

# Object - Oriented Programming 2

Week 5: Multithreading In Java( Synchronization, Interthread Communication, Thread group )

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

[Email: josebulinda@gmail.com](mailto:josebulinda@gmail.com)

[jose@kumiuniversity.ac.ug](mailto:jose@kumiuniversity.ac.ug)

# Agenda

1. Synchronization,
2. Interthread Communication,
3. Thread group

# 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 synchronization, 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.

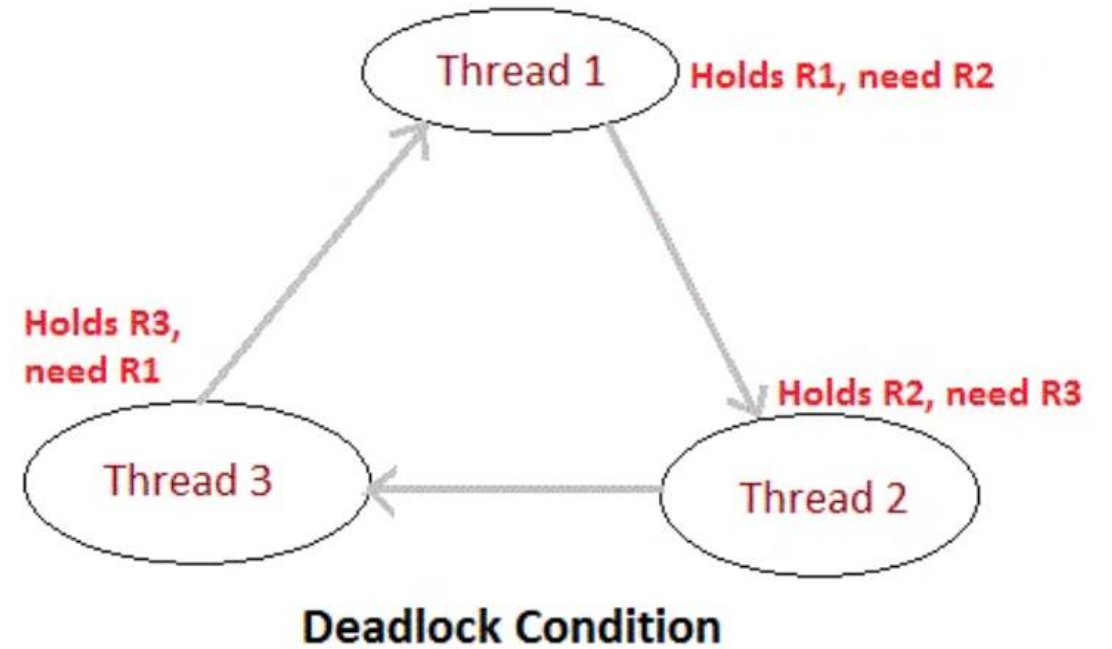
# Difference between `wait()` and `sleep()`

<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 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-Output

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

**Note** that this program build is not completing

# ThreadGroup

# 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();
14        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());
    }
}
```



**Note one**  
ThreadGroup  
called **o1** or **Parent**  
**thread group** is  
created.

# 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();
13        o2.destroy();
14        System.out.println(o2.getName()+" ==> Destroyed");
15        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");
    }
}
```

```

1 package Multithreading;
2 public class CopyActiveThread1 {
3     public static void main(String arg[]) {
4         ThreadGroup o1 = new ThreadGroup("Parent thread group");
5         ThreadGroup o2 = new ThreadGroup(o1, "Child thread group");
6         CopyActiveThread obj1 = new CopyActiveThread("**Thread 1**", o1);
7         System.out.println("Thread 1 Starts");
8         obj1.start();
9         CopyActiveThread obj2 = new CopyActiveThread("**Thread 2**", o1);
10        System.out.println("Thread 2 Starts");
11        obj2.start();
12        Thread[] tarray = new Thread[o1.activeCount()];
13        int count1 = o1.enumerate(tarray);
14        for (int i = 0; i < count1; i++)
15            System.out.println(tarray[i].getName() + " ==> Found");
16    }
17 }

```

```

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 works internally 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

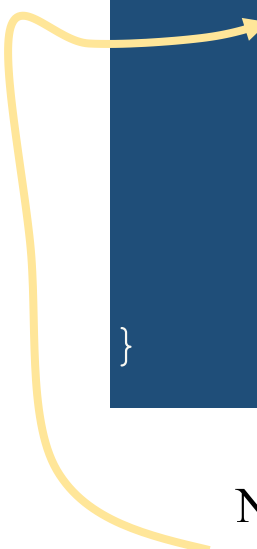
```
public final void interrupt()
```

# 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.

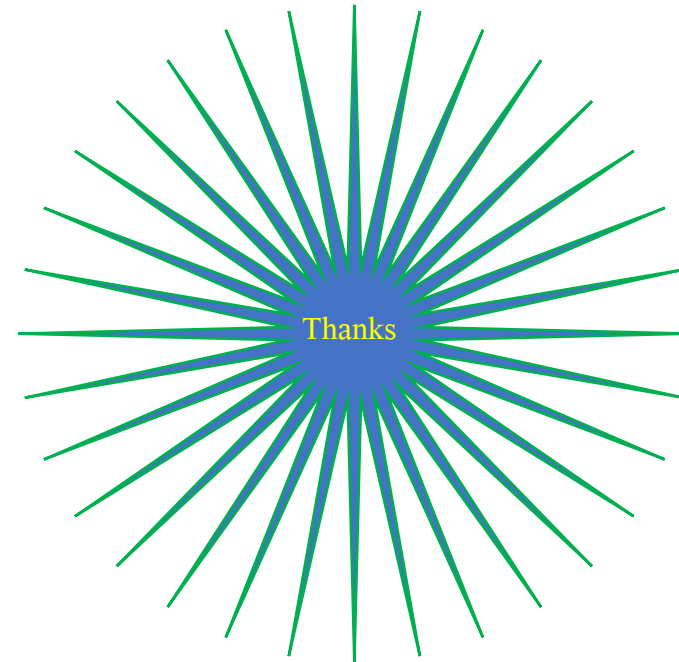
## 18. `stop()`

is used to stop threads of the thread group

# Summary

1. Synchronization,
2. Interthread Communication,
3. Thread group (looked at various methods that runs this this class)

Thank you for  
Listening



# References

Wikimedia Foundation. (2022, July 23). *Synchronization*. Wikipedia. Retrieved October 3, 2022, from <https://en.wikipedia.org/wiki/Synchronization>

*Java Thread Group*. Studytonight.com. (n.d.). Retrieved September 12, 2022, from <https://www.studytonight.com/java/java-thread-group.php>

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

*Java interthread communication*. Studytonight.com. (n.d.). Retrieved September 13, 2022, from <https://www.studytonight.com/java/interthread-communication.php>