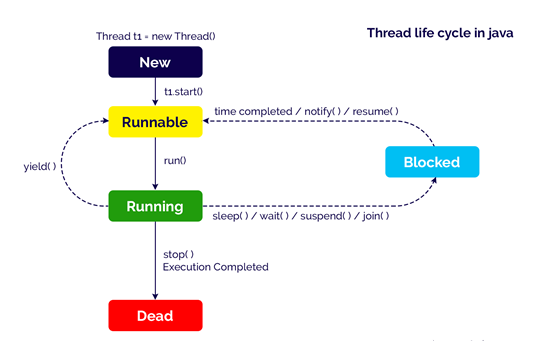
**Differences between thread-based multitasking and process-based multitasking**

|  |  |
| --- | --- |
| MULTIPROCESSING | MULTITHREADING |
| Heavy weight process | Light weight process |
| Every process has its own memory space | threads of a process share the memory of the parent process |
| Inter-process communication takes more time | Inter-thread communication takes less time |
| Context-switching is tedious | context-switching is easy |
| It allows the computer to run two or more programs concurrently | It allows the computer to run two or more threads concurrently |
| In this process is the smallest unit. | In this thread is the smallest unit. |

**Java thread model**

In java, a thread goes through different states throughout its execution. These stages are called thread life cycle states or phases. A thread may in any of the states like new, ready or runnable, running, blocked or wait, and dead or terminated state. The life cycle of a thread in java is shown in the following figure.



New

* When a thread object is created using new, then the thread is said to be in the New state.
* This state is also known as Born state.

Example

Thread t1 = new Thread();

Runnable / Ready

* When a thread calls start( ) method, then the thread is said to be in the Runnable state.
* This state is also known as a Ready state.

Example

t1.start( );

Running

* When a thread calls run( ) method, then the thread is said to be Running.
* The run( ) method of a thread called automatically by the start( ) method.

Blocked / Waiting

* A thread in the Running state may move into the blocked state due to various reasons like sleep( ) method called, wait( ) method called, suspend( ) method called, and join( ) method called, etc.
* When a thread is in the blocked or waiting state, it may move to Runnable state due to reasons like sleep time completed, waiting time completed, notify( ) or notifyAll( ) method called, resume( ) method called, etc.

Dead / Terminated

* A thread in the Running state may move into the dead state due to either its execution completed or the stop( ) method called.
* The dead state is also known as the terminated state.

**creating threads**

* In java, a thread is a lightweight process.
* Every java program executes by a thread called the main thread.
* When a java program gets executed, the main thread created automatically.
* All other threads called from the main thread.
* The java programming language provides two methods to create threads, and they are listed below.
  1. Using Thread class (by extending Thread class)
  2. Using Runnable interface (by implementing Runnable interface)

**Extending Thread class**

* The java contains a built-in class Thread inside the java.lang package. The Thread class contains all the methods that are related to the threads.
* To create a thread using Thread class, follow the step given below.

**Step-1**: Create a class as a child of Thread class. That means, create a class that extends Thread class.

**Step-2**: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

**Step-3**: Create the object of the newly created class in the main( ) method.

//CREATION OF THREADS USING THREAD CLASS

class ThreadOne extends Thread

{

ThreadOne( )

{

start();

}

public void run()

{

try

{

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

{

System.out.println("child"+i);

Thread.sleep(500);

}

}

catch(InterruptedException e)

{

System.out.println("child one interrupted");

}

}

}

class ThreadDemo1

{

public static void main(String args[])

{

new ThreadOne();

new ThreadOne();

new ThreadOne();

try

{

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

{

System.out.println("Parent thread"+i);

Thread.sleep(1000);

}

}

catch(InterruptedException e)

{

System.out.println("child one interrupted");

}

}}

The Thread class has the following consructors.

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

The Thread classs contains the following methods.

| **Method** | **Description** | **Return Value** |
| --- | --- | --- |
| run( ) | Defines actual task of the thread. | void |
| start( ) | It moves thre thread from Ready state to Running state by calling run( ) method. | void |
| setName(String) | Assigns a name to the thread. | void |
| getName( ) | Returns the name of the thread. | String |
| setPriority(int) | Assigns priority to the thread. | void |
| getPriority( ) | Returns the priority of the thread. | int |
| getId( ) | Returns the ID of the thread. | long |
| activeCount() | Returns total number of thread under active. | int |
| currentThread( ) | Returns the reference of the thread that currently in running state. | void |
| sleep( long ) | moves the thread to blocked state till the specified number of milliseconds. | void |
| isAlive( ) | Tests if the thread is alive. | boolean |
| yield( ) | Tells to the scheduler that the current thread is willing to yield its current use of a processor. | void |
| join( ) | Waits for the thread to end. | void |

## Implementing Runnable interface

* The java contains a built-in interface Runnable inside the java.lang package.

Step-1: Create a class as a child of Runnable interface. That means, create a class that implements that interface.

Step-2: create a object of Thread class by passing the object of this class as parameter. Using that thread object , call start() which will invoke the run().

Step-3: Override the run( ) method with the code that is to be executed by the thread. The run( ) method must be public while overriding.

Step-4: Create the object of the newly created class in the main( ) method.

//CREATION OF THREADS USING RUNNABLE INTERFACE

class ThreadThree implements Runnable

{

Thread t;

ThreadThree( )

{

t=new Thread(this,"child");

t.start();

}

public void run()

{

try

{

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

{

System.out.println("child"+i);

Thread.sleep(500);

}

}

catch(InterruptedException e)

{

System.out.println("child one interrupted");

}

}

}

class ThreadDemo3

{

public static void main(String args[])

{

new ThreadThree();

new ThreadThree();

new ThreadThree();

try

{

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

{

System.out.println("Parent thread"+i);

Thread.sleep(1000);

}

}

catch(InterruptedException e)

{

System.out.println("child one interrupted");

}

}}

**Creation of Multiple Threads**

class ThreadOne extends Thread

{

String cname;int sleeptime;String msg;

ThreadOne( String chname,int time,String msg1)

{

cname=chname;

sleeptime=time;

msg=msg1;

start();

}

public void run()

{

try

{

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

{

System.out.println(cname+"is running :"+msg);

Thread.sleep(sleeptime);

}

}

catch(InterruptedException e)

{

System.out.println("child one interrupted");

}

}

}

class ThreadDemo2

{

public static void main(String args[])

{

new ThreadOne("child1",1000,"Good Morning");

new ThreadOne("child2",5000,"Hello");

new ThreadOne("child3",10000,"Welcome");

try

{

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

{

System.out.println("Parent thread"+i);

Thread.sleep(1000);

}

}

catch(InterruptedException e)

{

System.out.println("child one interrupted");

}

}}

**Thread priorities:**

* In a java programming language, every thread has a property called priority.
* Most of the scheduling algorithms use the thread priority to schedule the execution sequence.
* In java, the thread priority range from 1 to 10.
* Priority 1 is considered as the lowest priority, and priority 10 is considered as the highest priority.
* The thread with more priority allocates the processor first.
* The java programming language Thread class provides two methods setPriority(int), and getPriority( ) to handle thread priorities.

The Thread class also contains three constants that are used to set the thread priority, and they are listed below.

* MAX\_PRIORITY - It has the value 10 and indicates highest priority.
* NORM\_PRIORITY - It has the value 5 and indicates normal priority.
* MIN\_PRIORITY - It has the value 1 and indicates lowest priority.
* The default priority of any thread is 5 (i.e. NORM\_PRIORITY).

**setPriority( ) method**

* The setPriority( ) method of Thread class used to set the priority of a thread.
* It takes an integer range from 1 to 10 as an argument and returns nothing (void).

Example

threadObject.setPriority(4);

or

threadObject.setPriority(MAX\_PRIORITY);

**getPriority( ) method**

* The getPriority( ) method of Thread class used to access the priority of a thread.

Example

int n = threadObject.getPriority();

**//PROGRAM**

//PROGRAM TO ILLUSTRATE CREATION OF MULTIPLE THREADS with priorities

class Threadone extends Thread

{

public void run()

{

try

{

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

{

System.out.println("child1: "+i);

Thread.sleep(1000); }}

catch(InterruptedException e)

{

System.out.println("child one interrupted");

}}}

class Threadtwo extends Thread

{

public void run()

{ try

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

{ System.out.println("child2: "+i);

Thread.sleep(1000);

}}

catch(InterruptedException e)

{System.out.println("child one interrupted");

}}}

class Threadthree extends Thread

{public void run()

{ try

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

{ System.out.println("child3: "+i);

Thread.sleep(1000);}}

catch(InterruptedException e)

{System.out.println("child one interrupted");

}}}

class ThreadPriorityDemo

{ public static void main(String args[])

{

Threadone a=new Threadone();

a.setPriority(Thread.MIN\_PRIORITY);

System.out.println(a.getPriority());

a.start();

Threadtwo b=new Threadtwo();

b.setPriority(7);

b.start();

Threadthree c=new Threadthree();

c.setPriority((Thread.MAX\_PRIORITY));

c.start();

try{

Thread.sleep(5000);

}

catch(InterruptedException e)

{ System.out.println("main interrupted");

}

System.out.println("main terminated");

}}

**synchronizing threads:**

* The java programming language supports multithreading.
* The problem of shared resources occurs when two or more threads get execute at the same time.
* In such a situation, we need some way to ensure that the shared resource will be accessed by only one thread at a time, and this is performed by using the concept called synchronization.
* The synchronization is the process of allowing only one thread to access a shared resource at a time.
* In java, the synchronization is achieved using the following concepts.
* Mutual Exclusion
* Inter thread communication

**Mutual Exclusion**

Using the mutual exclusion process, only one thread of multiple threads is given access while accessing the shared resource. In java, mutual exclusion is achieved using the following concepts.

* Synchronized method
* Synchronized block

### Synchronized method

* When a method created using a synchronized keyword, it allows only one object to access it at a time.
* When an object calls a synchronized method, it put a lock on that method so that other objects or thread that are trying to call the same method must wait, until the lock is released.
* Once the lock is released on the shared resource, one of the threads among the waiting threads will be allocated to the shared resource.

//PROGRAM TO ILLUSTRATE THREAD SYNCHRONIZATION

class Callme

{

synchronized void call(String msg)

{

System.out.print("["+msg);

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{}

System.out.println("]");

}}

class Caller implements Runnable

{

String msg;

Callme target;

Thread t;

public Caller(Callme targ,String s)

{

target=targ;

msg=s;

t=new Thread(this);

t.start();

}

public void run()

{

target.call(msg);

}}

class Synch

{

public static void main(String args[])

{

Callme target=new Callme();

Caller ob1=new Caller(target,"HELLO");

Caller ob2=new Caller(target,"SYNCHRONIZED");

Caller ob3= new Caller(target,"WORLD");

try

{

ob1.t.join();

ob2.t.join();

ob3.t.join();

}

catch(InterruptedException e){}

}}

### Synchronized block

* Instead of synchronizing the method definition , we can synchronize method call on a particular object.

//PROGRAM TO ILLUSTRATE THREAD SYNCHRONIZATION (Synchronized block)

class Callme

{

void call(String msg)

{

System.out.println("["+msg);

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{}

System.out.println("]");

}}

class Caller implements Runnable

{

String msg;

Callme target;

Thread t;

public Caller(Callme targ,String s)

{

target=targ;

msg=s;

t=new Thread(this);

t.start();

}

public void run()

{

synchronized(target){

target.call(msg);}

}}

class Synch2

{

public static void main(String args[])

{

Callme target=new Callme();

Caller ob1=new Caller(target,"HELLO");

Caller ob2=new Caller(target,"SYNCHRONIZED");

Caller ob3= new Caller(target,"WORLD");

try

{

ob1.t.join();

ob2.t.join();

ob3.t.join();

}

catch(InterruptedException e){}

}}

# Inter Thread Communication

* Inter thread communication is the concept where two or more threads communicate to solve the problem of polling.
* In java, polling is the situation to check some condition repeatedly, to take appropriate action, once the condition is true.
* That means, in inter-thread communication, a thread waits until a condition becomes true such that other threads can execute its task.
* The inter-thread communication allows the synchronized threads to communicate with each other.

Java provides the following methods to achieve inter thread communication.

* wait( )
* notify( )
* notifyAll( )

| **Method** | **Description** |
| --- | --- |
| **void wait( )** | It makes the current thread to pause its execution until other thread in the same monitor calls notify( ) |
| **void notify( )** | It wakes up the thread that called wait( ) on the same object. |
| **void notifyAll()** | It wakes up all the threads that called wait( ) on the same object. |

PRODUCER-CONSUMER problem is the example to illustrate inter-thread communication.

* The producer produces the item and the consumer consumes the same.
* But here, the consumer can not consume until the producer produces the item, and producer can not produce until the consumer consumes the item that already been produced.
* So here, the consumer has to wait until the producer produces the item, and the producer also needs to wait until the consumer consumes the same.
* Here we use the inter-thread communication to implement the producer and consumer problem.

//PROGRAM TO ILLUSTRATE PRODUCER-CONSUMER PROBLEM

//INTER-THREAD COMMUNICATION

class Q

{

int n;

boolean valueset=false;

synchronized int get()

{

if(!valueset)

{

try

{

wait();

}

catch(InterruptedException e){}

}

System.out.println("GOT: " +n);

valueset=false;

notify();

return n;

}

synchronized void put(int n)

{

if(valueset)

{

try

{

wait();

}

catch(InterruptedException e){}

}

this.n=n;

valueset=true;

System.out.println("put:"+n);

notify();

}}

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();

}}

public class PC

{

public static void main(String args[])

{

Q q=new Q();

new Producer(q);

new Consumer(q);

System.out.println("Press ctrl+c to stop");

}}