**Multithreaded Programming**

Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel.

A thread is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes commands sequentially, In fact, all main programs in our earlier examples can be called single threaded programs. Every program will have at least one thread.

A unique property of Java is its support for multi threading. That is, Java enables us to use multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny program (or module) known as a Thread that runs in parallel to others. A program that contains multiple flows of control is known as multithreaded program.

It is important to remember that threads running in parallel does not really mean that they actually run at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads. The Java interpreter handles the switching of control between the threads in such a way that it appear; they are running concurrently.

Multithreading is. useful in a number of ways, It enables programmer to do multiple things at one time. They can divide a long program (containing Operations that art conceptually concurrent) into threads and execute them in parallel.

Main Thread

Main method module

start start start

switching switching

Thread A Thread B Thread C

The above diagram illustrates a Java program with four threads, one main and three others. The main thread is actually the main method module, which is designed to create and start the other three threads, namely A , B and C.

**Creating Threads**

Creating threads in Java is simple. Threads arc 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 is the only method in which the thread's behavior can beimplemented. A typical run( ) would appear as follows:

public void run( )

{

……………

…………… (statements for implementing thread)

……………

}

The run( ) method should be invoked, by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called start ( ).

A new thread can be created in two ways.

1. **By creating a thread class**: Define a class that extends Thread class and override its run( ) method with the code required by the thread.
2. **By converting a class to a thread**: Define a class that implements Runnable interface. The Runnable interface has only one method, run( ), that is to be defined in the method with the code to be executed by the thread.

**Extending the Thread Class**

We can make our class runnable as thread by extending the class **java.lang.Thread**. This gives us access to all the thread methods directly, It includes the following steps:

1. Define the class as extending the Thread class
2. Implement the run( ) method that is responsible for executing the sequence of code that the thread will execute
3. Create a thread object and call the start( ) method to initiate the thread execution.

**Defining the Class**

The Thread class can be extended as follows:

class MyThread extends Thread

{

……………………

…………………..

}

Now we have a new type of thread MyThread.

**Implementing the *run( )* Method**

The run( ) method has been Inherited by the class MyThread, We have to override this method in order to implement the code to be executed by our thread. The basic implementation of run( ) will look like this.

public void run( )

{

……………

……………. // Thread code here…

}

When we start the new thread, Java calls, the thread's run( ) method, so it is the run( ) where all the action takes place.

**Creating the thread object ( starting new thread)**

To actually create and run an instance of our thread class, we must write the following

MyThread a= new MyThread( );

a.start( ); // invokces run( ) method.

**Example :**

class A extends Thread

{

public void run( )

{

for (int i=1;i<=10;i++)

{

System.out.println("From Thread A...i="+i);

}

System.out.println("Exit from A..");

}

}

class B extends Thread

{

public void run( )

{

for (int j=1;j<=10;j++)

{

System.out.println("From Thread B...j="+j);

}

System.out.println("Exit from B..");

}

}

class ThreadTest

{

public static void main(String args[ ])

{

A a=new A( );

a.start( );

B b=new B( );

b.start( );

}

}

**Stopping and Blocking a Thread**

**Stopping a Thread**

whenever we want to slop a thread from running further, we may do so by calling its stop( ) method, like:

a.stop( );

This statement causes the thread to move to the dead state, A thread will also move to the dead state automatically when it reaches the end of its method. The stop( ) method may be used when the *premature death* of a thread is desired.

**Blocking a Thread**

A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

sleep( ) // blocked for a specified time

suspend( ) //blocked until further Orders

wait( ) //blocked until certain condition occurs

These methods cause the thread to go into the blocked(or not-runnable) state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep( )**, the **resume( )** method is invoked in the case of **suspend()**, and the **notify( )**method is called in the case of **wait()**.

**Life Cycle of a Thread**

During the life time of a thread, there are many states it can enter. They include;

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in the following figure…

Dead

New Born

yield( )

New Thread

stop( )

start( )

Active Thread stop( ) Killed

Thread

sleep( )

suspend( ) resume( ) stop( )

wait( ) notify( )

Blocked

Idle Thread

(Not runnable)

**Newborn State**

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following things with it:

1. Schedule it for running using start( ) method.
2. Kill it using stop( ) method.

**Runnable State**

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is the thread has joined the queue of threads that are waiting for execution, If all threads have equal priority, then they are given time slots for execution in round robin fashion, i.e.,T first-come, first-serve manner. The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as time-slicing.

However, if we want a thread to relinquish control to another thread to equal priority before its turn comes, we can do so by using the yield( ) method.

yield()

Running Thread Runnable Threads

**Running State**

Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running thread may relinquish its control in one of the following situations.

1. It has been suspended using suspend( ) method. A suspended thread can be revived by using the resume() method, This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.

suspend( )

resume( )

Running Runnable Suspended

1. It has been made to sleep. We can put a thread to sleep for a specified time period using the method sleep(time) where time is in milliseconds. This means that the thread is out of the queue during this time period. The thread re-enters the runnable state as soon as this time period is elapsed. sleep(t)

after( t )

Running Runnable Sleeping

1. It been told to wait until some event occurs. This is done using the wait( ) method. The thread can be scheduled to run again using the notify( ) method.

wait( )

notify

Running Runnable Waiting

**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 therefore fully qualified to run again.

**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. However, we can kill it by sending the stop message to it at any state thus causing a premature death to it. A thread can be killed as soon it is born, or while it is running, or even when it is in “not runnable” (blocked) condition.

**Using Thread Methods**

We have discussed how Thread class methods can be used to control the behavior of a thread We have used the methods start( ) and run( ). There are also methods that can move a thread from one state to another . The following program illustrates the use of yield( ), sleep( ) and stop( ) methods.

class A extends Thread

{

public void run( )

{

for (int i= 1; i<=5; i++)

{

if(i ==1)

yield( );

System.out.println("From Thread A : i= " +i);

}

System.out. println("Exit from A..");

}

}

class B extends Thread

{

public void run( )

{

for(int j=1;j<=5; j++)

{

System.out.println("From Thread B : j =" + j);

if(j==3) stop();

}

System.out.println("Exit from B..");

}

}

class C extends Thread

{

public void run( )

{

for(int k=1;k<=5; k++)

{

System.out.println("From Thread C : k =" + k);

if(k==1)

try

{

sleep(1000);

}

catch( Exception e)

{

}

}

System.out.println("Exit from C..");

}

}

class ThreadMethods

{

public static void main(String args[])

{

A a =new A( );

B b=new B( );

C c=new C( );

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

a.start( );

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

b.start( );

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

c.start( );

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

}

}

**Thread Exceptions**

Note that the call to sleep( ) method is enclosed in a try block and followed by a catch block. This is necessary because the sleep( ) method throws an exception, which should be caught, If we fail to catch the exception, program will not compile.

Java run system will throw I**llegalThreadStateException** whenever we attempt to invoke a method that a thread cannot handle in the given state. For example, a sleeping thread cannot deal with the resume( )method because a sleeping thread cannot receive any instructions. The same is true with the suspend( ) method when it is used on a blocked (Not Runnable) thread.

Whenever we call a thread method that is likely to throw an exception, we have to supply an appropriate exception handler to catch it, The catch statement may take one of the following forms

catch (ThreadDeath e)

{

………….

………… // killed thread

}

catch (InterruptedException e)

{

………….

………… // cannot handle it in the current state

}

catch (IllegalArgumentException e)

{

………….

………… // illegal method arguments

}

catch (Exception e)

{

………….

………… // any other

}

**Thread Priority**

In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running. The threads that we have discussed so far are of the same priority. The threads of the same priority are given equal treatment by the Java scheduler and, therefore, they share the processor on a first-come, first-serve basis.

Java permits us to set the priority of a thread using the setPriority( )method as follows:

|  |
| --- |
| 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\_PRIORITY = 1

NORM\_PRIORITY=5

MAX\_PRIORITY = 10

The intNumber may assume one of these constants or any value between 1 and 10, Note that the default setting is NORM\_PRIORITY.

Eg: a.setPriority(Thread.MAX\_PRIORITY);

b.setPriority(a.getPriority( )+1);

where a and b are threads from the previous example…

**Synchronization**

Threads use their own data and methods provided inside their run( ) methods. What happens, when they try to use data and methods outside them selves'? On such occasions, they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as synchronization.

In case of Java, the keyword synchronized helps to solve such problems by keeping a watch on such locations, for example, the method that will read information from a file and the method that will update the same file may be declared as synchronized.

Example

synchronized void update( )

{

……………..

…………….. // code here is synchronised

}

When we declare a method synchronized, Java creates a “**monitor**” and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code, A monitor is like a key and the thread that holds the key can only open the lock.

Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

An interesting situation may occur when two or more threads are waiting to gain control of a resource. Due to some reasons, the condition on which the waiting threads rely on to gain control does not happen. This results in what is known as **DeadLock.**

**Implementing the “Runnable”Interface**

The Runnable interface declares the run( ) method that is required for implementing threads in our programs. To do this, we must perform the steps listed below:

1. Declare the class as implementing the Runnable interface.
2. Implement the run( ) method.
3. Create a thread by defining an object that is instantiated from this “runnable" class as the target of the thread.
4. Call the thread's start( ) method to run the thread.

Example :

class Example implements Runnable

{

public void run( )

{

for(int i=1; i<=10; i++)

{

System.out.println("Thread X : "+i);

}

System.out.println("End of Thread X " );

}

}

class RunnableTest

{

public static void main (String args[ ])

{

Example e = new Example( );

Thread threadE =new Thread(e);

threadE.start( );

System.out.println("End of main Thread");

}

}