Chapter - 16

Multithreading

What are the differences between program and process?

Program	**Process**
1. A program is set of optimized instructions. 2. Program always resides in secondary memory for a long time until we delete. 3. Based on the duration of the existence of a program is treated as static Object.	1. A program is under exception is known as process. 2. Process resides in main memory for limited span of time until execution complete. 3. Based on the duration of the existence of a program is treated as dynamic Object.

What is a process?

- A program under execution is called as process.
- Ex: opening an MS-word
- If multiple programs are running simultaneously, then it is called as multi-processing environment.
- Ex: in order to run a Java program we need to open the following programs:
    - Command prompt
    - Editor like notepad

What is a thread?

- Part of the program is called as Thread.
- Running multiple threads in a program is called as Multi-threading.
- Ex: while printing the document, we can do some modifications that won't reflect in print.
- One of the important features of Java that made the language popular is Multi-threading.

As real world developer in any language we may develop two types of applications. They are

1. Process based applications	**Thread based applications**
1. Process based applications always provides single flow of control. 2. All the applications of C,C++, Pascal, COBOL e.t.c., are comes under process based applications.	1. Thread based applications always provides and contains multiple flow of control. 2. All the applications of Java, .NET e.t.c., are comes under process based applications. 3. Context switch is less. 4. Because of less context switch

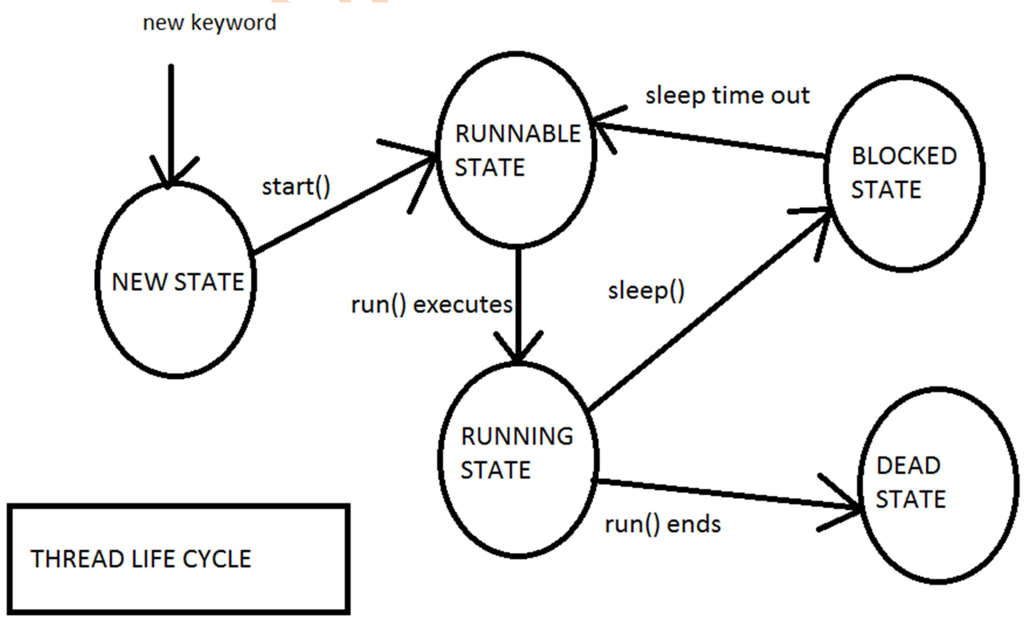
3. Context switch is more. 4. Because of more context switch process based applications takes more execution time. 5. All the process based applications are treated as heavy weight components. 6. Process based applications provide only sequential execution but not concurrent execution. 7. For each sub program in process based applications there exists a separate address space.	process based applications takes more execution time. 5. All the Thread based applications are treated as heavy light weight components. 6. Thread based applications provide both sequential and concurrent execution. 7. In Thread based applications irrespective of number of sub programs there exists a single address space.

Thred Life Cycle:

Whenever we write a multi threaded program there is possibility of existing many number of Threads. When multi threading program start executing, the threads of that program will undergo five states of Thread. They are

1. New State
2. Ready State or Runnable State
3. Running State
4. Waiting State or Sleep State
5. Halted State or Dead State

Thread State Chart Diagram:



New State: A new state is one in which Thread is created and it is about to enter in main memory.

Ready: A Ready state is one in which Thread is entered into the main memory. Memory space is created and first time waiting for CPU.

Running State: A Running State is one which the Thread is under the control of CPU in Java programming Threads of Java executing concurrently.

If Threads are executing hour by hour or minute by minute or second by second that type of Thread execution is known as sequential execution. If threads are executing millisecond by millisecond that type of Thread execution is known as Concurrent execution because no human being calculations may not be able to differentiable of milliseconds of time hence in Java programming all the Threads of our program executed by the CPU concurrently by following Round robin algorithm with the difference of milliseconds time.

Waiting state: A state of the Thread is said to be waiting state if and only if it satisfies the following factors.

a) Remaining **CPU burst time** (The amount of time required by the Thread from the CPU for its complete execution is known as CPU burst time).
b) Suspending the currently executing the Thread.
c) Making the currently executing Thread to sleep for a period of time in terms milliseconds.
d) Making the currently executing Thread to wait for a period of time in terms of milliseconds.
e) Making the currently executing Thread to wait without specifying the period of time.
f) Joining the currently executing Threads after completion of execution.

Halted State: A Halted State is one in which Thread has completed its execution in general as long as the Thread present in ready, running and waiting states whose execution state is true and these states are known as **in memory states.** As long as Threads are present in NEW and HALTED States whose execution status is false and these states are known as **out memory states.**

<u>In How many ways we can create a Thread?</u>

In Java, we can create a thread in two ways:

1. Extends Thread class
2. Implements Runnable interface

How to define a Thread class using Thread Class?

Step 1: extends from Thread class

Step 2: override public void run() method.

How to execute the thread which is extending thread class?

Step 1: create an object of the class which is defined in the above algorithm

Step 2: call start method using thread class object reference.

```
class MyThread extends Thread
{
        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println(i);
                        try
                        {
                                sleep(1000);
                        }
                        catch (InterruptedException e)
                        {
                                System.out.println(e);
                        }
                }
        }

}

class ExtendsThreadDemo
{
        public static void main(String[] args)
        {
                MyThread mt = new MyThread();
                mt.start();//--->internally calls run() method
        }
}
```

Implements Runnable Interface

How to define a Thread class using runnable interface?

Step 1: implements from Runnable interface

Step 2: override public void run() method.

How to execute the thread which is implementing runnable interface?

Step 1: create an object of the class who is implementing runnable interface class

Step 2: create an object of thread class and pass the above object as an argument.

Step 3: call start method using thread class object reference.

```
public class ServerBackup implements Runnable{

        public void run()
        {
                for(int i=0;i<5;i++)
                {
                        System.out.println("Taking server backup......");

                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println(e);
                        }
                        System.out.println("Back Up completed....");
                }
        }
}

public class ServerBackupDemo {

        public static void main(String[] args) {

                ServerBackup sb = new ServerBackup();
                Thread t = new Thread(sb);
                t.start();

        }

}
```

<u>Some of the methods defined in Thread class</u>

Public final void setName(String)

Public final String getName()

The above methods are used for setting the user friendly name to the Thread and obtaining same thing from the Thread.

EX: Thread t1=new Thread()

```
String tname=t1.getName();
System.out.println(tname);  // Thread 0
T1.setName("Java th");
Tname=t1.getName();
System.out.println(tname);//Java th
```

Public final void setPriority(int)

Public final int getPriority()

The above methods are used for setting and getting the priority of the Thread.

EX: Thread t1=new Thread();

```
Int pv=t.getPriority();
System.out.println(pv);
T1.setPriority(Thread.MAX_PRIORITY)
Pv=t1.getPriority();
System.out.println(pv);  //100→ MAX_PRIORITY
```

Public final void start():

This method is used for transferring the Thread from new state to ready state, provides internal services of multithreading and automatically calls run() which is containing the logic of the Thread.

Note: The internal service of the multithreading represents providing concurrency, synchronization, inter thread communication e.t.c.,

1. **Public void run():**

i. It is one of the **non final method** present in Thread class for providing logic of the Thread. Originally run () defined in Thread class with null body method.

ii. As a Java programmer to provide a logic of the Thread, run() must be overridden in the context of our derived class by extending java.lang.Thread class.

iii. This method will be called automatically by start () upon the Thread class Object we apply start (). That is run () can't be called by the Java programmer directly because that particular Thread never get the services of multithreading.

While we are overriding run() of the Thread class either we write block of statements which provides the logic of the Thread or we call user defined method which contains the logic of the Thread.

```
public class ToStringDemo {

    public static void main(String[] args) {
        ServerBackup sb = new ServerBackup();
        Thread t = new Thread(sb);

        System.out.println("Result is:" + t);

        /*

        Output: Thread[Thread-0,5,main]
        Thread-0: default name assigned to the thread by JVM
             to retrieve the thread Name ----> getName()
                  to set the new name ---> setName()

        5 : priority Number.
          every thread contains some number called priority Number
             that explains the execution sequence.

             0-4 ---> MIN_PRIORITY
              5 ---> NORM_PRIORITY
             6-10 ---> MAX_PRIORITY

             default priority of any thread is 5.
             to retrieve the priority ---> getPriority()
             to set the new priority ---> setPriority()

  main: Name of the thread group.

              ThreadGroup tg = new ThreadGroup();
              Thread t = new Thread(tg);
              t is the member of tg group.

              getThreadGroup().getName() used to retrieve the
              thread group name.

        */
        t.setName("Server backup Thread");
        t.setPriority(10);

        System.out.println("Result is" + t);
        System.out.println("Thread Name is" + t.getName());
        System.out.println("Thread Priority is:" + t.getPriority());
        System.out.println("Thread group Name is:" +
                                        t.getThreadGroup().getName());

    }

}
```

Public boolean isAlive(): This method determines the execution states of the Thread. This method returns true provided Thread is present in ready, running and waiting state. This method returns false when the Thread is in NEW and HALTED states.

```
class isAliveDemo
{
    public static void main(String[] args)
    {
        MyThread mt = new MyThread();
        System.out.println("Is Alive.....:" + mt.isAlive());
        mt.start();
        System.out.println("Is Alive.....:" + mt.isAlive());
        try
        {
            mt.join();
        }
        catch (InterruptedException e)
        {
            System.out.println(e);
        }

        System.out.println("Is Alive.....:" + mt.isAlive());

    }
```

}

public final void join() throws java.lang.InterruptedException:

This method is used for joining threads which are completed their execution as a single Thread. In another words this method joins the completed Thread group name collects all the combined joined Threads and handover to garbage Collector this approach improves the performance of multi Threading applications. The default environment of Java says as and when individual Threads are completed their execution individually collected Thread group name and handover to garbage collector which is not a recommended process.

Interrrupted exception occurs, interrupted exception never occurs in single/standalone applications but there is a possibility of occurring in Client/server environment. Lets assume n Threads are stated their execution and n-1 Threads are completed their execution and joined. Still nth Thread is executing due to mis memory management of serverside Operating System, Thread group name is **trying** collect n-1 joined Threads and handovering to garbage collector even through nth Thread is under execution. In this point of time JVM generates a predefined exception called java.lang.Interrupted Excetion with respective nth Thread. Interuppted Exception is one of the CheckedException.

```
public class JoinMethodDemo {

        public static void main(String[] args) {
                ServerBackup bkp = new ServerBackup();
                Thread t = new Thread(bkp);
                t.start();

                ServerBackup bkp1 = new ServerBackup();
                Thread t1 = new Thread(bkp1);
                t1.start();

                try
                {
                        t.join(3000);
                }
                catch(InterruptedException e)
                {
                        System.out.println(e);
                }
                System.out.println("Bye....Main Thread");

        }

}
```

How to create Multiple Threads?

```
public class NumberGenerator implements Runnable{

        Thread t;
        public NumberGenerator(String tName,int priority) {
                t = new Thread(this);
                t.setPriority(priority);
                t.setName(tName);
                t.start();
        }

        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println(t.getName() + "::" + i);

                        try
                        {
                                Thread.sleep(1000);
                        }
                catch(InterruptedException e)
                {
                        System.out.println(e);
                }
            }
        }

}

public class NumberGeneratorDemo {

        public static void main(String[] args) {
                NumberGenerator ng = new NumberGenerator("th1",10);
                NumberGenerator ng1 = new NumberGenerator("th2",5);
                NumberGenerator ng2 = new NumberGenerator("th3",1);

        }

}
```

Synchronization Techniques:

1. Synchronization is one of the distinct feature of multithreading to eliminate inconsistent results.
2. The concept of Synchronization must be always applied on common tasks for getting consistent results that is if multiple Threads are performing common task then those Threads generates inconsistent result. To avoid this inconsistent result it is recommended to apply Synchronization concept.

DEF: The mechanism of allowing only one Thread among multiple Threads into the area which is sharable/common to perform read and write operations known as Synchronization.

Need of Synchronization:

1. Let us assume variable balance whose initial value is 0. Let us assume there exists two types of Threads t1 and t2 want to update or deposit `10 and if `20 to the balance variable respectively.

2. Both Threads shared execution and completed if verify the value of the balance variable then it contains either 10 or 20. Which is one of the inconsistent result.
3. To avoid this inconsistent result we must apply the concept of **Synchronization.**

Synchronization concept applied:

1. Threads t1 and t2 started their execution (Assume t1 starts first and t2 starts later with the different of milliseconds of time). Thread t1 gets the value of balance(0) and balance variable value will be locked by JVM until t1 completes its execution.
2. Maintenance if Thread t2 trying to access the balance variable value then the JVM made the Thread t2 to wait until t1 completes its execution.
3. Once the Thread t1 completes its execution balance variable value(10) will be unlocked and given to t2 and once again balance variable value will be locked.
4. Thread t2 completed its execution, balance variable value(30) will be unlocked. Finally it we observer the value of the balance then it contains consistent result(30).
5. Hence during the Synchronization concept the process of locking and unlocking will be performed by the JVM until all Threads completed their execution.
6. Thread synchronization techniques are divided into two types they are

i. Synchronization Methods

ii. Synchronized blocks.

i. **Synchronized Methods:** If any ordinary method is accessed by more than one Thread then the ordinary method generates an inconsistent result. To avoid this inconsistent result the definition of ordinary method must be preceded by a keyword synchronized for making the method as Synchronized. Based on Synchronized keyword we write before the methods, they are classified into two types they are

a) Synchronized instance methods.

b) Synchronized static methods.

a) Synchronized instance methods: If an Ordinary instance method is accessed by multiple Threads then there is possibility of getting inconsistent result. To avoid this inconsistent result, the ordinary method must be made as synchronized by using synchronized keyword.

Syn:

```
 Synchronized Returntype MethodName(list of formal params if any)
{
Block of Statement(s);
}
```

If we make an ordinary instance method as synchronized then the **object** of corresponding class will be locked by JVM.

```
//package SynchronizedKeyword;


public class ATMCounter {
        /* synchronized keyword or synchronized method*/
        public  synchronized void withdrawl(String name)
        {
                System.out.println(name + "entered inside ATM Counter");
                System.out.println(name + "is counting money");
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.out.println(e);
                }
                System.out.println(name + "is leaving ATM Counter");
        }
}

/*

synchronized <<return-type>> <<method-name>>(Parameters)
{
        .
        .
}
*/
//package SynchronizedKeyword;
```

```
public class Customer implements Runnable{
        Thread t;
        ATMCounter atm;

        Customer(String nm,ATMCounter at)
        {
                t = new Thread(this,nm);
                atm = at;
                t.start();
        }
        public void run()
        {
                atm.withdrawl(t.getName());
        }
}




//package SynchronizedKeyword;

public class MainMethod1 {

        public static void main(String[] args) {
                ATMCounter atm = new ATMCounter();

                Customer c1 = new Customer("customer1",atm);
                Customer c2 = new Customer("customer2",atm);
                Customer c3 = new Customer("customer3",atm);
                Customer c4 = new Customer("customer4",atm);
        }
```

}

Synchronized static methods: If an ordinary static method accessed by multiple Threads then there is possibility of getting in consistent result. To avoid this inconsistent result the ordinary static method definition must be made as synchronized by using synchronized keyword.

Syn:

```
Synchronized static method name(list of formal params if any)
{
Block of statement(s);
}
```

If we make any ordinary static method as synchronized then corresponding class will be locked.

```
//package SynchronizedKeyword;


public class ATMCounter {
        /* synchronized keyword or synchronized method*/
        public static synchronized void withdrawl(String name)
        {
                System.out.println(name + "entered inside ATM Counter");
                System.out.println(name + "is counting money");
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                        System.out.println(e);
                }
                System.out.println(name + "is leaving ATM Counter");
        }
}
```

```
/*

synchronized  static <<return-type>> <<method-name>>(Parameters)
{
        .
        .
}
*/
//package SynchronizedKeyword;

public class Customer implements Runnable{
        Thread t;
        ATMCounter atm;

        Customer(String nm,ATMCounter at)
        {
                t = new Thread(this,nm);
                atm = at;
                t.start();
        }
        public void run()
        {
                atm.withdrawl(t.getName());
        }
}
//package SynchronizedKeyword;

public class MainMethod1 {

        public static void main(String[] args) {
                /* Class level locking demo */
                ATMCounter atm = new ATMCounter();
                ATMCounter atm1 = new ATMCounter();
                ATMCounter atm2 = new ATMCounter();
                ATMCounter atm3 = new ATMCounter();
                Customer c1 = new Customer("customer1",atm);
                Customer c2 = new Customer("customer2",atm1);
                Customer c3 = new Customer("customer3",atm2);
                Customer c4 = new Customer("customer4",atm3);

        }

}
```

Synchronized blocks:

Synchronized blocks are an alternative synchronization for achieving consistent results instead of using synchronized methods. Synchronized blocks must be always return ordinary instance methods and ordinary instance methods inherited non static methods for achieving the consistent result.

```
Synchronized(object of current class)
{
Block of statement(S);
}
```

Synchronized blocks always locks object of the class but not class because we override instance methods only but not recommended to static methods.

Need of Synchronized blocks: If a derived class inherits any method from an interface and if the inherited ordinary instance method is accessed by multiple Threads with respect to the 1derived class Object then the inherited ordinary instance method generates inconsistent result.

To avoid this inconsistent result, as a derived class programmer, we may attempt to write synchronized keyword before ordinary inherited instance method. This attempting is not possible because the derived class programmer does not have any privileges to change the prototype because interface methods. We are unable to solve inconsistent problem with concept of Synchronized methods. Hence to solve the problem of

Synchronized methods and to get consistent result we have a concept synchronized blocks.

```
//package SynchronizedBlockDemo;

public class Library {

        public void readingBook(String nm)
        {
                System.out.println(nm + "entered inside library");

                /* Synchronized block */
                synchronized(this)
                {
                        System.out.println(nm + "reading the book");
                        try
                        {
                                Thread.sleep(1000);
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println(e);
                        }
                        System.out.println(nm + "is leaving the library");
                }

        }

}
```

```
/*

/*

<<return-type>> <<method-name>>(Parameters)
{

        synchrnoized(this)
        {



    }
        .
        .
}
*/
//package SynchronizedBlockDemo;

public class Reader implements Runnable {
        Library lib;
        Thread t;

        Reader(String nm,Library lib)
        {
                t = new Thread(this,nm);
                this.lib = lib;
                t.start();
        }

        public void run()
        {
```

```
		lib.readingBook(t.getName());
	}
}

//package SynchronizedBlockDemo;

public class MainDemo {

	public static void main(String[] args) {
		Library lib = new Library();
		Reader r1 = new Reader("reader1",lib);
		Reader r2 = new Reader("reader2",lib);
		Reader r3 = new Reader("reader3",lib);
	}
}
```

Inter Thread Communication: ITC of Java is one of the specialized concept of inter process communication of Operating System.

1. ITC applications of Java are the fastest applications in the real industry compare to any of the applications in any language.
2. The real time implementations of ITC are

a) Implementation of real world server software by third party vendors.
b) Implementation of universal protocols such as http, FTP, SMTP e.t.c.,

Def: The mechanism of exchanging the data/information among multiple Threads is known as Inter Thread Communication (OR)

If the output of first Thread is given as input to second Thread, the output of second Thread is given to the input to Third Thread e.t.c., then the communication between first, second and third Threads are known as Inter Thread Communication. In order to develop Inter Thread Communication based applications we use the methods of java.lang.Object class. These methods are known as Inter Thread Communication.

Methods in java.lang.Object:

1. **public void wait(long milliseconds) throws interruptedException:** This method is used for making the Thread to wait for a period of time in terms of milliseconds. If the wailting time completed, automatically the Thread will be entered into Ready state from waiting state. This method is not recommended to use by Java programmer for making

the Thread to wait for an amount of time because Java programmer can't decide the CPU burst time of previous Thread to wait.

2. **public void wait():** This method is used for making the Thread to wait without specifying any time.
3. **public void notify():** This method is used for transferring one Thread at a time from waiting state to Ready state.
4. **public void notifyAll():** This method is used for transferring all the Threads into the form waiting state. The classical example of itc are

i) Producer – Consumer problems
ii) Dining – Philosophers problem
iii) Readers – writers problem
iv) Barbershop Problem

```
public class BankAccount {


    int money;

    boolean flag=false;


    public synchronized void deposit(String name)

    {

        if(flag)

        {


        System.out.println(name + "is going to waiting state");

            try {

                wait();


            } catch (InterruptedException e) {


                e.printStackTrace();

            }

        }


        money=1;

        flag=true;

        System.out.println(name + "is deposited Money");
```

```
                notify();

        }
        public synchronized void withdraw(String name)
        {
                if(!flag)
                        try {
                                System.out.println(name + "is going to waiting
state");
                                wait();
                        } catch (InterruptedException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }

                money=0;
                flag=false;
                System.out.println(name + "is withdrawn Money");
                notify();
        }

}
//package ITC;

public class Reciever implements Runnable {
        BankAccount account;
        Thread t;

        Reciever(String nm,BankAccount acc)
        {
                t = new Thread(this,nm);
                t.start();
```

```
		account = acc;
	}

	public void run()
	{
		for(int i=1;i<=5;i++)
			account.withdraw(t.getName());
	}


}
//package ITC;

public class Sender implements Runnable{

	BankAccount account;
	Thread t;
	Sender(String nm,BankAccount acc)
	{
		t = new Thread(this,nm);
		t.start();
		account = acc;
	}

	public void run()
	{
		for(int i=1;i<=5;i++)
			account.deposit(t.getName());
	}
}
class BankCustomerITC
```

```
{
        public static void main(String[] args)
        {
                BankAccount acc = new BankAccount();
                Sender s = new Sender("Sender",acc);
                Reciever r = new Reciever("Reciever",acc);


        }
}
```