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.

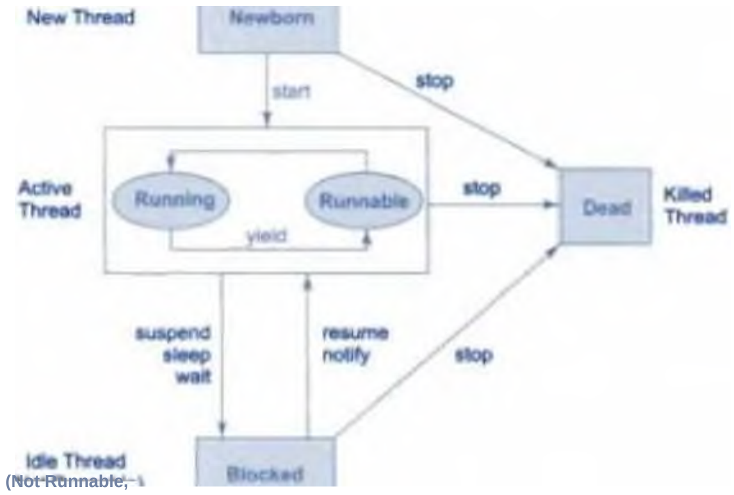

Fig 4.1 Life Cycle of a thread

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

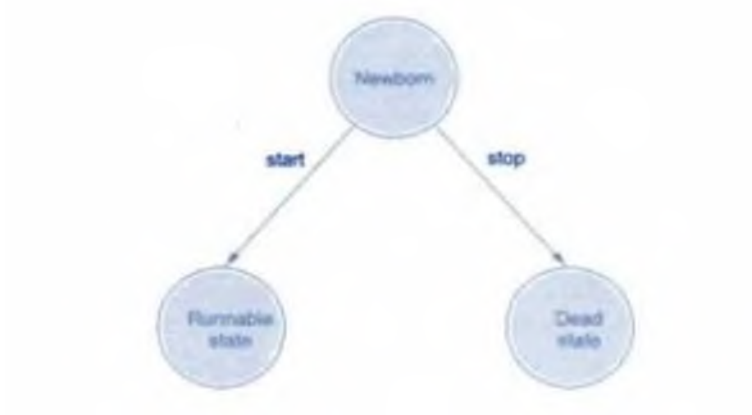

Fig 4.2 Scheduling a Newborn State

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 roundrobin 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.

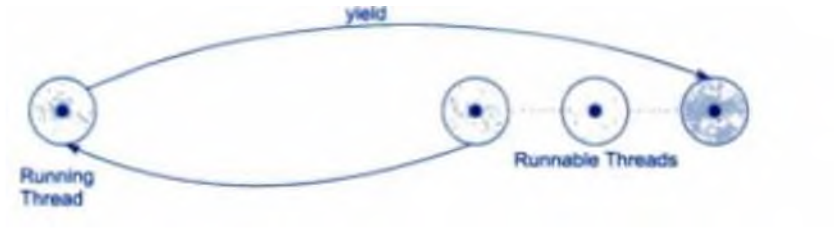

Fig 4.3 Yield() Method

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.

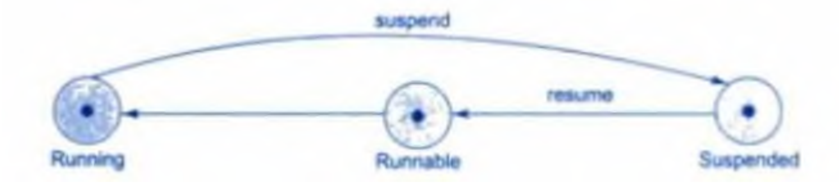

Fig 4.4 suspend() Method b)sleep() method:

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.

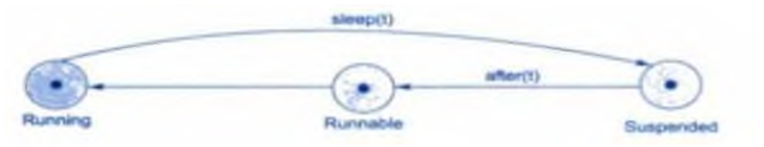

Fig 4.5 sleep() Method c)wait() method:

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.

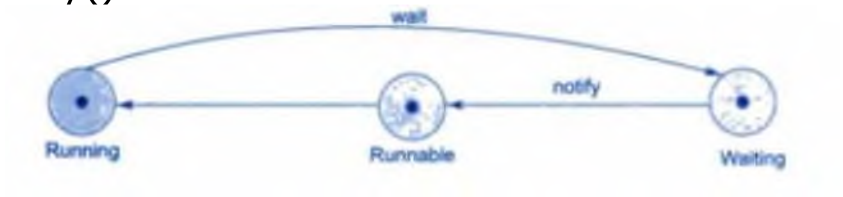

Fig 4.6 wait Method

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.

6 .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
```