

Introduction to Multithreading in Java

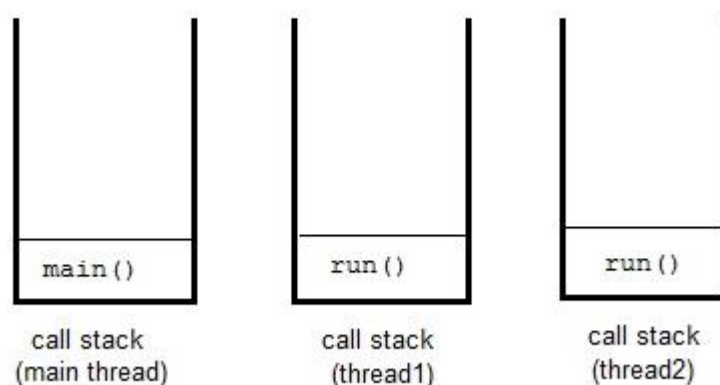
Multithreading is a concept of running multiple threads simultaneously. Thread is a lightweight unit of a process that executes in multithreading environment.

A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process). You can think of a lightweight process as a virtual CPU that executes code or system calls. You usually do not need to concern yourself with lightweight processes to program with threads. Multithreaded programs contain two or more threads that can run concurrently and each thread defines a separate path of execution. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

In Java, the word **thread** means two different things.

- An instance of **Thread** class.
- or, A thread of execution.

An instance of **Thread** class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.



Advantage of Multithreading

Multithreading **reduces** the CPU **idle time** that increase overall performance of the system. Since thread is lightweight process then it takes **less memory** and

perform **context switching** as well that helps to share the memory and reduce time of switching between threads.

Multitasking

Multitasking is a process of performing multiple tasks simultaneously. We can understand it by computer system that perform multiple tasks like: writing data to a file, playing music, downloading file from remote server at the same time.

Multitasking can be achieved either by using multiprocessing or multithreading. Multitasking by using multiprocessing involves multiple processes to execute multiple tasks simultaneously whereas Multithreading involves multiple threads to executes multiple tasks.

Why Multithreading ?

Thread has many advantages over the process to perform multitasking. Process is heavy weight, takes more memory and occupy CPU for longer time that may lead to performance issue with the system. To overcome these issue process is broken into small unit of independent sub-process. These sub-process are called threads that can perform independent task efficiently. So nowadays computer systems prefer to use thread over the process and use multithreading to perform multitasking.

How to Create Thread ?

To create a thread, Java provides a class **Thread** and an interface **Runnable** both are located into java.lang package.

We can create thread either by extending Thread class or implementing Runnable interface. Both includes a run method that must be override to provide thread implementation.

It is recommended to use Runnable interface if you just want to create a thread but can use Thread class for implementation of other thread functionalities as well.

We will discuss it in details in our next topics.

The `main` thread

When we run any java program, the program begins to execute its code starting from the main method. Therefore, the JVM creates a thread to start executing the code present in **main** method. This thread is called as **main** thread. Although the main thread is automatically created, you can control it by obtaining a reference to it by calling **currentThread()** method.

Two important things to know about **main** thread are,

- It is the thread from which other threads will be produced.
- It must be always the last thread to finish execution.

```
class MainThread
```

```

{

    public static void main(String[] args)

    {

        Thread t = Thread.currentThread();

        t.setName("MainThread");

        System.out.println("Name of thread is "+t);

    }

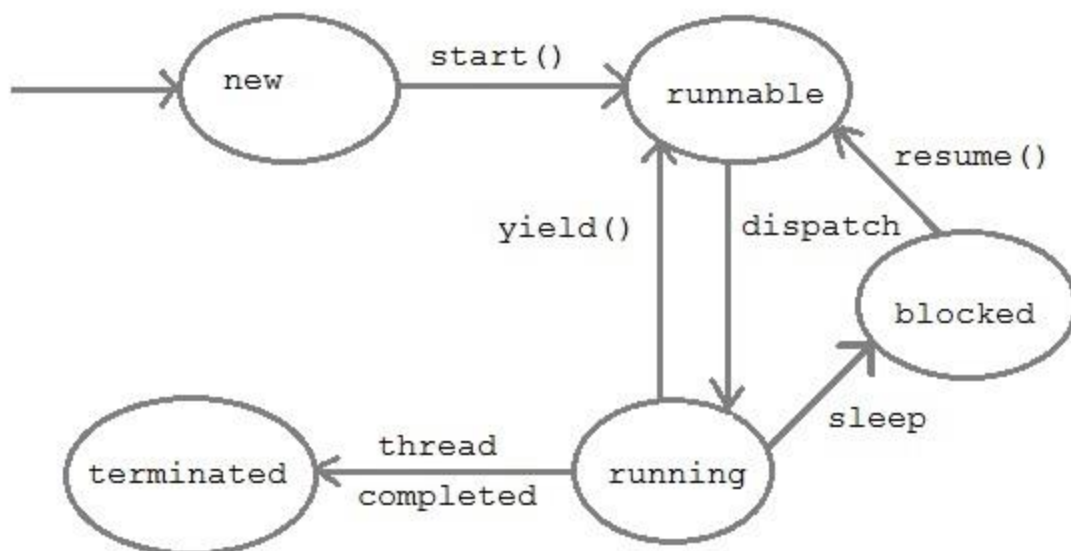
}

```

Name of thread is Thread[MainThread,5,main]

Life cycle of a Thread

Like process, thread have its life cycle that includes various phases like: new, running, terminated etc. we have described it using the below image.



1. **New** : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. **Runnable** : After invocation of start() method on new thread, the thread becomes runnable.
3. **Running** : A thread is in running state if the thread scheduler has selected it.

4. **Waiting** : A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
5. **Terminated** : A thread enter the terminated state when it complete its task.

Daemon Thread

Daemon threads is a low priority thread that provide supports to user threads. These threads can be user defined and system defined as well. Garbage collection thread is one of the system generated daemon thread that runs in background. These threads run in the background to perform tasks such as garbage collection. Daemon thread does allow JVM from existing until all the threads finish their execution. When a JVM founds daemon threads it terminates the thread and then shutdown itself, it does not care Daemon thread whether it is running or not.

Thread Pool

In Java, is used for reusing the threads which were created previously for executing the current task. It also provides the solution if any problem occurs in the thread cycle or in resource thrashing. In Java Thread pool a group of threads are created, one thread is selected and assigned job and after completion of job, it is sent back in the group.

Thread Priorities

In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler. The priority is given by the JVM or by the programmer itself explicitly. The range of the priority is between 1 to 10 and there are three variables which are static to define priority in a Thread Class.

Note: Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

Java Thread Class

Thread class is the main class on which Java's Multithreading system is based. Thread class, along with its companion interface Runnable will be used to create and run threads for utilizing Multithreading feature of Java.

It provides constructors and methods to support multithreading. It extends object class and implements Runnable interface.

Signature of Thread class

```
public class Thread extends Object implements Runnable
```

Thread Class Priority Constants

Field	Description
MAX_PRIORITY	It represents the maximum priority that a thread can have.
MIN_PRIORITY	It represents the minimum priority that a thread can have.
NORM_PRIORITY	It represents the default priority that a thread can have.

Constructors of Thread class

1. **Thread()**
2. **Thread(String str)**
3. **Thread(Runnable r)**
4. **Thread(Runnable r, String str)**
5. **Thread(ThreadGroup group, Runnable target)**
6. **Thread(ThreadGroup group, Runnable target, String name)**
7. **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**
8. **Thread(ThreadGroup group, String name)**

Thread Class Methods

Thread class also defines many methods for managing threads. Some of them are,

Method	Description
setName()	to give thread a name
getName()	return thread's name
getPriority()	return thread's priority
isAlive()	checks if thread is still running or not
join()	Wait for a thread to end
run()	Entry point for a thread
sleep()	suspend thread for a specified time
start()	start a thread by calling run() method
activeCount()	Returns an estimate of the number of active threads in the current thread's thread group and its subgroups.
checkAccess()	Determines if the currently running thread has permission to modify this thread.

Method	Description
currentThread()	Returns a reference to the currently executing thread object.
dumpStack()	Prints a stack trace of the current thread to the standard error stream.
getId()	Returns the identifier of this Thread.
getState()	Returns the state of this thread.
getThreadGroup()	Returns the thread group to which this thread belongs.
interrupt()	Interrupts this thread.
interrupted()	Tests whether the current thread has been interrupted.
isAlive()	Tests if this thread is alive.
isDaemon()	Tests if this thread is a daemon thread.
isInterrupted()	Tests whether this thread has been interrupted.
setDaemon(boolean on)	Marks this thread as either a daemon thread or a user thread.

Method	Description
setPriority(int newPriority)	Changes the priority of this thread.
yield()	A hint to the scheduler that the current thread is willing to yield its current use of a processor.

Some Important points to Remember

1. When we extend Thread class, we cannot override **setName()** and **getName()** functions, because they are declared final in Thread class.
2. While using **sleep()**, always handle the exception it throws.

```
static void sleep(long milliseconds) throws InterruptedException
```

Runnable Interface

It also used to create thread and should be used if you are only planning to override the `run()` method and no other Thread methods.

Signature

```
@FunctionalInterface  
public interface Runnable
```

Runnable Interface Method

It provides only single method that must be implemented by the class.

Method	Description
run()	It runs the implemented thread.

Shutdown hook

In Java, Shutdown hook is used to clean-up all the resource, it means closing all the files, sending alerts etc. We can also save the state when the JVM shuts down. Shutdown hook mostly used when any code is to be executed before any JVM shuts down. Following are some of the reasons when the JVM shut down:

- Pressing ctrl+c on the command prompt
- When the System.exit(int) method is invoked.
- When user logoff or shutdown etc

addShutdownHook(Thread hook)

The addShutdownHook(Thread hook) method is used to register the thread with the virtual machine. This method is of Runtime class.

Example:

```
class Demo6 extends Thread
{
    public void run()
    {
        System.out.println("Shutdown hook task is Now
completed...");
    }
}

public class ShutdownDemo1
{
    public static void main(String[] args) throws Exception
```

```

{

    Runtime obj=Runtime.getRuntime();

    obj.addShutdownHook(new Demo6());

    System.out.println("Now main method is
sleeping... For Exit press ctrl+c");

    try
    {

        Thread.sleep(4000);

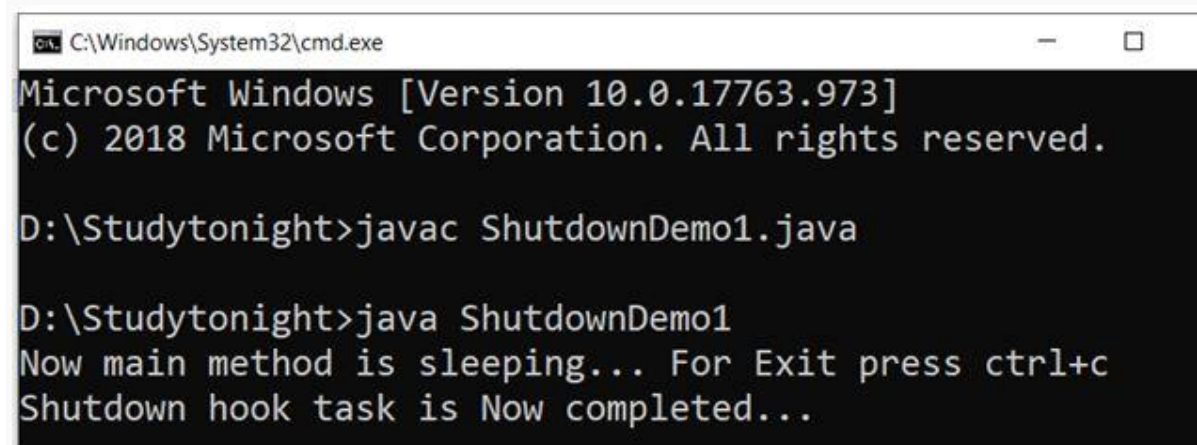
    }

    catch (Exception e) {}

}

}

```



```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

D:\Studytonight>javac ShutdownDemo1.java

D:\Studytonight>java ShutdownDemo1
Now main method is sleeping... For Exit press ctrl+c
Shutdown hook task is Now completed...

```

OutOfMemory Exception

In Java, as we know that all objects are stored in the heap. The objects are created using the new keyword. The OutOfMemoryError occurs as follow:

Exception in thread "main" java.lang.OutOfMemoryError: Java heap

This error occurs when Java Virtual Machine is not able to allocate the object because it is out of memory and no memory can be available by the garbage collector.

The meaning of OutOfMemoryError is that something wrong is in the program. Many times the problem can be out of control when the third party library caches strings.

Basic program in which OutOfMemoryError can occur

Example:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class OutOfMemoryDemo1 {

    public static void main(String[] args) {

        Listobj = new ArrayList<>();

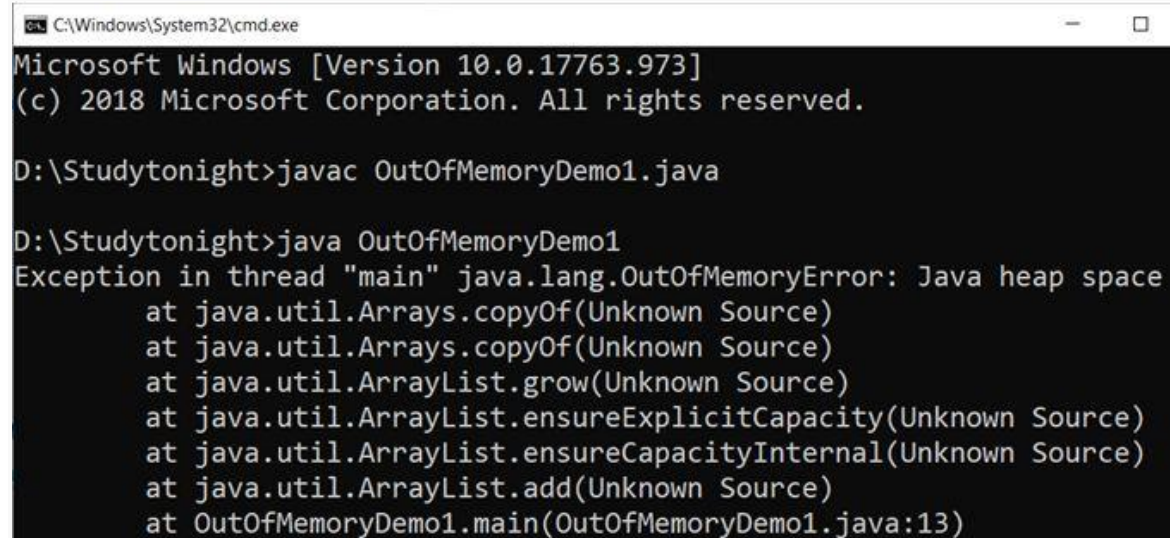
        Random obj1= new Random();

        while (true)

            obj.add(obj1.nextInt());

    }

}
```



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.17763.973]
(c) 2018 Microsoft Corporation. All rights reserved.

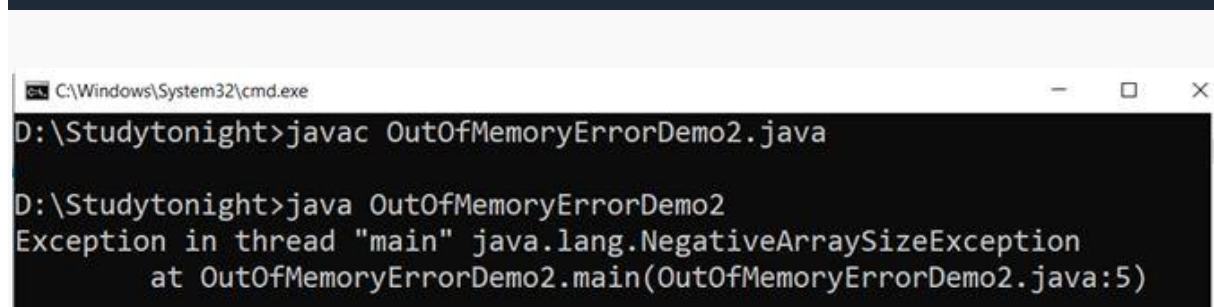
D:\Studytonight>javac OutOfMemoryDemo1.java

D:\Studytonight>java OutOfMemoryDemo1
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Unknown Source)
    at java.util.Arrays.copyOf(Unknown Source)
    at java.util.ArrayList.grow(Unknown Source)
    at java.util.ArrayList.ensureExplicitCapacity(Unknown Source)
    at java.util.ArrayList.ensureCapacityInternal(Unknown Source)
    at java.util.ArrayList.add(Unknown Source)
    at OutOfMemoryDemo1.main(OutOfMemoryDemo1.java:13)
```

Program in which OutOfMemoryError can occur because of low memory

Example:

```
public class OutOfMemoryErrorDemo2
{
    public static void main(String[] args)
    {
        Integer[] a = new Integer[100000*10000*1000];
        System.out.println("Done");
    }
}
```



```
C:\Windows\System32\cmd.exe
D:\Studytonight>javac OutOfMemoryErrorDemo2.java
D:\Studytonight>java OutOfMemoryErrorDemo2
Exception in thread "main" java.lang.NegativeArraySizeException
    at OutOfMemoryErrorDemo2.main(OutOfMemoryErrorDemo2.java:5)
```

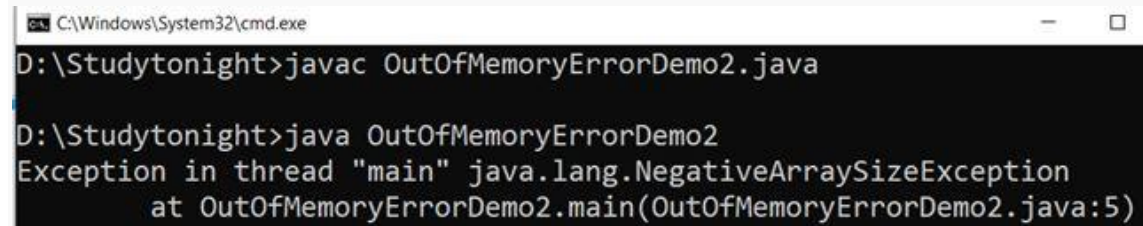
Program in which OutOfMemoryError can occur, when Garbage Collector exceed the limit

Example:

```
import java.util.*;

public class OutOfMemoryErrorDemo2{
    public static void main(String args[]) throws Exception
    {
        Map a = new HashMap();
    }
}
```

```
a = System.getProperties();  
  
Random b = new Random();  
  
while (true) {  
a.put(b.nextInt(), "randomValue");  
  
} }  
  
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\System32\cmd.exe'. The command prompt shows the following sequence of commands and output:

```
D:\Studytonight>javac OutOfMemoryErrorDemo2.java  
  
D:\Studytonight>java OutOfMemoryErrorDemo2  
Exception in thread "main" java.lang.NegativeArraySizeException  
    at OutOfMemoryErrorDemo2.main(OutOfMemoryErrorDemo2.java:5)
```

Creating a thread in Java

To implement multithreading, Java defines two ways by which a thread can be created.

- By implementing the **Runnable** interface.
- By extending the **Thread** class.

Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface, the class needs to implement the `run()` method.

Run Method Syntax:

```
public void run()
```

- It introduces a concurrent thread into your program. This thread will end when `run()` method terminates.
- You must specify the code that your thread will execute inside `run()` method.
- `run()` method can call other methods, can use other classes and declare variables just like any other normal method.

```
class MyThread implements Runnable
{
    public void run()
    {
        System.out.println("concurrent thread started running..");
    }
}

class MyThreadDemo
{
```

```

public static void main(String args[])
{
    MyThread mt = new MyThread();

    Thread t = new Thread(mt);

    t.start();

}
}

```

concurrent thread started running..

To call the `run()` method, `start()` method is used. On calling `start()`, a new stack is provided to the thread and `run()` method is called to introduce the new thread into the program.

Note: If you are implementing `Runnable` interface in your class, then you need to explicitly create a `Thread` class object and need to pass the `Runnable` interface implemented class object as a parameter in its constructor.

Extending Thread class

This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class. The extending class must override `run()` method which is the entry point of new thread.

```

class MyThread extends Thread
{
    public void run()
    {
        System.out.println("concurrent thread started
running..");
    }
}

class MyThreadDemo
{

```

```

public static void main(String args[])
{
    MyThread mt = new MyThread();
    mt.start();
}
}

```

concurrent thread started running..

In this case also, we must override the `run()` and then use the `start()` method to run the thread. Also, when you create `MyThread` class object, `Thread` class constructor will also be invoked, as it is the super class, hence `MyThread` class object acts as `Thread` class object.

What if we call `run()` method directly without using `start()` method?

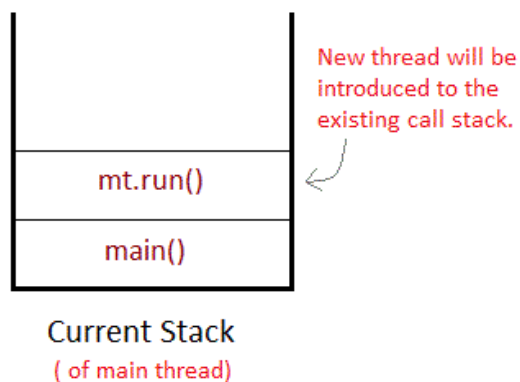
In above program if we directly call `run()` method, without using `start()` method,

```

public static void main(String args[])
{
    MyThread mt = new MyThread();
    mt.run();
}

```

Doing so, the thread won't be allocated a new call stack, and it will start running in the current call stack, that is the call stack of the **main** thread. Hence Multithreading won't be there.



Can we Start a thread twice?

No, a thread cannot be started twice. If you try to do so, **IllegalThreadStateException** will be thrown.

```
public static void main(String args[])
{
    MyThread mt = new MyThread();

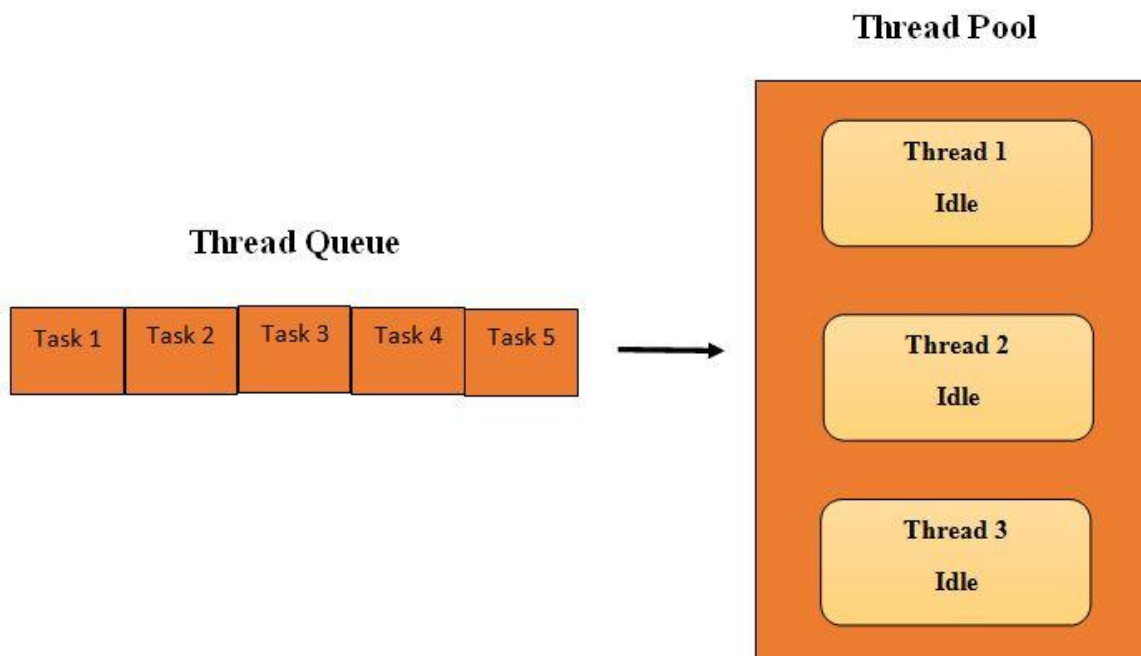
    mt.start();

    mt.start();           //Exception thrown
}
```

When a thread is in running state, and you try to start it again, or any method try to invoke that thread again using **start()** method, exception is thrown.

Thread Pool

In Java, is used for reusing the threads which were created previously for executing the current task. It also provides the solution if any problem occurs in the thread cycle or in resource thrashing. In Java Thread pool a group of threads are created, one thread is selected and assigned job and after completion of job, it is sent back in the group.



There are three methods of a Thread pool. They are as following:

1. newFixedThreadPool(int)
2. newCachedThreadPool()
3. newSingleThreadExecutor()

Following are the steps for creating a program of the thread pool

1. create a runnable object to execute.
2. using executors create an executor pool
3. Now Pass the object to the executor pool
4. At last shutdown the executor pool.

Example:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class WorkerThread implements Runnable
{
    private String message;

    public WorkerThread(String a)
    {
        this.message=a;
    }

    public void run()
    {
        System.out.println(Thread.currentThread().getName()+"
(Start) message = "+message);

        processmessage();

        System.out.println(Thread.currentThread().getName()+"
(End) ");
    }

    private void processmessage()
```

```

        {
            try
            {
                Thread.sleep(5000);
            }
            catch (InterruptedException e)
            {
                System.out.println(e);
            }
        }
    }
}

public class ThreadPoolDemo1
{
    public static void main(String[] args)
    {
        ExecutorService executor =
        Executors.newFixedThreadPool(5);

        for (int i = 0; i < 10; i++)
        {
            Runnable obj = new WorkerThread("" + i);
            executor.execute(obj);
        }

        executor.shutdown();

        while (!executor.isTerminated())
        {
        }

        System.out.println("*****All threads are
        Finished*****");
    }
}

```

```
}
```

```
C:\Windows\System32\cmd.exe
D:\Studytonight>Javaac ThreadPoolDemo1.java

D:\Studytonight>Java ThreadPoolDemo1
pool-1-thread-4 (Start) message = 3
pool-1-thread-5 (Start) message = 4
pool-1-thread-2 (Start) message = 1
pool-1-thread-1 (Start) message = 0
pool-1-thread-3 (Start) message = 2
pool-1-thread-2 (End)
pool-1-thread-5 (End)
pool-1-thread-5 (Start) message = 6
pool-1-thread-4 (End)
pool-1-thread-4 (Start) message = 7
pool-1-thread-2 (Start) message = 5
pool-1-thread-3 (End)
pool-1-thread-3 (Start) message = 8
pool-1-thread-1 (End)
pool-1-thread-1 (Start) message = 9
pool-1-thread-5 (End)
pool-1-thread-2 (End)
pool-1-thread-4 (End)
pool-1-thread-3 (End)
pool-1-thread-1 (End)
*****All threads are Finished*****
```

Joining threads in Java

Sometimes one thread needs to know when other thread is terminating. In java, **isAlive()** and **join()** are two different methods that are used to check whether a thread has finished its execution or not.

The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

```
final boolean isAlive()
```

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

```
final void join() throws InterruptedException
```

Using **join()** method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

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

As we have seen in the [Introduction to MultiThreading](#), the main thread must always be the last thread to finish its execution. Therefore, we can use Thread join() method to ensure that all the threads created by the program has been terminated before the execution of the main thread.

Java **isAlive** method

Lets take an example and see how the **isAlive()** method works. It returns true if thread status is live, false otherwise.

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("r1 ");
        try {
            Thread.sleep(500);
        }
        catch (InterruptedException ie)
        {
            // do something
        }
    }
}
```

```

    }

    System.out.println("r2 ");

}

public static void main(String[] args)

{

    MyThread t1=new MyThread();

    MyThread t2=new MyThread();

    t1.start();

    t2.start();

    System.out.println(t1.isAlive());

    System.out.println(t2.isAlive());

}

}

```

Output

```

r1
true
true
r1
r2
r2

```

Example of thread without `join()` method

If we run a thread without using `join()` method then the execution of thread cannot be predict. Thread scheduler schedules the execution of thread.

```

public class MyThread extends Thread

{

    public void run()

    {

        System.out.println("r1 ");

    }

}

```

```

        try {

            Thread.sleep(500);

        }

        catch (InterruptedException ie) { }

        System.out.println("r2 ");

    }

    public static void main(String[] args)

    {

        MyThread t1=new MyThread();

        MyThread t2=new MyThread();

        t1.start();

        t2.start();

    }

}

```

Output

r1

r1

r2

r2

In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 ms. At the same time Thread t2 will start its process and print "r1" on console and then go into sleep for 500 ms. Thread t1 will wake up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So you will get output like `r1 r1 r2 r2`

Example of thread with `join()` method

In this example, we are using `join()` method to ensure that thread finished its execution before starting other thread. It is helpful when we want to executes multiple threads based on our requirement.

```

public class MyThread extends Thread

{

```

```

public void run()
{
    System.out.println("r1 ");
    try {
        Thread.sleep(500);
    }catch(InterruptedException ie){ }
    System.out.println("r2 ");
}

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

    try{
        t1.join();    //Waiting for t1 to finish
    }catch(InterruptedException ie){}

    t2.start();
}
}

```

Output

r1

r2

r1

r2

In this above program join() method on thread t1 ensures that t1 finishes its process before thread t2 starts.

Specifying time with join()

If in the above program, we specify time while using **join()** with **t1**, then **t1** will execute for that time, and then **t2** will join it.

```
public class MyThread extends Thread
{
    MyThread(String str){
        super(str);
    }

    public void run()
    {
        System.out.println(Thread.currentThread().getName());
    }

    public static void main(String[] args)
    {
        MyThread t1=new MyThread("first thread");
        MyThread t2=new MyThread("second thread");
        t1.start();

        try{
            t1.join(1500);           //Waiting for t1
to finish

        }catch(InterruptedException ie){
            System.out.println(ie);
        }

        t2.start();
    }
}
```

```

        try{
            t2.join(1500);          //Waiting for t2 to
finish
        }catch(InterruptedException ie){
            System.out.println(ie);
        }
    }
}

```

Doing so, initially t1 will execute for 1.5 seconds, after which t2 will join it.

Java Sleeping Thread

To sleep a thread for a specified time, Java provides sleep method which is defined in Thread class. The sleep method is an overloaded method which are given below. It throws interrupted exception so make sure to provide proper handler.

It always pause the current thread execution. Any other thread can interrupt the current thread in sleep, in that case InterruptedException is thrown.

Syntax

```
sleep(long millis)throws InterruptedException
```

```
sleep(long millis, int nanos)throws InterruptedException
```

Example : Sleeping a thread

In this example, we are sleeping threads by using sleep method. Each thread will sleep for 1500 milliseconds and then resume its execution. See the below example.

```

public class MyThread extends Thread
{
    MyThread(String str){
        super(str);
    }
}

```

```

        public void run()
        {

System.out.println(Thread.currentThread().getName()+"
Started");

            try{

                MyThread.sleep(1500);

            }catch(InterruptedException ie){

                System.out.println(ie);

            }

System.out.println(Thread.currentThread().getName()+"
Finished");

        }

        public static void main(String[] args)

        {

            MyThread t1=new MyThread("first thread");

            MyThread t2=new MyThread("second thread");

            t1.start();

            t2.start();

        }

    }

```

Output

```

first thread Started
second thread Started
first thread Finished
second thread Finished

```

Example

```
public class MyThread extends Thread
{
    MyThread(String str){
        super(str);
    }

    public void run()
    {
        System.out.println(Thread.currentThread().getName()+"
        Started");

        try{
            MyThread.sleep(1500);

            System.out.println(Thread.currentThread().getName()+"
            Sleeping..");

            }catch(InterruptedException ie){
                System.out.println(ie);
            }

            System.out.println(Thread.currentThread().getName()+"
            Finished");
        }

        public static void main(String[] args)
        {
            MyThread t1=new MyThread("first thread");
            MyThread t2=new MyThread("second thread");

            System.out.println(t1.getName()+" state:
            "+t1.getState());
```

```

        t1.start();

        System.out.println(t1.getName()+" state:
"+t1.getState());

        System.out.println(t2.getName()+" state:
"+t2.getState());

        t2.start();

        System.out.println(t2.getName()+" state:
"+t2.getState());

    }

}

```

Output

```

first thread state: NEW
first thread state: RUNNABLE
second thread state: NEW
first thread Started
second thread state: RUNNABLE
second thread Started
first thread Sleeping..
second thread Sleeping..
second thread Finished
first thread Finished

```

`Thread.sleep()` interacts with the thread scheduler to put the current thread in wait state for specified period of time. Once the wait time is over, thread state is changed to runnable state and wait for the CPU for further execution. So the actual time that current thread sleep depends on the thread scheduler that is part of operating system.

Java Thread Priorities

Priority of a thread describes how early it gets execution and selected by the thread scheduler. In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler. The priority is given by the JVM or by the programmer itself explicitly. The range of the priority is between 1 to 10 and there are three constant variables which are static and used to fetch priority of a Thread. They are as following:

1. `public static int MIN_PRIORITY`

It holds the minimum priority that can be given to a thread. The value for this is 1.

2. `public static int NORM_PRIORITY`

It is the default priority that is given to a thread if it is not defined. The value for this is 0.

3. `public static int MAX_PRIORITY`

It is the maximum priority that can be given to a thread. The value for this is 10.

Get and Set methods in Thread priority

1. `public final int getPriority()`

In Java, `getPriority()` method is in `java.lang.Thread` package. it is used to get the priority of a thread.

2. `public final void setPriority(int newPriority)`

In Java `setPriority(int newPriority)` method is in `java.lang.Thread` package. It is used to set the priority of a thread. The `setPriority()` method throws `IllegalArgumentException` if the value of new priority is above minimum and maximum limit.

Example: Fetch Thread Priority

If we don't set thread priority of a thread then by default it is set by the JVM. In this example, we are getting thread's default priority by using the `getPriority()` method.

```
class MyThread extends Thread
{
    public void run()
    {
```

```

        System.out.println("Thread Running...");
    }

    public static void main(String[] args)
    {
        MyThread p1 = new MyThread();
        MyThread p2 = new MyThread();
        MyThread p3 = new MyThread();
        p1.start();

        System.out.println("P1 thread priority : " +
p1.getPriority());

        System.out.println("P2 thread priority : " +
p2.getPriority());

        System.out.println("P3 thread priority : " +
p3.getPriority());

    }
}

```

Output

P1 thread priority : 5

Thread Running...

P2 thread priority : 5

P3 thread priority : 5

Example: Thread Constants

We can fetch priority of a thread by using some predefined constants provided by the Thread class. these constants returns the max, min and normal priority of a thread.

```

class MyThread extends Thread

```

```

{

    public void run()

    {

        System.out.println("Thread Running...");

    }

    public static void main(String[] args)

    {

        MyThread p1 = new MyThread();

        p1.start();

        System.out.println("max thread priority : " +
p1.MAX_PRIORITY);

        System.out.println("min thread priority : " +
p1.MIN_PRIORITY);

        System.out.println("normal thread priority : " +
p1.NORM_PRIORITY);

    }

}

```

Output

Thread Running...

max thread priority : 10

min thread priority : 1

normal thread priority : 5

Example : Set Priority

To set priority of a thread, `setPriority()` method of thread class is used. It takes an integer argument that must be between 1 and 10. see the below example.


```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Thread Running...");
    }

    public static void main(String[] args)
    {
        MyThread p1 = new MyThread();
        // Starting thread
        p1.start();
        // Setting priority
        p1.setPriority(2);
        // Getting priority
        int p = p1.getPriority();

        System.out.println("thread priority : " + p);

    }
}
```

Output

thread priority : 2

Thread Running...

Example:

In this example, we are setting priority of two thread and running them to see the effect of thread priority. Does setting higher priority thread get CPU first. See the below example.

```
class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Thread Running...
"+Thread.currentThread().getName());
    }

    public static void main(String[] args)
    {
        MyThread p1 = new MyThread();
        MyThread p2 = new MyThread();

        // Starting thread
        p1.start();
        p2.start();

        // Setting priority
        p1.setPriority(2);

        // Getting -priority
        p2.setPriority(1);

        int p = p1.getPriority();

        int p22 = p2.getPriority();

        System.out.println("first thread priority : " +
p);
```

```

        System.out.println("second thread priority : " +
p22);

    }

}

```

Output

Thread Running... Thread-0

first thread priority : 5

second thread priority : 1

Thread Running... Thread-1

Note: Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

Java Daemon Thread

Daemon threads is a low priority thread that provide supports to user threads. These threads can be user defined and system defined as well. Garbage collection thread is one of the system generated daemon thread that runs in background. These threads run in the background to perform tasks such as garbage collection. Daemon thread does allow JVM from existing until all the threads finish their execution. When a JVM finds daemon threads it terminates the thread and then shutdown itself, it does not care Daemon thread whether it is running or not.

Following are the methods in Daemon Thread

1. void setDaemon(boolean status)

In Java, this method is used to create the current thread as a daemon thread or user thread. If there is a user thread as obj1 then obj1.setDaemon(true) will make it a Daemon thread and if there is a Daemon thread obj2 then calling obj2.setDaemon(false) will make it a user thread.

Syntax:

```
public final void setDaemon(boolean on)
```

2. boolean isDaemon()

In Java, this method is used to check whether the current thread is a daemon or not. It returns true if the thread is Daemon otherwise it returns false.

Syntax:

```
public final boolean isDaemon()
```

Example:

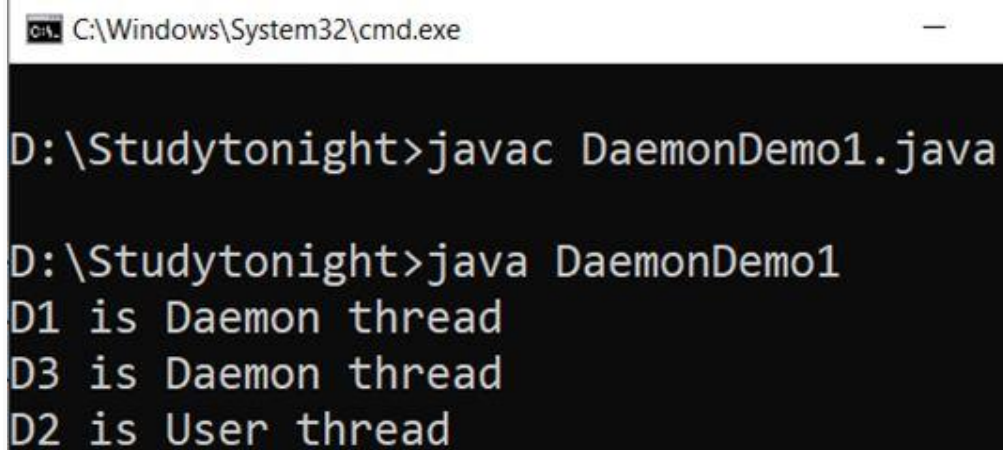
Lets create an example to create daemon and user threads. To create daemon thread `setDaemon()` method is used. It takes boolean value either true or false.

```
public class DaemonDemo1 extends Thread
{
    public DaemonDemo1(String name1)
    {
        super(name1);
    }

    public void run()
    {
        if(Thread.currentThread().isDaemon())
        {
            System.out.println(getName() + " is Daemon
thread");
        }
        else
        {
            System.out.println(getName() + " is User thread");
        }
    }

    public static void main(String[] args)
    {
        DaemonDemo1 D1 = new DaemonDemo1("D1");
        DaemonDemo1 D2 = new DaemonDemo1("D2");
        DaemonDemo1 D3 = new DaemonDemo1("D3");
    }
}
```

```
D1.setDaemon(true);  
  
D1.start();  
  
D2.start();  
  
D3.setDaemon(true);  
  
D3.start();  
  
}  
}
```



A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\System32\cmd.exe'. The command prompt shows the following sequence of commands and output:

```
D:\Studytonight>javac DaemonDemo1.java  
  
D:\Studytonight>java DaemonDemo1  
D1 is Daemon thread  
D3 is Daemon thread  
D2 is User thread
```

Example : Daemon thread Priority

Since daemon threads are low level threads then lets check the priority of these threads. The priority we are getting is set by the JVM.

```
public class DaemonDemo1 extends Thread  
{  
  
    public DaemonDemo1(String name1)  
    {  
        super(name1);  
    }  
  
    public void run()
```

```
{

    if(Thread.currentThread().isDaemon())

    {

        System.out.println(getName() + " is Daemon thread");

    }

    else

    {

        System.out.println(getName() + " is User thread");

    }

    System.out.println(getName()+" priority "+Thread.currentThread().getPriority());

}

public static void main(String[] args)

{

    DaemonDemo1 D1 = new DaemonDemo1("D1");

    DaemonDemo1 D2 = new DaemonDemo1("D2");

    DaemonDemo1 D3 = new DaemonDemo1("D3");


    D1.setDaemon(true);

    D1.start();

    D2.start();

    D3.setDaemon(true);

    D3.start();

}

}
```

Output

D1 is Daemon thread

D1 priority 5

D2 is User thread

D3 is Daemon thread

D2 priority 5

D3 priority 5

Example

While creating daemon thread make sure the `setDaemon()` is called before starting of the thread. Calling it after starting of thread will throw an exception and terminate the program execution.

```
public class DaemonDemo1 extends Thread
{
    public DaemonDemo1(String name1)
    {
        super(name1);
    }

    public void run()
    {
        if(Thread.currentThread().isDaemon())
        {
            System.out.println(getName() + " is Daemon
thread");
        }
        else
        {
            System.out.println(getName() + " is User thread");
        }
    }
}
```

```

        System.out.println(getName()+" priority
"+Thread.currentThread().getPriority());
    }

    public static void main(String[] args)
    {
        DaemonDemo1 D1 = new DaemonDemo1("D1");
        DaemonDemo1 D2 = new DaemonDemo1("D2");
        DaemonDemo1 D3 = new DaemonDemo1("D3");

        D1.setDaemon(true);
        D1.start();
        D2.start();
        D3.start();
        D3.setDaemon(true);

    }
}

```

Output

```

D1 is Daemon threadException in thread "main"
D1 priority 5
D3 is User thread
D2 is User thread
D2 priority 5java.lang.IllegalThreadStateException
D3 priority 5
at java.base/java.lang.Thread.setDaemon(Thread.java:1410)
at myjavaproject.DaemonDemo1.main(DaemonDemo1.java:32)

```


Java Synchronization

Synchronization is a process of handling resource accessibility by multiple thread requests. The main purpose of synchronization is to avoid thread interference. At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

General Syntax:

```
synchronized (object)
{
    //statement to be synchronized
}
```

Every Java object with a critical section of code gets a lock associated with the object. To enter critical section a thread need to obtain the corresponding object's lock.

Why we need Synchronization?

If we do not use synchronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, Suppose we have two different threads **T1** and **T2**, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file *temporary.txt* (*temporary.txt* is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be **locked**(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

Using Synchronized Methods

Using Synchronized methods is a way to accomplish synchronization. But lets first see what happens when we do not use synchronization in our program.

Example with no Synchronization

In this example, we are not using synchronization and creating multiple threads that are accessing display method and produce the random output.

```
class First
```

```
{  
  
    public void display(String msg)  
    {  
        System.out.print ("["+msg);  
        try  
        {  
            Thread.sleep(1000);  
        }  
        catch (InterruptedException e)  
        {  
            e.printStackTrace();  
        }  
        System.out.println ("]");  
    }  
}
```

```
class Second extends Thread  
{  
    String msg;  
    First fobj;  
    Second (First fp, String str)  
    {  
        fobj = fp;  
        msg = str;  
        start();  
    }  
    public void run()  
    {
```

```

        fobj.display(msg);
    }
}

public class Syncro
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1= new Second(fnew, "new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

```

[welcome [ new [ programmer]
]
]

```

In the above program, object **fnew** of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(void **display**). Hence the result is nonsynchronized and such situation is called **Race condition**..

Synchronized Keyword

To synchronize above program, we must *synchronize* access to the shared **display()** method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display() method.

```
synchronized void display (String msg)
```

Example : implementation of synchronized method

```

class First
{
    synchronized public void display(String msg)

```

```
{  
  
    System.out.print ("["+msg);  
  
    try  
  
    {  
  
        Thread.sleep(1000);  
  
    }  
  
    catch (InterruptedException e)  
  
    {  
  
        e.printStackTrace();  
  
    }  
  
    System.out.println ("]");  
  
}  
}
```

```
class Second extends Thread  
{  
  
    String msg;  
  
    First fobj;  
  
    Second (First fp,String str)  
  
    {  
  
        fobj = fp;  
  
        msg = str;  
  
        start();  
  
    }  
  
    public void run()  
  
    {  
  
        fobj.display(msg);  
  
    }  
}
```

```

}

public class MyThread
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1= new Second(fnew, "new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

[welcome]

[programmer]

[new]

Using Synchronized block

If want to synchronize access to an object of a class or only a part of a method to be synchronized then we can use synchronized block for it. It is capable to make any part of the object and method synchronized.

Example

In this example, we are using synchronized block that will make the display method available for single thread at a time.

```

class First
{
    public void display(String msg)
    {
        System.out.print ("["+msg);
        try

```

```

        {
            Thread.sleep(1000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println ("]");
    }
}

class Second extends Thread
{
    String msg;
    First fobj;
    Second (First fp, String str)
    {
        fobj = fp;
        msg = str;
        start();
    }
    public void run()
    {
        synchronized(fobj)    //Synchronized block
        {
            fobj.display(msg);
        }
    }
}

```

```

}

public class MyThread
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1= new Second (fnew,"new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

[welcome]

[new]

[programmer]

Because of synchronized block this program gives the expected output.

Difference between synchronized keyword and synchronized block

When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution.

synchronized block acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

Which is more preferred - Synchronized method or Synchronized block?

In Java, synchronized keyword causes a performance cost. A synchronized method in Java is very slow and can degrade performance. So we must use synchronization

keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

Java Interthread Communication

Java provide benefits of avoiding thread pooling using inter-thread communication. The `wait()`, `notify()`, and `notifyAll()` methods of Object class are used for this purpose. These method are implemented as **final** methods in Object, so that all classes have them. All the three method can be called only from within a **synchronized** context

- `wait()` tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
- `notify()` wakes up a thread that called `wait()` on same object.
- `notifyAll()` wakes up all the thread that called `wait()` on same object.

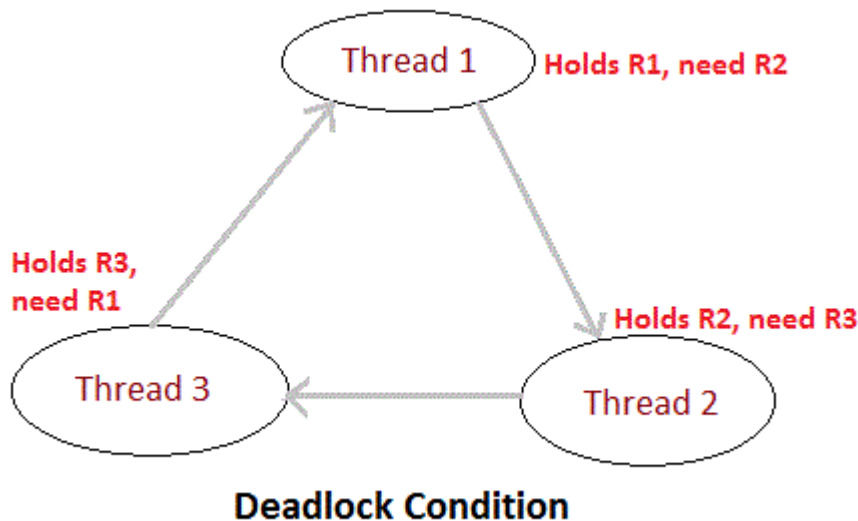
Difference between `wait()` and `sleep()`

<code>wait()</code>	<code>sleep()</code>
called from synchronised block	no such requirement
monitor is released	monitor is not released
gets awake when <code>notify()</code> or <code>notifyAll()</code> method is called.	does not get awake when <code>notify()</code> or <code>notifyAll()</code> method is called
not a static method	static method
<code>wait()</code> is generally used on condition	<code>sleep()</code> method is simply used to put your thread on sleep.

Thread Pooling

Pooling is usually implemented by loop i.e to check some condition repeatedly. Once condition is true appropriate action is taken. This waste CPU time.

Thread Deadlock in Java



Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.

Example

In this example, multiple threads are accessing same method that leads to deadlock condition. When a thread holds the resource and does not release it then other thread will wait and in deadlock condition wait time is never ending.

```
class Pen{}  
  
class Paper{}  
  
public class Write {  
  
    public static void main(String[] args)  
    {
```

```
final Pen pn =new Pen();

final Paper pr =new Paper();

Thread t1 = new Thread() {

    public void run()

    {

        synchronized(pn)

        {

            System.out.println("Thread1 is holding Pen");

            try{

                Thread.sleep(1000);

            }

            catch(InterruptedException e){

                // do something

            }

            synchronized(pr)

            {

                System.out.println("Requesting for Paper");

            }

        }

    }

};

Thread t2 = new Thread() {

    public void run()

    {

        synchronized(pr)

        {
```

```

        System.out.println("Thread2 is holding
Paper");

        try {
            Thread.sleep(1000);
        }

        catch (InterruptedException e) {
            // do something
        }

        synchronized (pn)
        {
            System.out.println("requesting for
Pen");
        }
    }

}

};

t1.start();

t2.start();

}

}

```

Output

Thread1 is holding Pen

Thread2 is holding Paper

Java Thread group

ThreadGroup is a class which is used for creating group of threads. This group of threads are in the form of a tree structure, in which the initial thread is the parent thread. A thread can have all the information of the other threads in the groups but can have the information of the threads of the other groups. It is very useful in the case where we want to suspend and resume numbers of threads. This thread group is implemented by java.lang.ThreadGroup class.

There are two types of Constructors in the Thread group they are as follow:

1. public ThreadGroup(String name)
2. public ThreadGroup(ThreadGroup parent, String name)

Following are the methods present in Thread group

1. checkAccess()

In Java, `checkAccess()` method is of ThreadGroup class. It is used to check whether the running thread has permission for modification or not in the ThreadGroup.

Syntax

```
public final void checkAccess()
```

Example:

```
class ThreadDemo1_1 extends Thread
{
    ThreadDemo1_1(String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for (int i = 0; i < 10; i++)
```

```

        {

            try

            {

                Thread.sleep(10);

            }

            catch (InterruptedException ex)

            {

                System.out.println(Thread.currentThread().getName());

            }

        }

        System.out.println(Thread.currentThread().getName());

    }

}

public class ThreadDemo1

{

    public static void main(String arg[]) throws
    InterruptedException, SecurityException

    {

        ThreadGroup obj1 = new ThreadGroup("Parent thread
=====> ");

        ThreadGroup obj2 = new ThreadGroup(obj1, "child thread
=====> ");

        ThreadDemo1_1 t1 = new ThreadDemo1_1("*****Thread-
1*****", obj1);

        t1.start();

        ThreadDemo1_1 t2 = new ThreadDemo1_1("*****Thread-
2*****", obj1);

        t2.start();

        obj1.checkAccess();

```

```

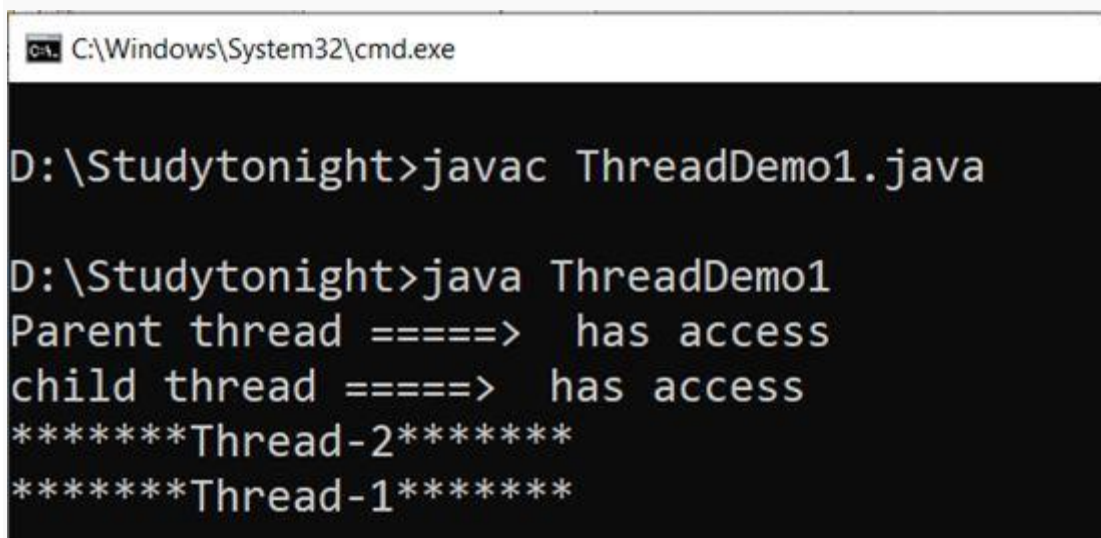
        System.out.println(obj1.getName() + " has access");

        obj2.checkAccess();

        System.out.println(obj2.getName() + " has access");

    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo1.java

D:\Studytonight>java ThreadDemo1
Parent thread ==> has access
child thread ==> has access
*****Thread-2*****
*****Thread-1*****

```

2. activeCount()

In Java, `activeCount()` method is of `ThreadGroup` class. It is used to count the active threads in a group which are currently running.

```

public static int activeCount()

```

Example:

```

class Demo2 extends Thread
{
    Demo2 (String a, ThreadGroup b)
    {

```

```

        super(b, a);

    }

    public void run()
    {
        for (inti = 0; i< 10; i++)
        {
            try
            {
                Thread.sleep(10);
            }

            catch (InterruptedException ex)
            {
                System.out.println(Thread.currentThread().getName());
            }
        }

        System.out.println(Thread.currentThread().getName());
    }
}

public class ThreadDemo2
{
    public static void main(String arg[])
    {
        ThreadGroup o1 = new ThreadGroup("parent thread group");

        Demo2 obj1 = new Demo2 ("Thread 1 =====> ", o1);
        Demo2 obj2 = new Demo2 ("Thread 2 =====> ", o1);
        Demo2 obj3 = new Demo2 ("Thread 3 =====> ", o1);
    }
}

```

```

        Demo2 obj4 = new Demo2 ("Thread 4 =====> ", o1);

        Demo2 obj5 = new Demo2 ("Thread 5 =====> ", o1);

        Demo2 obj6 = new Demo2 ("Thread 6 =====> ", o1);

        obj1.start();

        obj2.start();

        obj3.start();

        obj4.start();

        obj5.start();

        obj6.start();

        System.out.println("Total number of active thread
=====> "+ o1.activeCount());
    }
}

```

```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo2.java

D:\Studytonight>java ThreadDemo2
Total number of active thread =====> 6
Thread 5 =====>
Thread 3 =====>
Thread 1 =====>
Thread 6 =====>
Thread 4 =====>
Thread 2 =====>

```

3. activeGroupCount()

In Java, `activeGroupCount()` method is of `ThreadGroup` class. It is used to count the active groups of threads which are currently running.

Syntax `public int activeGroupCount()`.


```
public int activeGroupCount().
```

Example:

```
class Demo2 extends Thread
{
    Demo2 (String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for (inti = 0; i< 10; i++)
        {
            try
            {
                Thread.sleep(10);
            }
            catch (InterruptedException ex)
            {
                System.out.println(Thread.currentThread().getName());
            }
        }
        System.out.println(Thread.currentThread().getName()+"
=====> completed executing");
    }
}
```

```

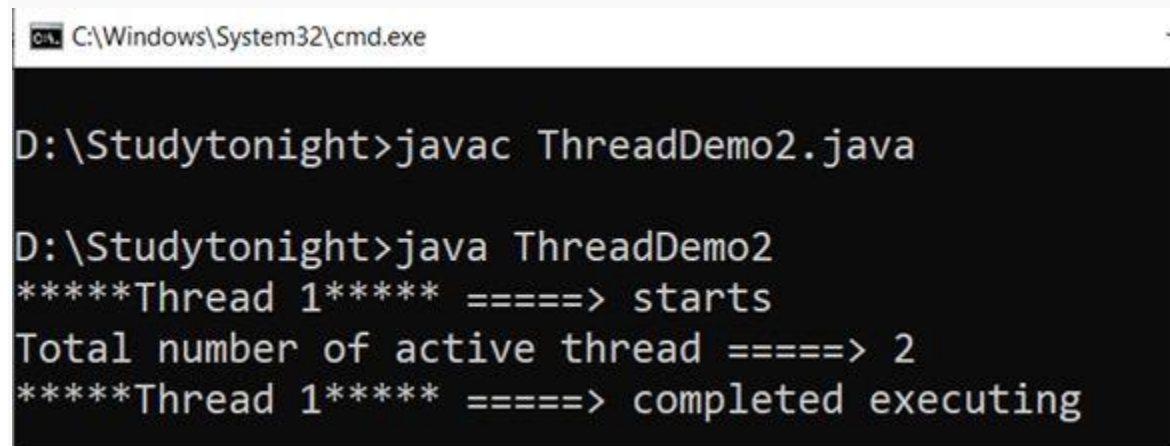
public class ThreadDemo2
{
    public static void main(String arg[])
    {
        ThreadGroup o1 = new ThreadGroup("parent thread
group");
        ThreadGroup o2 = new ThreadGroup(o1,"Child thread
group");
        ThreadGroup o3 = new ThreadGroup(o1,"parent thread
group");

        Demo2 obj1 = new Demo2("*****Thread 1*****",o1);
        System.out.println(obj1.getName() + " =====> starts");

        obj1.start();

        System.out.println("Total number of active thread
=====> " + o1.activeGroupCount());
    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo2.java

D:\Studytonight>java ThreadDemo2
*****Thread 1***** =====> starts
Total number of active thread =====> 2
*****Thread 1***** =====> completed executing

```

4. destroy()

In Java, the `destroy()` method is of ThreadGroup. It used to destroy a thread group. For destroying any thread group it is important that all the threads in that group should be stopped.

syntax

```
public void destroy()
```

Example:

```
class Demo2 extends Thread
{
    Demo2 (String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for (inti = 0; i< 10; i++)
        {
            try
            {
                Thread.sleep(10);
            }
            catch (InterruptedException ex)
            {
                System.out.println(Thread.currentThread().getName());
            }
        }
    }
}
```

```

        System.out.println(Thread.currentThread().getName()+"
=====> completed executing");
    }
}

public class ThreadDemo2
{
    public static void main(String arg[]) throws
InterruptedException, SecurityException
    {
        ThreadGroup o1 = new ThreadGroup("****parent thread
group****");

        ThreadGroup o2 = new ThreadGroup(o1,"****Child thread
group****");

        Demo2 obj1 = new Demo2("****Thread 1****",o1);
        obj1.start();

        Demo2 obj2 = new Demo2("****Thread 2****",o1);
        obj2.start();

        obj1.join();
        obj2.join();

        o2.destroy();

        System.out.println(o2.getName()+" ====> Destroyed");

        o1.destroy();

        System.out.println(o1.getName()+" ====> Destroyed");

    }
}

```

```
C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo2.java

D:\Studytonight>java ThreadDemo2
*****Thread 2***** =====> completed executing
*****Thread 1***** =====> completed executing
*****Child thread group***** =====> Destroyed
*****parent thread group***** =====> Destroyed
```

5. enumerate(Thread[] list)

In Java, `enumerate()` method is of `ThreadGroup` class. It is used for copying active threads into a specified array.

```
public int enumerate(Thread[] array)
```

Example:

```
class Demo2 extends Thread
{
    Demo2 (String a, ThreadGroup b)
    {
super(b, a);
    }
    public void run()
    {
        for (inti = 0; i< 10; i++)
        {
            try
```

```

        {
Thread.sleep(10);

        }

        catch (InterruptedException ex)

        {

System.out.println(Thread.currentThread().getName());

        }

    }

System.out.println(Thread.currentThread().getName()+" =====>
completed executing");

    }

}

public class ThreadDemo2

{

    public static void main(String arg[])

    {

ThreadGroup o1 = new ThreadGroup("*****parent thread
group*****");

ThreadGroup o2 = new ThreadGroup(o1,"*****Child thread
group*****");


        Demo2 obj1 = new Demo2("*****Thread 1*****",o1);

        System.out.println("Thread 1 Starts");

        obj1.start();

        Demo2 obj2 = new Demo2("*****Thread 2*****",o1);

        System.out.println("Thread 2 Starts");

        obj2.start();


        Thread[] tarray = new Thread[o1.activeCount()];

```

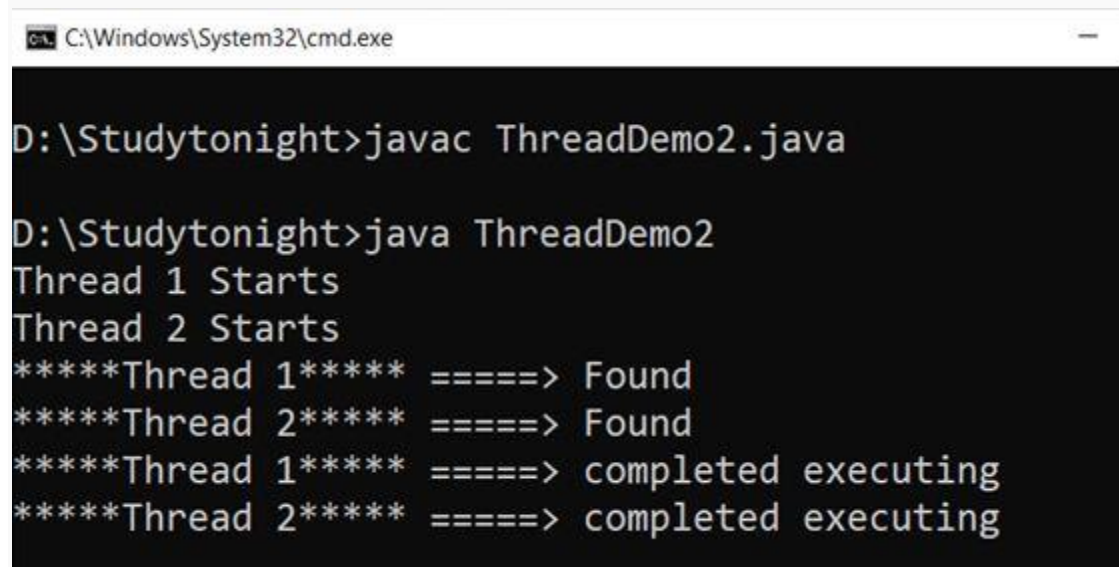
```

int count1 = o1.enumerate(tarray);

    for (inti = 0; i< count1; i++)
System.out.println(tarray[i].getName() + " =====> Found");

    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo2.java

D:\Studytonight>java ThreadDemo2
Thread 1 Starts
Thread 2 Starts
*****Thread 1***** =====> Found
*****Thread 2***** =====> Found
*****Thread 1***** =====> completed executing
*****Thread 2***** =====> completed executing

```

6. getMaxPriority()

In Java, getMaxPriority() method is of ThreadGroup class. It is used for check the maximum priority of the thread group.

Syntax

```

public final int getMaxPriority()

```

Example:

```

class Demo2 extends Thread
{

```

```

    Demo2 (String a, ThreadGroup b)
    {
        super(b, a);
    }

    public void run()
    {
        for (inti = 0; i< 10; i++)
        {
            try
            {
                Thread.sleep(10);
            }

            catch (InterruptedException ex)
            {

            }

            System.out.println(Thread.currentThread().getName());

        }

        System.out.println(Thread.currentThread().getName()+"
=====> completed executing");
    }
}

public class ThreadDemo2
{
    public static void main(String arg[])
    {
        ThreadGroup o1 = new ThreadGroup("****parent thread
group****");

        System.out.println("Maximum priority of Parent Thread:
" + o1.getMaxPriority());
    }
}

```



```

        ThreadGroup o2 = new ThreadGroup(o1, "*****Child thread
group*****");

        System.out.println("Maximum priority of Child Thread:
" + o2.getMaxPriority());

        Demo2 obj1 = new Demo2("*****Thread 1*****", o1);

        System.out.println("Thread 1 Starts");

        obj1.start();

        Demo2 obj2 = new Demo2("*****Thread 2*****", o1);

        System.out.println("Thread 2 Starts");

        obj2.start();

    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo2.java

D:\Studytonight>java ThreadDemo2
Maximum priority of Parent Thread: 10
Maximum priority of Child Thread: 10
Thread 1 Starts
Thread 2 Starts
*****Thread 2***** =====> completed executing
*****Thread 1***** =====> completed executing

```

7. getName()

In Java, `getName()` method is of `ThreadGroup` class. It is used for getting the name of the current thread group.

```
public final String getName()
```

Example:

```
class Demo3 extends Thread
{
    Demo3(String a, ThreadGroup b)
    {
        super(b, a);
        start();
    }
    public void run()
    {
        System.out.println(Thread.currentThread().getName());
    }
}

public class ThreadDemo3
{
    public static void main(String arg[]) throws
    InterruptedException,
    SecurityException, Exception
    {
        ThreadGroup o1 = new ThreadGroup("****Parent
thread****");
        ThreadGroup o2 = new ThreadGroup(o1, "****Child
thread****");

        Demo3 obj1 = new Demo3("Thread-1", o1);

        System.out.println("Name of First threadGroup : "
+obj1.getThreadGroup().getName());
    }
}
```

```

        Demo3 obj2 = new Demo3("Thread-2", o2);

        System.out.println("Name of Second threadGroup: "
+obj2.getThreadGroup().getName());
    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo3.java

D:\Studytonight>java ThreadDemo3
Name of First threadGroup : *****Parent thread*****
Thread-1
Thread-2
Name of Second threadGroup: *****Child thread*****

```

8. getParent()

In Java, `getParent()` method is of `ThreadGroup` class. It is used to get the parent thread from the thread group.

```

public final ThreadGroup getParent()

```

Example:

```

class Demo3 extends Thread
{
    Demo3(String a, ThreadGroup b)
    {
        super(b, a);
    }
}

```

```

        start();

    }

    public void run()

    {

        System.out.println(Thread.currentThread().getName());

    }

}

public class ThreadDemo3

{

    public static void main(String arg[]) throws
    InterruptedException, SecurityException, Exception

    {

        ThreadGroup o1 = new ThreadGroup("*****Parent
thread*****");

        ThreadGroup o2 = new ThreadGroup(o1, "*****Child
thread*****");

        Demo3 obj1 = new Demo3("Thread-1", o1);

        System.out.println("Thread one starting");

        obj1.start();

        Demo3 obj2 = new Demo3("Thread-2", o2);

        System.out.println("Thread second starting");

        obj2.start();

        System.out.println("Parent Thread Group for " +
o1.getName() + " is " + o1.getParent().getName());

        System.out.println("Parent Thread Group for " +
o2.getName() + " is " + o2.getParent().getName());
    }
}

```

```

    }
}

```



```

C:\Windows\System32\cmd.exe
D:\Studytonight>javac ThreadDemo3.java

D:\Studytonight>java ThreadDemo3
Thread one starting
Thread second starting
Parent Thread Group for *****Parent thread***** is main
Parent Thread Group for *****Child thread***** is *****Parent thread*****
Thread-1 ===> completed executing
Thread-2 ===> completed executing

```

9. interrupt()

In Java, `interrupt()` method is of `ThreadGroup` class. It is used to interrupt all the threads of the thread group.

Syntax

```
public final void interrupt()
```

Example:

```

class Demo2 extends Thread
{
    Demo2 (String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for (inti = 0; i< 10; i++)

```

```

        {

            try

            {

                Thread.sleep(10);

            }

            catch (InterruptedException ex)

            {

                System.out.println(Thread.currentThread().getName() + " ====>
interrupted");

            }

        }

        System.out.println(Thread.currentThread().getName() + "
====> completed executing");

    }

}

public class ThreadDemo2

{

    public static void main(String arg[]) throws
InterruptedException, SecurityException

    {

        ThreadGroup o1 = new ThreadGroup("****parent thread
group****");

        ThreadGroup o2 = new ThreadGroup(o1, "****Child thread
group****");

        Demo2 obj1 = new Demo2("****Thread 1****", o1);

        System.out.println(obj1.getName() + "Thread 1 Starts");

        obj1.start();

        o1.interrupt();
    }
}

```

```


        Demo2 obj2 = new Demo2("*****Thread 2*****",o1);

        System.out.println(obj2.getName()+"Thread 2 Starts");

        obj2.start();

    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo3.java

D:\Studytonight>java ThreadDemo3
Thread one starting
Thread second starting
Parent Thread Group for *****Parent thread***** is main
Parent Thread Group for *****Child thread***** is *****Parent thread*****
Thread-1 ==> completed executing
Thread-2 ==> completed executing

```

10. isDaemon()

In Java, `isDaemon()` method is of `ThreadGroup` class. It is used to check whether a thread is Daemon thread or not.

Syntax

```

public final boolean isDaemon()

```

Example:

```

class Demo4 extends Thread
{
    Demo4(String a, ThreadGroup b)
    {

```

```

        super(b, a);

    }

    public void run()

    {

        for(int i = 0; i < 10; i++)

        {

            i++;

        }

        System.out.println(Thread.currentThread().getName() +
" ==> Execution Finished");

    }

}

public class ThreadDemo4

{

    public static void main(String arg[]) throws
InterruptedException, SecurityException, Exception

    {

        ThreadGroup o1 = new ThreadGroup("****Parent
thread****");

        Demo4 obj1 = new Demo4 ("****Thread-1****", o1);

        obj1.start();

        System.out.println("Is " + o1.getName() + " a daemon
threadGroup? " + o1.isDaemon());

    }

}

```


C:\Windows\System32\cmd.exe

```
D:\Studytonight>javac ThreadDemo4.java
```

```
D:\Studytonight>java ThreadDemo4
```

```
Is *****Parent thread***** a daemon threadGroup? false  
*****Thread-1***** ===> Execution Finished
```

11. setDaemon(Boolean daemon)

In Java, `setDaemon()` method is of `ThreadGroup` class. It is used to make a thread as a daemon thread.

Syntax:

```
public final void setDaemon(boolean daemon)
```

Example:

```
class Demo4 extends Thread  
{  
    Demo4(String a, ThreadGroup b)  
    {  
        super(b, a);  
    }  
    public void run()  
    {  
        for(int i = 0; i < 10; i++)  
        {  
            i++;  
        }  
        System.out.println(Thread.currentThread().getName() +  
" ===> Execution Finished");  
    }  
}
```

```

    }

}

public class ThreadDemo4
{
    public static void main(String arg[]) throws
InterruptedException, SecurityException, Exception
    {

        ThreadGroup o1 = new ThreadGroup("*****Parent
thread*****");

        o1.setDaemon(true);

        ThreadGroup o2 = new ThreadGroup("*****Child
thread*****");

        o2.setDaemon(true);

        Demo4 obj1 = new Demo4 ("*****Thread-1*****", o1);
        obj1.start();

        Demo4 obj2 = new Demo4 ("*****Thread-2*****", o2);
        obj2.start();

        System.out.println("Is " + o1.getName() + " a daemon
threadGroup? " + o1.isDaemon());

        System.out.println("Is " + o2.getName() + " a daemon
threadGroup? " + o2.isDaemon());

    }
}

```

```
C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo4.java

D:\Studytonight>java ThreadDemo4
*****Thread-1***** ==> Execution Finished
*****Thread-2***** ==> Execution Finished
Is *****Parent thread***** a daemon threadGroup? true
Is *****Child thread***** a daemon threadGroup? true
```

12. isDestroyed()

In Java, `isDestroyed()` method is of `ThreadGroup` class. It is used to check whether the thread group is destroyed or not.

Syntax

```
public boolean isDestroyed()
```

Example:

```
class Demo4 extends Thread
{
    Demo4(String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for(int i = 0; i < 10; i++)
        {
            i++;
        }
    }
}
```

```

        System.out.println(Thread.currentThread().getName() +
" ==> Execution Finished");
    }
}

public class ThreadDemo4
{
    public static void main(String arg[]) throws
InterruptedException, SecurityException, Exception
    {

        ThreadGroup o1 = new ThreadGroup("*****Parent
thread*****");

        ThreadGroup o2 = new ThreadGroup("*****Child
thread*****");

        Demo4 obj1 = new Demo4 ("*****Thread-1*****", o1);
        System.out.println("Starting Thread 1");
        obj1.start();

        Demo4 obj2 = new Demo4 ("*****Thread-2*****", o2);
        System.out.println("Starting Thread 2");
        obj2.start();

        if(o1.isDestroyed()==true)
            System.out.println("The Group is destroyed");
        else
            System.out.println("The Group is not destroyed");
    }
}

```

```
C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo4.java

D:\Studytonight>java ThreadDemo4
Starting Thread 1
Starting Thread 2
The Group is not destroyed
*****Thread-2***** ==> Execution Finished
*****Thread-1***** ==> Execution Finished
```

13. list()

In Java, the `list()` method is of ThreadGroup class. It is used to getting all the information about a thread group. It is very useful at the time of debugging.

Syntax

```
public void list()
```

Example:

```
class Demo4 extends Thread
{
    Demo4(String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for(int i = 0; i < 10; i++)
        {
            i++;
        }
    }
}
```

```

    }

    System.out.println(Thread.currentThread().getName() +
" ==> Execution Finished");

    }
}

public class ThreadDemo4
{
    public static void main(String arg[]) throws
InterruptedException, SecurityException, Exception
    {

        ThreadGroup o1 = new ThreadGroup("****Parent
thread****");

        ThreadGroup o2 = new ThreadGroup("****Child
thread****");

        Demo4 obj1 = new Demo4("****Thread-1****", o1);
        System.out.println(obj1.getName()+"Starting Thread
1");
        obj1.start();

        Demo4 obj2 = new Demo4 ("****Thread-2****", o2);
        System.out.println(obj2.getName()+"Starting Thread
2");
        obj2.start();

        System.out.println("List of parent Thread Group: " +
o1.getName() + ":");
        o1.list();
    }
}

```

```
C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo4.java

D:\Studytonight>java ThreadDemo4
*****Thread-1*****Starting Thread 1
*****Thread-2*****Starting Thread 2
*****Thread-1***** ==> Execution Finished
*****Thread-2***** ==> Execution Finished
List of parent Thread Group: *****Parent thread*****:
java.lang.ThreadGroup[name=*****Parent thread*****,maxpri=10]
```

14. ParentOf(ThreadGroup g)

In Java, `ParentOf()` method is of `ThreadGroup` class. It is used to check whether the current running thread is the Parent thread of which `ThreadGroup`.

Syntax

```
public final boolean parentOf(ThreadGroup g)
```

Example:

```
class Demo4 extends Thread
{
    Demo4(String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for(int i = 0;i < 10;i++)
```

```

        {
            i++;
        }

        System.out.println(Thread.currentThread().getName() +
" ==> Execution Finished");
    }
}

public class ThreadDemo4
{
    public static void main(String arg[]) throws
InterruptedException, SecurityException, Exception
    {

        ThreadGroup o1 = new ThreadGroup("****Parent
thread****");

        ThreadGroup o2 = new ThreadGroup("****Child
thread****");

        Demo4 obj1 = new Demo4("****Thread-1****", o1);
        System.out.println(obj1.getName()+"Starting Thread
1");

        obj1.start();

        Demo4 obj2 = new Demo4 ("****Thread-2****", o2);
        System.out.println(obj2.getName()+"Starting Thread
2");

        obj2.start();

        boolean isParent = o2.parentOf(o1);

        System.out.println(o2.getName() + " is the parent of "
+ o1.getName() + ": "+ isParent);
    }
}

```



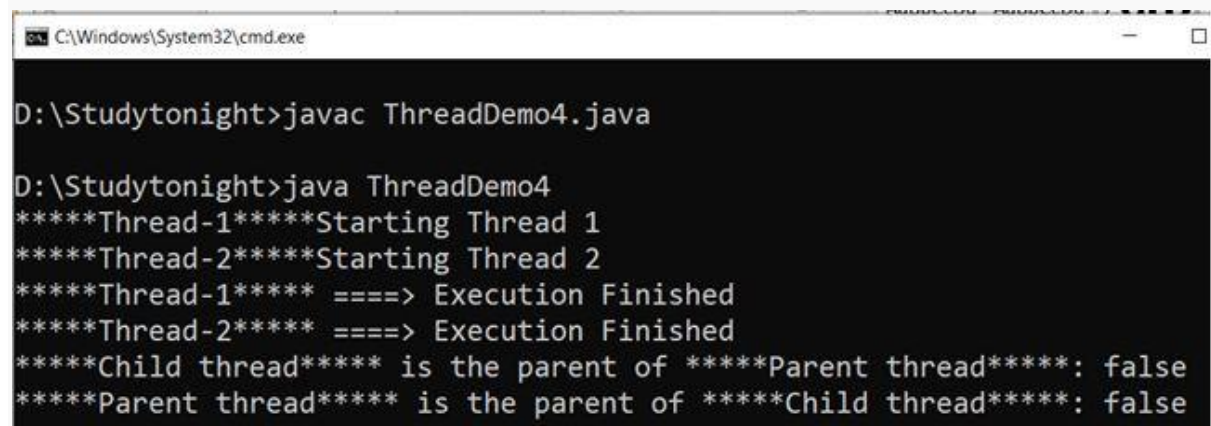
```

        isParent = o1.parentOf(o2);

        System.out.println(o1.getName() + " is the parent of "
+ o2.getName() + ": " + isParent);

    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo4.java

D:\Studytonight>java ThreadDemo4
*****Thread-1*****Starting Thread 1
*****Thread-2*****Starting Thread 2
*****Thread-1***** ==> Execution Finished
*****Thread-2***** ==> Execution Finished
*****Child thread***** is the parent of *****Parent thread*****: false
*****Parent thread***** is the parent of *****Child thread*****: false

```

15. suspend()

In Java, `suspend()` method is of `ThreadGroup` class. It is used to suspend all threads of the thread group.

Syntax

```

public final void suspend()

```

Example:

```

class Demo4 extends Thread
{
    Demo4(String a, ThreadGroup b)
    {
        super(b, a);
    }
}

```

```

    }

    public void run()
    {
        for(int i = 0;i < 10;i++)
        {
            i++;
        }

        System.out.println(Thread.currentThread().getName() +
" ==> Execution Finished");
    }
}

public class ThreadDemo4
{
    public static void main(String arg[]) throws
InterruptedException, SecurityException, Exception
    {

        ThreadGroup o1 = new ThreadGroup("****Parent
thread****");

        ThreadGroup o2 = new ThreadGroup("****Child
thread****");

        Demo4 obj1 = new Demo4("****Thread-1****", o1);
        System.out.println(obj1.getName()+"Starting Thread
1");
        obj1.start();

        Demo4 obj2 = new Demo4 ("****Thread-2****", o2);
        System.out.println(obj2.getName()+"Starting Thread
2");
        obj2.start();
    }
}

```

```

        o1.suspend();
    }
}

```

C:\Windows\System32\cmd.exe - java ThreadDemo4

```

D:\Studytonight>javac ThreadDemo4.java
Note: ThreadDemo4.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

D:\Studytonight>java ThreadDemo4
*****Thread-1*****Starting Thread 1
*****Thread-2*****Starting Thread 2
*****Thread-1***** ==> Execution Finished

```

16. resume()

In Java, `resume()` method is of ThreadGroup class. It is used to resume all threads of the thread group which were suspended.

Syntax

```
public final void resume()
```

Example:

```

class Demo4 extends Thread
{
    Demo4(String a, ThreadGroup b)
    {
        super(b, a);
    }
}

```

```

    }

    public void run()
    {
        for(int i = 0;i < 10;i++)
        {
            i++;
        }

        System.out.println(Thread.currentThread().getName() +
" ==> Execution Finished");
    }
}

public class ThreadDemo4
{
    public static void main(String arg[]) throws
InterruptedException, SecurityException, Exception
    {

        ThreadGroup o1 = new ThreadGroup("****Parent
thread****");

        ThreadGroup o2 = new ThreadGroup("****Child
thread****");

        Demo4 obj1 = new Demo4("****Thread-1****", o1);
        System.out.println(obj1.getName()+"Starting Thread
1");

        obj1.start();

        o1.suspend();

        Demo4 obj2 = new Demo4 ("****Thread-2****", o2);
        System.out.println(obj2.getName()+"Starting Thread
2");
    }
}

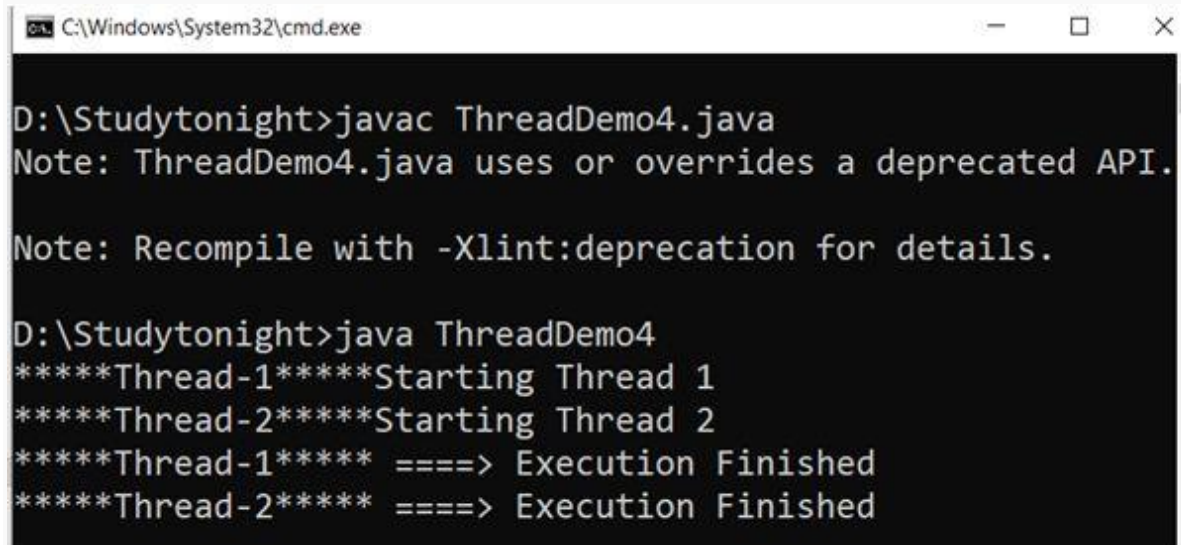
```

```
        obj2.start();

        o1.resume();

    }

}
```



A screenshot of a Windows command prompt window titled "C:\Windows\System32\cmd.exe". The prompt shows the following commands and output:

```
D:\Studytonight>javac ThreadDemo4.java
Note: ThreadDemo4.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

D:\Studytonight>java ThreadDemo4
*****Thread-1*****Starting Thread 1
*****Thread-2*****Starting Thread 2
*****Thread-1***** ==> Execution Finished
*****Thread-2***** ==> Execution Finished
```

17. setMaxPriority(intpri)

In Java, `setMaxPriority()` method is of ThreadGroup class. It is used to set the maximum priority of the thread group.

Syntax

```
public final void setMaxPriority(int pri)
```

Example:

```
class Demo5 extends Thread
{
    Demo5(String a, ThreadGroup b)
    {
```

```

        super(b, a);
    }

    public void run()
    {
        for (int i = 0; i < 10; i++)
        {
            try
            {
                Thread.sleep(10);
            }

            catch (InterruptedException ex) {
                System.out.println("Thread " +
Thread.currentThread().getName() + " ====> Interrupted"); }

        }

        System.out.println(Thread.currentThread().getName() +
" [Priority = " +
            Thread.currentThread().getPriority() + "]" );

        System.out.println(Thread.currentThread().getName() + "
====> Execution finish");
    }
}

public class ThreadDemo5
{
    public static void main(String arg[]) throws
InterruptedException,
        SecurityException, Exception
    {
        ThreadGroup o1 = new ThreadGroup("****Parent
threadGroup****");

        ThreadGroup o2 = new ThreadGroup(o1, "****Child
threadGroup****");
    }
}

```

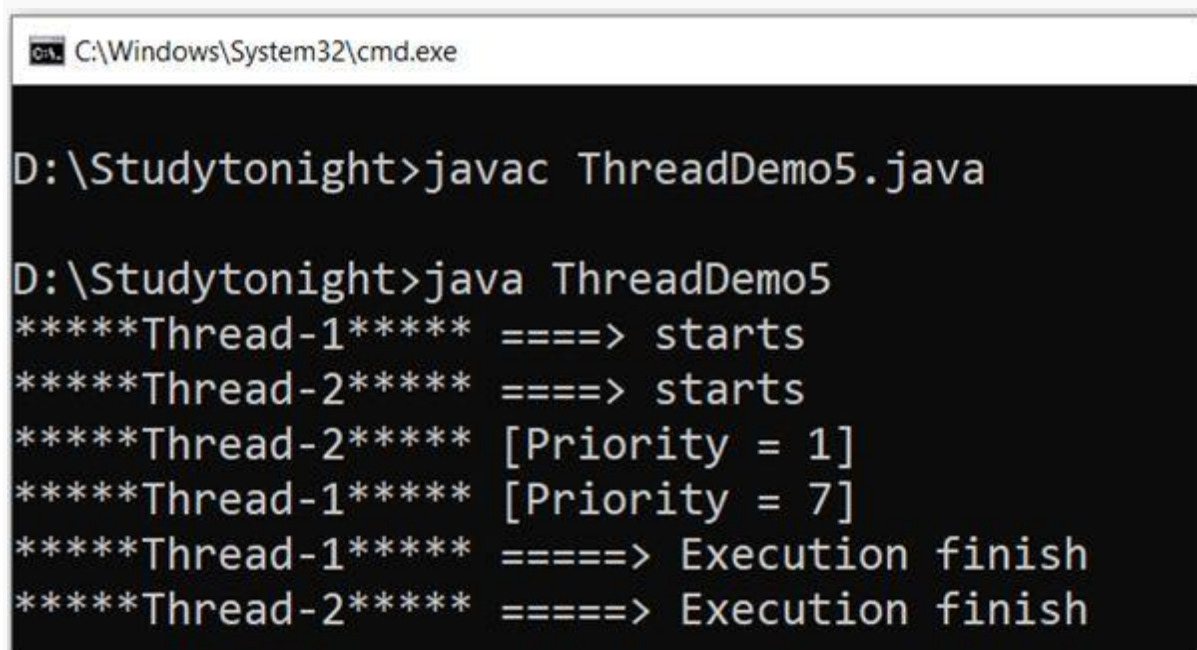
```

        o1.setMaxPriority(Thread.MAX_PRIORITY-3);
        o2.setMaxPriority(Thread.MIN_PRIORITY);

        Demo5 obj1 = new Demo5("*****Thread-1*****", o1);
        obj1.setPriority(Thread.MAX_PRIORITY);
        System.out.println(obj1.getName() + " ====> starts");
        obj1.start();

        Demo5 obj2 = new Demo5("*****Thread-2*****", o2);
        obj2.setPriority(Thread.MAX_PRIORITY);
        System.out.println(obj2.getName() + " ====> starts");
        obj2.start();
    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo5.java

D:\Studytonight>java ThreadDemo5
*****Thread-1***** ====> starts
*****Thread-2***** ====> starts
*****Thread-2***** [Priority = 1]
*****Thread-1***** [Priority = 7]
*****Thread-1***** =====> Execution finish
*****Thread-2***** =====> Execution finish

```

18. stop()

In Java, `stop()` method is of `ThreadGroup` class. It is used to stop threads of the thread group.

Syntax

```
public final void stop()
```

Example:

```
class Demo5 extends Thread
{
    Demo5(String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for (int i = 0; i < 10; i++)
        {
            try
            {
                Thread.sleep(10);
            }
            catch (InterruptedException ex) {
                System.out.println("Thread " +
Thread.currentThread().getName() + " =====> Interrupted"); }
        }
        System.out.println(Thread.currentThread().getName() +
" [Priority = " +
                Thread.currentThread().getPriority() + "]" );
    }
}
```



```

        System.out.println(Thread.currentThread().getName()+"
=====> Execution finish");
    }
}

public class ThreadDemo5
{
    public static void main(String arg[]) throws
InterruptedException,
        SecurityException, Exception
    {
        ThreadGroup o1 = new ThreadGroup("*****Parent
threadGroup*****");

        ThreadGroup o2 = new ThreadGroup(o1, "*****Child
threadGroup*****");

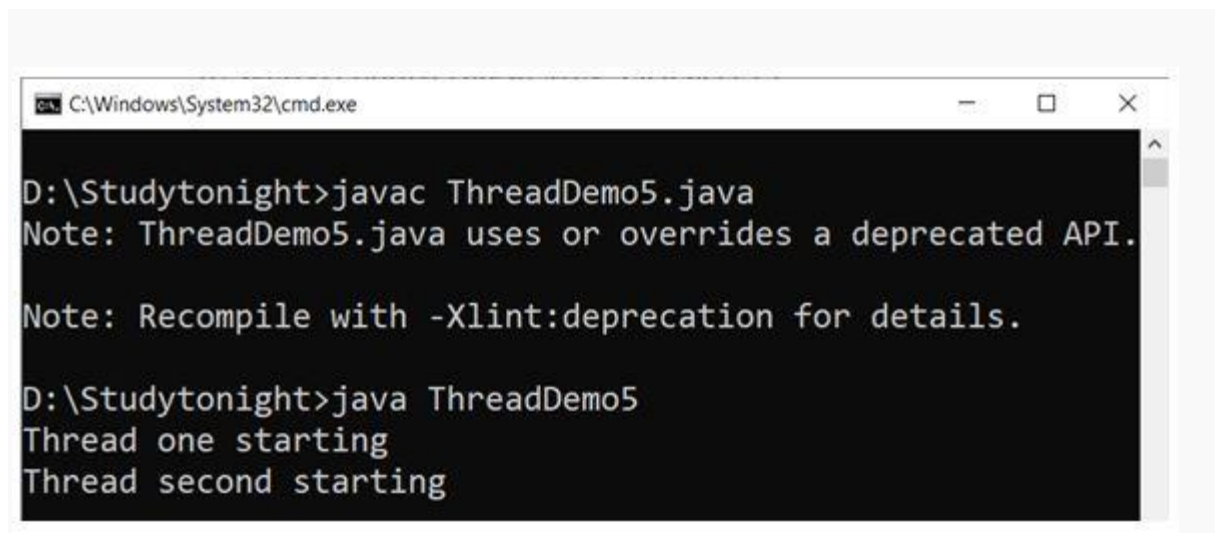
        o1.setMaxPriority(Thread.MAX_PRIORITY-3);
        o2.setMaxPriority(Thread.MIN_PRIORITY);

        Demo5 obj1 = new Demo5("*****Thread-1*****", o1);
        System.out.println("Thread one starting");
        obj1.start();

        Demo5 obj2 = new Demo5("*****Thread-2*****", o2);
        System.out.println("Thread second starting");
        obj2.start();

        o1.stop();
    }
}

```

A screenshot of a Windows command prompt window. The title bar shows 'C:\Windows\System32\cmd.exe'. The command prompt shows the following text:
D:\Studytonight>javac ThreadDemo5.java
Note: ThreadDemo5.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
D:\Studytonight>java ThreadDemo5
Thread one starting
Thread second starting

19. toString()

In Java, `toString()` method is of ThreadGroup class. It is used to get the string representation of a thread group.

Syntax

```
public String toString()
```

Example:

```
class Demo5 extends Thread
{
    Demo5(String a, ThreadGroup b)
    {
        super(b, a);
    }
    public void run()
    {
        for (int i = 0; i< 10; i++)
        {
            try
```

```

        {

            Thread.sleep(10);

        }

        catch (InterruptedException ex) {

            System.out.println("Thread " +
Thread.currentThread().getName() + " ====> Interrupted"); }

        }

        System.out.println(Thread.currentThread().getName() +
" [Priority = " +

            Thread.currentThread().getPriority() + "]" );

        System.out.println(Thread.currentThread().getName() + "
====> Execution finish");

    }

}

public class ThreadDemo5

{

    public static void main(String arg[]) throws
InterruptedException,

        SecurityException, Exception

    {

        ThreadGroup o1 = new ThreadGroup("****Parent
threadGroup****");

        ThreadGroup o2 = new ThreadGroup(o1, "****Child
threadGroup****");


        Demo5 obj1 = new Demo5("****Thread-1****", o1);

        System.out.println(obj1.getName() + " ====> starts");

        obj1.start();


        Demo5 obj2 = new Demo5("****Thread-2****", o2);

```

```

        System.out.println(obj2.getName() + " ==> starts");

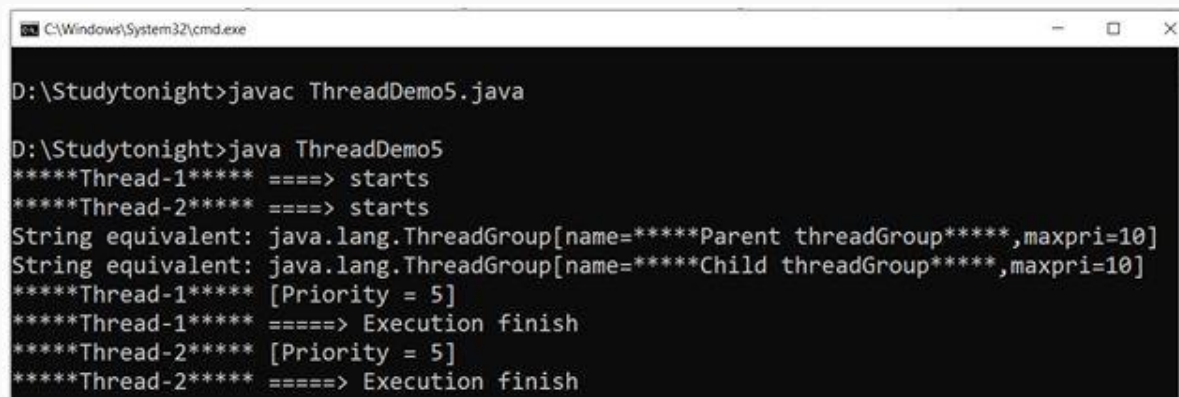
        obj2.start();

        System.out.println("String equivalent: " +
o1.toString());

        System.out.println("String equivalent: " +
o2.toString());

    }
}

```



```

C:\Windows\System32\cmd.exe

D:\Studytonight>javac ThreadDemo5.java

D:\Studytonight>java ThreadDemo5
****Thread-1**** ==> starts
****Thread-2**** ==> starts
String equivalent: java.lang.ThreadGroup[name=****Parent threadGroup****,maxpri=10]
String equivalent: java.lang.ThreadGroup[name=****Child threadGroup****,maxpri=10]
****Thread-1**** [Priority = 5]
****Thread-1**** ==> Execution finish
****Thread-2**** [Priority = 5]
****Thread-2**** ==> Execution finish

```