

Multithreading

Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A **process** consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process.

There are two distinct types of **Multitasking** i.e. Processor-Based and Thread-Based multitasking.

Q: What is the difference between thread-based and process-based multitasking?

Ans: As both are types of multitasking there is very basic difference between the two. **Process-Based multitasking** is a feature that allows your computer to run two or more programs concurrently. For example you can listen to music and at the same time chat with your friends on Facebook using browser. In Thread-based multitasking, thread is the smallest unit of code, which means a single program can perform two or more tasks simultaneously. For example a text editor can print and at the same time you can edit text provided that those two tasks are performed by separate threads.

Q: Why multitasking thread requires less overhead than multitasking processor?

Ans: A multitasking thread requires less overhead than multitasking processor because of the following reasons:

- Processes are heavyweight tasks where threads are lightweight
- Processes require their own separate address space where threads share the address space
- Interprocess communication is expensive and limited where Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.

Benefits of Multithreading

1. Enables programmers to do multiple things at one time

2. Programmers can divide a long program into threads and execute them in parallel which eventually increases the speed of the program execution
3. Improved performance and concurrency
4. Simultaneous access to multiple applications

Life Cycle of Thread

A thread can be in any of the five following states

1. **Newborn State:** When a thread object is created a new thread is born and said to be in Newborn state.
2. **Runnable State:** If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor. If all threads in queue are of same priority then they are given time slots for execution in round robin fashion
3. **Running State:** It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs
 - a. Thread give up its control on its own and it can happen in the following situations
 - i. A thread gets suspended using **suspend()** method which can only be revived with **resume()** method
 - ii. A thread is made to sleep for a specified period of time using **sleep(time)** method, where time in milliseconds
 - iii. A thread is made to wait for some event to occur using **wait ()** method. In this case a thread can be scheduled to run again using **notify ()** method.
 - b. A thread is pre-empted by a higher priority thread
4. **Blocked State:** If a thread is prevented from entering into runnable state and subsequently running state, then a thread is said to be in Blocked state.
5. **Dead State:** A runnable thread enters the Dead or terminated state when it completes its task or otherwise terminates.

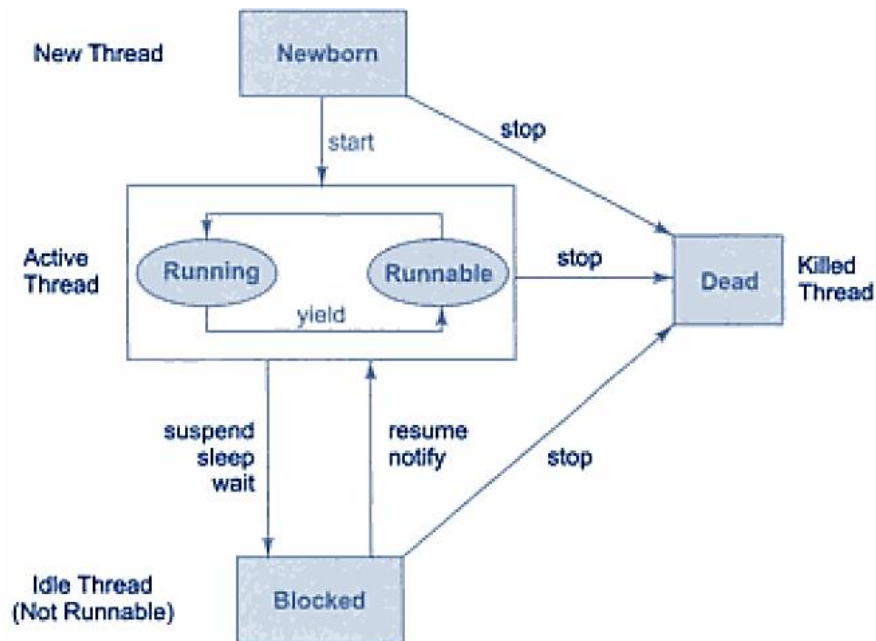


Fig: Life Cycle of Thread

Main Thread

Every time a Java program starts up, one thread begins running which is called as the main thread of the program because it is the one that is executed when your program begins.

- Child threads are produced from main thread
- Often it is the last thread to finish execution as it performs various shut down operations

Creating a Thread

Java defines two ways in which this can be accomplished:

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

Create Thread by Implementing Runnable

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

```
public void run()
```

You will define the code that constitutes the new thread inside **run()** method. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can.

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);

Here threadOb is an instance of a class that implements the Runnable interface and 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. The start() method is shown here:

void start();

Example to Create a Thread using Runnable Interface

```
class t1 implements Runnable
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t1 obj1 = new t1();
        Thread t = new Thread(obj1);
        t.start();
    }
}
```

Output:

```
C:\NIEC Java>javac t1.java
C:\NIEC Java>java t1
Thread is Running
C:\NIEC Java>
```

Create Thread by 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.

Example to Create a Thread by Extending Thread Class

```
class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
    }
}
```

Output:

```
C:\NIEC Java>javac t2.java
C:\NIEC Java>java t2
Thread is Running
C:\NIEC Java>
```

Thread Methods

SN	Methods with Description
1	public void start() Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	public void run() If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	public final void setName(String name) Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	public final void setPriority(int priority) Sets the priority of this Thread object. The possible values are between 1 and 10.
5	public final void setDaemon(boolean on) A parameter of true denotes this Thread as a daemon thread.
6	public final void join(long millisec) The current thread invokes this method on a second thread, causing the current thread

	to block until the second thread terminates or the specified number of milliseconds passes.
7	public void interrupt() Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	public final boolean isAlive() Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

Q: Can we start a thread twice?

Ans: No, if a thread is started it can never be started again, if you do so, an `illegalThreadStateException` is thrown. Example is shown below in which a same thread is coded to start again

```
class t2 extends Thread
{
    public void run()
    {
        System.out.println("Thread is Running");
    }
    public static void main(String args[])
    {
        t2 obj1 = new t2();
        obj1.start();
        obj1.start();
    }
}
```

As you can see two statements to start a same thread is written in the code which will not give error during compilation but when you run it you can see an Exception as shown in the Output Screenshot.

Output:

```
C:\NIEC Java>javac t2.java
C:\NIEC Java>java t2
Thread is Running
Exception in thread "main" java.lang.IllegalThreadStateException
    at java.lang.Thread.start(Unknown Source)
    at t2.main(t2.java:11)
```

Use of Yield() Method

Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled

Example

```

class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            if(i==2) yield();
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
}

class B extends Thread
{
    public void run()
    {
        for(int j=1; j<=5; j++)
        {
            System.out.println("B:" +j);
        }
        System.out.println("Exit from B");
    }
}

class yieldtest
{
    public static void main(String args[])
    {
        A a = new A();
        B b = new B();
        a.start();
        b.start();
    }
}

```

Condition is checked and when i==2 yield() method is evoked taking control to thread B

As you can see in the output below, thread A gets started and when condition if(i==2) gets satisfied yield() method gets evoked and the control is relinquished from thread A to thread B which run to its completion and only after that thread a regain the control back.

Output

```
C:\NIEC Java>javac yieldtest.java
C:\NIEC Java>java yieldtest
A:1
B:1
B:2
B:3
B:4
B:5
Exit from B
A:2
A:3
A:4
A:5
Exit from A
```

Use of stop() Method

The stop() method kills the thread on execution

Example

```
class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            if(i==2) stop();
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
    public static void main(String args[])
    {
        C c = new C();
        c.start();
    }
}
```

Condition is checked and when i==2 stop() method is evoked causing termination of thread execution

Output

```
C:\NIEC Java>java C
A:1
```

Use of sleep() Method

Causes the currently running thread to block for at least the specified number of milliseconds. You need to handle exception while using sleep() method.

Example

```

class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            try
            {
                if(i==2) sleep(1000);
            }
            catch(Exception e)
            {
            }
            System.out.println("A: " +i);
        }
        System.out.println("Exit from A");
    }
    public static void main(String args[])
    {
        C c = new C();
        c.start();
    }
}

```

Condition is checked and when i==2 sleep() method is evoked which halts the execution of the thread for 1000 milliseconds. When you see output there is no change but there is delay in execution.

Output

```

C:\NIEC Java>javac C.java
C:\NIEC Java>java C
A:1
A:2
A:3
A:4
A:5
Exit from A

```

Use of suspend() and resume() 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.

Following is the example in which two threads C and A are created. Thread C is started ahead of Thread A, but C is suspended using suspend() method causing Thread A to get hold of the processor allowing it to run and when Thread C is resumed using resume() method it runs to its completion.

Example

```

class C extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("C:" +i);
        }
        System.out.println("Exit from C");
    }
}

class A extends Thread
{
    public void run()
    {
        for(int i=1; i<=5; i++)
        {
            System.out.println("A:" +i);
        }
        System.out.println("Exit from A");
    }
}

class suspendtest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        c.start();
        a.start();
        c.suspend();
        c.resume();
    }
}

```

Although Thread 'C' is started earlier than Thread 'A' but due to suspend method Thread 'A' gets completed ahead of Thread 'C'

Output

```

C:\Achin Jain>java suspendtest
A:1
A:2
A:3
A:4
A:5
Exit from A
C:1
C:2
C:3
C:4
C:5
Exit from C

```

Thread Priority

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between **MIN_PRIORITY (a constant of 1)** and **MAX_PRIORITY (a constant of 10)**. By default, every thread is given priority **NORM_PRIORITY (a constant of 5)**.

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Example

```
class prioritytest
{
    public static void main(String args[])
    {
        C c = new C();
        A a = new A();
        a.setPriority(10);
        c.setPriority(1);
        c.start();
        a.start();
    }
}
```

In the above code, you can see Priorities of Thread is set to maximum for Thread A which lets it to run to completion ahead of C which is set to minimum priority.

Output:

```
C:\Achin Jain>java prioritytest
A:1
A:2
A:3
A:4
A:5
Exit from A
C:1
C:2
C:3
C:4
C:5
Exit from C
```

Use of isAlive() and join() method

The java.lang.Thread.isAlive() method tests if this thread is alive. A thread is alive if it has been started and has not yet died. Following is the declaration for java.lang.Thread.isAlive() method

public final boolean isAlive()

This method returns true if this thread is alive, false otherwise.

join() method waits for a thread to die. It causes the currently thread to stop executing until the thread it joins with completes its task.

Example

```
class A extends Thread
{
    public void run()
    {
        System.out.println("Status:" + isAlive());
    }
}
class alivetest
{
    public static void main(String args[])
    {
        A a = new A();
        a.start();
        try
        {
            a.join();
        }
        catch (InterruptedException e)
        {}
        System.out.println("Status:" + a.isAlive());
    }
}
```

At this point Thread A is alive so the value gets printed by **isAlive()** method is "**true**"

join() method is called from Thread A which stops executing of further statement until A is Dead

Now isAlive() method returns the value false as the Thread A is complete

Output

```
C:\Achin Jain>java alivetest
Status:true
Status:false
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called thread synchronization. The synchronized keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

synchronized(object)

```
{  
    // statements to be synchronized  
}
```

Problem without using Synchronization

In the following example method updatesum() is not synchronized and access by both the threads simultaneously which results in inconsistent output. Making 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 the code. Writing the method as synchronized will make one thread enter the method and till execution is not complete no other thread can get access to the method.

```
synchronized void updatesum(int i)  
{  
    Thread t = Thread.currentThread();  
    for(int n=1; n<=5; n++)  
    {  
        System.out.println(t.getName()+" : "+(i+n));  
    }  
}
```

```

class update
{
    void updatesum(int i)
    {
        Thread t = Thread.currentThread();
        for(int n=1; n<=5; n++)
        {
            System.out.println(t.getName()+" : "+(i+n));
        }
    }
}

class A extends Thread
{
    update u = new update();
    public void run()
    {
        u.updatesum(10);
    }
}

class syntest
{
    public static void main(String args[])
    {
        A a = new A();
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(a);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
    }
}
    
```

Diagram illustrating the synchronization of the `updatesum` method:

- The original method signature `void updatesum(int i)` is shown in a red box.
- An arrow points to the modified method signature `synchronized void updatesum(int i)`, also in a red box.
- A red arrow points from the modified signature to the output window.

Output when method is declared as synchronized

```

C:\Achin Jain>java syntest
Thread A : 11
Thread A : 12
Thread A : 13
Thread A : 14
Thread A : 15
Thread B : 11
Thread B : 12
Thread B : 13
Thread B : 14
Thread B : 15
    
```

Output

```

C:\Achin Jain>java syntest
Thread A : 11
Thread B : 11
Thread A : 12
Thread B : 12
Thread A : 13
Thread B : 13
Thread A : 14
Thread B : 14
Thread A : 15
Thread B : 15
    
```

Interthread Communication

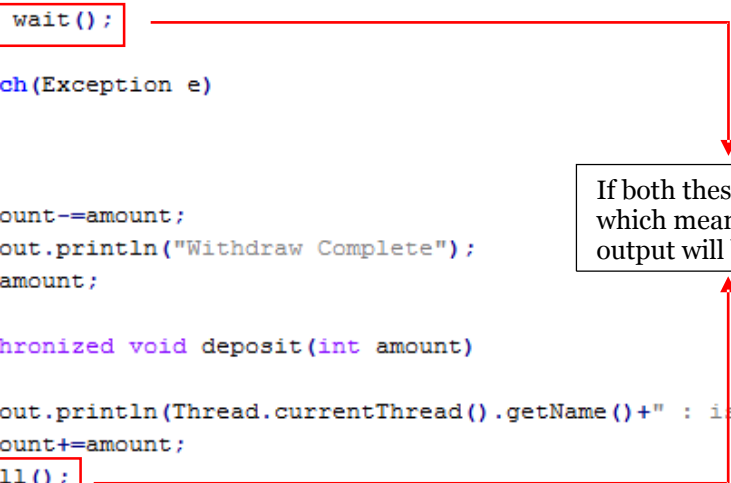
It is all about making synchronized threads communicate with each other. It is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter in the same critical section to be executed. It is implemented by the following methods of Object Class:

- **wait() :** This method 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() :** This method wakes up the first thread that called wait() on the same object.
- **notifyAll() :** This method wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

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.

Example

```
class customer
{
int amount = 0;
int flag = 0;
public synchronized int withdraw(int amount)
{
    System.out.println(Thread.currentThread().getName()+" : is going to withdraw");
    if(flag==0)
    {
        try
        {
            System.out.println("Waiting...");
            wait();
        }
        catch(Exception e)
        {
        }
    }
    this.amount-=amount;
    System.out.println("Withdraw Complete");
    return amount;
}
public synchronized void deposit(int amount)
{
    System.out.println(Thread.currentThread().getName()+" : is going to Deposit");
    this.amount+=amount;
    notifyAll();
    System.out.println("Deposit Complete");
    flag=1;
}
}
```



If both these methods are commented which means there is no communication, output will be inconsistent. See [Output 2](#)

```

class threadcomm
{
    public static void main(String args[])
    {
        final customer c = new customer();
        Thread t1 = new Thread()
        {
            public void run()
            {
                c.withdraw(5000);
                System.out.println("After withdraw Amount is :"+ c.amount);
            }
        };
        Thread t2 = new Thread()
        {
            public void run()
            {
                c.deposit(10000);
                System.out.println("After Deposit Amount is :"+ c.amount);
            }
        };
        t1.start();
        t2.start();
    }
}

```

Output 1:

```

C:\Achin Jain>java threadcomm
Thread-0 : is going to withdraw
Waiting....
Thread-1 : is going to Deposit
Deposit Complete
After Deposit Amount is :10000
Withdraw Complete
After withdraw Amount is :5000

```

Output 2:

```

C:\Achin Jain>java threadcomm
Thread-0 : is going to withdraw
Waiting....
Withdraw Complete
After withdraw Amount is :-5000
Thread-1 : is going to Deposit
Deposit Complete
After Deposit Amount is :5000

```