Client Server Application Programming

Week 7: Multi-threaded Applications (Overview, Multi-threading in Java, Synchronization, Interthread Communication, Thread Groups and Thread Priorities)

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Summary of previous Lecture

1. Transmission Control Protocol - ServerSocket Class,
2. Creating a TCP Server,
3. Running TCPClient against TCPServer
4. Exception Handling and
5. Socket-Specific Exceptions

Agenda

1. Overview of Multi-threading in Java,
2. Synchronization,
3. Interthread Communication,
4. Thread Groups and
5. Thread Priorities

Overview of Multi-threading in Java

Introduction to Java Multithreading

Multithreading is a concept of running multiple threads simultaneously. like many modern programming languages, Java support multi-threaded applications. In Java, threads of execution are represented by the **java.lang.Thread class**, while code for tasks that are designed to run in a separate thread is represented by the **java.lang.Runnable** interface. It is very important that developers be aware of both, **(**David R.,Michael R. 2002).

Thread is a lightweight unit of a process that executes in multithreading environment.

Introduction to java Multithreading+

When a program is divided into a number of small processes, it is said to be a multi-threaded program. Each small process can be addressed as a single thread (a lightweight process).

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.(Studytonigh.com, n.d.).

Thread

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

1. An **instance** of **Thread** class, or
2. 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.

How Can One Create Thread ?

To create a thread, Java provides a class called **Thread** and an interface called **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 overridden 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, (Studytonight.com,n.d.)

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 known as **main** thread.

Although the **main thread** is automatically created, you can control it by obtaining a reference to it through calling **currentThread()** method.

The main thread+ important things to know

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

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

The main thread+ calling currentThread()

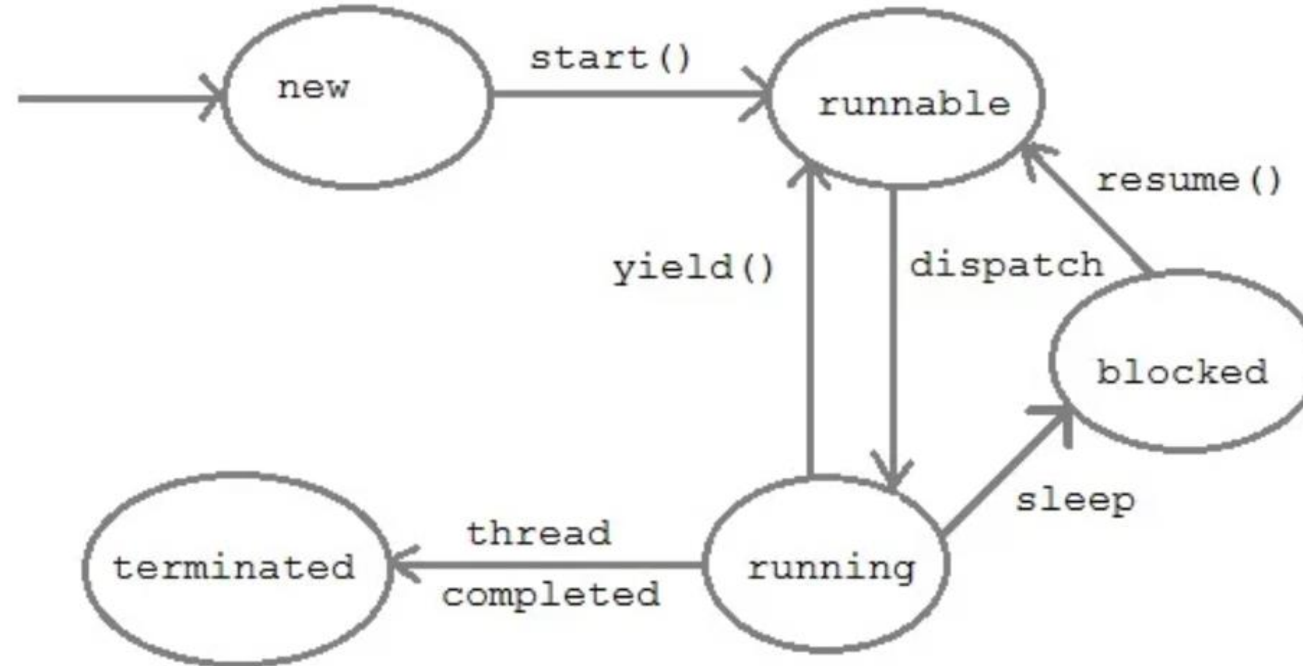
```
class MainThread{
    public static void main(String[] args){
    Thread ob = Thread.currentThread();
    ob.setName("MainThread");
    System.out.println("Name of thread is "+ob);
    }
}
```

(Studytonight.com,n.d.)

```
run:
Name of thread is Thread[MainThread,5,main]
```

Life cycle of a Thread

Like process, thread has a life cycle that includes various phases like: new, running, terminated etc. as see in the figure below.



(studytonight.com/java/multithreading-in-java.php)

Life cycle of a Thread Explained

Like process, thread has a life cycle that includes various phases like: -

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

(studytonight.com/java/multithreading-in-java.php)

Thread Creation in Java

Thread Creation in Java

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

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

Creating a thread by 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.

Syntax:

```
class MyThreadPro1 implements Runnable { }

public void run()
```

Run Method Syntax

Three things we need to know about the run() are: -

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

Example1

```
class MyThreadPro1 implements Runnable {
	public void run() {
	System.out.println("concurrent thread started 
running..");
	}
}
```

```
class MyThreadPro2 {
	public static void main(String args[]){
	MyThreadPro1 mt = new MyThreadPro1();
	Thread ob = new Thread(mt);
	ob.start(); //executes run() of MyThreadPro1 Class
	}
}
```

Output

```
package Multithreading;
public class MyThreadPro1 implements Runnable {
    public void run() {
        System.out.println("Concurrent thread started running.
    }
}
```

```
2  package Multithreading;
3  public class MyThreadPro2 {
4      public static void main(String args[]){
5          MyThreadPro1 mt = new MyThreadPro1();
6          Thread ob = new Thread(mt);
7          ob.start();
8      }
9  }
```

```
run:
Concurrent thread started running..
```

Note that start() method of Thread class is called inside MyThreadPro2 class is able to execute the run() method of MyThreadPro1 class

Key Notes

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

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.

Creating a thread by 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 MyThreadExtend extends Thread {
        public void run(){
                System.out.println("concurrent thread started running..");
        }
}
```

```
Class MyThreadExtend1{
        public static void main(String args[]){
        MyThreadExtend ob = new MyThreadExtend();
        ob.start();
        }
}
```

```
run:
Concurrent thread started running..
```

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 Synchronization

According to Wikimedia Foundation.(2022). Synchronization is the coordination of events to operate a system in unison.

However, in multithreading, 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, (studytonight.com, n.d.).

Synchronization+

General **Syntax:**

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

```
synchronized public void display (String on) {
    //statement to be synchronized
}
```

Why we need Synchronization

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

For Example, Suppose we have two different threads **X1** and **X2**, X1 starts execution and save certain values in a file *FileTodRead.txt* which will be used to calculate some result when X1 returns. Meanwhile, X2 starts and before X1 returns, X2 change the values saved by X1 in the file *FileTodRead*.txt (*FileTodRead*.txt is the shared resource). Now obviously X1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once X1 starts using *FileTodRead.txt* file, this file will be **locked**(LOCK mode), and no other thread will be able to access or modify it until X1 returns,(Studytonight.com,n.d).

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.

Example with NO Synchronization of methods

The program below shows us a **Race situation,** where object **fc** of class FirstC is shared by all the three running threads(**sc**, **sc1** and **sc2**) to call the shared **display()** method. Hence the result is nonsynchronized and such situation is called **Race condition**..

```
class FirstC{
      public void display(String msg){
      System.out.print ("["+msg);
      try {
      Thread.sleep(1500);
      }
      catch(InterruptedException e){
      e.printStackTrace();
      }
      System.out.println ("]");
}
}
```

Example with NO Synchronization of methods

```
class SecondC extends Thread{
      String msg; FirstC fob;
      SecondC (FirstC fp, String s){
      fob = fp;
      msg = s;
      start();
      }
      public void run() {
      fob.display(msg);
      }
}
```

Example with NO Synchronization of methods

```
public class SyncroC{
    public static void main (String[] args){
    FirstC fc = new FirstC();
    SecondC sc = new SecondC(fc, "welcome");
    SecondC sc1= new SecondC(fc,"new");
    SecondC sc2 = new SecondC(fc, "programmer");
}
}
```

```
run:
[new[programmer[welcome]
]
]
```

Synchronizing a program

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

Syntax

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

Example : implementation of synchronized method-FirstD class

Note that display() method is now synchronized

```
class FirstD{
      synchronized public void display(String msg){
      System.out.print ("["+msg);
      try {
      Thread.sleep(1500);
      }
      catch(InterruptedException e){
      e.printStackTrace();
      }
      System.out.println ("]");
}
}
```

Example : implementation of synchronized method-SecondD class

```
class SecondD extends Thread{
      String msg;
      FirstD fob;
      SecondD(FirstD fp, String s){
      fob = fp;
      msg = s;
      start();
      }
      public void run() {
      fob.display(msg);
      }
}
```

Example : implementation of synchronized method-SyncronD class

```
public class SyncroD{
    public static void main (String[] args){
    FirstD fc = new FirstD ();
    SecondD sc = new SecondD (fc, "welcome");
    SecondD sc1= new SecondD (fc,"new");
    SecondD sc2 = new SecondD (fc, "programmer");
}
}
```

```
run:
[welcome]
[programmer]
[new]
```

Using Synchronized block

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

Using Synchronized block

Note that display() method is NOT synchronized

```
class FirstE{
        public void display(String msg){
        System.out.print ("["+msg);
        try {
        Thread.sleep(1500);
        }
        catch(InterruptedException e){
        e.printStackTrace();
        }
        System.out.println ("]");
}
}
```

Example : implementation of synchronized method-SecondE class

```
class SecondE extends Thread{
       String msg;
       FirstE fob;
       SecondE(FirstE fp, String s){
       fob = fp;
       msg = s;
       start();
       }
       public void run(){
             synchronized(fob){
             fob.display(msg);
             }
       }
}
```

Note the introduction of synchronized (){} block in run() method.

Example : implementation of synchronized method-SyncroE class

```
public class SyncroE{
    public static void main (String[] args){
    FirstE fc = new FirstE();
    SecondE sc = new SecondE(fc, "welcome");
    SecondE sc1= new SecondE(fc,"new");
    SecondE sc2 = new SecondE(fc, "programmer");
}
}
```

Example : implementation of synchronized method-SyncroE class

Because of synchronized block in this program, it is able to produce the expected output as below;-

[welcome]

[new]

[programmer]

```
run:
[welcome]
[new]
[programmer]
```

Difference between synchronized keyword and synchronized block

1. 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.

2. 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**.

Interthread Communication,

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 methods can be called **only** from within a **synchronized** context

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

Difference between wait() and sleep()

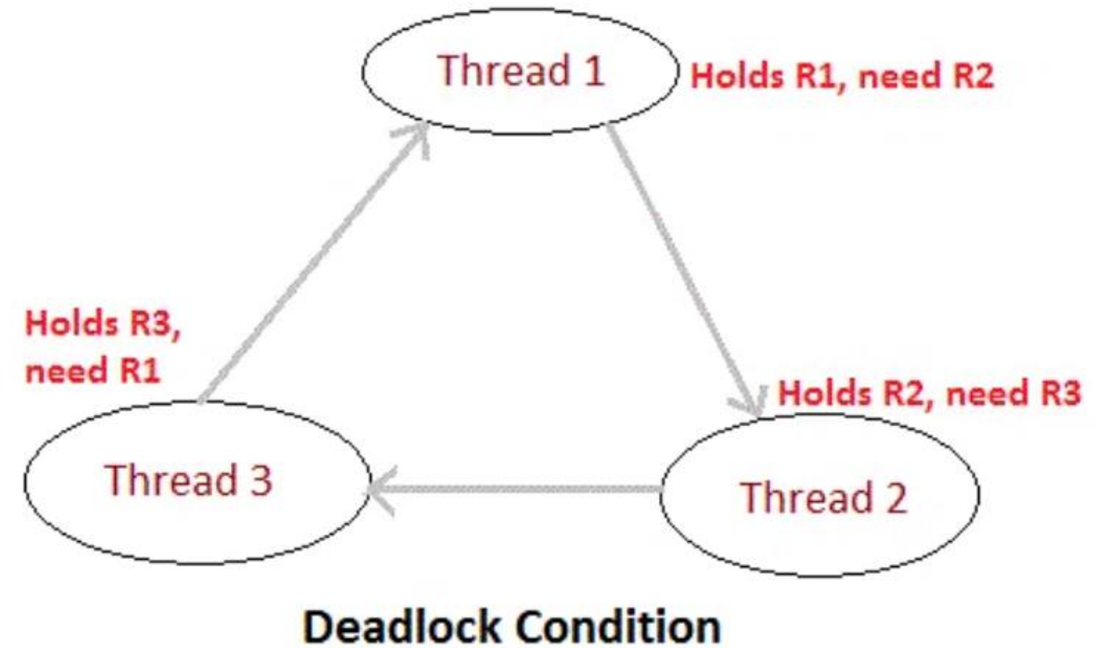
wait()	sleep()
called from synchronised block	no such requirement
monitor is released	monitor is not released
gets awake when notify() or notifyAll() method is called.	does not get awake when notify() or notifyAll() method is called
not a static method	static method
wait() is generaly used on condition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

(Studytonight.com, n.d.)

Thread Deadlock in Java

Deadlock is a situation of complete Lock, when no thread can complete its execution because of lack of resources.

For example. In the below 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.



(Studytonight.com,n.d)

Example: Deadlock

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 yet deadlock condition wait time is never ending. We will run three classes pen, paper and writer.

```
class Pen{}
```

```
class Paper{}
```

Example: Deadlock+

```
public class Writer{
      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){   }
            synchronized(pr){
            System.out.println("Requesting for Paper");
            }
      }
      }
};
```

Example: Deadlock++

```
Thread t2 = new Thread(){
      public void run(){
      synchronized(pr) {
            System.out.println("Thread2 is holding Paper");
      try {
            Thread.sleep(1000);
      }
      catch(InterruptedException e){ }
      synchronized(pn) {
      System.out.println("requesting for Pen");
      }
      }
      }
      };
      t1.start();
      t2.start();
}
}
```

Example: Deadlock-Ouput

```
run:
Thread1 is holding Pen
Thread2 is holding Paper
```

Note that this program build is not completing

Thread Group

ThreadGroup Class

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 group **but cannot have the information of the threads of the other groups**. It is very useful in the case where we want to suspend and resume a number of threads. This thread group is implemented by **java.lang.ThreadGroup class,** (studytonight.com, n.d).

Types of Constructors in the ThreadGroup Class

There are two types of Constructors in the Thread group. As follow:

1. public ThreadGroup(String name)

2. public ThreadGroup(ThreadGroup parent, String name)

Methods in the ThreadGroup Class

Following are the methods present in Thread group

1. checkAccess()
2. activeCount()
3. activeGroupCount()
4. destroy()
5. enumerate(Thread[] list)
6. getMaxPriority()
7. getName()
8. getParent()
9. interrupt()
10. isDaemon()
11. setDaemon(Boolean daemon)

And more 8

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: checkAccess()

```
class ThreadCAccess extends Thread {
     ThreadCAccess(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());
     }
}
```

Example: checkAccess()+

```
public class ThreadCAccess1 extends Thread{
        public ThreadCAccess1(String thread1, ThreadGroup obj1) {   }
        public static void main(String arg[]) throws InterruptedException,
        SecurityException{
        ThreadGroup obj1 = new ThreadGroup("Parent thread ==> ");
        ThreadGroup obj2 = new ThreadGroup(obj1, "Child thread ==> ");
        ThreadCAccess1 t1 = new ThreadCAccess1("*Thread-1*", obj1);
        t1.start();
        ThreadCAccess1 t2 = new ThreadCAccess1("*Thread-2*", obj1);
        t2.start();
        obj1.checkAccess();
        System.out.println(obj1.getName() + " has access");
        obj2.checkAccess();
        System.out.println(obj2.getName() + " has access");
        }
}
```

Two Throups created. i.e Obj1 and Obj2

Two threads t1 and t2 placed in ThreadGroup Obj1

Example: checkAccess()+ Output

```
1   package Multithreading;
2   public class ThreadCAccess1 extends Thread{
3       public ThreadCAccess1(String thread1, ThreadGroup obj1) {
4       }
5       public static void main(String arg[]) throws
6               InterruptedException,SecurityException{
7           ThreadGroup obj1 = new ThreadGroup("Parent thread ==> ");
8           ThreadGroup obj2 = new ThreadGroup(obj1, "Child thread ==> ");
9           ThreadCAccess1 t1 = new ThreadCAccess1("**Thread-1**", obj1);
10          t1.start();
11          ThreadCAccess1 t2 = new ThreadCAccess1("**Thread-2**", obj1);
12          t2.start();
13          obj1.checkAccess();
            System.out.println(obj1.getName() + " has access");
15          obj2.checkAccess();
16          System.out.println(obj2.getName() + " has access");
17          }
18
19      }
```

```
run:
Parent thread ==>  has access
Child thread ==>  has access
```

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.

Syntax

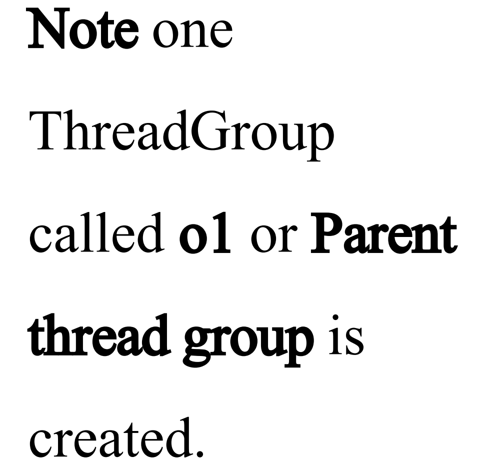
```
public static int activeCount()
```

Example: activeCount()

```
class ThreadACounter extends Thread {
      ThreadACounter (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());
      }
}
```

Example: activeCount()+

```
public class ThreadACounter1 extends Thread{
        public static void main(String arg[]) {
        ThreadGroup o1 = new ThreadGroup("Parent thread group");
        ThreadACounter obj1 = new ThreadACounter("Thread 1 ==> ", o1);
        ThreadACounter obj2 = new ThreadACounter("Thread 2 ==> ", o1);
        ThreadACounter obj3 = new ThreadACounter("Thread 3 ==> ", o1);
        ThreadACounter obj4 = new ThreadACounter("Thread 4 ==> ", o1);
        ThreadACounter obj5 = new ThreadACounter("Thread 5 ==> ", o1);
        ThreadACounter obj6 = new ThreadACounter("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());
}
}
```



Example: activeCount()+ Output

```
1    package Multithreading;
2    public class ThreadACounter1 extends Thread {
3        public static void main(String arg[]) {
4        ThreadGroup o1 = new ThreadGroup("parent thread group");
5        ThreadACounter obj1 = new ThreadACounter("Thread 1 ==> ", o1);
6        ThreadACounter obj2 = new ThreadACounter("Thread 2 ==> ", o1);
7        ThreadACounter obj3 = new ThreadACounter("Thread 3 ==> ", o1);
8        ThreadACounter obj4 = new ThreadACounter("Thread 4 ==> ", o1);
9        ThreadACounter obj5 = new ThreadACounter("Thread 5 ==> ", o1);
10       ThreadACounter obj6 = new ThreadACounter("Thread 6 ==> ", o1);
11       obj1.start();
12       obj2.start();
13       obj3.start();
14       obj4.start();
15       obj5.start();
16       obj6.start();
17       System.out.println("Total number of active"
18               + " thread ==> "+ o1.activeCount());
19   }
20   }
```

```
run:
Total number of active thread ==> 6
Thread 4 ==>
Thread 5 ==>
Thread 2 ==>
Thread 6 ==>
Thread 1 ==>
Thread 3 ==>
```

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()
```

Example: activeGroupCount()

```
class activeGroupC extends Thread {
	activeGroupC (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 
execution");
	}
}
```

Example: activeGroupCount()

```
public class activeGroupC1 {
      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");
      activeGroupC obj1 = new activeGroupC("**Thread 1**",o1);
      System.out.println(obj1.getName() + " ==> starts");
      obj1.start();
      System.out.println("Total number of active ThreadGroups ==> "+
      o1.activeGroupCount());
}
}
```

```
run:
**Thread 1** ==> starts
Total number of active ThreadGroup ==> 2
**Thread 1** ==> completed execution
```

4. destroy()

is used to destroy a thread group. To destroy any thread group, all the threads in that group should be stopped first.

syntax

```
public void destroy()
```

Example: destroy()

```
class threadGDestroy extends Thread {
	threadGDestroy (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 
execution");
	}
}
```

Example: destroy()+

```
public class threadGDestroy1 {
      public static void main(String arg[]) throws InterruptedException,
                                          SecurityException {
      ThreadGroup o1 = new ThreadGroup("Parent thread group");
      ThreadGroup o2 = new ThreadGroup(o1, "Child thread group");
      threadGDestroy obj1 = new threadGDestroy("Thread 1",o1);
      obj1.start();
      threadGDestroy obj2 = new threadGDestroy("*****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");
}
}
```

Example: destroy()+ Output

```
1   package Multithreading;
2   public class threadGDestroy1 {
3       public static void main(String arg[]) throws InterruptedException,
4                                                    SecurityException {
5           ThreadGroup o1 = new ThreadGroup("Parent thread group");
6           ThreadGroup o2 = new ThreadGroup(o1, "Child thread group");
7           threadGDestroy obj1 = new threadGDestroy("Thread 1",o1);
8           obj1.start();
9           threadGDestroy obj2 = new threadGDestroy("**Thread 2**",o1);
10          obj2.start();
11          obj1.join();
12          obj2.join();
            o2.destroy();
14          System.out.println(o2.getName()+" ====> Destroyed");
            o1.destroy();
16          System.out.println(o1.getName()+" ====> Destroyed");
17      }
18  }
```

```
run:
Thread 1 ==> completed execution
**Thread 2** ==> completed execution
Child thread group ====> Destroyed
Parent thread group ====> Destroyed
```

5. enumerate(Thread[] list)

is used for copying active threads into a specified array.

Syntax

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

Example: **enumerate(Thread[] list)**

```
class CopyActiveThread extends Thread {
      CopyActiveThread (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 execution");
      }
}
```

Example: **enumerate(Thread[] list)+**

```
public class CopyActiveThread1 {
        public static void main(String arg[]) {
        ThreadGroup o1 = new ThreadGroup("Parent thread group");
        ThreadGroup o2 = new ThreadGroup(o1,"Child thread group");
        CopyActiveThread obj1 = new CopyActiveThread("**Thread 1**",o1);
        System.out.println("Thread 1 Starts");
        obj1.start();
        CopyActiveThread obj2 = new CopyActiveThread("**Thread 2**",o1);
        System.out.println("Thread 2 Starts");
        obj2.start();
        Thread[] tarray = new Thread[o1.activeCount()];
        int count1 = o1.enumerate(tarray);
        for (int i = 0; i< count1; i++)
        System.out.println(tarray[i].getName() + " ==> Found");
        }
}
```

```java
package Multithreading;
public class CopyActiveThread1 {
        public static void main(String arg[]) {
        ThreadGroup o1 = new ThreadGroup("Parent thread group");
        ThreadGroup o2 = new ThreadGroup(o1,"Child thread group");
        CopyActiveThread obj1 = new CopyActiveThread("**Thread 1**",o1);
        System.out.println("Thread 1 Starts");
        obj1.start();
        CopyActiveThread obj2 = new CopyActiveThread("**Thread 2**",o1);
        System.out.println("Thread 2 Starts");
        obj2.start();
        Thread[] tarray = new Thread[o1.activeCount()];
        int count1 = o1.enumerate(tarray);
        for (int i = 0; i< count1; i++)
        System.out.println(tarray[i].getName() + " ==> Found");
        }
}
```

```
run:
Thread 1 Starts
Thread 2 Starts
**Thread 1** ==> Found
**Thread 2** ==> Found
**Thread 2** ==> completed execution
**Thread 1** ==> completed execution
```

6. getMaxPriority()

is used to check the maximum priority of the thread group.

Syntax

```
public final int getMaxPriority()
```

7. getName()

is used for getting the name of the current thread group. We looked at how this method work eerily in this lecture.

8. getParent()

is used to get the parent thread from the thread group.

```
public final ThreadGroup getParent()
```

Example: **getParent()**

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

Example: **getParent()+**

```
public class getParentThread1{
      public static void main(String arg[]) throws InterruptedException,
                        SecurityException, Exception {
      ThreadGroup o1 = new ThreadGroup("*Parent thread*");
      ThreadGroup o2 = new ThreadGroup(o1, "*Child thread*");
      getParentThread obj1 = new getParentThread("Thread-1", o1);
      System.out.println("First Thread starting");
      try{obj1.start();} catch(IllegalThreadStateException e){}
      getParentThread obj2 = new getParentThread("Thread-2", o2);
      System.out.println("Second Thread starting");
      try{obj2.start();} catch(IllegalThreadStateException e){}
      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());
}
}
```

Example: **getParent()+ Output**

```
1   package Multithreading;
2   public class getParentThread1{
3       public static void main(String arg[]) throws InterruptedException,
4                               SecurityException, Exception {
5       ThreadGroup o1 = new ThreadGroup("*Parent thread*");
6       ThreadGroup o2 = new ThreadGroup(o1, "*Child thread*");
7       getParentThread obj1 = new getParentThread("Thread-1", o1);
8       System.out.println("First Thread  starting");
9      try{
10      obj1.start();
11     }catch(IllegalThreadStateException e){}
12      getParentThread obj2 = new getParentThread("Thread-2", o2);
13      System.out.println("Second Thread  starting");
14      try{
15      obj2.start();
16     }catch(IllegalThreadStateException e){}
17      System.out.println("Parent Thread Group for " + o1.getName() +
18              " is " + o1.getParent().getName());
19      System.out.println("Parent Thread Group for " + o2.getName() +
20              " is " + o2.getParent().getName());
21  }
22  }
```

```
run:
First Thread  starting
Second Thread  starting
Thread-1
Thread-2
Parent Thread Group for *Parent thread* is main
Parent Thread Group for *Child thread* is *Parent thread*
```

9. interrupt()

is used to interrupt all the threads of the thread group.

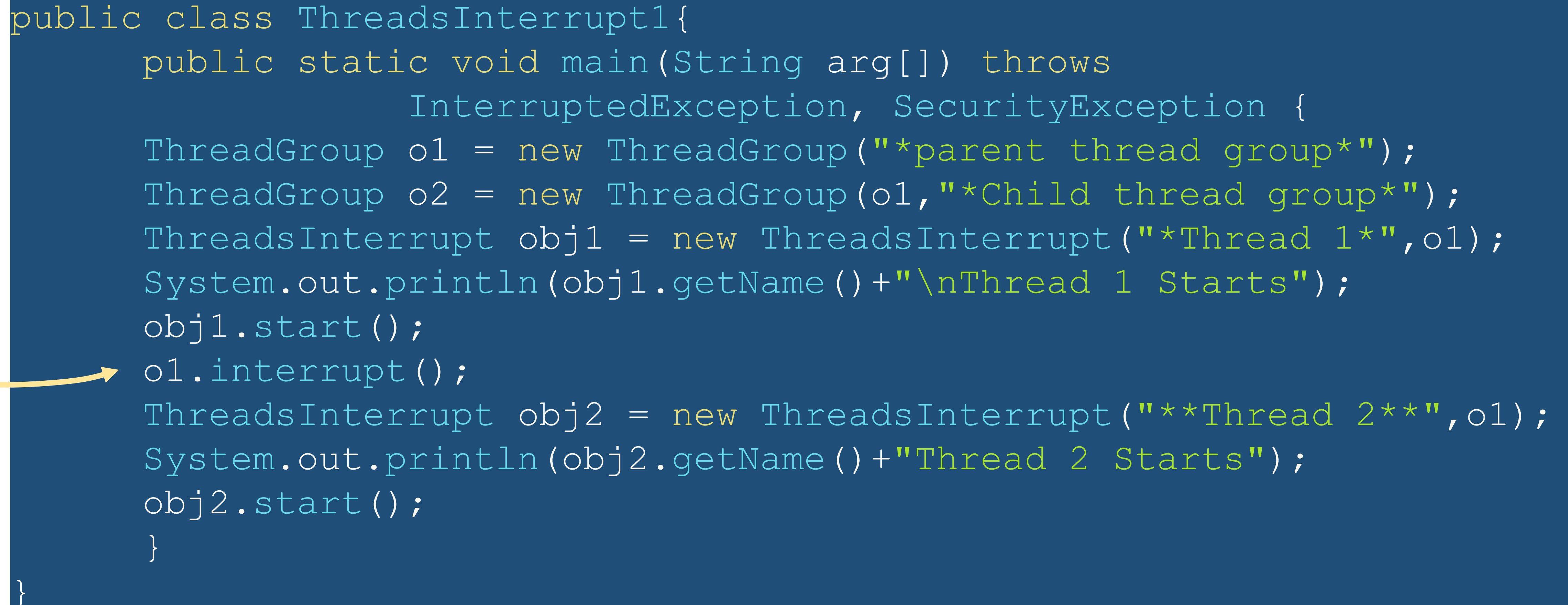
Syntax

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

Example of Interrupt Thread

```
class ThreadsInterrupt extends Thread {
        ThreadsInterrupt (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
execution");
}
}
```

Example of Interrupt thread+



```
public class ThreadsInterrupt1{
      public static void main(String arg[]) throws
                  InterruptedException, SecurityException {
      ThreadGroup o1 = new ThreadGroup("*parent thread group*");
      ThreadGroup o2 = new ThreadGroup(o1,"*Child thread group*");
      ThreadsInterrupt obj1 = new ThreadsInterrupt("*Thread 1*",o1);
      System.out.println(obj1.getName()+"\nThread 1 Starts");
      obj1.start();
      o1.interrupt();
      ThreadsInterrupt obj2 = new ThreadsInterrupt("**Thread 2**",o1);
      System.out.println(obj2.getName()+"Thread 2 Starts");
      obj2.start();
      }
}
```

Note that thread obj1 was interrupted but thread obj2 wasn't

Example of Interrupt thread Output

```
1   package Multithreading;
2   public class ThreadsInterrupt1{
3       public static void main(String arg[]) throws
4           ThreadGroup o1 = new ThreadGroup("*parent thread group*");
5           ThreadGroup o2 = new ThreadGroup(o1,"*Child thread group*");
6           ThreadsInterrupt obj1 = new ThreadsInterrupt("*Thread 1*",o1);
7           System.out.println(obj1.getName()+"\nThread 1 Starts");
8           obj1.start();
9           o1.interrupt();
10          ThreadsInterrupt obj2 = new ThreadsInterrupt("**Thread 2**",o1);
11          System.out.println(obj2.getName()+"\nThread 2 Starts");
12          obj2.start();
13          }
14  }
```

```
run:
*Thread 1*
Thread 1 Starts
**Thread 2**
Thread 2 Starts
*Thread 1* ==> interrupted
*Thread 1* ==> completed execution
**Thread 2** ==> completed execution
```

10. isDaemon()

is used to check whether a thread is Daemon thread or not.

Syntax

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

11. setDaemon(Boolean daemon)

is used to make a thread as a daemon thread.

Syntax:

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

Other methods

12. list()

is used to getting all the information about a thread group. It is very useful at the time of debugging.

13. isDestoryed()

is used to check whether the thread group is destroyed or not.

14. ParentOf(ThreadGroup g)

is used to check whether the current running thread is the Parent thread of which ThreadGroup.

15. suspend()

is used to suspend all threads of the thread group

16. resume()

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

17. setMaxPriority(intpri)

is used to set the maximum priority of the thread group.

18. stop()

is used to stop threads of the thread group

Thread Priority in Java

Thread Priority in Java

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 as below:-

Thread Priority in Java + Types of priority

Java provide three kinds of priority options that can be applied to threads, as below;

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. (Studytonight.com. (n.d.).

Get and Set methods in Thread priority

1. **public final int getPriority();** In Java, getPriority() method is found in java.lang.Thread package. it is used to get the priority of a thread.
2. **public final void setPriority(intnewPriority);** In Java setPriority(intnewPriority) 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 Default Thread Priority

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

Example: Fetch Thread Priority+

```
class getThreadPrio extends Thread {
      public void run() {
      System.out.println("Thread Running...");
      }
      public static void main(String[]args) {
            getThreadPrio p1 = new getThreadPrio();
            getThreadPrio p2 = new getThreadPrio();
            getThreadPrio p3 = new getThreadPrio();
            p1.start();
            System.out.println("First thread priority : " +
            p1.getPriority());
      System.out.println("Second thread priority : " + p2.getPriority());
      System.out.println("Third thread priority : " + p3.getPriority());
      }
}
```

Example: Fetch Thread Priority++ Output

```java
package Multithreading;
class getThreadPrio extends Thread {
        public void run() {
        System.out.println("Thread Running...");
        }
    public static void main(String[]args) {
        getThreadPrio p1 = new getThreadPrio();
        getThreadPrio p2 = new getThreadPrio();
        getThreadPrio p3 = new getThreadPrio();
        p1.start();
        System.out.println("First thread priority : " + p1.getPriority());
        System.out.println("Second thread priority : " + p2.getPriority());
        System.out.println("Third thread priority : " + p3.getPriority());
        }
}
```

```
run:
Thread Running...
First thread priority : 5
Second thread priority : 5
Third thread priority : 5
```

Example: Fetch Thread priority based on 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 getThreadPrio1 extends Thread {
      public void run() {
      System.out.println("Thread Running...");
      }
      public static void main(String[]args) {
      getThreadPrio1 p1 = new getThreadPrio1();
      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);
      }
}
```

Example: Fetch Thread priority based on **Constants-Output**

```
1   package Multithreading;
2   class getThreadPrio1 extends Thread {
        public void run() {
4           System.out.println("Thread Running...");
5       }
6       public static void main(String[]args) {
7           getThreadPrio1 p1 = new getThreadPrio1();
8           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);
12      }
13  }
```

```
run:
Thread Running...
Max thread priority : 10
Min thread priority : 1
Normal thread priority : 5
```

Example 1: Set Thread Priority

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

```
class setThreadPrio extends Thread {
      public void run() {
      System.out.println("Thread Running...");
      }
      public static void main(String[]args) {
       setThreadPrio p1 = new setThreadPrio();
      p1.start();
      p1.setPriority(3);
      int p = p1.getPriority();
      System.out.println("Thread priority : " + p);
      }
}
```

Example 1 : Set Thread Priority-Output

```
1   package Multithreading;
2   class setThreadPrio extends Thread {
3       public void run() {
4       System.out.println("Thread Running...");
5       }
6       public static void main(String[]args) {
7       setThreadPrio p1 = new setThreadPrio();
8       p1.start();
9       p1.setPriority(3); // Seting priority
10      int p = p1.getPriority();
11      System.out.println("Thread priority : " + p);
12      }
13  }
```

```
run:
Thread Running...
Thread priority : 3
```

Example2 : Set Thread Priority

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 setThreadPrio1 extends Thread {
      public void run() {
      System.out.println("Thread Running...
                        "+Thread.currentThread().getName()); }
      public static void main(String[]args) {
      setThreadPrio1 p1 = new setThreadPrio1();
      setThreadPrio1 p2 = new setThreadPrio1(); // Starting thread
      p1.start();
      p2.start(); // Setting priority
      p1.setPriority(2);
      p2.setPriority(1); // Getting -priority
      int p = p1.getPriority();
      int q = p2.getPriority();
      System.out.println(" First thread priority : " + p);
      System.out.println(" Second thread priority : " + q); } }
```

Example2 : Set Thread Priority-Output

```
1  package Multithreading;
2  class setThreadPrio1 extends Thread {
3      public void run() {
4          System.out.println("Thread Running..."
5                  + ""+Thread.currentThread().getName()); }
6          public static void main(String[]args) {
7          setThreadPrio1 p1 = new setThreadPrio1();
8          setThreadPrio1 p2 = new setThreadPrio1();
9              p1.start();
10             p2.start(); // Setting priority
11             p1.setPriority(2);
12             p2.setPriority(1); // Getting -priority
13             int p = p1.getPriority();
14             int q = p2.getPriority();
15         System.out.println("First thread priority : " + p);
16         System.out.println("Second thread priority : " + q);
17         }
18 }
```

```
run:
Thread Running...Thread-1
Thread Running...Thread-0
First thread priority : 2
Second thread priority : 1
```

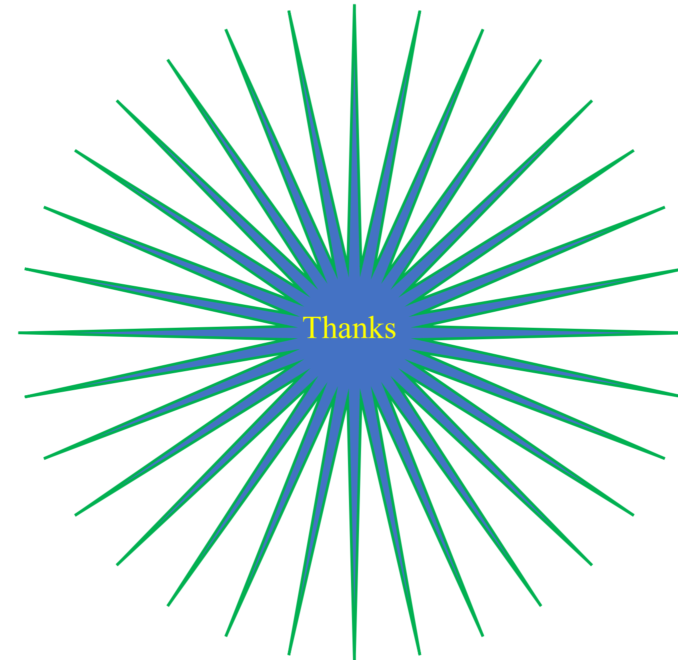
Note: Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

Summary

Summary

1. Overview of Multi-threading in Java,
2. Synchronization,
3. Interthread Communication,
4. Thread Groups and
5. Thread Priorities

Thank you for Listening



References

Java™ Network Programming and Distributed Computing, (David R.,Michael R. 2002), Publisher : Addison Wesley; ISBN: 0201710374

Introduction to multithreading in Java. Studytonight.com. (n.d.). Retrieved September 23, 2022, from https://www.studytonight.com/java/multithreading-in-java.php

Java thread class. Studytonight.com. (n.d.). Retrieved September 23, 2022, from https://www.studytonight.com/java/thread-class-and-functions.php

Creating a thread in Java. Studytonight.com. (n.d.). Retrieved September 23, 2022, from https://www.studytonight.com/java/creating-a-thread.php

Java thread priorities. Studytonight.com. (n.d.). Retrieved September 22, 2022, from https://www.studytonight.com/java/thread-priorities-in-java.php