



CORE JAVA

By : Kalpesh Chauhan



MULTITHREADING IN JAVA

INTRODUCTION

Multithreading in java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

ADVANTAGES OF JAVA MULTITHREADING

- It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
- You **can perform many operations together, so it saves time.**
- Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

MULTITASKING

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

PROCESS-BASED MULTITASKING (MULTIPROCESSING)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

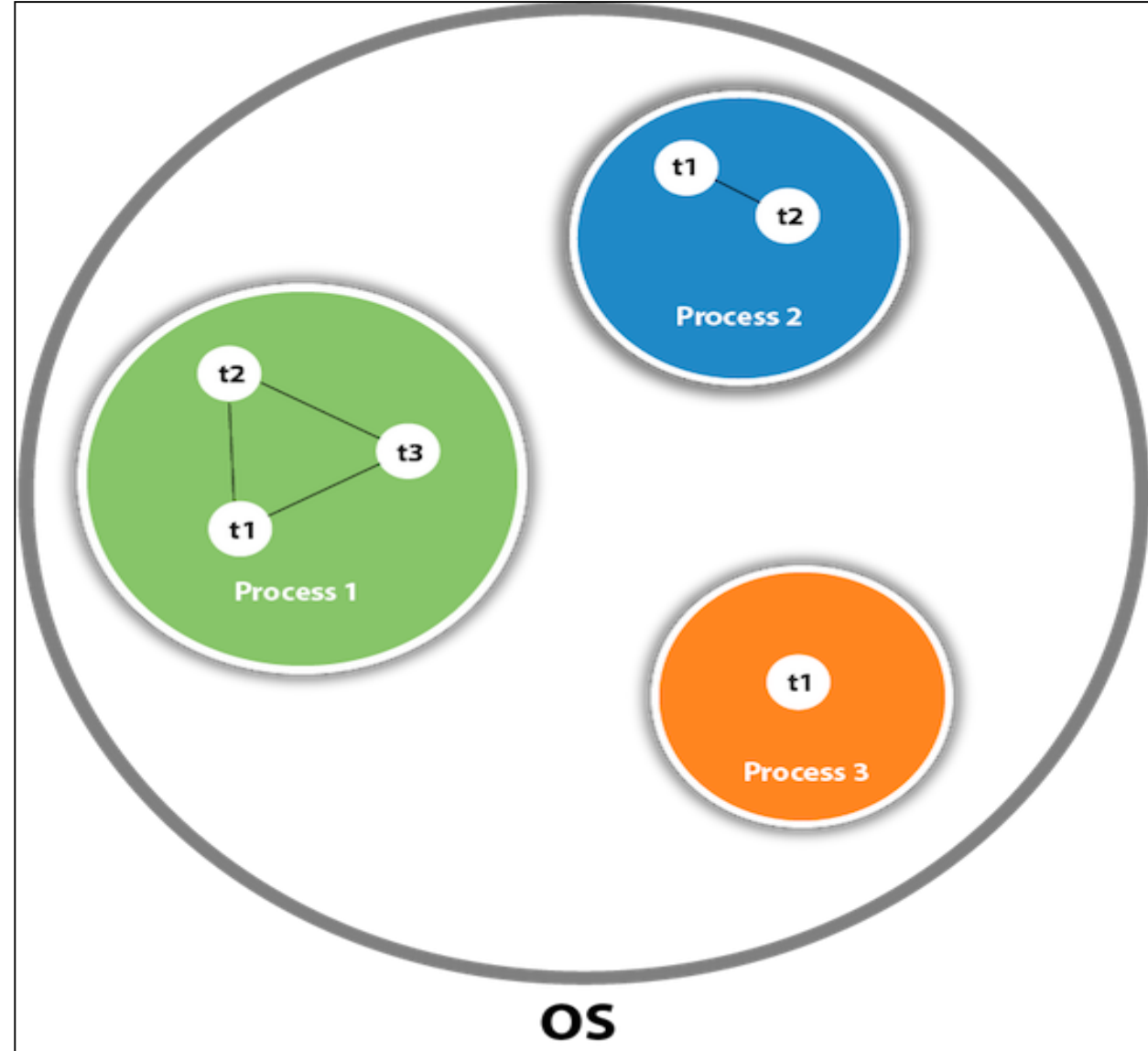
THREAD-BASED MULTITASKING (MULTITHREADING)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

WHAT IS THREAD IN JAVA

- A thread is a lightweight subprocess, the 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 uses a shared memory area.

As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.



LIFE CYCLE OF A THREAD (THREAD STATES)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:



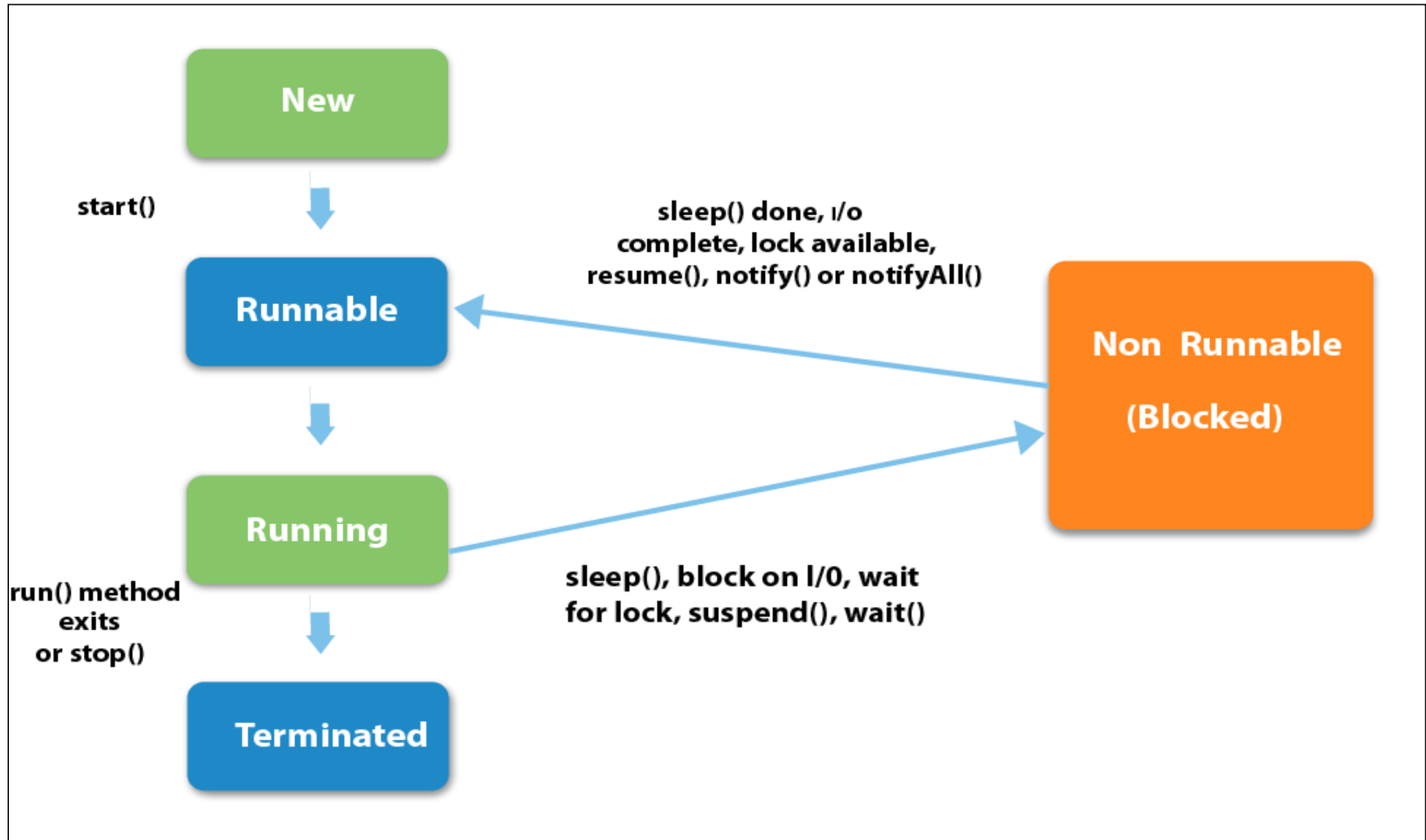
➤ New

➤ Runnable

➤ Running

➤ Non-Runnable (Blocked)

➤ Terminated



New

The thread is in new state if you create an instance of Thread class but before the invocation of **start()** method.

Runnable

The thread is in runnable state after invocation of **start()** method, but the thread scheduler has not selected it to be the running thread.

Running

The thread is in running state if the thread scheduler has selected it.

Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

Terminated

A thread is in terminated or dead state when its **run()** method exits.

HOW TO CREATE THREAD

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

THREAD CLASS:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

COMMONLY USED CONSTRUCTORS OF THREAD CLASS:

`Thread()`

`Thread(String name)`

`Thread(Runnable r)`

`Thread(Runnable r,String name)`

RUNNABLE INTERFACE:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

public void run(): is used to perform action for a thread.

STARTING A THREAD:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

1. A new thread starts(with new callstack).
2. The thread moves from New state to the Runnable state.
3. When the thread gets a chance to execute, its target run() method will run.

JAVA THREAD EXAMPLE BY EXTENDING THREAD CLASS

```
class Multi extends Thread{  
    public void run(){  
        System.out.println("thread is running...");  
    }  
  
    public static void main(String args[]){  
        Multi t1=new Multi();  
        t1.start();  
    }  
}
```

JAVA THREAD EXAMPLE BY IMPLEMENTING RUNNABLE INTERFACE

```
class Multi3 implements Runnable{  
    public void run(){  
        System.out.println("thread is running...");  
    }  
  
    public static void main(String args[]){  
        Multi3 m1=new Multi3();  
        Thread t1 =new Thread(m1);  
        t1.start();  
    }  
}
```

THREAD SCHEDULER IN JAVA

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

DIFFERENCE BETWEEN PREEMPTIVE SCHEDULING AND TIME SLICING

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

SLEEP METHOD IN JAVA

The `sleep()` method of `Thread` class is used to sleep a thread for the specified amount of time.

The `Thread` class provides two methods for sleeping a thread:

1. `public static void sleep(long milliseconds)throws InterruptedException`
2. `public static void sleep(long milliseconds, int nanos)throws InterruptedException`

EXAMPLE OF SLEEP METHOD IN JAVA

```
class TestSleepMethod1 extends Thread{  
    public void run(){  
        for(int i=1;i<5;i++){  
            try{  
                Thread.sleep(500);  
            }catch(InterruptedException e){  
                System.out.println(e);  
            }  
            System.out.println(i);  
        }  
    }  
}
```



```
public static void main(String args[])
{
    TestSleepMethod1 t1=new TestSleepMethod1();
    TestSleepMethod1 t2=new TestSleepMethod1();

    t1.start();
    t2.start();
}
}
```



As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on

CAN WE START A THREAD TWICE

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

```
public class TestThreadTwice1 extends Thread{
    public void run(){
        System.out.println("running...");
    }
    public static void main(String args[]){
        TestThreadTwice1 t1=new TestThreadTwice1();
        t1.start();
        t1.start();
    }
}
```

WHAT IF WE CALL RUN() METHOD DIRECTLY INSTEAD START() METHOD?

Each thread starts in a separate call stack.

Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

THE JOIN() METHOD

The `join()` method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

```
public void join()throws InterruptedException
```

```
public void join(long milliseconds)throws InterruptedException
```

GETNAME(),SETNAME(String) AND GETID() METHOD:

```
class TestJoinMethod3 extends Thread{  
    public void run(){  
        System.out.println("running...");  
    }  
    public static void main(String args[]){  
        TestJoinMethod3 t1=new TestJoinMethod3();  
        TestJoinMethod3 t2=new TestJoinMethod3();  
        System.out.println("Name of t1:"+t1.getName());  
        System.out.println("Name of t2:"+t2.getName());  
        System.out.println("id of t1:"+t1.getId());  
    }  
}
```



```
t1.start();
```

```
t2.start();
```

```
t1.setName("Karan Ayan");
```

```
System.out.println("After changing name of t1:"+t1.getName());
```

```
}
```

```
}
```


THE CURRENTTHREAD() METHOD:


The `currentThread()` method returns a reference to the currently executing thread object.

```
class TestJoinMethod4 extends Thread
{
    public void run(){
        System.out.println(Thread.currentThread().getName());
    }
}

public static void main(String args[]){
    TestJoinMethod4 t1=new TestJoinMethod4();
    TestJoinMethod4 t2=new TestJoinMethod4();
    t1.start();
    t2.start();
}
}
```



PRIORITY OF A THREAD (THREAD PRIORITY):



Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 CONSTANTS DEFINED IN THREAD CLASS

1. `public static int MIN_PRIORITY`
2. `public static int NORM_PRIORITY`
3. `public static int MAX_PRIORITY`

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

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.

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.

POINTS TO REMEMBER FOR 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.

WHY JVM TERMINATES THE DAEMON THREAD IF THERE IS NO USER THREAD?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread

METHODS FOR JAVA DAEMON THREAD BY THREAD CLASS

The `java.lang.Thread` class provides two methods for java daemon thread.

No.	Method	Description
1)	<code>public void setDaemon(boolean status)</code>	is used to mark the current thread as daemon thread or user thread.
2)	<code>public boolean isDaemon()</code>	is used to check that current is daemon.

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

JAVA THREAD POOL

Java Thread pool represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

ADVANTAGE OF JAVA THREAD POOL

Better performance It saves time because there is no need to create new thread.

Real time usage

It is used in Servlet and JSP where container creates a thread pool to process the request.

EXAMPLE OF JAVA THREAD POOL

Let's see a simple example of java thread pool using `ExecutorService` and `Executors`.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class WorkerThread implements Runnable {
    private String message;
    public WorkerThread(String s){
        this.message=s;
    }
```

```
public void run() {

    System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);

    processmessage();//call processmessage method that sleeps the thread for 2 seconds

    System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name

}
```

```
private void processmessage() {  
    try {  
        Thread.sleep(2000);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```


```
public class TestThreadPool {  
    public static void main(String[] args) {  
  
        ExecutorService executor = Executors.newFixedThreadPool(5); //creating a pool of 5 threads  
        for (int i = 0; i < 10; i++) {  
            Runnable worker = new WorkerThread("" + i);  
  
            executor.execute(worker); //calling execute method of ExecutorService  
        }  
  
        executor.shutdown();  
        while (!executor.isTerminated()) {}  
  
        System.out.println("Finished all threads");  
    }  
}
```


THREADGROUP IN JAVA

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Note: Now `suspend()`, `resume()` and `stop()` methods are deprecated.

Java thread group is implemented by `java.lang.ThreadGroup` class.



A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates 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 it cannot access the information about its thread group's parent thread group or any other thread groups.

CONSTRUCTORS OF THREADGROUP CLASS

There are only two constructors of ThreadGroup class.

No	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name.

ThreadGroup Example

```
public class ThreadGroupDemo implements Runnable{  
    public void run() {  
        System.out.println(Thread.currentThread().getName());  
    }  
  
    public static void main(String[] args) {  
        ThreadGroupDemo runnable = new ThreadGroupDemo();  
        ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");  
        Thread t1 = new Thread(tg1, runnable, "one");  
        t1.start();  
        Thread t2 = new Thread(tg1, runnable, "two");  
        t2.start();  
        Thread t3 = new Thread(tg1, runnable, "three");  
        t3.start();  
  
        System.out.println("Thread Group Name: "+tg1.getName());  
        tg1.list();  
    }  
}
```

SYNCHRONIZATION IN JAVA

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

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

WHY USE SYNCHRONIZATION

The synchronization is mainly used to

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

TYPES OF SYNCHRONIZATION

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

THREAD SYNCHRONIZATION

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

1. Mutual Exclusive

- Synchronized method.
- Synchronized block.
- static synchronization.

2. Cooperation (Inter-thread communication in java)

UNDERSTANDING THE PROBLEM WITHOUT SYNCHRONIZATION

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
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();  
    }  
}
```

DEADLOCK IN JAVA

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

INTER-THREAD COMMUNICATION IN JAVA

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

wait()

notify()

notifyAll()

WAIT() METHOD

Causes current thread to release the lock and wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

NOTIFY() METHOD

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

```
public final void notify()
```

NOTIFYALL() METHOD

Wakes up all threads that are waiting on this object's monitor. Syntax:

```
public final void notifyAll()
```

WHY WAIT(), NOTIFY() AND NOTIFYALL() METHODS ARE DEFINED IN OBJECT CLASS NOT THREAD CLASS?

It is because they are related to lock and object has a lock.

DIFFERENCE BETWEEN WAIT AND SLEEP?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

```
class Customer{  
    int amount=10000;  
    synchronized void withdraw(int amount){  
        System.out.println("going to withdraw...");  
        if(this.amount<amount){  
            System.out.println("Less balance; waiting for deposit...");  
            try{  
                wait();}catch(Exception e){}  
        }  
        this.amount-=amount;  
        System.out.println("withdraw completed...");  
    }  
}
```

```
synchronized void deposit(int amount){  
    System.out.println("going to deposit...");  
    this.amount+=amount;  
    System.out.println("deposit completed... ");  
    notify();  
}  
}
```

```
class Test{  
    public static void main(String args[]){  
        final Customer c=new Customer();  
        new Thread(){  
            public void run(){c.withdraw(15000);}  
        }.start();  
        new Thread(){  
            public void run(){c.deposit(10000);}  
        }.start();  
    }  
}
```




CONTINUE IN NEXT UNIT.....