UNIT IV

MULTITHREADING AND GENERIC PROGRAMMING

Differences between multithreading and multitasking , thread life cycle, creating threads, creating threads, synchronizing threads, Inter-thread communication, daemon threads, thread groups. Generic Programming - Generic classes-generic methods-Bounded Types-Restrictions and Limitations

Thread:

A thread is a single sequential (separate) flow of control within program.

Sometimes, it is called an execution context or light weight process.

Multithreading

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Multitasking

Executing several tasks simultaneously is called multi-tasking.

There are 2 types of multi-tasking

1. Process-based multitasking
2. Thread-based multi-tasking

1. Process-based multi-tasking

Executing various jobs together where each job is a separate independent operation is called process-based multi-tasking.

2. Thread-based multi-tasking

Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread-based multitasking and each independent part is called Thread. It is best suitable for the programmatic

level. The main goal of multi-tasking is to make or do a better performance of the system by reducing response time

Multithreading vs Multitasking	
Multithreading is to execute multiple threads in a process concurrently.	Multitasking is to run multiple processes on a computer concurrently.
Execution	
In Multithreading, the CPU switches between multiple threads in the same process.	In Multitasking, the CPU switches between multiple processes to complete the execution.
Resource Sharing	
In Multithreading, resources are shared among multiple threads in a process.	In Multitasking, resources are shared among multiple processes.
Complexity	
Multithreading is light-weight and easy to create.	Multitasking is heavy-weight and harder to create.

Life Cycle of Thread

A thread can be in any of the five following states

1. **Newborn State:**

When a thread object is created a new thread is born and said to be in Newborn state.

2. **Runnable State:**

If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion

3. Running State:

It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs

(a) Thread give up its control on its own and it can happen in the following situations

i. A thread gets suspended using suspend() method which can only be revived with resume() method

ii. A thread is made to sleep for a specified period of time using sleep(time) method, where time in milliseconds

iii. A thread is made to wait for some event to occur using wait () method. In this case a thread can be scheduled to run again using notify () method.

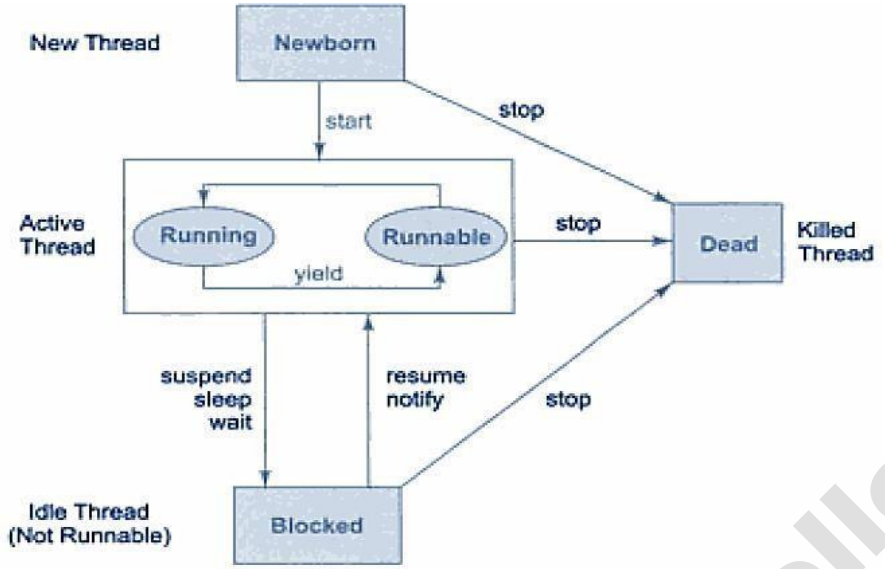
(b) A thread is pre-empted by a higher priority thread

4. Blocked State:

If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.

5. Dead State:

A runnable thread enters the Dead or terminated state when it completes its task or otherwise



The Main Thread

When we run any java program, the program begins to execute its code starting from the main method. Therefore, the JVM creates a thread to start executing the code present in main method. This thread is called as main thread. Although the main thread is automatically created, you can control it by obtaining a reference to it by calling currentThread() method.

Two important things to know about main thread are,

- It is the thread from which other threads will be produced.
- main thread must be always the last thread to finish execution.

```
class MainThread
{
 public static void main(String[] args)
 {
  Thread t1=Thread.currentThread();
  t.setName("MainThread");
  System.out.println("Name of thread is "+t1);
 }
}
```

Output:

Name of thread is Thread[MainThread,5,main]

Creation Of Thread

Thread Can Be Implemented In Two Ways

1) **Implementing Runnable Interface**
2) **Extending Thread Class**

1. Create Thread by Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run()

Example:

```
public class ThreadSample implements Runnable
{
        public void run()
        {
                try
                {
                        for (int i = 5; i > 0; i--)
                        {
                                System.out.println("Child Thread" + i);
                                Thread.sleep(1000);
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("Child interrupted");
                }
```

```
                System.out.println("Exiting Child Thread");
        }
}
public class MainThread
{
        public static void main(String[] arg)
        {
                ThreadSample d = new ThreadSample();
                Thread s = new Thread(d);
                s.start();
                try
                {
                        for (int i = 5; i > 0; i--)
                        {
                                System.out.println("Main Thread" + i);
                                Thread.sleep(5000);
                        }
                }
                catch (InterruptedException e)
                {
                        System.out.println("Main interrupted");
                }
                System.out.println("Exiting Main Thread");
        }}
```

2. Extending Thread Class

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of

the new thread.

Example:

```
public class ThreadSample extends Thread
{
	public void run()
	{
		try
		{
			for (int i = 5; i > 0; i--)
			{
				System.out.println("Child Thread" + i);
				Thread.sleep(1000);
			}
		}
		catch (InterruptedException e)
		{
			System.out.println("Child interrupted");
		}
	System.out.println("Exiting Child Thread");
	}
}
public class MainThread
{
	public static void main(String[] arg)
	{
		ThreadSample d = new ThreadSample();
		d.start();
		try
		{
```

```
                for (int i = 5; i > 0; i--)
                {
                        System.out.println("Main Thread" + i);
                        Thread.sleep(5000);
                }
            }
            catch (InterruptedException e)
            {
                System.out.println("Main interrupted");
            }
            System.out.println("Exiting Main Thread");
        }
}
```

Thread priority:

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

public static int MIN_PRIORITY

public static int NORM_PRIORITY

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.

Example :

```
public class MyThread1 extends Thread {
    MyThread1(String s)
    {
```

```
        super(s);
        start();
    }
    public void run()
    {
    for(int i=0;i<5;i++)
    {
        Thread cur=Thread.currentThread();
        cur.setPriority(Thread.MAX_PRIORITY);
        int p=cur.getPriority();
    System.out.println("Thread
Name"+Thread.currentThread().getName());
        System.out.println("Thread Priority"+cur);
    }
    }
}
class MyThread2 extends Thread {
    MyThread2(String s)
    {
        super(s);
        start();
    }
    public void run()
    {
    for(int i=0;i<5;i++)
    {
        Thread cur=Thread.currentThread();
        cur.setPriority(Thread.MIN_PRIORITY);
        System.out.println(cur.getPriority());
```

```
            int p=cur.getPriority();
        System.out.println("Thread
Name"+Thread.currentThread().getName());
            System.out.println("Thread Priority"+cur);
        }
        }
}
public class ThreadPriority {
        public static void main(String[] args)
        {
            MyThread1 m1=new MyThread1("MyThread1");
            MyThread2 m2=new MyThread2("MyThread2");
        }
        }
```

Synchronizing Threads

- Synchronization in java is the capability to control the access of multiple threads to any shared resource.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource

General Syntax :

```
synchronized(object)
{
//statement to be synchronized
}
```

Why use Synchronization

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

- Process Synchronization
- Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   - Synchronized method.
   - Synchronized block.
   - static synchronization.
2. Cooperation (Inter-thread communication in java)

Synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Example of synchronized method

```
package Thread;
public class SynThread
{
    public static void main(String args[])
    {
        share s = new share();
        MyThread m1 = new MyThread(s, "Thread1");
        MyThread m2 = new MyThread(s, "Thread2");
```

```
            MyThread m3 = new MyThread(s, "Thread3");
        }
}
class MyThread extends Thread
{
        share s;
        MyThread(share s, String str)
        {
            super(str);
            this.s = s;
            start();
        }
        public void run()
        {
            s.doword(Thread.currentThread().getName());
        }
}
class share
{
        public synchronized void doword(String str)
        {
            for (int i = 0; i < 5; i++)
            {
                System.out.println("Started:" + str);
                try
                {
                    Thread.sleep(1000);
                }
                catch (Exception e)
```

```
                {
                }}}}
```

Synchronized block.

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.
- If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Example of synchronized block

```
class Table{
 void printTable(int n){
   synchronized(this){//synchronized block
   for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}
  class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
```

```
}
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}
public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{
 synchronized static void printTable(int n){
   for(int i=1;i<=10;i++){
```

```
    System.out.println(n*i);
    try{
      Thread.sleep(400);
    }catch(Exception e){}
  }
 }
}
class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
}
}
class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
}
}
class MyThread3 extends Thread{
public void run(){
Table.printTable(100);
}
}
class MyThread4 extends Thread{
public void run(){
Table.printTable(1000);
}
}
public class TestSynchronization4{
public static void main(String t[]){
```

```
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();
MyThread3 t3=new MyThread3();
MyThread4 t4=new MyThread4();
t1.start();
t2.start();
t3.start();
t4.start();
}
}
```

Inter-thread communication

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

Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

wait()

tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.

notify()

wakes up a thread that called wait() on same object.

notifyAll()

wakes up all the thread that called wait() on same object.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example of inter thread communication in java

```
class Customer{
int amount=10000;
synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
```

```
}
}
class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();
}}
```

Daemon Thread in Java

Daemon thread in java is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

Example:

```
public class TestDaemonThread1 extends Thread{
 public void run(){
  if(Thread.currentThread().isDaemon()){//checking for daemon thread
   System.out.println("daemon thread work");
  }
  else{
  System.out.println("user thread work");
 }
 }
 public static void main(String[] args){
  TestDaemonThread1 t1=new TestDaemonThread1();//creating thread
  TestDaemonThread1 t2=new TestDaemonThread1();
```

```
  TestDaemonThread1 t3=new TestDaemonThread1();
  t1.setDaemon(true);//now t1 is daemon thread
  t1.start();//starting threads
  t2.start();
  t3.start();
 }
}
```

Thread Group

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

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

1. ThreadGroup(String name)-creates a thread group with given name.
2. ThreadGroup(ThreadGroup parent, String name)-creates a thread group with given parent group and name.

Important methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below.

1) int activeCount()-returns no. of threads running in current group.

2) int activeGroupCount()-returns a no. of active group in this thread group.

3) void destroy()-destroys this thread group and all its sub groups.

4) String getName()-returns the name of this group.

5) ThreadGroup getParent()-returns the parent of this group.

6) void interrupt()-interrupts all threads of this group.

7)void list()-prints information of this group to standard console.

Let's see a code to group multiple threads.

```
ThreadGroup tg1 = new ThreadGroup("Group A");
Thread t1 = new Thread(tg1,new MyRunnable(),"one");
Thread t2 = new Thread(tg1,new MyRunnable(),"two");
Thread t3 = new Thread(tg1,new MyRunnable(),"three");
```

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

ThreadGroup Example

```
public class ThreadGroupDemo implements Runnable{
   public void run() {
        System.out.println(Thread.currentThread().getName());
   }
   public static void main(String[] args) {
     ThreadGroupDemo runnable = new ThreadGroupDemo();
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
        Thread t1 = new Thread(tg1, runnable,"one");
        t1.start();
        Thread t2 = new Thread(tg1, runnable,"two");
        t2.start();
        Thread t3 = new Thread(tg1, runnable,"three");
        t3.start();
```

```
            System.out.println("Thread Group Name: "+tg1.getName());
            tg1.list();
        }
    }
```

Output:

```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]
    Thread[one,5,Parent ThreadGroup]
    Thread[two,5,Parent ThreadGroup]
    Thread[three,5,Parent ThreadGroup]
```

Generic Programming

Generic programming enables the programmer to create classes,interfaces and methods that automatically works with all types of data(Integer, String, Float etc). It has expanded the ability to reuse the code safely and easily.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.

2) Type casting is not required: There is no need to typecast the object.

3)Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Generic class

- A class that can refer to any type is known as generic class.
- Generic class declaration defines set of parameterized type one for each possible invocation of the type parameters

Example:

```
class TwoGen<T, V>
 {
        T ob1;
        V ob2;
       TwoGen(T o1, V o2)
        {
              ob1 = o1;
              ob2 = o2;
        }
       void showTypes() {
              System.out.println("Type of T is " + ob1.getClass().getName());
              System.out.println("Type of V is " + ob2.getClass().getName());
       }
       T getob1()
       {
              return ob1;
       }

       V getob2()
       {
              return ob2;
       }
}
public class MainClass
```

```
{
        public static void main(String args[])
            {
TwoGen<Integer, String> tgObj = new TwoGen<Integer, String>(88,"Generics");
            tgObj.showTypes();
              int v = tgObj.getob1();
            System.out.println("value: " + v);
              String str = tgObj.getob2();
            System.out.println("value: " + str);
              }
}
```

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

```
public class TestGenerics4{
  public static < E > void printArray(E[] elements) {
      for ( E element : elements){
         System.out.println(element );
       }
       System.out.println();
  }
   public static void main( String args[] ) {
      Integer[] intArray = { 10, 20, 30, 40, 50 };
      Character[] charArray = { 'J', 'A', 'V', 'A'};
      System.out.println( "Printing Integer Array" );
      printArray( intArray );
     System.out.println( "Printing Character Array" );
```

```
        printArray( charArray );
    }
}
```

Bounded type

The type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter

Syntax :

<T extends superclass>

Example

```
class Stats<T extends Number> {
  T[] nums;
   Stats(T[] o) {
   nums = o;   }
   double average() {
   double sum = 0.0;
    for(int i=0; i < nums.length; i++)
    sum  +=  nums[i].doubleValue();
    return sum / nums.length;
  }
}
public class MainClass {
  public static void main(String args[]) {
    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is " + v);
    Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
```

```
    double w = dob.average();
    System.out.println("dob average is " + w);
  }
}
```

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type