);
   }
}
```

javac -d . Scinsp.java
java com.aditya.mca.acet.Scinsp

Program3: - Creating (Scinop) sub class in other package.

```
/* private and default members are not accessible in a subclasses of
other package. */
package com.aditya.mca.acet.boys;
import com.aditya.mca.acet.Abcd;
public class Scinop extends Abcd
{
   public void printit()
   {
      System.out.println("a="+a); //error private member not accessible
      System.out.println("b="+b); //error default member not accessible
      System.out.println("\nc="+c+"\nd="+d);
   }
}
```

javac -d . Scinop.java
java com.aditya.mca.acet.boys.Scinop

Program4: - Creating (Ocinsp) Other class in same package.

```
/* private member is not accessible in other classes of same package */
package com.aditya.mca.acet;
import com.aditya.mca.acet.Abcd;
public class Ocinsp
{
   public void printit()
   {
     Abcd x = new Abcd();
     System.out.println("a="+x.a);//error private member not accessible
     System.out.println("b="+x.b+"\nc="+x.c+"\nd="+x.d);
   }
   public static void main(String args[])
   {
     Ocinsp k = new Ocinsp();
     k.printit();
   }
```

```
}
```

javac -d . Ocinsp.java
java com.aditya.mca.acet.Ocinsp

Program5: - Creating (Ocinsp) Other class in other package.

```
/* private, default and protected members are not accessible in other
classes of other package */
package com.aditya.mca.acet.boys;
import com.aditya.mca.acet.Abcd;
public class Ocinop
{
  public void printit()
  {
   Abcd x = new Abcd();
   System.out.println("a="+x.a);//error private member not accessible
   System.out.println("b="+x.b);//error default member not accessible
   System.out.println("c="+x.c);//error protected member not accessible
   System.out.println("d="+x.d);
  }
  public static void main(String args[])
  {
   Ocinop k = new Ocinop();
   k.printit();
  }
}
```

javac -d . Ocinop.java
java com.aditya.mca.acet.boys.Ocinop

User-defined Exception in Java

An exception is a run time error that occurs during the execution of a program. When an exception occurred the program gets terminated abnormally and, the code after the line that generated the exception never gets executed. Java provides us the facility to create our own error conditions by creating user defined exceptions. These are also known as customized exceptions. In order to create a custom exception, we need to extend the Exception class that belongs to java.lang package.

Steps to create a customized exception: -

1. Create a sub class to Exception class that acts as a user defined exception.
2. Create constructor and methods in that class which are needed for exception handling.
3. Use try, catch, throw, throws and finally keywords to perform exception handling.
4. Use **throws** keyword with the method signature in which there is a change to raise run time error.
5. Verify the error with if statement and invoke the exception explicitly using **throw** keyword.

Keyword	Purpose

try	The try statement allows you to define a block of code to be tested for errors while it is being executed.
catch	The catch block contains code that is executed when the exception handler is invoked it.
throws	The throws keyword is used to declare which exceptions can be thrown from a method
throw	The throw keyword is used to throw an exception explicitly when corresponding error occurs.
finally	The finally keyword is used to define a block of code whose execution is mandatory no matter if there is an exception or not.

Example1: - /* Create a user defined exception that will respond whenever the input value is zero */ Prepare This for Exam Purpse

```
import   java.io.*;
import   java.util.Scanner;
class ZeroInputException extends Exception
{
}
class Mysums
{
  void  sum() throws ZeroInputException
  {
     int a,b;
     Scanner sc = new Scanner(System.in);
     System.out.println("Enter two numbers ");
     a=sc.nextInt();
     b=sc.nextInt();
     if(a==0 || b==0)
            throw new ZeroInputException();
     System.out.println("Sum = " + (a+b));
  }
}
class Eclass1
{
  public static void main(String args[])
  {
     Mysums  x = new  Mysums();
     try
     {
        x.sum();
     }
     catch(ZeroInputException  e)
     {
        System.out.println("ZeroInputException");
        System.out.println("Please enter non-zero numbers");
     }
  }
}
```

Output: -
Enter two numbers
22
33

```
Sum = 55
Enter two numbers
9
0
ZeroInputException
Please enter non-zero numbers
```

Example2: - /* Create user defined exception to identify invalid hall ticket number and invalid marks in student class and respond accordingly. */

```
import java.lang.Exception;
class  IllegalValueException extends Exception
{
   void getMsg()
   {
     System.out.println("Error due to invalid hall ticket number.");
     System.out.println("Number should not be negative");
   }
}
class  InvalidMarksException extends Exception
{
   void getMsg()
   {
     System.out.println("Error due to invalid marls.");
     System.out.println("Marks Range is 0-100");
   }
}
class Student
{
   int htno,pm;
   String sn;
   public Student(int a, String b, int c)
   {
       htno=a;
       sn=b;
       pm=c;
   }
   void show() throws IllegalValueException
   {
       System.out.println("Student Name = " + sn);
       if (htno<=0)
       {
           throw new IllegalValueException();
       }
       System.out.println("HTNO = " + htno);
   }
   void showMe() throws InvalidMarksException
   {
       if (pm<0 || pm>100)
       {
           throw new InvalidMarksException ();
       }
       System.out.println("Percentage of Marks = " + pm);
```

```
    }
}
class Eclass2
{
   public static void main(String args[])
   {
       //Student s = new Student(1020,"SREEJA",181);
       Student s = new Student(-1021,"SRINIJA",95);
       try
       {
           s.show();
       }
       catch(IllegalValueException e)
       {
           e.getMsg();
       }
       try
       {
           s.showMe();
       }
       catch(InvalidMarksException  e)
       {
           e.getMsg();
       }
   }
}
```

Output: -
Student Name = SRINIJA
Error due to invalid hall ticket number.
Number should not be negative.
Percentage of Marks = 95

Example3: - /* Write java program to create user defined exceptions related to banking transactions */

```
import  java.io.*;
import  java.util.Scanner;
class  InsufficiantBalanceException  extends  Exception
{
}
class  InvalidDenominationException  extends  Exception
{
}
class  Bank
{
    int  acno;
    String  cn;
    double   bal;
    Bank(int  a, String  b, double  c)
    {
       acno=a;
       cn=b;
       bal=c;
       System.out.println("Account Created");
```

```
    }
    void showAccount()
    {
        System.out.println("Account Number = " + acno);
        System.out.println("Customer Name = " + cn);
        System.out.println("Balance Amount = " + bal);
    }
    void  deposite(double  a)  throws  InvalidDenominationException
    {
         if(a%100 !=0)
            throw  new InvalidDenominationException();
         bal=bal+a;
         System.out.println("Transaction Complete");
         System.out.println("Account Balance = " + bal);
    }
    void  withdraw(double  a)  throws  InsufficiantBalanceException
    {
         if(bal-a<1000)
            throw  new InsufficiantBalanceException();
         bal=bal-a;
         System.out.println("Transaction Complete");
         System.out.println("Account Balance = " + bal);
    }
    void checkBalance()
    {
         System.out.println("Balance Amount = " + bal);
    }
}
class   Eclass3
{
   public static void  main(String  args[])
   throws IOException, InterruptedException
   {
      Bank  obj = new Bank(2212,"Chinmayi",10000);
      Scanner  sc = new  Scanner(System.in);
      int  ch;
      double  amt;
      do
      {
         System.out.println("AXIS BANK LIMITED\n  Main Raod, KKD");
         System.out.println("-----------------");
         System.out.println("Account Number :  " + obj.acno);
         System.out.println("\nAvailable Operations:-");
         System.out.println("1. Deposite\n2. Withdraw\n3. Show");
         System.out.println("Account\n4. Check Balance\n5. Quit");
         System.out.println("Enter your choice");
         ch=sc.nextInt();
         switch(ch)
         {
             case  1:
                try
                {
                  System.out.println("Enter  Deposite Amount ");
```

```
                  amt=sc.nextDouble();
                  obj.deposite(amt);
                }
                catch(InvalidDenominationsException  e)
                {
                  System.out.println("InvalidDenominationException");
                  System.out.println("Only accepted denominations are
                  Rs.100/-, Rs.200/-, Rs.500/- and Rs.2000/-");
                }
                break;
             case  2:
                try
                {
                  System.out.println("Enter  Withdraw Amount ");
                  amt=sc.nextDouble();
                  obj.withdraw(amt);
                }
                catch(InsufficiantBalanceException  e)
                {
                 System.out.println("InsufficiantBalanceException");
                 System.out.println("\nMin Balance should be Rs.1000");
                }
                break;
             case  3:
                obj.showAccount();
                break;
             case  4:
                obj.checkBalance();
                break;
             case  5:
                System.out.println("Thank You ... Once visit again");
                break;
             default:
                 System.out.println("Wrong Choice ... Try Again!");
         }
         System.out.println("*** PRESS ANY KEY TO CONTINUE ***");
         System.in.read();           // works like getch() function.
new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
// works like clrscr() function.
     }
     while(ch!=5);
   }
}
```