UNIT IV MULTITHREADING AND GENERIC PROGRAMMING

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

4.1 Differences between multi-threading and multitasking

4.1.1 Multithreading

A multithreaded program runs two or more programs run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution.

4.1.2 Multitasking

Multitasking is the process of running two or more programs concurrently. There are two types

1. Process based multitasking-A process is nothing but a program that is executing. It is the feature that runs two or more programs concurrently.

2. Thread based multitasking-The thread is the smallest unit of dispatchable code. A program can perform two or more tasks simultaneously.

4.2 Thread Life Cycle

Definition of Thread

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.
Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

During the life time of a thread, it enters into various states.
The states are

1. Newborn State
2. Runnable State
3. Running State

4. Blocked State
5. Dead State

A thread can move from one state to another state. It is always in one of these five states.

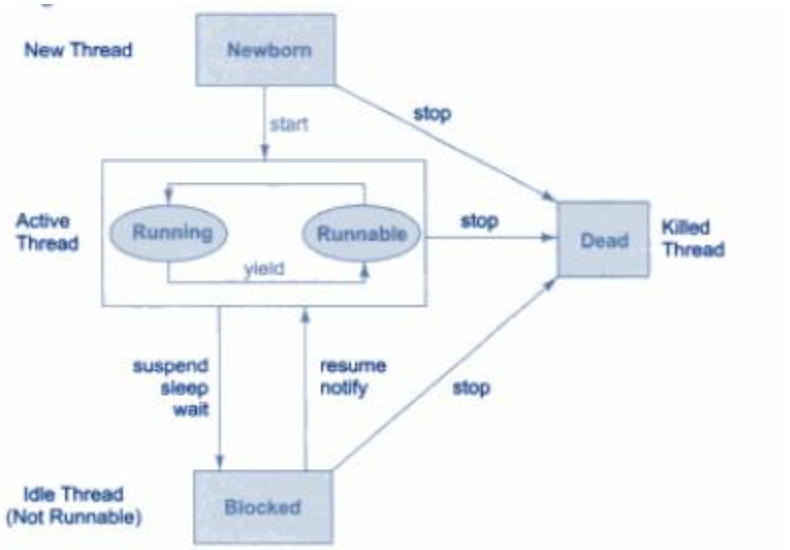


Fig 4.1 Life Cycle of a thread

1.Newborn State

When we create a thread object, the thread is born and is said to be newborn -state. In this state, we can do the following tasks

- Schedule a thread for running using start() method
- Kill a thread using stop() method

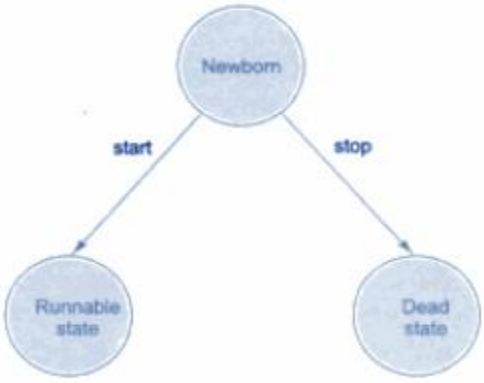


Fig 4.2 Scheduling a Newborn State

If a newborn thread is scheduled,it moves to the runnable state.

2.Runnable State

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.The thread is waiting in the queue for its execution.If all threads have equal priority,then they are given time slots for execution in round-robin fashion,that means first-come,first-serve manner.This process of assigning time to threads is known as time-slicing.

If we want a thread to relinquish control to another thread to equal priority before it turns comes, we can do the same by using yield() method.

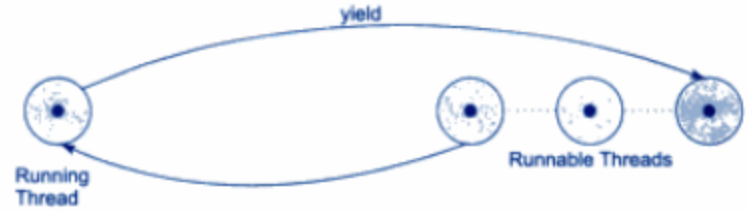


Fig 4.3 Yield() Method

3. Running State

Running means that the thread is allotted with the processor for its execution.The thread runs until higher priority thread comes.A running thread may relinquish its control in one of the following situations

a)suspend() method:

We can suspend the running thread for some time by using suspend() method. A suspended thread may resume by using resume() method.

Fig 4.4 suspend() Method

b)sleep() method:

We can put the running thread into sleep mode for some specified time period by using sleep(time)where time is in milliseconds. This means that the thread is out of the queue during the time

period.The thread re-enters into runnable state as soon as this time period is elapsed.

Fig 4.5 sleep() Method

c)wait() method:

The thread is in wait state until some event occurs. This is done using wait() method. The thread can be scheduled to run again using the notify() method.

Fig 4.6 wait Method

4.Blocked State

A thread is said to be blocked when it 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. A blocked thread is considered "not runnable" but not dead and fully qualified to run again.

5.Dead State

Every thread has a life cycle. A running thread ends its life when it has completed executing its run() method. It is a natural death. We can kill it by sending the stop message to it at any state.

4.3 Creating Threads

Creating threads in java is simple. Threads are implemented in the form of objects that contain a method called run(). The run() method is the heart and soul of any thread.It makes up the entire body of a thread and implements the thread's behavior.

Syntax:

```
public void run()
{
.....
.....
..... //Statements for implementing thread
}
```

The run() method must be invoked by an object of the concerned thread. This can be achieved by creating the thread and instantiating it with the help of stop() method.

A new thread can be created in two ways

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

The **Thread** class defines several methods that help manage threads.

Method	Meaning
getName()	Obtain a thread's name.
getPriority()	Obtain a thread's priority.
isAlive()	Determine if a thread is still running.
join()	Wait for a thread to terminate.
run()	Entry point for the thread.
sleep()	Suspend a thread for a period of time.
start()	Start a thread by calling its run method.

Table 4.1 Methods of Thread Class

4.3.1 The Main Thread

When a Java program starts up, one thread begins running immediately.This is called as the main thread of the program,because it is the one that is executed when the progam begins.

The main thread is important for two reasons

- 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.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**.

General Form:

static Thread currentThread( )

Example Program:

```
class CurrentThreadDemo
{
public static void main(String args[])
{
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
try
{
for(int n = 5; n > 0; n--)
{
System.out.println(n);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Main thread interrupted");
}
}
}
```

Output:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
```

5
4
3
2
1

4.3.2 Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

public void run( )

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread(Runnable *threadOb*, String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here:

void start( )

Example Program:

```
class NewThread implements Runnable
{
Thread t;
NewThread()
{
```

```
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
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 ThreadDemo
{
public static void main(String args[ ] )
{
new NewThread(); // create a new thread
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[Demo Thread,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.
```

4.3.2 Extending Thread

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. Here is the preceding program rewritten to extend **Thread**:

Example Program:

```
class NewThread extends Thread
{
NewThread()
{
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start(); // Start the thread
```

```
}
// This is the entry point for the second thread.
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[])
{
new NewThread(); // create a new thread
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[Demo Thread,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.
```

4.3.3 Creating Multiple Threads

Multiple threads can be created and executed concurrently at the same time.

Example Program:

```
class NewThread implements Runnable
{
String name; // name of thread
Thread t;
NewThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
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[])
{
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
}
catch (InterruptedException e)
{
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread 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.

4.3.4 Using isAlive( ) and join( )

isAlive() method

To determine whether a thread has finished, **we can use isAlive() method on the thread.**

Syntax:

final boolean isAlive( )

The isAlive() returns true if the thread is still running.It returns false otherwise.

join() method

The method that you will more commonly use to wait for a thread to finish is called **join( )**

Syntax:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Example Program:

```
class NewThread implements Runnable
{
String name; // name of thread
Thread t;
NewThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
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 DemoJoin
{
public static void main(String args[])
{
```

```
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
// wait for threads to finish
try
{
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch (InterruptedException e)
{
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Output:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
```

```
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

4.3.5 Thread Priority

Each thread is assigned a priority. Based on the priority, the thread will be scheduled for running. The threads of the same priority are given equal treatment by the Java scheduler, they share the processor on a first come, first serve basis.

The thread is set with the priority, we can use setPriority() method.

Syntax:
ThreadName.setPriority(intNumber);

The intNumber is an integer value to which the thread's priority is set. The Thread class defines several priority constants:

MIN_PRORITY = 1
NORM_PRORITY = 5
MAX_PRORITY = 10

Whenever multiple thread are ready for execution, the Java system chooses the highest priority and executes it. If another thread of a higher priority comes, the currently running thread is preempted by the incoming thread. Now preempted thread thread goes to runnable state.

Example Program:

```
class A extends Thread
{
public void run()
{
System.out.println("threadA started");
for(int i=1;i<=4;i++)
{
System.out.println("From Thread A:i="+i);
}
System.out.println("Exit from A");
}
}
class B extends Thread
{
public void run()
{
System.out.println("threadB started");
for(int j=1;j<=4;j++)
{
System.out.println("From Thread B: j="+j);
}
System.out.println("Exit from B");
}
}
class C extends Thread
{
public void run()
{
System.out.println("threadC started");
for(int k=1;k<=4;k++)
{
```

```
System.out.println("From Thread C: k="+k);
}
System.out.println("Exit from C");
}
}
class ThreadPriority
{
public static void main(String args[])
{
A threadA=new A();
B threadB=new B();
C threadC=new C();

threadC.setPriority(Thread.MAX_PRIORITY);
threadB.setPriority(threadA.getPriority()+1);
threadA.setPriority(Thread.MIN_PRIORITY);

System.out.println("Start thread A");
threadA.start();

System.out.println("Start thread B");
threadB.start();

System.out.println("Start thread C");
threadC.start();

System.out.println("End of main thread");
}
}
```

Output:

```
Start thread A
Start thread B
Start thread C
threadB started
From Thread B : j=1
From Thread B : j=2
```

```
threadC started
From Thread C : k=1
From Thread C : k=2
From Thread C : k=3
From Thread C : k=4
Exit from C
End of main thread
From Thread B : j=3
From Thread B : j=4
Exit from B
threadA started
From Thread A : i=1
From Thread A : i=2
From Thread A : i=3
From Thread A : i=4
Exit from A
```

4.4 synchronizing threads

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

When we want to share the resource with multiple threads, we can use the Java synchronization concept. 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. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

4.4.1 Purpose of using synchronization

1. To prevent thread interference
2. To prevent consistency problem

4.4.2 Types of Thread synchronization

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

1. Mutual Exclusive
    1. Synchronized method.

2. Synchronized block.
3. static synchronization.

2. Cooperation (Inter-thread communication in java)

4.4.3 Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

4.4.3.1 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 Program: Without Synchronized Method

```
class Table
{
void printTable(int n)
{//method not synchronized
for(int i=1;i<=5;i++)
{
System.out.println(n*i);
try
{
Thread.sleep(400);
}
catch(Exception e){System.out.println(e);}
}
}
}
class MyThread1 extends Thread
{
Table t;
```

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

class MyThread2 extends Thread
{
Table t;
MyThread2(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(100);
}
}
class TestSynchronization1
{
public static void main(String args[])
{
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
100
10
```

200
15
300
20
400
25
500

Example Program: With Synchronized Method

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

```
}
class MyThread2 extends Thread
{
Table t;
MyThread2(Table t)
{
this.t=t;
}
public void run()
{
t.printTable(100);
}
}
public class TestSynchronization2
{
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();
}
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

4.4.3.2 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.

Purpose of Synchronized Block

1. Synchronized block is used to lock an object for any shared resource.
2. Scope of synchronized block is smaller than the method.

Syntax:

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

Example program:

```
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();
}
}
```

Output:
5
10
15
20
25
100
200
300
400
500

4.4.3.3 Static Synchronization Block

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

Example Program:

```
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();
}
}
```

Output:

1
2
3
4
5
6
7
8
9
10
10
20
30
40
50
60
70
80
90
100
100
200
300
400
500
600
700
800
900
1000
1000
2000
3000
4000
5000
6000
7000

8000
9000
10000

4.5 Inter-thread Communication

Polling is a process in which the condition is repeatedly checked. If the condition is true, appropriate action is taken. This wastes CPU time.

For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.

To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

The following methods are declared within Object:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notify All( )
```

Example Program:

```
class Q
```

```
{
int n;
synchronized int get()
{
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n)
{
this.n = n;
System.out.println("Put: " + n);
}
}
class Producer implements Runnable
{
Q q;
Producer(Q q)
{
this.q = q;
new Thread(this, "Producer").start();
}
public void run()
{
int i = 0;
while(true)
{
q.put(i++);
}
}
}
class Consumer implements Runnable
{
Q q;
Consumer(Q q)
{
this.q = q;
new Thread(this, "Consumer").start();
}
```

```
public void run()
{
while(true)
{
q.get();
}
}
}
class PC
{
public static void main(String args[])
{
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Output:

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

4.6 Daemon Threads

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.
There are many java daemon threads running automatically e.g. gc, finalizer etc.
You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

4.6.1 Purpose of Daemon thread:

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

4.6.2 Methods of Daemon Thread

Method	Description
public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
public boolean isDaemon()	is used to check that current is daemon.

Table 4.2 Methods of Daemon Thread

Example Program:

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

Output:
daemon thread work
user thread work
user thread work

4.7 Thread Groups

ThreadGroup creates a group of threads. It offers a convenient way to manage groups of threads as a unit. This is particularly valuable in situation in which you want to suspend and resume a number of related threads.

- The thread group form a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group but not to access information about its thread group's parent thread group or any other thread group.

Constructors:

1. **public ThreadGroup(String name):** Constructs a new thread group. The parent of this new group is the thread group of the currently running thread.
2. **public ThreadGroup(ThreadGroup parent, String name):** Creates a new thread group. The parent of this new group is the specified thread group.

Methods:

1.int activeCount(): This method returns the number of threads in the group plus any group for which this thread is parent.

Example Program:

```
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 1000; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Exception encounterted");
}
}
}
}
public class ThreadGroupDemo
{
public static void main(String arg[])
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("parent thread group");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting one");
NewThread t2 = new NewThread("two", gfg);
```

```
System.out.println("Starting two");
// checking the number of active thread
System.out.println("number of active thread: "+
gfg.activeCount());
}
}
```

Output:

```
Starting one
Starting two
number of active thread: 2
```

2. int activeGroupCount(): This method returns an estimate of the number of active groups in this thread group.

Example Program:

```
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 1000; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Exception encounterted");
}
}
```

```
System.out.println(Thread.currentThread().getName()+  "finished executing");
}
}
public class ThreadGroupDemo1
{
public static void main(String arg[]) throws InterruptedException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("gfg");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting one");
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting two");
// checking the number of active thread
System.out.println("number of active thread group: "+
gfg.activeGroupCount());
}
}
```

Output:
Starting one
Starting two
number of active thread group: 2
one finished executing
two finished executing

3.void checkAccess(): Causes the security manager to verify that the invoking thread may access and/ or change the group on which **checkAccess()** is called.

Example Program:

```
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
```

```
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 1000; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Exception encounterted");
}
}
System.out.println(Thread.currentThread().getName() + "
finished executing");
}
}
public class ThreadGroupDemo2
{
public static void main(String arg[]) throws
InterruptedException,SecurityException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting one");
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting two");
gfg.checkAccess();
System.out.println(gfg.getName() + " has access");
gfg_child.checkAccess();
System.out.println(gfg_child.getName() + " has access");
}
}
```

Output:
Starting one
Starting two
Parent thread has access
child thread has access
one finished executing
two finished executing

4. void destroy(): Destroys the thread group and any child groups on which it is called.

Example Program:

```
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Exception encounterted");
}
}
}
}
public class ThreadGroupDemo3
{
```

```
public static void main(String arg[]) throws InterruptedException,SecurityException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting one");
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting two");
// block until other thread is finished
t1.join();
t2.join();
// destroying child thread
gfg_child.destroy();
System.out.println(gfg_child.getName() + " destroyed");
// destroying parent thread
gfg.destroy();
System.out.println(gfg.getName() + " destroyed");
}
}
```

Output:
Starting one
Starting two
child thread destroyed
Parent thread destroyed

5.**int enumerate(Thread group[]):** The thread that comprise the invoking thread group are put into the group array.

Example Program:

```
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
```

```
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Exception encounterted");
}
}
System.out.println(Thread.currentThread().getName() + "
finished executing");
}
}
public class ThreadGroupDemo4
{
public static void main(String arg[]) throws
InterruptedException,SecurityException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting one");
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting two");
// returns the number of threads put into the array
Thread[] group = new Thread[gfg.activeCount()];
int count = gfg.enumerate(group);
for (int i = 0; i < count; i++)
{
System.out.println("Thread " + group[i].getName() + " found");
}
```

```
}
}
```

Output:

```
Starting one
Starting two
Thread one found
Thread two found
one finished executing
two finished executing
```

6. **int getMaxPriority():** Returns the maximum priority setting for the group.

Example Program:

```
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Exception encounterted");
}
}
System.out.println(Thread.currentThread().getName() + "
finished executing");
```

```
}
}
public class ThreadGroupDemo5
{
public static void main(String arg[]) throws
InterruptedException,SecurityException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
// checking the maximum priority of parent thread
System.out.println("Maximum priority of ParentThreadGroup = "+
gfg.getMaxPriority());
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting one");
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting two");
}
}
```

Output:
Maximum priority of ParentThreadGroup = 10
Starting one
Starting two
two finished executing
one finished executing

7. **String getName():** This method returns the name of the group.

Example Program:

```
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
```

```
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Exception encounterted");
}
}
System.out.println(Thread.currentThread().getName() +" finished
executing");
}
}
public class ThreadGroupDemo6
{
public static void main(String arg[]) throws
InterruptedException,SecurityException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting " + t2.getName());
}
}
```

Output:

```
Starting one
Starting two
two finished executing
one finished executing
```

8. ThreadGroup getParent(): Returns null if the invoking ThreadGroup object has no parent. Otherwise, it returns the parent of the invoking object.

Syntax: final ThreadGroup getParent().

Returns: the parent of this thread group.

The top-level thread group is the only thread group whose parent is null.

Exception: SecurityException - if the current thread cannot modify this thread group.

Example Program:

```
// Java code illustrating getParent() method
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Exception encounterted");
}
}
System.out.println(Thread.currentThread().getName()+ " finished executing");
}
}
```

```
public class ThreadGroupDemo7
{
public static void main(String arg[]) throws InterruptedException,SecurityException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting " + t2.getName());
// prints the parent ThreadGroup
// of both parent and child threads
System.out.println("ParentThreadGroup for " + gfg.getName()
+ " is " + gfg.getParent().getName());
System.out.println("ParentThreadGroup for " +
gfg_child.getName() + " is " + gfg_child.getParent().getName());
}
}
```

Output:

```
Starting one
Starting two
ParentThreadGroup for Parent thread is main
ParentThreadGroup for child thread is Parent thread
one finished executing
two finished executing
```

9. void interrupt(): Invokes the **interrupt()** methods of all threads in the group.

Syntax: public final void interrupt().

Returns: NA.

Exception: SecurityException - if the current thread is not allowed to access this thread group or any of the threads in the thread group.

Example Program:

```
// Java code illustrating interrupt() method
```

```
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Thread " +
Thread.currentThread().getName()+ " interrupted");
}
}
System.out.println(Thread.currentThread().getName() +" finished
executing");
}
}
public class ThreadGroupDemo8
{
public static void main(String arg[]) throws
InterruptedException,SecurityException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");

NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
```

```
System.out.println("Starting " + t2.getName());
// interrupting thread group
gfg.interrupt();
}
}
```

Output:

```
Starting one
Starting two
Thread two interrupted
Thread one interrupted
one finished executing
two finished executing
```

10. boolean isDaemon():

Tests if this thread group is a daemon thread group. A daemon thread group is automatically destroyed when its last thread is stopped or its last thread group is destroyed.

Syntax: public final boolean isDaemon().

Example Program:

```
// Java code illustrating isDaemon() method
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
```

```
catch (InterruptedException ex)
{
System.out.println("Thread" + Thread.currentThread().getName()
                                + " interrupted");
}
}
System.out.println(Thread.currentThread().getName() +" finished
executing");
}
}
public class ThreadGroupDemo9
{
public static void main(String arg[]) throws InterruptedException,
SecurityException
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting " + t2.getName());
if (gfg.isDaemon() == true)
System.out.println("Group is Daemon group");
else
System.out.println("Group is not Daemon group");
}
}
```

Output:

```
Starting one
Starting two
Group is not Daemon group
two finished executing
one finished executing
```

11. **boolean isDestroyed():** This method tests if this thread group has been destroyed.

Syntax: public boolean isDestroyed().

Example Program:

```
// Java code illustrating isDestroyed() method
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Thread " +
Thread.currentThread().getName()
                                + " interrupted");
}
}
System.out.println(Thread.currentThread().getName() + "
finished executing");
}
}
public class ThreadGroupDemo10
{
public static void main(String arg[]) throws InterruptedException,
SecurityException, Exception
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
```

```
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting " + t2.getName());
if (gfg.isDestroyed() == true)
System.out.println("Group is destroyed");
else
System.out.println("Group is not destroyed");
}
}
```

Output:

```
Starting one
Starting two
Group is not destroyed
one finished executing
two finished executing
```

12. **void list():** Displays information about the group.

Syntax: public void list().

Example Program:

```
// Java code illustrating list() method.
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
```

```
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Thread " +
Thread.currentThread().getName()
                            + " interrupted");
}
}
System.out.println(Thread.currentThread().getName() +
          " finished executing");
}
}
public class ThreadGroupDemo11
{
public static void main(String arg[]) throws InterruptedException,
SecurityException, Exception
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting " + t2.getName());
// listing contents of parent ThreadGroup
System.out.println("\nListing parentThreadGroup: " +
gfg.getName() + ":");
// prints information about this thread group
// to the standard output
gfg.list();
}
}
```

Output:

```
Starting one
Starting two
```

```
Listing parentThreadGroup: Parent thread:
java.lang.ThreadGroup[name=Parent thread, maxpri=10]
    Thread[one, 5, Parent thread]
    Thread[two, 5, Parent thread]
    java.lang.ThreadGroup[name=child thread, maxpri=10]
one finished executing
two finished executing
```

13. boolean parentOf(ThreadGroup group): This method tests if this thread group is either the thread group argument or one of its ancestor thread groups.
Syntax: final boolean parentOf(ThreadGroup group).

Example Program:

```
// Java code illustrating parentOf() method
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Thread " +
Thread.currentThread().getName()+ " interrupted");
```

```
}
}
System.out.println(Thread.currentThread().getName() + "
finished executing");
}
}
public class ThreadGroupDemo12
{
public static void main(String arg[]) throws InterruptedException,
        SecurityException, Exception
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting " + t2.getName());
// checking who is parent thread
if (gfg.parentOf(gfg_child))
System.out.println(gfg.getName() + " is parent of "
+ gfg_child.getName());
}
}
```

Output:

```
Starting one
Starting two
Parent thread is parent of child thread
two finished executing
one finished executing
```

14. **void setDaemon(boolean isDaemon):** This method changes the daemon status of this thread group. A daemon thread group is automatically destroyed when its last thread is stopped or its last thread group is destroyed.
Syntax: final void setDaemon(boolean isDaemon).

Example Program:

```
// Java code illustrating setDaemon() method
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Thread " +
Thread.currentThread().getName()
                        + " interrupted");
}
}
System.out.println(Thread.currentThread().getName() +
            " finished executing");
}
}
public class ThreadGroupDemo13
{
public static void main(String arg[]) throws InterruptedException,
SecurityException, Exception
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
// daemon status is set to true
```

```
gfg.setDaemon(true);
// daemon status is set to true
gfg_child.setDaemon(true);
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting " + t2.getName());
if (gfg.isDaemon() && gfg_child.isDaemon())
System.out.println("Parent Thread group and "+ "child thread
group"
                        + " is daemon");
}
}
```

Output:

```
Starting one
Starting two
Parent Thread group and child thread group is daemon
one finished executing
two finished executing
```

15.**void setMaxPriority(int priority):** Sets the maximum priority of the invoking group to priority.
Syntax: final void setMaxPriority(int priority).

Example Program

```
// Java code illustrating setMaxPriority() method
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
}
public void run()
{
for (int i = 0; i < 10; i++)
{
```

```
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Thread " +
Thread.currentThread().getName()
                              + " interrupted");
}
}
System.out.println(Thread.currentThread().getName() +
          " [priority = " + Thread.currentThread().getPriority() + "]
            finished executing.");
}
}
public class ThreadGroupDemo14
{
public static void main(String arg[]) throws InterruptedException,
       SecurityException, Exception
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
gfg.setMaxPriority(Thread.MAX_PRIORITY - 2);
gfg_child.setMaxPriority(Thread.NORM_PRIORITY);
NewThread t1 = new NewThread("one", gfg);
t1.setPriority(Thread.MAX_PRIORITY);
System.out.println("Starting " + t1.getName());
t1.start();
NewThread t2 = new NewThread("two", gfg_child);
t2.setPriority(Thread.MAX_PRIORITY);
System.out.println("Starting " + t2.getName());
t2.start();
}
}
```

Output:

Starting one
Starting two
two [priority = 5] finished executing.
one [priority = 8] finished executing.

16. **String toString():** This method returns a string representation of this Thread group.
Syntax: public String toString().

Example Program:

```
// Java code illustrating toString() method
import java.lang.*;
class NewThread extends Thread
{
NewThread(String threadname, ThreadGroup tgob)
{
super(tgob, threadname);
start();
}
public void run()
{
for (int i = 0; i < 10; i++)
{
try
{
Thread.sleep(10);
}
catch (InterruptedException ex)
{
System.out.println("Thread " +
Thread.currentThread().getName()
                        + " interrupted");
}
}
System.out.println(Thread.currentThread().getName() +
            " finished executing");
}
}
```

```
public class ThreadGroupDemo15
{
public static void main(String arg[]) throws InterruptedException,
      SecurityException, Exception
{
// creating the thread group
ThreadGroup gfg = new ThreadGroup("Parent thread");
ThreadGroup gfg_child = new ThreadGroup(gfg, "child thread");
// daemon status is set to true
gfg.setDaemon(true);
// daemon status is set to true
gfg_child.setDaemon(true);
NewThread t1 = new NewThread("one", gfg);
System.out.println("Starting " + t1.getName());
NewThread t2 = new NewThread("two", gfg);
System.out.println("Starting " + t2.getName());
// string equivalent of the parent group
System.out.println("String equivalent: " + gfg.toString());
}
}
```

Output:

```
Starting one
Starting two
String equivalent: java.lang.ThreadGroup[name=Parent thread,
maxpri=10]
one finished executing
two finished executing
```

4.8 Generic Programming

The term ***generics*** means ***parameterized types***. Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class,

interface, or method that operates on a parameterized type is called *generic*, as in ***generic class* or *generic method*.**
Object is the superclass of all other classes, an **Object** reference can refer to any type object. Thus, in pre-generics code, generalized classes, interfaces, and methods used **Object** references to operate on various types of objects.
Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.
Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

4.9 Generic Classes

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

General Form:

```
class class-name<type-param-list>
{
//Body of the class
}
```

Class Reference Declaration:

class-name<type-arg-list> var-name=new class-name<type-arg-list>(cons-arg-list)

4.9.1 Generic class with single type parameters

The following program defines two classes. The first is the generic class Gen and the second class GenDemo, which uses Gen. Here T is a type parameter that will be replaced by a real type when an object of type Gen is created.

Example Program:

```
class Gen<T>
{
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o)
{
ob = o;
}
// Return ob.
T getob()
{
return ob;
}
// Show type of T.
void showType()
{
System.out.println("Type of T is " + ob.getClass().getName());
}
}
// Demonstrate the generic class.
class GenDemo
{
public static void main(String args[])
{
// Create a Gen reference for Integers.
Gen<Integer> iOb;
// Create a Gen<Integer> object and assign its
// reference to iOb. Notice the use of autoboxing
// to encapsulate the value 88 within an Integer object.
iOb = new Gen<Integer>(88);
// Show the type of data used by iOb.
iOb.showType();
// Get the value in iOb. Notice that
// no cast is needed.
int v = iOb.getob();
System.out.println("value: " + v);
```

```
System.out.println();
// Create a Gen object for Strings.
Gen<String> strOb = new Gen<String> ("Generics Test");
// Show the type of data used by strOb.
strOb.showType();
// Get the value of strOb. Again, notice
// that no cast is needed.
String str = strOb.getob();
System.out.println("value: " + str);
}
}
```

Output:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

4.9.2 Generic class with two type parameters

In a generic type, more than one type parameter can be declared. To specify two or more type parameters, simply use a comma-separated list.

Example Program:

```
// A simple generic class with two type
// parameters: T and V.
class TwoGen<T, V>
{
T ob1;
V ob2;
// Pass the constructor a reference to
// an object of type T and an object of type V.
TwoGen(T o1, V o2)
{
ob1 = o1;
ob2 = o2;
}
// Show types of T and V.
```

```
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;
}
}
// Demonstrate TwoGen.
class SimpGen
{
public static void main(String args[])
{
TwoGen<Integer, String> tgObj =new TwoGen<Integer,
String>(88, "Generics");
// Show the types.
tgObj.showTypes();
// Obtain and show values.
int v = tgObj.getob1();
System.out.println("value: " + v);
String str = tgObj.getob2();
System.out.println("value: " + str);
}
}
```

Output:

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

4.10 Generic Methods

Generic method is a method with type parameters. In this Generic concept , types and methods can be generic.

Syntax:

<type-param-list>re-type meth-name(param-list)
{
// Function Body
……..
}

where

type-param-list is a list of type parameters separated by commas

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods –

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example Program1:

```
public class GenericMethodTest
{
// generic method printArray
```

```
public static < E > void printArray( E[] inputArray )
{
// Display array elements
for(E element : inputArray)
{
System.out.printf("%s ", element);
}
System.out.println();
}
public static void main(String args[])
{
// Create arrays of Integer, Double and Character
Integer[] intArray = { 1, 2, 3, 4, 5 };
Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
System.out.println("Array integerArray contains:");
printArray(intArray);   // pass an Integer array
System.out.println("\nArray doubleArray contains:");
printArray(doubleArray);   // pass a Double array
System.out.println("\nArray characterArray contains:");
printArray(charArray);   // pass a Character array
}
}
```

Output:

```
Array integerArray contains:
1 2 3 4 5
Array doubleArray contains:
1.1 2.2 3.3 4.4
Array characterArray contains:
H E L L O
```

Example Program2:

```
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', 'T','P','O','I','N','T' };
System.out.println( "Printing Integer Array" );
printArray( intArray  );
System.out.println( "Printing Character Array" );
printArray( charArray );
}
}
```

Output:

```
Printing Integer Array
        10
        20
        30
        40
        50
        Printing Character Array
        J
        A
        V
        A
        T
        P
        O
        I
        N
        T
```

Example Program3:

```
// Demonstrate a simple generic method.
class GenMethDemo
```

```
{
// Determine if an object is in an array.
static <T, V extends T> boolean isIn(T x, V[] y)
{
for(int i=0; i < y.length; i++)
if(x.equals(y[i])) return true;
return false;
}
public static void main(String args[])
{
// Use isIn() on Integers.
Integer nums[] = { 1, 2, 3, 4, 5 };
if(isIn(2, nums))
System.out.println("2 is in nums");
if(!isIn(7, nums))
System.out.println("7 is not in nums");
System.out.println();
// Use isIn() on Strings.
String strs[] = { "one", "two", "three","four", "five" };
if(isIn("two", strs))
System.out.println("two is in strs");
if(!isIn("seven", strs))
System.out.println("seven is not in strs");
// Oops! Won't compile! Types must be compatible.
// if(isIn("two", nums))
// System.out.println("two is in strs");
}
}
```

Output:
2 is in nums
7 is not in nums
two is in strs
seven is not in strs

4.10.1 Generic Constructors

A generic constructor is a constructor with type parameters.

Example Program:

```
// Use a generic constructor.
class GenCons
{
private double val;
<T extends Number> GenCons(T arg)
{
val = arg.doubleValue();
}
void showval()
{
System.out.println("val: " + val);
}
}
class GenConsDemo
{
public static void main(String args[])
{
GenCons test = new GenCons(100);
GenCons test2 = new GenCons(123.5F);
test.showval();
test2.showval();
}
}
```

Output:

```
val: 100.0
val: 123.5
```

4.11 Bounded Types

Bounded type parameters can be a type parameter with one or more bounds. The bounds restrict the set of types that can be used as type arguments and give access to the methods defined by the bounds.

General Form:

To declare a parameter with bounded type, the list of type parameter names can be followed by the extends keyword.

<T extends superclass>

This specifies that T can be replaced by superclass.Superclass defines an inclusive,upper limit.

Example Program1:

```
public class MaximumTest
{
// determines the largest of three Comparable objects
public static <T extends Comparable<T>> T maximum(T x, T y,
T z)
{
T max = x;   // assume x is initially the largest
if(y.compareTo(max) > 0)
{
max = y;   // y is the largest so far
}
if(z.compareTo(max) > 0)
{
max = z;   // z is the largest now
}
return max;   // returns the largest object
}
public static void main(String args[])
{
System.out.printf("Max of %d, %d and %d is %d\n\n",3, 4, 5,
maximum( 3, 4, 5 ));
System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",6.6,
8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));
System.out.printf("Max of %s, %s and %s is
%s\n","pear","apple", "orange", maximum("pear", "apple",
"orange"));
}
}
```

Output:

```
Max of 3, 4 and 5 is 5
Max of 6.6,8.8 and 7.7 is 8.8
Max of pear, apple and orange is pear
```

Example Program2:

```
class Stats<T extends Number>
{
T[] nums; // array of Number or subclass
// Pass the constructor a reference to
// an array of type Number or subclass.
Stats(T[] o)
{
nums = o;
}
// Return type double in all cases.
double average()
{
double sum = 0.0;
for(int i=0; i < nums.length; i++)
sum += nums[i].doubleValue();
return sum / nums.length;
}
}
// Demonstrate Stats.

class BoundsDemo
{
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);
```

```
// This won't compile because String is not a
// subclass of Number.
// String strs[] = { "1", "2", "3", "4", "5" };
// Stats<String> strob = new Stats<String>(strs);
// double x = strob.average();
// System.out.println("strob average is " + v);
}
}
```

Output:

```
iob average is 3.0
dob average is 3.3
```

4.11.1 Wild Card Arguments

A wildcard is a syntactic construct that denotes a family of types. A wildcard describes a family of types. The different types of wildcards are given below

Notation	Meaning
<T>	Concrete type
<?>	The unbounded wildcard. It stands for the family of all types
<?super subclass>	A bounded wildcard supertype of T. It stands for the family of all types that are supertypes of Type,type Type being included
<?extends superclass>	A bounded wildcard subtype of T
<U extends T>	U must be a supertype of T

4.11.2 Unbounded Wildcard

The wildcard "?" matches any type of valid stats object.

Example Program:

```
class Stats<T extends Number>
{
```

```
T[] nums; // array of Number or subclass
// Pass the constructor a reference to
// an array of type Number or subclass.
Stats(T[] o)
{
nums = o;
}
// Return type double in all cases.
double average()
{
double sum = 0.0;
for(int i=0; i < nums.length; i++)
sum += nums[i].doubleValue();
return sum / nums.length;
}
// Determine if two averages are the same.
// Notice the use of the wildcard.
boolean sameAvg(Stats<?> ob)
{
if(average() == ob.average())
return true;
return false;
}
}
// Demonstrate wildcard.
class WildcardDemo
{
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);
Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
```

```
Stats<Float> fob = new Stats<Float>(fnums);
double x = fob.average();
System.out.println("fob average is " + x);
// See which arrays have same average.
System.out.print("Averages of iob and dob ");
if(iob.sameAvg(dob))
System.out.println("are the same.");
else
System.out.println("differ.");
System.out.print("Averages of iob and fob ");
if(iob.sameAvg(fob))
System.out.println("are the same.");
else
System.out.println("differ.");
}
}
```

Output:
iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.

4.11.3 Bounded Wildcards

Wildcard arguments can be bounded in much the same way that a type parameter can be bounded. A bounded wildcard is especially important when you are creating a generic type that will operate on a class hierarchy.

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the upper bound, which is created using an **extends** clause in much the same way it is used to create a bounded type.

Example Program:

```
class TwoD
{
int x, y;
TwoD(int a, int b)
{
x = a;
y = b;
}
}
// Three-dimensional coordinates.
class ThreeD extends TwoD
{
int z;
ThreeD(int a, int b, int c)
{
super(a, b);
z = c;
}
}
// Four-dimensional coordinates.
class FourD extends ThreeD
{
int t;
FourD(int a, int b, int c, int d)
{
super(a, b, c);
t = d;
}
}
// This class holds an array of coordinate objects.
class Coords<T extends TwoD>
{
T[] coords;
Coords(T[] o)
{
coords = o;
}
```

```
}
// Demonstrate a bounded wildcard.
class BoundedWildcard
{
static void showXY(Coords<?> c)
{
System.out.println("X Y Coordinates:");
for(int i=0; i < c.coords.length; i++)
System.out.println(c.coords[i].x + " " +
c.coords[i].y);
System.out.println();
}
static void showXYZ(Coords<? extends ThreeD> c)
{
System.out.println("X Y Z Coordinates:");
for(int i=0; i < c.coords.length; i++)
System.out.println(c.coords[i].x + " " +
c.coords[i].y + " " +
c.coords[i].z);
System.out.println();
}
static void showAll(Coords<? extends FourD> c)
{
System.out.println("X Y Z T Coordinates:");
for(int i=0; i < c.coords.length; i++)
System.out.println(c.coords[i].x + " " +
c.coords[i].y + " " +
c.coords[i].z + " " +
c.coords[i].t);
System.out.println();
}
public static void main(String args[])
{
TwoD td[] = {
new TwoD(0, 0),
new TwoD(7, 9),
new TwoD(18, 4),
new TwoD(-1, -23)
```

```
};
Coords<TwoD> tdlocs = new Coords<TwoD>(td);
System.out.println("Contents of tdlocs.");
showXY(tdlocs); // OK, is a TwoD
// showXYZ(tdlocs); // Error, not a ThreeD
// showAll(tdlocs); // Error, not a FourD
// Now, create some FourD objects.
FourD fd[] = {
new FourD(1, 2, 3, 4),
new FourD(6, 8, 14, 8),
new FourD(22, 9, 4, 9),
new FourD(3, -2, -23, 17)
};
Coords<FourD> fdlocs = new Coords<FourD>(fd);
System.out.println("Contents of fdlocs.");
// These are all OK.
showXY(fdlocs);
showXYZ(fdlocs);
showAll(fdlocs);
}
}
```

Output:

```
Contents of tdlocs.
X Y Coordinates:
0 0
7 9
18 4
-1 -23

Contents of fdlocs.
X Y Coordinates:
1 2
6 8
22 9
3 -2

X Y Z Coordinates:
```

1 2 3
6 8 14
22 9 4
3 -2 -23

X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17

4.11.4 Inheritance and Generics

A generic class can act as a superclass or be a subclass.

4.11.4.1 Using a Generic Superclass

in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses.

Example Program:

```
// A subclass can add its own type parameters.
class Gen<T>
{
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o)
{
ob = o;
}
// Return ob.
T getob()
{
return ob;
}
}
// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T>
```

```
{
V ob2;
Gen2(T o, V o2)
{
super(o);
ob2 = o2;
}
V getob2()
{
return ob2;
}
}
// Create an object of type Gen2.
class HierDemo
{
public static void main(String args[])
{
// Create a Gen2 object for String and Integer.
Gen2<String, Integer> x =
new Gen2<String, Integer>("Value is: ", 99);
System.out.print(x.getob());
System.out.println(x.getob2());
}
}
```

Output:
Value is: 99

4.11.4.2 A Generic Subclass

A subclass can add its own type parameters,if needed.

Example Program:

```
class NonGen
{
int num;
NonGen(int i)
{
num = i;
```

```
}
int getnum()
{
return num;
}
}
// A generic subclass.
class Gen<T> extends NonGen
{
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o, int i)
{
super(i);
ob = o;
}
// Return ob.
T getob()
{
return ob;
}
}
// Create a Gen object.
class HierDemo2
{
public static void main(String args[])
{
// Create a Gen object for String.
Gen<String> w = new Gen<String>("Hello", 47);
System.out.print(w.getob() + " ");
System.out.println(w.getnum());
}
}
```

Output:

```
Hello 47
```

4.11.5 Overriding Methods in a Generic Clas

A method in a generic class can be overridden just like any other method.

Example Program:

```
class Gen<T>
{
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o)
{
ob = o;
}
// Return ob.
T getob()
{
System.out.print("Gen's getob(): " );
return ob;
}
}
// A subclass of Gen that overrides getob().
class Gen2<T> extends Gen<T>
{
Gen2(T o)
{
super(o);
}
// Override getob().
T getob()
{
System.out.print("Gen2's getob(): ");
return ob;
}
}
// Demonstrate generic method override.
class OverrideDemo
{
```

```
public static void main(String args[])
{
// Create a Gen object for Integers.
Gen<Integer> iOb = new Gen<Integer>(88);
// Create a Gen2 object for Integers.
Gen2<Integer> iOb2 = new Gen2<Integer>(99);
// Create a Gen2 object for Strings.
Gen2<String> strOb2 = new Gen2<String> ("Generics Test");
System.out.println(iOb.getob());
System.out.println(iOb2.getob());
System.out.println(strOb2.getob());
}
}
```

Output:

```
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```

4.12 Restrictions and Limitations

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays.

4.12.1 Type Parameters Can't Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

Example Program:

```
// Can't create an instance of T.
class Gen<T>
{
T ob;
Gen()
{
ob = new T(); // Illegal!!!
}
}
```

Here, it is illegal to attempt to create an instance of **T**. The reason should be easy to
understand: because **T** does not exist at run time, how would the compiler know what type
of object to create?

4.12.2 Restrictions on Static Members

No **static** member can use a type parameter declared by the enclosing class. For example,
both of the **static** members of this class are illegal:

Example Program:

```
class Wrong<T>
{
// Wrong, no static variables of type T.
static T ob;
// Wrong, no static method can use T.
static T getob()
{
return ob;
}
}
```

Although you can't declare **static** members that use a type parameter declared by the enclosing class, you can declare **static** generic methods, which define their own type parameters

4.12.3 Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose element type is a type parameter. Second, you cannot create an array of type-specific generic references.

Example Program:

```
// Generics and arrays.
class Gen<T extends Number>
{
T ob;
T vals[]; // OK
```

```
Gen(T o, T[] nums)
{
ob = o;
// This statement is illegal.
// vals = new T[10]; // can't create an array of T
// But, this statement is OK.
vals = nums; // OK to assign reference to existent array
}
}
class GenArrays
{
public static void main(String args[])
{
Integer n[] = { 1, 2, 3, 4, 5 };
Gen<Integer> iOb = new Gen<Integer>(50, n);
// Can't create an array of type-specific generic references.
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
// This is OK.
Gen<?> gens[] = new Gen<?>[10]; // OK
}
}
```

4.12.4 Generic Exception Restriction A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.

2 Marks Questions and Answers

1. Difference between multitasking and multithreading.

Multithreading	**Multitasking**
The system executes multiple threads of the same or different processes at the same time.	The system allows executing multiple programs and tasks at the same time
CPU has to **switch** between **multiple threads** to make it	**CPU** has to **switch** between **multiple programs** so that it appears that multiple

appear that all threads are running simultaneously	programs are running simultaneously.
Threads belonging to the same process **shares the same memory and resources** as that of the process.	Multitasking allocates **separate memory and resources** for each process/program

2. What is thread?

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

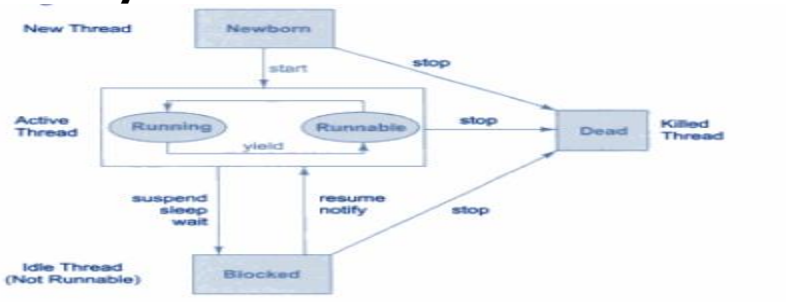
3. Write the states of a thread.

During the life time of a thread, it enters into various states.

The states are

- Newborn State
- Runnable State
- Running State
- Blocked State
- Dead State

4. Draw the life cycle of thread.



5. What is the role of Newborn State?

When we create a thread object, the thread is born and is said to be newborn -state. In this state, we can do the following tasks

- Schedule a thread for running using start() method
- Kill a thread using stop() method

6. What is the role of Runnable State?

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.The thread is waiting in the queue for its execution.If all threads have equal priority,then they are given time slots for execution in round-robin fashion,that means first-come,first-serve manner.

7. Write down the role of Running State.

Running means that the thread is allotted with the processor for its execution.The thread runs until higher priority thread comes.A running thread may relinquish its control in one of the following situations

8. Write down the role of Blocked State.

A thread is said to be blocked when it 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. A blocked thread is considered "not runnable" but not dead and fully qualified to run again.

9. Write down the role of Dead State.

Every thread has a life cycle. A running thread ends its life when it has completed executing its run() method. It is a natural death. We can kill it by sending the stop message to it at any state.

10. What are the ways of creating threads?

A new thread can be created in two ways

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

11. What are the methods in Thread class?

Method	Meaning
getName()	Obtain a thread's name.
getPriority()	Obtain a thread's priority.
isAlive()	Determine if a thread is still running.
join()	Wait for a thread to terminate.
run()	Entry point for the thread.
sleep()	Suspend a thread for a period of time.
start()	Start a thread by calling its run method.

12. What is the difference between yielding and sleeping?

When a task invokes its yield() method, it returns to the ready state. When a task invokes its sleep() method, it returns to the waiting state.

13. Can I implement my own start() method?

The Thread start() method is not marked final, but should not be overridden. This method contains the code that creates a new executable thread and is very specialised. Your threaded application should either pass a Runnable type to a new Thread, or extend Thread and override the run() method.

14. How do you set the priority to a thread?

Each thread is assigned a priority. Based on the priority, the thread will be scheduled for running. The threads of the same priority are given equal treatment by the Java scheduler, they share the processor on a first come, first serve basis.

The thread is set with the priority, we can use setPriority() method.

Syntax:

ThreadName.setPriority(intNumber);

The intNumber is an integer value to which the thread's priority is set. The Thread class defines several priority constants:

MIN_PRORITY = 1
NORM_PRORITY = 5
MAX_PRORITY = 10

15. How to synchronize the threads in Java?

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

When we want to share the resource with multiple threads, we can use the Java synchronization concept. 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. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

16. Write the purpose of using synchronization in threads?

- To prevent thread interference
- To prevent consistency problem

17. What are the types of Thread Synchronization?

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

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

18. Write down the use 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.

19. What is the use of 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.

20. What is the use of static synchronization block?

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

21. What is meant by 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.

22. How does the interprocess communication happen?

Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods.

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

23. Write down the purpose of using Daemon thread in java.

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.

- Its life depends on user threads.
- It is a low priority thread.

24. What are the methods of daemon thread?

Method	Description
public void setDaemon(boolean status)	is used to mark the current thread as daemon thread or user thread.
public boolean isDaemon()	is used to check that current is daemon.

25. What is meant by ThreadGroup?

ThreadGroup creates a group of threads. It offers a convenient way to manage groups of threads as a unit. This is particularly valuable in situation in which you want to suspend and resume a number of related threads.

- The thread group form a tree in which every thread group except the initial thread group has a parent.
- A thread is allowed to access information about its own thread group but not to access information about its thread group's parent thread group or any other thread group.

26. Write down the constructors of ThreadGroup.

public ThreadGroup(String name): Constructs a new thread group. The parent of this new group is the thread group of the currently running thread.

public ThreadGroup(ThreadGroup parent, String name): Creates a new thread group. The parent of this new group is the specified thread group.

27. Write down some of the methods in ThreadGroup.

activeCount(),activeGroupCount(),**checkAccess(),destroy(), enumerate(Thread group[]),getMaxPriority(),getName()**

28. Write the purpose of using generic programming.

The term ***generics*** means ***parameterized types***. Parameterized types are important because they enable you to

create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.

29. What is meany by generic class?

A class that can refer to any type is known as generic class. Here, we are using **T** type parameter to create the generic class of specific type.

General Form:

```
class class-name<type-param-list>
{
//Body of the class
}
```

30. What is meany by generic method?

Generic method is a method with type parameters. In this Generic concept , types and methods can be generic.

Syntax:

```
<type-param-list>re-type meth-name(param-list)
{
// Function Body
……..
}
```

31. Write down the rules to define Generic methods.

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only

reference types, not primitive types (like int, double and char).

32. Write down the use of bounded type parameters.

Bounded type parameters can be a type parameter with one or more bounds. The bounds restrict the set of types that can be used as type arguments and give access to the methods defined by the bounds.

33. What is meant by Wild Card argument?

A wildcard is a syntactic construct that denotes a family of types. A wildcard describes a family of types. The different types of wildcards are given below

Notation	**Meaning**
<T>	Concrete type
<?>	The unbounded wildcard. It stands for the family of all types
<?super subclass>	A bounded wildcard supertype of T. It stands for the family of all types that are supertypes of Type,type Type being included
<?extends superclass>	A bounded wildcard subtype of T
<U extends T>	U must be a supertype of T

34. What is meant by Bounded Wildcards?

A bounded wildcard specifies either an upper bound or a lower bound for the type argument. This enables you to restrict the types of objects upon which a method will operate. The most common bounded wildcard is the upper bound, which is created using an **extends** clause in much the same way it is used to create a bounded type.

13 Marks Questions

1. What is thread? Explain the life cycle of threads.
2. Write short notes on synchronization.

3. Explain the properties of thread in detail.
4. Explain in detail about generic classes and methods.
5. Write the difference between multithreading and multitasking. What are the ways of creating a thread in Java?
6. Explain various Java functions which support inter-process communication.
7. Write a program to create a thread by implementing Runnable interface.
8. Explain the life cycle of thread and synchronization with a neat diagram.
9. What is multithreading? Write a Java program to create multiple threads.
10. Develop a program to create multiple threads using isAlive() and join method.
11. Explain in detail about generic inheritance and generic interfaces. Discuss exploring the impact of inheritance in generic classes with example.
12. Write about inheritance rules for generic types with example.

15 Marks Questions

1. Define thread. Explain the states of thread briefly. State the reasons for synchronization in thread. Write a simple concurrent programming to create, sleep, and delete the threads.
2. State the motivations of generic programming. Explain the generic classes and methods with example.
3. Write a Java program to display the different combinations of 20-20 cricket teams with team member names using multithreading.
4. Write a Java program that correctly implements producer consumer problem using the concept of inter thread communication
5. How will you resolve deadlock in Interthread communication?