Discord link :: https://discord.com/invite/wyEAPaQEBN
Telegram link :: https://t.me/+PIaEH8mE9sxkMzll

How to kill a Thread in the middle of the execution?

stop()-> This method is used to kill the thread in the middle of execution.
It would enter into dead state immediately.
This method is deperecated in java and it is not recomended to use.

How to suspend() and resume() of a Thread?

=> remember the example of governement office guy getting suspend and later on getting resumed for his work.

```
public void suspend()
public void resume()
```

=> we can suspend a thread by using suspend() method call of Thread class,then thread will enter into suspended state.

=> we can resume a thread by using resume() method call of Thread class,then suspended thread can contain its execution.

=> All these methods are deprecated method of Thread class, it is not a good practise to use in our programming.

ThreadGroup
+++++++++++

```
class MyThread extends Thread
{

      MyThread(ThreadGroup g, String name)
      {
            super(g,name);
      }

      @Override
      public void run()
      {
            System.out.println("Child Thread");
            try
            {
                  Thread.sleep(2000);
            }
            catch (InterruptedException ie)
            {
            }
      }
}

public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{

                  ThreadGroup indGroup = new ThreadGroup("IND");
                  System.out.println(indGroup);
                  ThreadGroup benGroup = new ThreadGroup(indGroup,"BENG");
                  System.out.println(benGroup);

                  MyThread t1 =new MyThread(indGroup,"sachin");
                  MyThread t2 =new MyThread(indGroup,"dravid");
                  System.out.println(t1);
                  System.out.println(t2);
                  System.out.println();

                  t1.start();
                  t2.start();

                  System.out.println("Active Threads in a Group::
"+indGroup.activeCount());

                  System.out.println("Active ThreadGroup ::
"+indGroup.activeGroupCount());

                  indGroup.list();

                  Thread.sleep(5000);

                  System.out.println("Active Threads in a Group::
"+indGroup.activeCount());

                  indGroup.list();

                  System.out.println("End of main method...");
      }
}
```

Output

```
java.lang.ThreadGroup[name=IND,maxpri=10]
java.lang.ThreadGroup[name=BENG,maxpri=10]
Thread[sachin,5,IND]
Thread[dravid,5,IND]

Child Thread
Child Thread
Active Threads in a Group:: 2
Active ThreadGroup :: 1
java.lang.ThreadGroup[name=IND,maxpri=10]
    Thread[sachin,5,IND]
    Thread[dravid,5,IND]
    java.lang.ThreadGroup[name=BENG,maxpri=10]
Active Threads in a Group:: 0
java.lang.ThreadGroup[name=IND,maxpri=10]
    java.lang.ThreadGroup[name=BENG,maxpri=10]
End of main method...
```

eg#2.

```
public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{
            ThreadGroup system=Thread.currentThread().getThreadGroup().getParent();
            System.out.println(system);

            Thread[] t= new Thread[system.activeCount()];
            //Copy all active subThreadGroups into ThreadGroup Array
            system.enumerate(t);

            for ( Thread obj:t)
            {
                  System.out.println(obj);
                  System.out.println(obj.getName()+" Is Daemon::
"+obj.isDaemon());
                  System.out.println();
            }

      }
}
```

```
Output
java.lang.ThreadGroup[name=system,maxpri=10]
Thread[Reference Handler,10,system]
Reference Handler Is Daemon::  true

Thread[Finalizer,8,system]
Finalizer Is Daemon::  true

Thread[Signal Dispatcher,9,system]
Signal Dispatcher Is Daemon::  true

Thread[Attach Listener,5,system]
Attach Listener Is Daemon::  true

Thread[main,5,main]
main Is Daemon::  false
```

ThreadLocal
+++++++++++

=> It provides ThreadLocal varaibles.

=> It maintains a value on the basis of Threads.

=> Each ThreadLocal varaible maintains a seperate value like userId,transactionId for each thread which can access the object.

=> Thread can access ThreadLocal value, it can manipulate and it can also remove the value from ThreadLocal object.

=> A thread can access its own ThreadLocal varaible value, but not other threads Local varaibles.

=> Once the Thread enters into dead state, the Thread Local variable by default will be eligible for Garabage Collection.

eg#1.

```
public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args)throws Exception{

            ThreadLocal tl = new ThreadLocal(){

                  @Override
                  protected Object initialValue(){
                        System.out.println("Method getting called...");
                        return "sachin";
                  }
            };
            System.out.println("Getting the TL variable :: "+tl.get());

            System.out.println();

            tl.set("dhoni");
            System.out.println("Getting the TL variable :: "+tl.get());

            System.out.println();

            tl.remove();
            System.out.println("Getting the TL variable :: "+tl.get());

      }
}
```

Output

```
Method getting called...
Getting the TL variable :: sachin

Getting the TL variable :: dhoni

Method getting called...
Getting the TL variable :: sachin
```

JDK8 Features
+++++++++++++

```
java 1.7 ----> july 2011
              |=> 3 years of hardwork by Oracle Community
java 1.8 ----> March 2014
```

Major Version of java earlier was JDK1.5 with features like

a. Annotations
b. Enum
c. Wrapper classes
d. foreach loop.......

After JDK1.5Version being the major development SUNMS didn't focussed on releasing major things in java.
But when java was sold to Oracle communicty, Oracle people gave more importance for java to come up with "Major changes"
in programming.
Before JDK1.8 => java ---------> Object Oriented Programming[Oops]
After JDK1.8 => java ---------> Object Oriented programming[Ooops, Functional Aspects of Programming]

Features

a. Lambda Expression
b. Functional Interfaces
c. Default Methods
d. Predicates,Supplier,Consumer
e. Double colon operation[::]
f. Stream API
g. Date and Time API
h. Optional API

How to write Lambda Expression?
While writing lambda expression

a. method name is not required
b. return type of method is not required

c. access modifier for a method is not required
d. if a body of a method contains only one instruction then {} is also optional.
e. No need to give dataype for parameters also.
f. If it contains only arguments, then don't specify paranthesis() also.
g. If a method is returning something then we need not use return keyword also to return the value.

1. Method with no parameters, no return type.

```
 public void m1()
 {
      System.out.println("hello");
 }

 () -> System.out.println("hello");
```

2. Method with parameters, no return type

```
  1>
   public void m1(int a, int b)
   {
      System.out.println(a+b)
   }

   (a,b)-> System.out.println(a+b)

  2>
   public void m1(String str)
   {
      System.out.println(str.toUpperCase());
   }

   str -> System.out.println(str.toUpperCase());
```

3. Method with parameters and return type

```
      public String m1(String str)
      {
            return str.toUpperCase();
      }

      str-> str.toUpperCase();
```

Note:
 1. Similar to method body,lamda expression can have single statements or multiple statements.
 2. if multiple statements are there then we need to keep those statements under {}.
 3. After writing Lambda Expression,we can call that expression,but to call that Lambda Expression we need "FunctionalInterfaces".

Functional Interfaces
+++++++++++++++++++++
 => If an interface conatins only one abstract method then such type of interfaces are called as "Functional interface".
 => To indicate an interface is "FunctionalInterface",they gave one Annotation

"@FunctionalInterface".

eg:: Runnable

```
interface Runnable
{
      void run();
}
```

eg:: Callable

```
interface Callable
{
      T call();
}
```

eg:: Comparable

```
interface Comparable
{
      int compareTo();
}
```

CaseStudies

1. Valid

```
 @FunctionalInterface
 interface Interf
 {
      void m1();
 }
```

2. In-Valid

```
 @FunctionalInterface
 interface Interf
 {
      void m1();
      void m2();
 }
```

3. In-Valid

```
 @FunctionalInterface
 interface Interf
 {

 }
```

4. Valid

```
 @FunctionalInterface
 interface Interf1
 {
      void m1();
 }

 @FunctionalInterface
 interface Interf2 extends Interf1
 {

 }
```

5. Valid

```
 @FunctionalInterface
 interface Interf1
 {
      void m1();
 }

 @FunctionalInterface
 interface Interf2 extends Interf1
 {
      void m1();
 }
```

6. In-Valid

```
 @FunctionalInterface
 interface Interf1
 {
      void m1();
 }

 @FunctionalInterface
 interface Interf2 extends Interf1
 {
      void m2();
 }
```

7.Valid

```
 @FunctionalInterface
 interface Interf1
 {
      void m1();
 }

 interface Interf2 extends Interf1
 {
      void m2();
 }
```

Working with Lambda Expression through Functional Interfaces
++

```
@FunctionalInterface
interface Interf
{
      public void add(int a,int b);
}
public class Test{
      //JVM -> main thread created and started
      public static void main(String[] args){
            //Interface called Interf add(int a,int b) :: void
            //Binding Lambda-Expression
            Interf i = (a,b)-> System.out.println("The sum is :: "+(a+b));
             i.add(10,20);
      }
}
```

output

```
The sum is :: 30
```