1. Introduction to Single / multi tasking
2. The ways to define a thread

By extending Thread class

By implementing Runnable (I)

1. Getting & Setting Name of Thread
2. Thread Priority
3. The method to prevent Thread execution
4. Yield()
5. Join()
6. Sleep()
7. Synchronization
8. Inter Thread communication
9. Deadlock
10. Demon Threads
11. Multithreading Enhancement

**Introduction to Single / multi tasking:**

Executing several tasks simultaneously is nothing but multitasking. The best example in the real world is student in a class doing several task (like- Listening, Writing, sometime mobile checking, sometime sleeping)

There are two type of multitasking.

1. Process based 2- Thread based

**Process Based Multitasking:**

Executing several task simultaneously where each task is a separate independent process (means not dependent on each other) is called process based multitasking.

For example in our desktop or laptop we can do several tasks simultaneously as listed below.

1. Typing a java program in editor
2. Listening audio songs from the same system
3. Download a file from internet in the same system

The above listed tasks are completely independent to each other, while typing, only editor related process will be executing, on the same time audio related mp3 player process and file downloading process will also be executing without any dependency on each other.

So process based multitasking is best suitable at operating system (OS) level not at programmatic level.

**Thread Based Multitasking:**

Let’s say we have to write a programme having 10k line of code. As we know that JVM is an interpreter which executes the program line by line. Let’s say if JVM takes one second execution time for each line then in that case for executing 100000s. Now let’s say the 50k line (half of the codes) can be executed independently then we can divide this big task into two independent task and both the task can be executed parallel and here this each independent task is called thread. So here two threads can be run simultaneously and then it will take less time to execute 100k line of code.

50k – Task1 or Thread-1

50k- Task 2 or Thread 2

**So executing several task simultaneously where each task is a separate independent part of the same program is called thread based multitasking and each independent task is called a thread**. So here a big program of 100k line of code now executing into two separate independent parts (thread 1 and thread 2) so this is nothing but Thread based multitasking.

So as per the above examples, in process based multitasking three programme running independently while in case of thread based multitasking in one program two tasks or jobs are running independently.

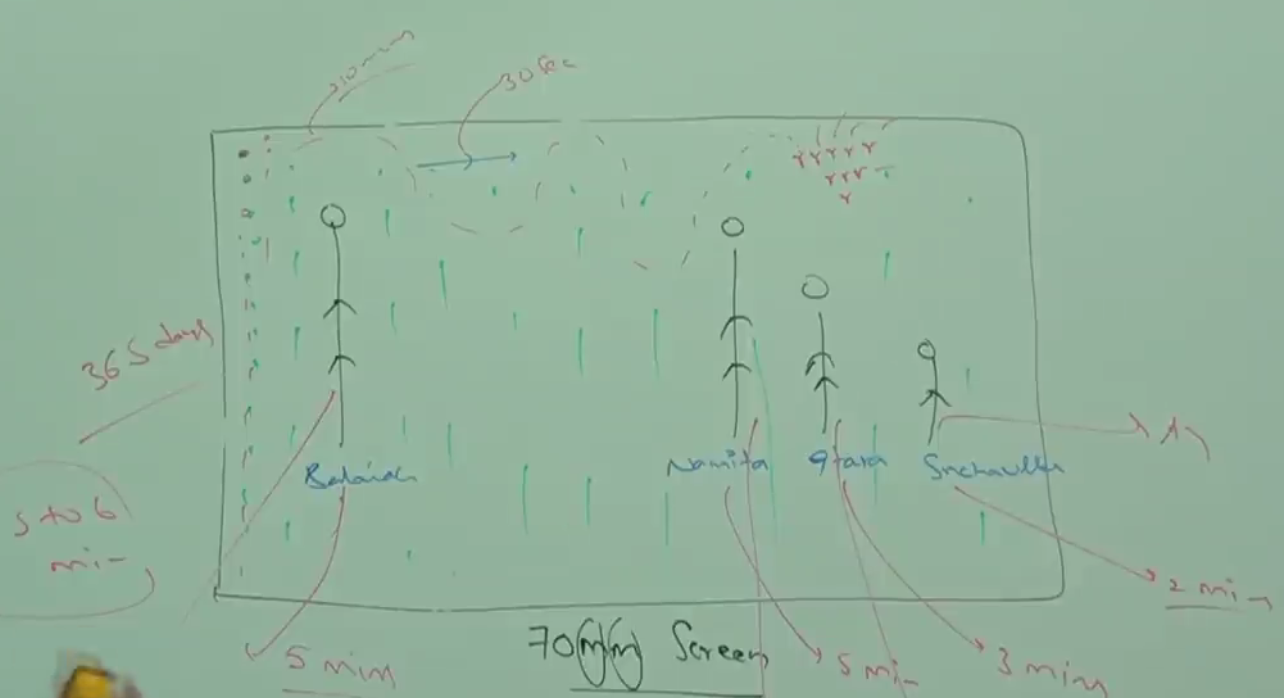
So Thread based multitasking is best suitable at programmatic level not at OS level.

**Advantage of multitasking:**

Whether it is process based or thread based multitasking, the main objective of the multitasking is to reduce processor’s CPU response time and to increase performance of the system.

**Multi Thread Concept:**

Best suitable to develop multimedia graphics, to develop animation movies, to develop games, server development where the thousands of request are processed simultaneously, for example Gmail server, which daily processes several request (email) simultaneously and provide the result on real time and this happens because every server internally maintains multiple threads and hence for each request one separate thread will be allotted to process that request. For example Tomcat server which contains default number of threads is 60



As here each object is getting executed simultaneously. If each one gets executed one by one then would not look like videos.

Example: Let’s say we have to search a particular word in the system from C, D, E drive.

Now if we write one programme which start searching drive by drive then in that case first of all it will perform search operation inside C drive then it will start from D and then from E drive. As we can see that searching operation in each drive is separate and independent so we can create thread for each drive and then each drive will start searching simultaneously in all the three drive. Now searching the particular word in the system will take less time.   
Similarly inside the C drive we will have several folders where the same search operation will be performed, so like each drive we can create one separate thread for each folder inside each drive. Now it will take even lesser time than earlier.

Hence with the help of multi thread we can search the particular word in the system very fast.

**So this concludes that where ever in the program a multiple independent tasks are identified there we can apply multi threading concept.**

When compared with old languages developing multithreaded applications in java is very easy because java provides inbuilt support for multithreading with rich API [Thread, Runnable, ThreadGroup….]

**Session-2**

Here we will learn.

1. What is thread and in how many ways we can define thread?
2. Thread scheduler

**Thread:** So as per the above discussion we can say that **a separate flow of execution of each independent task of the same program is called thread.** So if only one task is there in the program then it will be single thread, if there is multiple independent tasks in one program then it will multi thread. So every thread is executing or performing a separate independent job or task of the program.

**Defining Threads**: Two ways

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

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyThread **extends** Thread {  // overriding run() method of Thread class  **public** **void** run() {  // Executing job of thread MyThread  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println("Child Thread");  }  }  } | 1-So here we are **extending Class Thread** for user defined class MyThread.  2- **Overriding run ()** present in the Thread class.   1. Inside the run method executing the for loop   Note- Point 1 and 2 are nothing but the defining a thread (extending and overriding run () method for MyThread class.  The code inside the run () method is called the job for MyThread class |

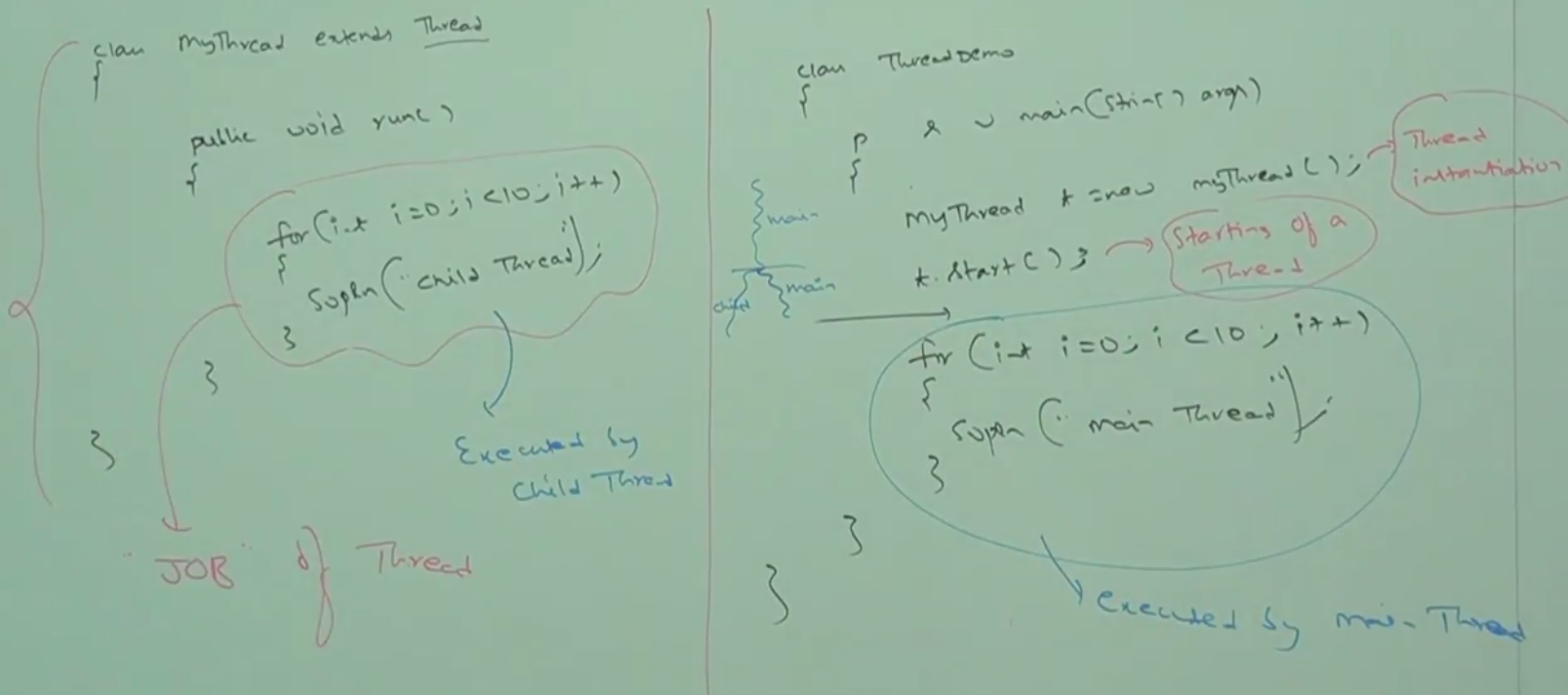
|  |  |
| --- | --- |
|  | So we can create a user defined Thread class by extending Thread class and we can define the user-defined job inside run () method. So whatever lines of code we will write inside run () method will be called job of thread. |

So now we have got to know of defining the Thread and defining job of the thread. Now we have to start thread.

By default every java program will have one thread [main () thread] .In-fact several demon thread are running like GC but main thread will be only one thread. This main thread will start the child thread present inside the main thread.

**Now Start the Thread:**

|  |  |
| --- | --- |
| **class** **ThreadDemo** {  **public** **static** **void** main(String[] args) {  // Here at this point we have main thread only running  MyThread t = **new** MyThread();// Main Thread instantiating child thread  t.start();// Here main thread starting the child thread  // Now at this point two threads are running, the main and child threads  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println("Main Thread");  }  }  } | **Steps:**  -Before instantiation there will be only one thread main thread.  -The child thread class will be instantiated by main thread only.  -t.start () - this line the child thread will be started by main thread only and this line will execute the child thread run () method to execute child thread job. When the above line will execute then first of all Thread class start () method will execute and internally it will execute the run() method of MyThread class  -Now at this point child thread will start running and now at this point two threads will be in running state.  -Now both the threads (main thread and child threads) are in running state and now we will get mix result because the scheduler some time execute main thread first and some time child thread first. |

****

**Here both the threads run independently so we cannot predict the order of execution as the thread scheduler will execute both the threads randomly.**

**Case1- Thread scheduler:**

What is the job of thread scheduler? Ans: Scheduling thread executions

Thread scheduler is the part of JVM and it is responsible of thread execution. Let’s say if multiple threads are there, and are waiting their chance to execute then in which order they are going to execute is decided by the thread scheduler only i.e. thread schedule is working as supervisor which decides the order of thread executions.

**Note:** The thread scheduler follows some algorithms to execute multiple threads and the algorithm which, the threads scheduler uses is varied from JVM to JVM. Some scheduler may follow First Come First Serve, some may follow Round Robin, Some may follow shortest job first. That is why whenever the program having multiple threads cannot be guaranteed for their order of execution and their exact output but we can provide several possible outputs.

The followings are the various possible output of the above program

|  |  |  |  |
| --- | --- | --- | --- |
| Main Thread  Main Thread  Child Thread  Child Thread | Child Thread  Child Thread  Main Thread  Main Thread | Child Thread  Main Thread  Child Thread  Main Thread | Main Thread  Child Thread  Child Thread  Main Thread |

Question: if we cannot guarantee the output then why we use multithread program in our application?

**Case-2:** As in the above program to execute the **run ()** of MyThread class we have written t.start (). We can execute the run () method directly as **t.run ()** instead of using **t.start ().** So what is the difference between **t.start () and t.run ()**.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  // Here main thread only running  MyThread t = **new** MyThread();// Main Thread instantiating child thread  t.start();// Here main thread starting the child thread  // Now at this point two threads are running main and child threads  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }  }  }  Here two thread will running | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  // This here main thread only running  MyThread t = **new** MyThread();// Main Thread instantiating child thread  t.run();// Here run() will be called just like normal method and at this point only one thread(main thread) will be available and child thread won’t be started or created at all. In whole program execution only one thread will be there  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }  }  } |

In the case of **t.start ()** a new thread will be created which is the responsible for execution of run() method but in the case of **t.run()**, a new thread would not be created and run() method will be executed just like a normal method call by main thread. Hence in the above program if we replace **t.start ()** with **t.run ()** the output is always will be child thread followed by main thread. This total output will executed by only main thread.

**Question**: If we can call the run () using **t.run ()** then why we have to call t.start ()?

**Case-3: Importance of t.start () method:** t.start () method is responsible for following as listed below.

**Start () {**

1. It registers the newly created thread with Thread scheduler
2. Perform all other mandatory activities
3. Finally invoke the run() method

**}**

Hence without executing t.start () method there is no chance of starting a new thread in java. Due to these t.start() method is considered as heart of multithreading.

Q: Can we overload run ()? Ans: Yes  
Case-4: **Overloading of run () method:** Overloading of run () method is always possible but **t.start ()** method can invoke no argument run () method only and we have to call explicitly the other over-loaded run (int j) method like a normal method call

|  |  |
| --- | --- |
| **package com.threadconcept;**  **public class MyThread extends Thread {**  **// overriding run() method of Thread class**  **public void run() {**  **// Executing job of thread MyThread**  **for (int i = 0; i < 2; i++) {**  **System.*out*.println("Child Thread");**  **}**  **}**  **public void run(int j) {**  **// Executing job of thread MyThread**  **for (int i = j; i < 5; i++) {**  **System.*out*.println("Normal overloaded");**  **}**  **}**  **}** | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  // This here main thread only running  MyThread t = **new** MyThread();// Main Thread instantiating child thread  t.start();// Here main thread starting the child thread  // Now at this point two threads are running main and child threads  t.run(2); // called like a normal method  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }  }  } |

Q: Now if MyThread class does not override run () then which start () method will execute.

**Case-5: If we are not overriding run () method then Thread class run () method will be executed, which has empty implementation. Hence we won’t get any output.**

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyThread **extends** Thread {  // did not override Thread class run() in MyThread class  }  So here Thread class will execute its own run () method which has empty implementation so we will not get any output. | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  // This here main thread only running  MyThread t = **new** MyThread();// Main Thread instantiating child thread  t.start();  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }  }  } |

**Note**: If we are not overriding run () method in our user-defined thread class then we should not go for multithreading concept i.e. it is highly recommended to override run ()

**Case-6: Overriding start () method:**

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyThread **extends** Thread {  //Overriding Thread class start() method  **public** **void** start() {  System.***out***.println("Start Method");  }  //Overriding Thread class run() method  **public** **void** run() {  System.***out***.println("Run method");  }  }  Output:  Start Method  Main Thread  Main Thread | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyThread t = **new** MyThread();  t.start();  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }  }  }  So when t.start() will execute it first check whether start() method is available in the child class or not. If start() has been overridden in Thread’s child class MyThread then first priority will child class(MyThread) start () method to execute and it won’t execute Parent class Thread start() method. |

Now if Thread class start () method won’t be run then in that case no child thread will be created and hence run () method also would not be executed. Hence in this case start() will be executed like a normal method only and in this case whole programme will be executed by main thread only and only one thread (main thread) will be there.

So if we override start () method then our start () will be executed just like a normal method call and new thread won’t be created.

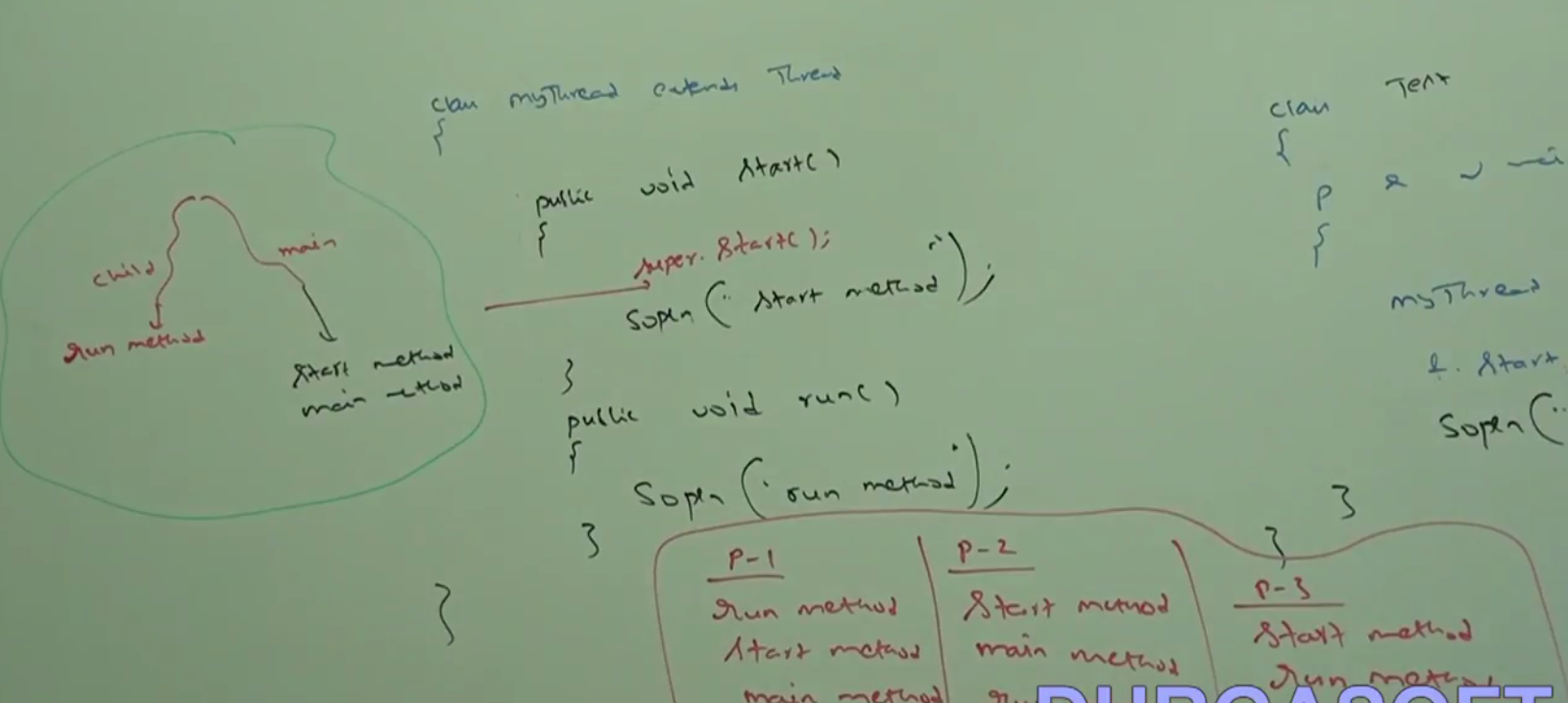
**Note**: It is not recommended to override start () method otherwise don’t go for multithreading concept.

**Case-7: if call super.start () inside our own start ()**

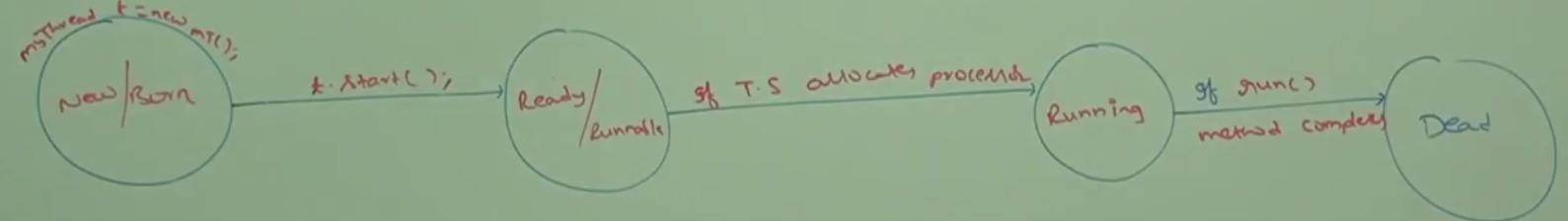
In this case Thread class start () method will be executed which in-turn execute run() method and hence we will have two thread main thread which will execute our own start() and remaining and child thread will execute run() method.

So we can get several possible outputs as given below.

|  |  |  |  |
| --- | --- | --- | --- |
| **package** com.threadconcept;  **public** **class** MyThread **extends** Thread {  //Overriding Thread class start() method  **public** **void** start() {  **super.start()//** calling Thread class start method  System.***out***.println("Start Method");  }  //Overriding Thread class run() method  **public** **void** run() {  System.***out***.println("Run method");  } | | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyThread t = **new** MyThread();  t.start();  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }  }  } | |
| Start Method  Main Thread  Main Thread  Run method | Start Method  Run method  Main Thread  Main Thread | Run method  Start Method  Main Thread  Main Thread | Main Thread  Main Thread  Run method  Start Method |



Case-8: **Thread lifecycle**:



The above diagram says the following listed below

1. First of all a new thread will be created or born and hence it is called as new or born state.
2. When we perform t.start () it goes into Ready or Runnable state.
3. Now if Thread scheduler allocates process for this thread then it will go into **Running** state.
4. If everything goes fine and if run() method executed or completed then thread will finally goes into dead or stop state.

It is very similar to human being life-cycle.

But as the normal live person can perform some other activity so in the same way a running live thread also can go to sleep() or wait() or join() or yield() activity from Running state.

Case-9: After starting a thread (t.start) if we are trying to restart (t.start() ) thread then we will get runtime exception saying IllegalThreadStateExceptio.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyThread t = **new** MyThread();  t.start();  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }  t.start(); //Starting the same thread again.  }  } | Start Method  Run method  Main Thread  Main Thread  Exception in thread "main" java.lang.IllegalThreadStateException  at java.lang.Thread.start(Thread.java:708)  at com.threadconcept.MyThread.start(MyThread.java:7)  at com.threadconcept.ThreadDemo.main(ThreadDemo.java:11) |

So in the above example first time t.start() the program will be executed like normal multithread class. But again when JVM will try to execute t.start() second time then it will observe that thread is already started and hence it will give runtime exception as mentioned above.

It is very similar to re-born the dead thread, which is impossible because one thread complete its lifecycle it goes to dead state or stop state and then it is impossible to start the same thread again.

**Session-3**

**Defining a Thread by implementing Runnable (I) Interface & Thread class constructor**

In the first approach we have created thread by extending Thread class. The Thread class intern implements Runnable Interface.

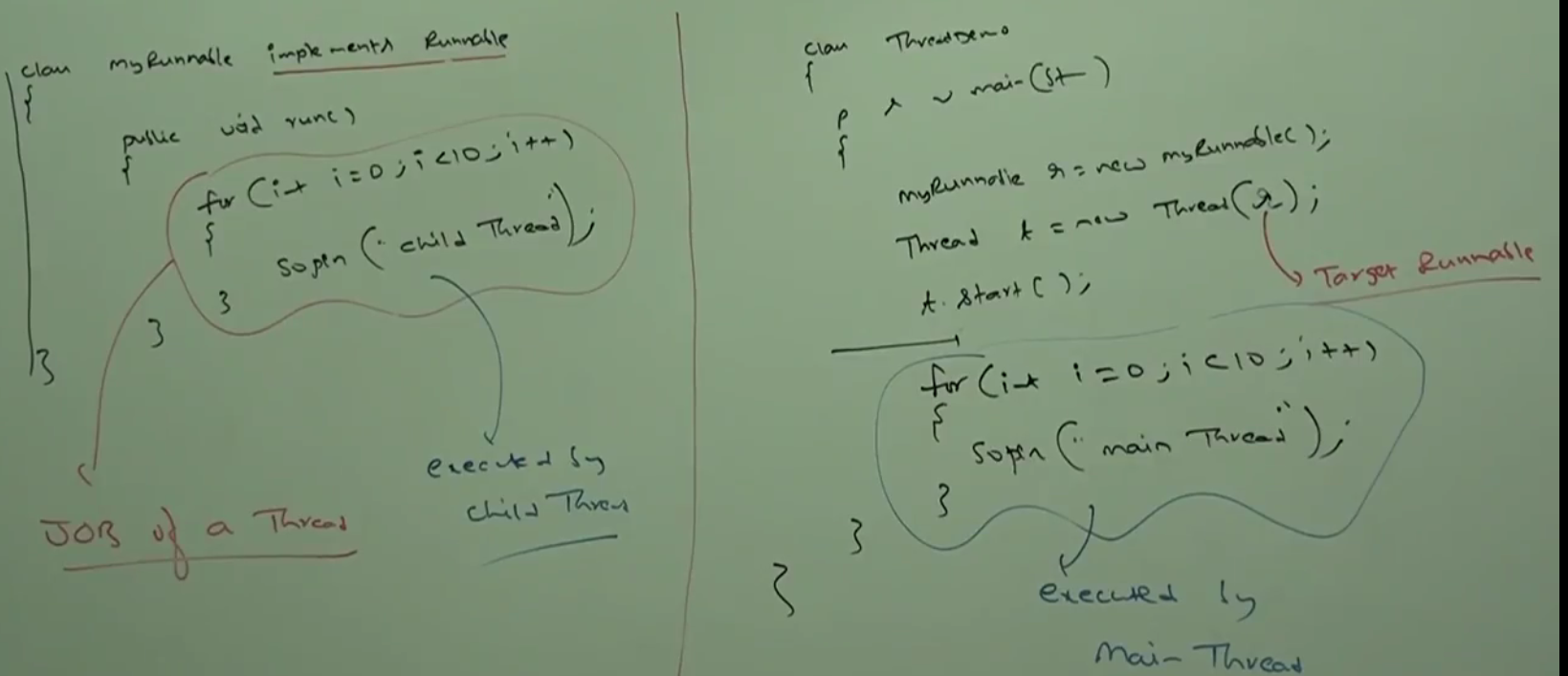
[MyThread🡺Thread🡺Runnable(I)]

Now in the second approach instead of extending Thread class, we will directly implement Runnable interface into our own MyThread class [MyThread🡺Runnable (I)].

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyRunnable **implements** Runnable {  @Override  **public** **void** run() {  System.***out***.println("Run Method executed");  }  } | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyRunnable r = **new** MyRunnable();  Thread t = **new** Thread(r); // Linking Runnable (I) and create new thread  t.start(); // starting thread  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }}} |

1. Runnable Interface is present in java.lang package and it contains only one method [public void run ()].
2. As we already know that the MyRunnable class will be instantiated by main thread and at this point only main thread is running.
3. As the Runnable Interface contains only one run () method and it does not contain any start () method like Thread class i.e. Runnable Interface does not have the capability to start thread.
4. So to get start capability for Runnable Interface instantiate Thread as [Thread t = new Thread(r);] and pass the r so that Thread Class can create the thread for MyRunnable class. Now there a link got established between MyRunnable classes (Implementing Runnable Interface).Here (r) is called Target Runnable or target run () method to be executed.
5. Now using t.start () method we can call the run () of our own MyRunnable class and now at this point we will have main thread and child thread.
6. Now as soon as the above line executed run () of MyRunnable class will be executed by child thread.
7. So now two thread are running and hence we will get several possible output

|  |  |  |
| --- | --- | --- |
| Main Thread  Run Method executed  Main Thread | Main Thread  Main Thread  Run Method executed | Run Method executed  Main Thread  Main Thread |
| MyRunnable r = **new** MyRunnable();  Thread t = **new** Thread();  t.start(); | | As here we can see that we have not pass (r) into Thread class constructor and now if we execute t.start () then it will execute Thread class run () method instead of executing MyRunnable class run() method. |



Case Study:

|  |  |
| --- | --- |
| **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyRunnable r = **new** MyRunnable();  Thread t1 = **new** Thread();  Thread t2 = **new** Thread(r);  t1.start(); // Create one thread of Thread class & which in-turn execute Thread class run () method  t1.run(); // Won't create any thread and run () of Tread t1 will be executed like normal method    t2.start(); // Create new thread for MyRunnable Class which implements Runnable (I)  t2.run(); // Execute run () just like a normal method  // r.start(); // start() method not available for MyRunnable class as it implement Runnable(I)  r.run(); // Execute run () of MyRunnable class just like a normal method  }  } | |
| **Case 1-t1.start()** | A new thread will be created which is responsible for execution of Thread class run () method, which has empty implementation. |
| **Case 2- t.run()** | No new thread will be created and Thread class run () method will be executed just like a normal method call. |
| **Case 3- t2.start()** | A new thread will be created which is responsible for execution of MyRunnable class run () method. |
| **Case 4- t2.run()** | A new thread won’t be created and MyRunnable run method will be executed just like a normal method call. |
| **Case 5- r.start()** | We will get compile time error saying MyRunnable class does not have start capability |
| **Case 6- r.run()** | Now new thread will be created and MyRunnable run () method will be executed like a normal method call |

**Among the above two approach which one is best to define a thread?**

Among tow ways of defining a thread, Implements Runnable Interface approach is recommended.

In the first approach our class always extends Thread class and there is no chance of extending any other class because java does not support multiple inheritances and hence we are missing inheritance benefit.

But in the second approach while implementing Runnable Interface we can extend any other class hence we won’t miss any inheritance benefit.

Because of above reason Implementing Runnable Interface approach is recommended than extending Thread class.

**Thread Class Constructors:**

|  |  |
| --- | --- |
| Thread t = new Thread() | Create a new thread extending Thread class |
| Thread t = new Thread(Runnable r) | Create a new thread implementing Runnable (I) |
| Thread t = new Thread(String name) | Create a new thread and providing name to that thread |
| Thread t = new Thread(Runnable r, String name) | Create a new thread implementing Runnable (I) and providing name to that thread |
| Thread t = new Thread(ThreadGroup g, String name) | Create a new thread which belongs to ThreadGroup g and providing name to that thread. |
| Thread t = new Thread(ThreadGroup g, Runnable r) | Create a new thread which belongs to ThreadGroup g and implementing Runnable (I) |
| Thread t = new Thread(ThreadGroup g, Runnable r, String name) | Create a new thread which belongs to ThreadGroup g and implementing Runnable (I) and providing name to that thread |
| Thread t = new Thread(ThreadGroup g, Runnable r, String name, long StackSize) | Create a new thread which belongs to ThreadGroup g and implementing Runnable (I) and providing name to that thread and allocating stack memory to that thread |

**Durga’s approach to define a thread (Not recommended to use) – Hybrid Approach:**

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyThread **extends** Thread {  //Overriding Thread class run() method  **public** **void** run() {  System.***out***.println("Child thread");  }  } | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyThread t = **new** MyThread();  Thread t1 = **new** Thread(t);  t1.start();  System.***out***.println("Main thread");  }} |

**Since here we are extending Thread class so we don’t need to pass MyThread class object into Thread class. We directly can use t to call start () method of MyThread class. But this approach is 100% correct; the program will be executed successfully because**

**MyThread🡺extends Thread 🡺 implements Runnable (I)**

**Note: for every thread in java there is some name is defined for them if programmer does not define name of thread explicitly then a default name is gets generated by JVM.**

**Getting & Setting Name of Thread:**

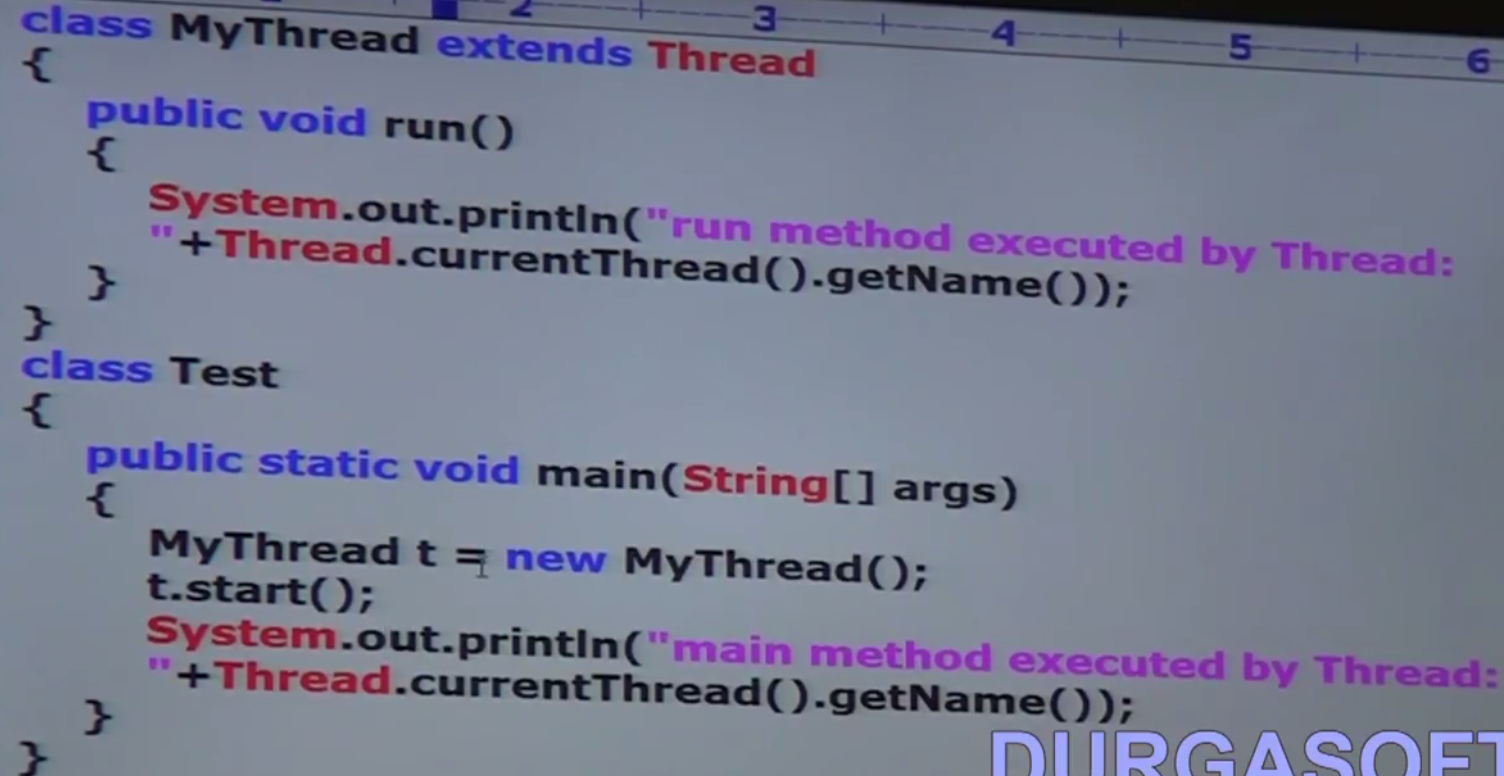
Every thread in java has some name. It may be default name generated by JVM or customized name provided by programmer.

We can get and set name of a Thread by using below two method of Thread class

1. public final String getName() or
2. public final void setName(String name)

|  |
| --- |
| **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  System.***out***.println("Main Thread Name:-" + Thread.*currentThread*().getName());  // Changing or setting main thread name  Thread.*currentThread*().setName("ArunThread");  System.***out***.println("Changed name of main thread:-" + Thread.*currentThread*().getName());  MyThread t = **new** MyThread();  t.start();  System.***out***.println("Child Thread Name:-" + t.getName());  t.setName("ChildThreadT1");  System.***out***.println("Changed Child Thread Name:-" + t.getName());  }  } |
| **Output:**  Main Thread Name:-main  Changed name of main thread:-ArunThread  Child Thread Name:-Thread-0  Changed Child Thread Name:-ChildThreadT1  ChildThreadT1 |

Note: We can get current executing thread object by using Thread.currentThread() method



|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyThread **extends** Thread {  //Overriding Thread class run() method  **public** **void** run() {  System.***out***.println("Run method executed by Thread:-" + **Thread.*currentThread*().getName());**  }  }  Output:  Main Thread Name:-main  Changded name of main thread:-**ArunThread**  Child Thread Name:-**Thread-0**  Changed Child Thread Name:-**ChildThreadT1**  Main method executed by Thread:-**ArunThread**  Run method executed by Thread:-**ChildThreadT1** | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  System.***out***.println("Main Thread Name:-" + **Thread.*currentThread*().getName());**  // Chenging or setting main thread name  **Thread.*currentThread*().setName("ArunThread");**  System.***out***.println("Changded name of main thread:-" + **Thread.*currentThread*().getName());**  MyThread t = **new** MyThread();  t.start();  System.***out***.println("Child Thread Name:-" + t.getName());  **t.setName("ChildThreadT1");**  System.***out***.println("Changed Child Thread Name:-" + t.getName());  System.***out***.println("Main method executed by Thread:-" + **Thread.*currentThread*().getName**());  }  } |
| **Thread.*currentThread*().getName());- To get current thread name**  **Thread.*currentThread*().setName("ArunThread");- To set or change the current thread name**  **t.start();t.getName());t.setName("ChildThreadT1"); Setting and getting child thread name** | |

**Callable Interface**

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyRunnable **implements** Runnable {  @Override  **public** **void** run() {  System.***out***.println("Run Method executed");  }  } | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyRunnable r = **new** MyRunnable();  Thread t = **new** Thread(r); // Linking Runnable (I) and create new thread  t.start(); // starting thread  **for** (**int** i = 0; i < 2; i++) {  System.***out***.println("Main Thread");  }}} |

As we have seen that Runnable (I) interface contains one method [**public void run ()**] which return type is void and it does not return anything. In the same we have Thread class which is the implementation class of Runnable Interface also contains [public void run ()] method and does not return anything back.

So once the job completed, the run () method does not return to the caller [**t.start ()**], and immediately thread will be terminated by JVM.

But sometime after completing the job thread has to return something to the caller then we should go for Callable (I) interface instead of Runnable (I) interface. This Callable interface contains one method [**public Object call ()**] whose return type is object.

So the above program using callable interface will look like as given below.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **import** java.util.concurrent.Callable;  **public** **class** MyCallable **implements** Callable<Object> {    @Override  **public** Object call() **throws** Exception {  // **TODO** Auto-generated method stub  **return** **null**;  }  } | **package** com.threadconcept;  **import** java.util.concurrent.Callable;  **public** **class** MyCallable **implements** Callable<String> {    @Override  **public** String call() **throws** Exception {  // **TODO** Auto-generated method stub  **return** **"Callable call method returned- String";**  }  } |
| **package** com.threadconcept;  **public** **class** TestRunnableDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyCallable c = **new** MyCallable();  String call = c.call();  System.***out***.println(call);  System.***out***.println("Main Thread");  }  } | **Output:**  **Callable call method returned- String**  **Main Thread** |

**One more example will see in Thread Pool Executer framework session**

**Session -4**

**Thread Priorities**

Just like every Thread have some name; in the same way every thread in java has some priority. It may be default priority generated by JVM or customized priority provided by programmer.

The valid rage of priority is from 1 to 10. Where 1 is min priority and 10 is max priority. Thread class defines the following constants to represents some standard priorities.

|  |  |
| --- | --- |
| /\*\*\* The minimum priority that a thread can have. \*/  **public** **final** **static** **int** ***MIN\_PRIORITY*** = 1;  /\*\*\* The default priority that is assigned to a thread. \*/  **public** **final** **static** **int** ***NORM\_PRIORITY*** = 5;  /\*\*\* The maximum priority that a thread can have. \*/  **public** **final** **static** **int** ***MAX\_PRIORITY*** = 10; | Thread.***MIN\_PRIORITY*** = 1;  Thread***.NORM\_PRIORITY***= 5;  Thread.***MAX\_PRIORITY*** = 10; |

**Q: Who is going to use this Thread priorities and when the Thread priority will be required?**

Let’s assume in one traffic signal President, PM, CM & Minister is coming separately from all four direction of traffic signal. Then in this situation, everyone is waiting for their chance to cross traffic signal. Now it is the Police man responsibility to allow based on the priority to go ahead. President has MAX\_PRIORIY so he would get first change to cross signal, then PM and then CM and finally minister will have MIN\_PRIORITY so he will get last chance cross signal.

Similarly when multiple thread are waiting to get the chance to be allocated for the processor, now it is the thread scheduler responsibility to give the change based on thread priorities i.e. the thread having ***MAX\_PRIORITY*** priority will get first chance to be allocated for processing by processor. And the thread having ***MIN\_PRIORITY*** will get last chance.

Let’s say we have four threads **[t1 (8), t2 (1), t3 (5) & t4 (2)]**. Here t1 will get first chance and t2 will get last chance.

So thread scheduler use thread priority while allocating processor. The thread having high priority will get first chance to be allocated for processing by processor.

If two threads having same priority **[t1 (7), t2 (7)]** then we cannot expect exact execution order, it depends on Thread scheduler to allocate the processor. Thread scheduler either may use Round-Robin or Fist-Come-First-Serve or Shortest-Job-First algorithm to allocate the processor for the thread having same priorities.

**Getting & Setting Priority of Thread**

Thread class defines two methods getPriority () & setPriority (int newPriority) for getting and setting priority of a thread.

1. **public** **final** **int** getPriority()
2. **public** **final** **void** setPriority(**int** newPriority)

**Note**: Allowed valid range for setting newPriority is 1 to 10 into setPriority (int newPriority) method. By mistake if we try to set other than this range like setPriority (**100**) then we will get runtime exception saying **illegalargumentException**

**Default Priority:** The default priority for every thread is 5 or the default priority only for main thread is 5? Which one is correct?

So the default priority only for main thread is 5 and for all remaining threads the default priority will be inherited from parent to child i.e. whatever the priority the parent thread will have, the same priority will be inherited into child thread.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  System.***out***.println("Main Thread Name:-" + Thread.*currentThread*().getName());  System.***out***.println("Main Thread priority:-" + Thread.*currentThread*().getPriority());  // Chenging or setting main thread name  Thread.*currentThread*().setName("ArunThread");  System.***out***.println("Changded name of main thread:-" + Thread.*currentThread*().getName());  MyThread t = **new** MyThread();  t.start();  System.***out***.println("Child Thread Name:-" + t.getName());  t.setName("ChildThreadT1");  System.***out***.println("Changed Child Thread Name:-" + t.getName());  System.***out***.println("Changed Child Thread Priority:-" + t.getPriority());  System.***out***.println("Main method executed by Thread:-" + Thread.*currentThread*().getName());  }  } | |
| Main Thread Name:-main  Main Thread priority:-5  Changded name of main thread:-ArunThread  Child Thread Name:-Thread-0  Changed Child Thread Name:-ChildThreadT1  Changed Child Thread Priority:-5  Run method executed by ThreadChildThreadT1  Main method executed by Thread:-ArunThread | So here we can see that the main thread default priority is 5. Since here the main thread is the parent thread that is responsible to create child thread (t) so the we can see that the child thread priority also 5 only.  Now let’s change the main thread (parent thread) priority and then see the child thread priority. |
| **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  System.***out***.println("Main Thread priority:-" + Thread.*currentThread*().getPriority());  // Setting new priority  Thread.*currentThread*().setPriority(7);  System.***out***.println("New set priority of main thread:-" + Thread.*currentThread*().getPriority());  MyThread t = **new** MyThread();  t.start();  System.***out***.println("Child Thread Priority:-" + t.getPriority());  }  } | Output:  Main Thread priority:-5  New set priority of main thread:-7  Child Thread Priority:-7  So we can see that once the main thread priority is set to 7 then its child thread priority is also set to 7 as the same priority now inherited from parent. |

**Note**: Don’t get confuse between Parent class and parent thread. For the above program the parent class is Thread class as the MyThread class extends Thread class but her parent thread is main thread which is responsible to create and execute the child thread and its run method.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyThread **extends** Thread {  //Overriding Thread class run() method  **public** **void** run() {  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Child Thread");  }  }  } | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyThread t = **new** MyThread();  t.setPriority(10);// **Line-1**  t.start();  **System.*out*.println("MainThreadPriority-" + Thread.*currentThread*().getPriority());**  System.***out***.println("Child Thread priority-" + t.getPriority());  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Main Thread");  }}} |

|  |  |
| --- | --- |
| Child Thread  Child Thread  Child Thread  Main Thread priority-5  Child Thread priority-10  Main Thread  Main Thread  Main Thread | So here we can see that we have set the child thread priority 10 and hence child thread got ***MAX\_PRIORITY*** than main thread (5). So now here first child thread will be executed and then main thread remaining line of code will be executed.  If we are commenting line -1 then both child and main thread will have same priority (5) and hence we can expect execution order and exact output.  If we are not commenting line-1 then main thread has priority-5 and child thread has proirority-10 and hence child thread will get the chance first followed by main thread. In this case output is child thread….3 times followed by main thread …..3 times |

Note: Some platforms won’t provide proper support for thread priority.

Here in the above program also we are not getting expected output so here my window-10 plateform does not support for thread- priority and in this case we need to write email the Microsoft people to install some path so that it start support for thread priority.

|  |  |  |
| --- | --- | --- |
| Main Thread priority-5  Child Thread priority-10  Main Thread  Main Thread  Main Thread  Child Thread  Child Thread  Child Thread | Main Thread priority-5  Child Thread priority-10  Main Thread  Main Thread  Child Thread  Child Thread  Child Thread  Main Thread | Child Thread  Child Thread  Child Thread  Main Thread priority-5  Child Thread priority-10  Main Thread  Main Thread  Main Thread |

**Session-5**

**Preventing Thread from Execution: 1-yield (), 2- join (), 3-sleep ()**

**Q: How many ways we can prevent a thread from execution temporarily?**

**We can prevent a thread execution by using the following methods.**

1. **yield()**
2. **Join()**
3. **Sleep()**

**yield () method: What is the purpose of yield() method and When we can go for yield () method?**

This method causes (or says) to left current executing thread to give the chance for waiting thread of same priority. If there is no waiting thread or all waiting thread have low priority then same thread can continue its execution.

Let’s say if multiple thread (t1, t2, t3, t4) are waiting their chance to be executed and let’s say and let’s say thread scheduler use First-Come-First-Serve algorithm to execute the threads. Let’s say t1 comes first and get the chance to be executed by processor. Now let’s say t1 takes 10k hour to get executed. In that case other has to wait for 10k hour even though the reaming thread will take less time to execute. Hence this is not good at all.

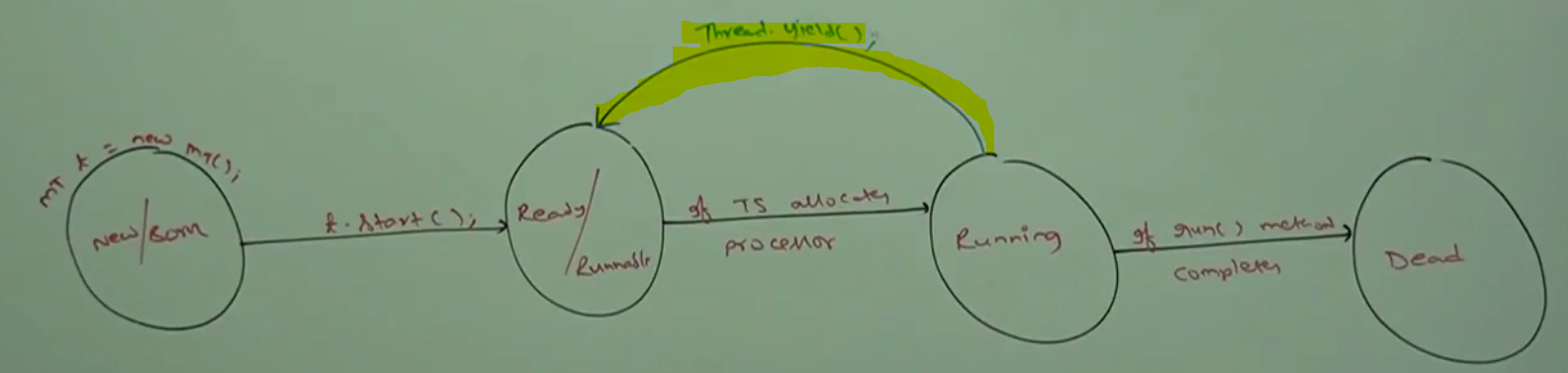
To handle this problem we will use yield () method. The yield () method ask the thread to left its current execution and to give chance for waiting thread if the other thread have the same priority and if there is no waiting thread or all waiting thread have low priority then same thread can continue its execution.

**Note:** If multiple threads are waiting with same priority then which waiting thread will get the chance we cannot decide and it depends on thread scheduler.

The thread which yielded (which left the processor), when it will get the chance once again, it depends on thread scheduler and we cannot expect or decide exactly.

Prototype of yield(): **public static native void yield() //** Native means not implemented in java**.**

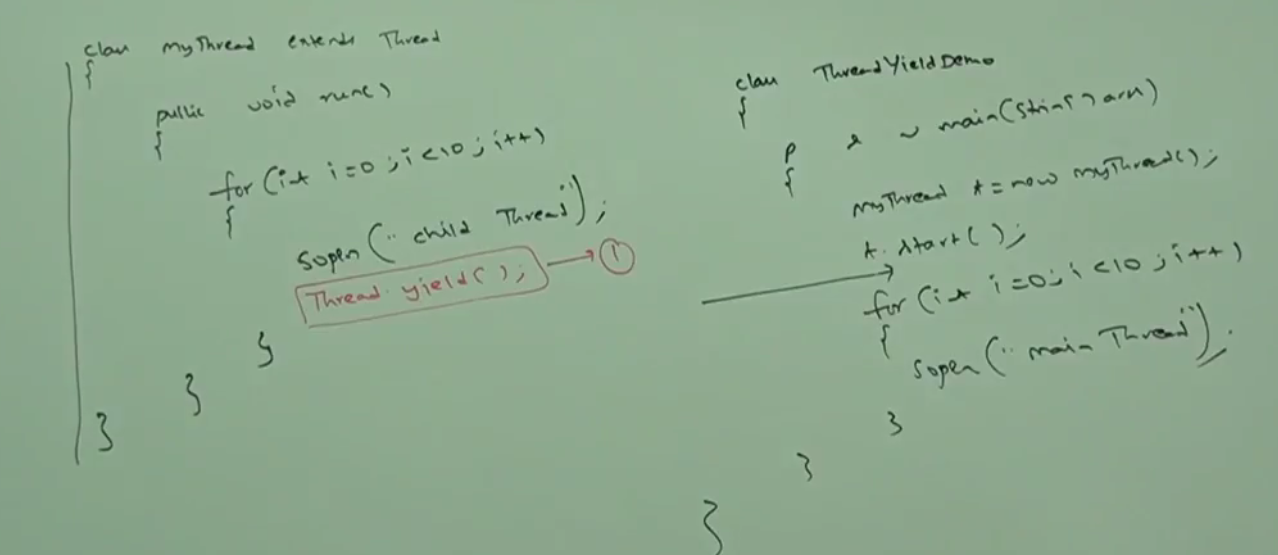
**If we call the yield() method on the running thread then thread will reach to which state?**

****

**So as per the diagram when Thread.yiled() gets called for current running thread then it left its current execution (or left the processer) and reach to ready or Runnable state and again start waiting for their chance to be allocated and start execution.**

**Note:**

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MyThread **extends** Thread {  //Overriding Thread class run() method  **public** **void** run() {  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Child Thread");  **Thread.*yield*();**  }  }  } | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) {  MyThread t = **new** MyThread();  t.start();  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Main Thread");  }  }  } |

****

In the above program if we are commenting line-1 then both the thread executed simultaneously and we cannot expect which thread will complete first.

If we are not commenting line-1 then child thread always calls yield () method because of it, main thread will get chance number of times and the chance of completing main thread first is high.

**Who requires yield() method:** The thread which require more processing time then in the middle recommended to call yield() frequently so that other waiting thread having same priority can get chance to execute**.**

**Note:** Since it is written in native (not in java language) so some platform (OS) won’t provide proper support for yield () method.

**Join () method: What is the purpose of join() method.**

Let’s say me and my friend is studying in Durga classes and my friend class got completed very early but my class is still running and will take some time to complete. Now my friend has two choices, if he want to go alone then he can go, but if he wants to go with me then he has to wait until my class gets completed. So if my friend wants to wait until completing my class then this is nothing but join condition as stated below in case of thread. Since here my friends is waiting to join me so my friend has to call join () to join me.

**If a thread wants to wait until completing other thread then we should go for join () method.** Let’s say we have two threads t1 and t2 and both are executing now if t1 wants to wait until completing t2 (because may be t1 wants to use t2 output) then t1 has to call join () method. Now since here t1 is waiting so t1 has to call join () method one thread (t2) like as [**t2.join ()**] and this method or line will be executed by t1.

So once t1 calls [**t2.join ()**] method then immediately t1 enters into waiting state until t2 complete its execution. But Note, there is no any impact on t2 execution because t2 even don’t know whether t1 has call join () method or not and t2 can continue its execution and once t2 completes its execution then t1 can continue its execution.

**Here t1 is waiting, so t1 will call join method using t2 object like [t2.join ()] so that t1 can start its own execution once t2 complete its own execution.**

Example: Let’s say there is marriage at my home then we will have to perform the following activities as listed below.

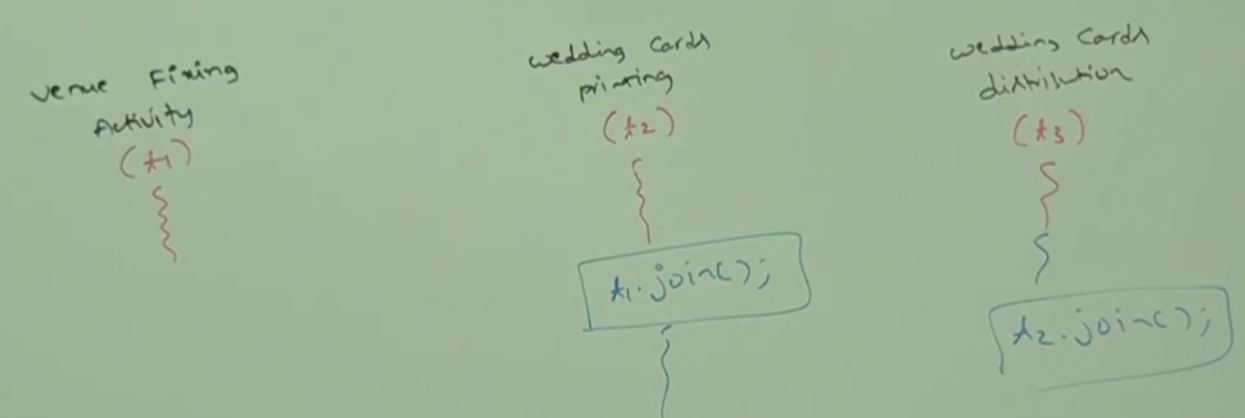
1. Venue Finding activity (t1)
2. Wedding cards printing activity(t2)
3. Wedding card distribution activity(t3)
4. Marriage function activity(t4)

Let’s consider the above 4 activity as job or thread. Then how we can apply join () method here for above activities.

As we know that once Venue finding activity(t1) will be completed then only we can start wedding card printing activity (t2) because in the wedding card Marriage venue will have to printed .So as per the join () concept, t2 has to wait until t1 completes so t2 will call join() method on t1 like as **[t1.join()]**

Similarly wedding card distribution activity (t3) has to wait until wedding card printing activity (t2) gets completed. So t3 will call join () method on t2 like as **[t2.join ()].**

Similarly marriage function activity (t4) has to wait until wedding card distribution activity (t3) complete. So t4 will call join method on t3 like as **[t3.join ()].**



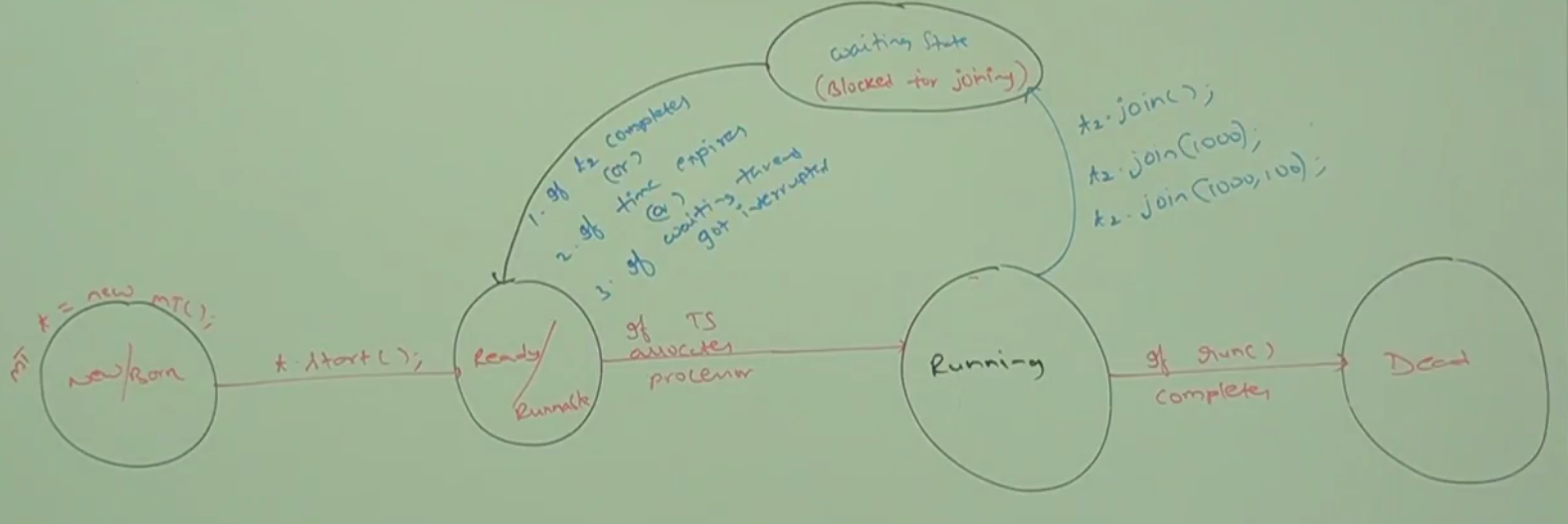
**Prototype of join ():**

|  |  |  |
| --- | --- | --- |
| 1- **public final void join ()**  **[t1.join()]**  Here once t1 complete its execution then only waiting thread will start its execution | 2- **public final void join (long timeInMs)**  **[t1.join()]**  Here the other thread will not wait for completing t1 execution, once the time defined in the above method reached immediately other waiting thread will start its execution. Here the joining time is given in Millisecond | 3- **public final void join (long timeInMs, int nanoSec)**  **[t1.join()]**  **This is more specific regarding waiting time. Same way** other thread will not wait for completing t1 execution, once the time defined in the above method reached immediately other waiting thread will start its execution. Here the joining time is given in Millisecond: nanosecond like (20:99) |

While using join () method then there may be chance to get interruption from other waiting thread so all the above method throws **InterruptionException** (IE). These are checked exception so either we will handle this exception by throwing It using throws keyword or by using try-catch block otherwise we will get compile time error. So the above all three methods can be written as.

1. **public final void join () throws** InterruptionException
2. **public final void join (long timeInMs) throws** InterruptionException
3. **public final void join (long timeInMs, int nanoSec) throws** InterruptionException

**Q: What is the impact of join method in our thread cycle.**



As per the diagram let’s say we have two thread t1 and t2 both are executing. Let’s t1 wants to complete t2 first i.e. t1 wants to wait until t2 completes then t1 will call join () method on t2 in one of the three ways [t2.join () or t2.join (1000) or t2.join (1000, 100). Once t1 execute this method then t1 immediately goes to waiting state (or blocked for joining state).

Now t1 will come out from the waiting state in one of the following condition.

1. If t2 completes or
2. If the time mentioned inside the join () method expires
3. If waiting thread got interrupted

Note: The thread once come from running state to waiting state then it never come back directly into running state. Before going to running state it will go he ready or Runnable state and once thread scheduler allocates the processor then only it will come again into running state for further execution

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept;  **public** **class** ThreadJoin **extends** Thread {  @Override  **public** **void** run() {  // super.run();  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Child Thread");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }}}} | | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ThreadJoin t = **new** ThreadJoin();  t.start();  t.join();  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Main Thread");  }  }  } |
| Output:  **Child Thread**  **Child Thread**  **Child Thread**  **Main Thread**  **Main Thread**  **Main Thread** | **So here in the main method once t.start () executes it enters into child thread run method and start executing.**  **In the child run() method we have included Thread.sleep(2000) so in every iteration of for loop it went into sleep for 2 sec and then start executing next iteration.**  **As we can see in the main method we have included t.join() i.e. join () is called on child thread and this line is getting executed by main thread so it means main thread will be waiting until child thread complete its execution and once child thread completed its execution then only main thread stat its own execution. So here we will get exact outputs instead of getting several possible outputs.** | |
| **Note:** Here in the above program since main thread is calling join () on child thread without any time condition. So main thread has to wait until child thread complete its execution.  **Now let’s call the join () method with time limit as t.join (1000). He** | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ThreadJoin t = **new** ThreadJoin();  t.start();  **t.join(1000);**  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Main Thread");  }  }  } | |
| **Output:**  Child Thread  Child Thread  Main Thread  Main Thread  Main Thread  Child Thread | **So here in this case main thread will not wait to complete child thread execution. The main thread will wait for 1000 millisecond and if child thread is not able to complete its execution in 1000 ms then main thread will start its own execution and then once main thread compete its execution then child thread again will start its remaining execution.**  **Note: if we comment t.join () method then both the thread (main and child) will be executing simultaneously and we cannot get exact output and we will get several possible output.** | |

**Case-2:** So far we have discussed where main thread has to wait until child thread complete its execution. Now we will see the case where child thread has to wait until main thread complete its execution.

In the above case where main thread had to wait until child thread complete its execution, so here it was very easy to create child thread using child or sub class object which was extending Thread class.

But the question is here in case two how to get main thread inside child class? We can get main thread by using below steps.

**Step-1:** Create one static Thread type variable inside the child class. Like as: [**static** Thread *mt*;]

**Step-2:** Now instantiate this variable with main thread inside main () method like as given below.

[ThreadJoin.*mt* = Thread.*currentThread* ();]

So once main () gets executed the current thread(main thread) will be assigned to static variable mt and this way the main thread would be available in the child thread. Now using mt.join() we can execute the main thread first and child thread will have to wait until main thread complete its execution

|  |  |
| --- | --- |
| Child class:  **package** com.threadconcept;  **public** **class** ThreadJoin **extends** Thread {  **static** Thread *mt*;  @Override  **public** **void** run() {  // super.run();  **try** {  *mt*.join();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Child Thread");  }  }  } | **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ThreadJoin.*mt* = Thread.*currentThread*();  ThreadJoin t = **new** ThreadJoin();  t.start();  // t.join();  **for** (**int** i = 0; i < 3; i++) {  System.***out***.println("Main Thread");  Thread.*sleep*(2000);  }  }  }  **Output**:  **Main Thread**  **Main Thread**  **Main Thread**  **Child Thread**  **Child Thread**  **Child Thread** |

Now here in the above example child thread calls join () method on main thread object, hence child thread has to wait until completing main thread. In this case output will be main thread….3 times child thread…3.

**Case-3;** If main thread calls join () method on child thread object and child thread calls join () method on main thread object then both thread will wait forever and the program will be stuck (this is something like deadlock situation)

**Case-4:** If a thread calls join() method on the same thread itself then the program will be stuck (this is also something like deadlock). In this case thread has to wait to complete itself for infinite time.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **class** ThreadDemo {  **public** **static** **void** **main**(String[] args) **throws** InterruptedException {  **Thread.*currentThread*().join();**  } | Here **Thread.*currentThread*()** is nothing but main thread only and this calls join () on itself hence main thread has to wait until its own completion which is impossible to complete. So program will be in deadlock situation. |

**Session-6**

**How a thread can interrupt another thread**

**Thread.sleep ():** When we can go for **Thread.sleep ()** method.

**As we know that as a human being we keep performing while we are awake and while sleep then we are in idle situation and we don’t perform any thing during sleep time and after completing sleep time we again start performing.**

**So whenever a temporary pause/halt is required there we should go for sleep () method or whenever we don’t want to perform for a particular of time then we should go for sleep () method. So in the same way….**

**So if a thread don’t want to perform any operation for a particular amount of time then we should go for sleep () method.**

Prototype: There are two methods in java one is native not written in java and another is written in java

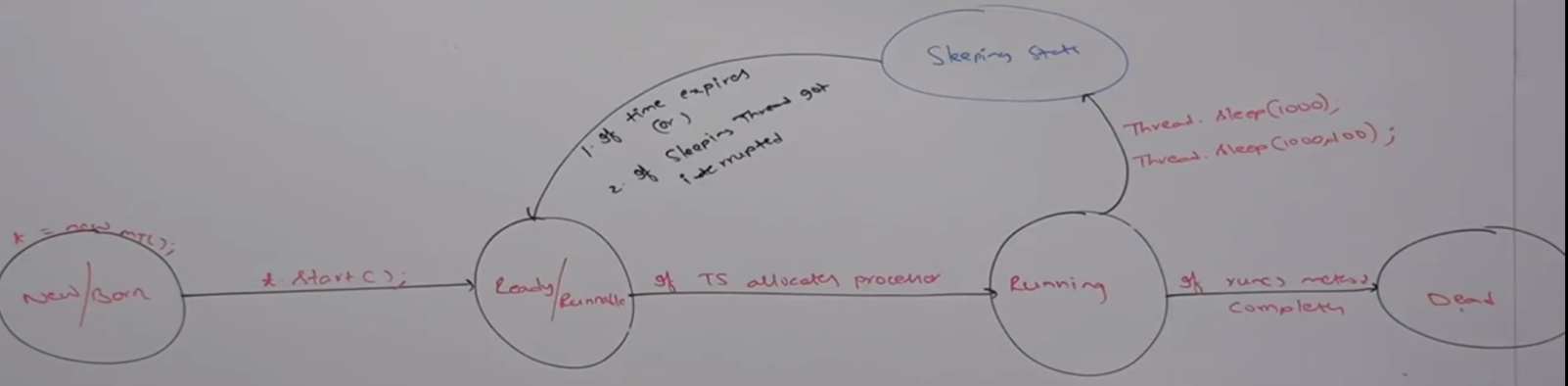
|  |  |
| --- | --- |
| 1. **public final void join () throws** InterruptionException 2. **public final void join (long timeInMs) throws** InterruptionException 3. **public final void join (long timeInMs, int nanoSec) throws** InterruptionException | 1. public static native void sleep (long ms) **throws** InterruptionException // not written in java 2. public static void sleep (long ms, int ns) **throws** InterruptionException |

**Q: Why in join we have three methods and in sleep we have two methods**?

Here in case of join () we have three method and in one of the three method we are not passing any argument while in case of sleep () we don’t have such type method, because we cannot put any thread on sleep mode forever just like human being, every human being will go for sleep for a particular amount of time.

Similar to join every sleep method throws InterruptionException exception, which is checked exception, hence whenever we are using sleep() method, compulsory we should handle InterruptionException exception either by try-catch or by throws keyword, otherwise compile time error.

Q: What is impact on our thread cycle



So like join () method when a running thread calls sleep method using [**Thread.sleep (1000) or Thread.sleep (1000, 100)**] then thread immediately goes into sleep state.

The thread will come out from the sleep state: **if time expires or sleeping thread got interrupted.**

**Note:** once the thread will come out from sleep state then it will not go back directly into running state, first the thread will go to ready/Runnable state and from there if thread scheduler allocates processor then thread once again comes into running state for further processing.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** SlideRotatorUsingSleep {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  **for** (**int** i = 1; i < 5; i++) {  System.***out***.println("Slide:- " + i);  Thread.*sleep*(5000);  }  }  } | Output:  **Slide:- 1**  **Slide:- 2**  **Slide:- 3**  **Slide:- 4**  **Slide:- 5**  **Here in this program the slide is getting rotate after each 5 second. Here we have kept sleep(5000) in each slide rotation. So each slide will be paused for 5 second in each rotation.** |

**Q: In the previous discussion we have used a word a lot like in join or sleep a thread can be interrupted, so how a thread can be interrupted or can interrupt another thread and interruption is good or bad?**

A: Thread interruption is done based on situation, so sometime interruption is good and sometime it is bad.

Thread class contains one method **public void interrupt () and a thread can interrupt the sleeping thread or waiting thread. OR**

**A thread can interrupt a sleeping thread or waiting thread by using interrupt () method of Thread © class.**

**Prototype: public void interrupt ();**

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MySleepThread **extends** Thread {  @Override  **public** **void** run() {  // super.run();  **try** {  **for** (**int** i = 0; i < 10; i++) {  System.***out***.println ("Child lazy thread...”);  **Thread.*sleep*(5000);**  }  } **catch** (InterruptedException e) {  System.***out***.println("I got interrupted");  e.printStackTrace();  }}} | **package** com.threadconcept;  **public** **class** SleepThreadDemo {  **public** **static** **void** main(String[] args) {  MySleepThread t = **new** MySleepThread();  t.start();  **t.interrupt();**  // t.isInterrupted();// return boolean  System.***out***.println("Main Thread");  }  } |
| **Output:**  Main Thread  Child lazy thread...  I got interrupted  java.lang.InterruptedException: sleep interrupted  at java.lang.Thread.sleep(Native Method) at com.threadconcept.MySleepThread.run(MySleepThread.java:11) | Here when **t.start ()** executes then run method of child class is executes and in the first iteration of for loop SOP line executes and after that thread goes into sleep state, and we know the interrupt () gets called when any thread is in sleeping state or waiting state so now the interrupt method got called on child thread object by main thread and hence the execution got interrupted and we get interrupt excerption as shown in the output. |

Note: Whenever we are calling interrupt () method if the target thread (child thread) not in sleeping or waiting state then there is no impact of interrupt () immediately. Interrupt call will be waited until target thread entered into sleeping or waiting state. If the target thread entered into sleeping or waiting state, then immediately interrupt call will interrupt the target thread.

If the target thread never entered into sleeping or waiting state in its life time then there is no impact of interrupt call, this is the only case where interrupt call will be wasted.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MySleepThread **extends** Thread {  @Override  **public** **void** run() {  // super.run();  **for** (**int** i = 0; i < 5; i++) {  System.***out***.println("Child lazy thread..");  }  **try** {  **Thread.*sleep*(5000);**  } **catch** (InterruptedException e) {  System.***out***.println("I got interrupted");  // e.printStackTrace();  }  }  } | **package** com.threadconcept;  **public** **class** SleepThreadDemo {  **public** **static** **void** main(String[] args) {  MySleepThread t = **new** MySleepThread();  t.start();  t.interrupt();  // t.isInterrupted();// return boolean  System.***out***.println("Main Thread");  }  } |
| **Output:**  Main Thread  Child lazy thread..  Child lazy thread..  Child lazy thread..  Child lazy thread..  Child lazy thread..  I got interrupted  java.lang.InterruptedException: sleep interrupted  at java.lang.Thread.sleep(Native Method)  at com.threadconcept.MySleepThread.run(MySleepThread.java:12) | **So here as per the output we can see that the for loop got executed successful and wherever the child thread entered into sleeping state, immediately interrupt () method gets called and hence catch block gets executed.** |

**Comparison between yield() , join () & sleep().**

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Yield ()** | **Join()** | **Sleep()** |
| **1- Purpose** | **When a running thread paused to give the chance to another thread having same priority then we should go for yield()** | **When a running thread wants to wait until another thread completes its execution then we should go for join()** | **When a running thread don’t want to perform any operation for a particular amount of time then we should go for sleep () method.** |
| **2-it overloaded or not?** | **No** | **Yes** | **Yes** |
| **3-It is final?** | **No** | **Yes** | **No** |
| **4-It is throwing interrupt ex** | **No** | **Yes** | **Yes** |
| **5-It is native?** | **Yes** | **No** | **Sleep(logn ms) is native**  **Sleep(long ms , int ns) non native** |
| **6-It is static** | **Yes** | **No** | **Yes** |

**Session-7**

1. **Synchronization (Part-1)**

It is most valuable concept in multithreading.

1. Synchronized is the modifier applicable only for methods and block but not for classes and variables.
2. If multiple threads are trying to operate simultaneously on the same java object then there may be a chance of data inconsistency problem. To overcome this problem we should go for Synchronized keyword.
3. If a method or block declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object so that data inconsistency problem will be resolved.
4. Advantage: The main advantage of synchronized keyword is, we can resolve data inconsistency problems but the main disadvantage of synchronized keyword is, it increases waiting time of threads and creates performance problems. Hence if there is no specific requirement then it is not recommended to use synchronized keyword.

Example: Public telephone booth with lock is the best example for Synchronized, let’s when very first person comes then he entered into PTB and lock the booth, now second person came and sees that booth is already occupied and locked by one person. Then in this case second person has to wait until the person complete his conversation. Now when first person completes his conversations, then the second person entered into booth and locks it for conversation. Similarly for third, fourth and for other person mechanism will work. Similarly for an object in case of synchronization …

Internally synchronization concept is implemented by using lock. Every object in java has a unique lock, whenever we are using synchronized keyword then only lock concept into picture.

If a thread wants to execute synchronized method on the given object, then first it has to get lock of that object. Once thread got the lock then it is allowed to execute any synchronized method on that object

Once method execution completes then automatically thread releases a lock.

Acquiring and releasing a lock internally takes care by JVM and programmer not responsible for this activity.

Note: While a thread executing a synchronized method on the given object then the remaining threads are not allowed to execute any synchronized method simultaneously on the same object. But remaining threads are allowed to execute non synchronized method simultaneously

|  |  |
| --- | --- |
| **package** com.threadconcept;  **public** **class** MySynchronizedMethod {  **synchronized** **public** **void** m1() {  System.***out***.println("First synchronized method");  }  **synchronized** **public** **void** m2() {  System.***out***.println("Second synchronized method");  }  **public** **void** m3() {  System.***out***.println("Normal method");  }  } |  |

So as per the program and diagram we have four threads (t1, t2, t3, t4). First of all thread t1 comes and try to execute m1 () method and acquires the object and lock it. Now another thread t2 comes and try to execute the same method m1(). Since m1() is a synchronized method and apart form that object is already locked by t1 so t2 will have to wait until t1 release this lock and so t2 will go into waiting state.

Now another thread t3 comes and try to execute another synchronized method m2 ().As we know that for synchronized method or block first of all a lock is required then only any thread can execute any synchronized method or block, and So still object is locked by thread t1, so JVM will not allow and t3 has to wait until t1 releases the lock and so t3 will also go into waiting state.

Now another thread t4 comes and try to execute non synchronized method and as we know that for non synchronized method or block lock is not required so JVM will allow to execute m3() method by thread t4.

So lock concept is implemented based on object but not on method.

Q: if object are locked by thread t1 already then it is correct that t2 and t3 will have to wait, but how come t4 is allowed to execute non synchronized method?

A: A java object is divided into two areas

1. **Synchronized Area:** This area can be accessed by only one thread at a time and this area can be used when we are performing [update, add, delete] i.e. where the state of object is changing.
2. **Non Synchronized Area:** This area can be accessed by any number of threads simultaneously and this area can be used when we are performing read operation i.e. where the state of object is not changing.

|  |  |
| --- | --- |
|  |  |
| **package** com.threadconcept;  **public** **class** ReservationSystem {  **public** **int** checkAvailabiliy() {  // Read operation  **return** 10;  }  **synchronized** **public** String bookTicket() {  // Update operation  // Add operation  // Delete operation  **return** "Successfully booked...”;  }  } | So here we have got idea that when we can go for synchronized and when can go for non synchronized .  checkAvialabilit() method returns only number of availble tickety i.e. read operation so it is non synchronized mehtod.  While in bookTicket we are performing update, add and delete operation so we made this method synchronized so that only one thread can access this method at a time. |

**Example:**

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.sync;  **public** **class** MySynchronizedMethod {  //synchronized  **public** **void** wish(String name) {  **for** (**int** i = 0; i < 10; i++) {  System.***out***.print("Good Morning:");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  }  System.***out***.println(name);  }  }  }  Created a **non** **synchronized** wish method and include sleep () for 2 sec. | **package** com.threadconcept.sync;  **public** **class** MySyncThread **extends** Thread {  MySynchronizedMethod ms;  String name;  **Public** MySyncThread (MySynchronizedMethod ms, String name) {  **super**();  **this**.ms = ms;  **this**.name = name;  }  @Override  **public** **void** run() {  ms.wish(name);  }  }  Created a thread class and in the run() called the wish() method | **package** com.threadconcept.sync;  **public** **class** TestSynchronization {  **public** **static** **void** main(String[] args) {  MySynchronizedMethod ms = **new** MySynchronizedMethod();  **MySyncThread t1 = new MySyncThread(ms, "Dhoni");**  **MySyncThread t2 = new MySyncThread(ms, "Yuraj");**  t1.start();  t2.start();  }  }  Here we have created two child threads and each calls Child class run() method which in turn calls non synchronized method wish(). Since here two thread are trying to call one non synchronized method simultaneously so let’s see the output |

|  |  |  |  |
| --- | --- | --- | --- |
| **Output for non-synchronized method:**  Good Morning:Good Morning:Dhoni  Good Morning:Yuraj  Good Morning:Dhoni  Good Morning:Yuraj  Good Morning:Dhoni  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Dhoni  Good Morning:Dhoni  Yuraj  As we can see here we are getting mixed result- **irregular** result. | **package** com.threadconcept.sync;  **public** **class** **MySynchronizedMethod** {  // made the method synchronized  **public** **synchronized void** wish(String name) {  **for** (**int** i = 0; i < 10; i++) {  System.***out***.print("Good Morning:");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  }  System.***out***.println(name);  }  }  }  Created a **non** **synchronized** wish method and include sleep () for 2 sec. | **Output for synchronized method:**  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj | Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni So as we can see that when we made a method synchronized then in that case we are getting the expected result. |

**Note: As here in the above example we have created two threads, now it up to thread scheduler JVM which will decide which thread will get first chance to execute method we cannot decide at all. So sometime t1 get first chance and [Good Morning: Dhoni] will print first and sometime t2 will get first chance and [Good Morning: Yuraj] will be printed.**

**Hence If we are not declaring wish () as synchronized then both the threads will be executed simultaneously and hence we will get irregular output.**

**Hence If we declaring wish () as synchronized then at a time only one threads is allow to execute on given MySynchronizedMethod** class **object**(**ms**) **and hence we will get regular output.**

**So here on one object (ms) multiple tread are trying to operate on same object hence synchronization is required.**

**Case Study: Create two different object of** MySynchronizedMethod class object

|  |  |
| --- | --- |
| **package** com.threadconcept.sync;  **public** **class** TestSynchronization {  **public** **static** **void** main(String[] args) {  MySynchronizedMethod **d1** = **new** MySynchronizedMethod();  MySynchronizedMethod **d2** = **new** MySynchronizedMethod();  MySyncThread t1 = **new** MySyncThread(**d1**, "Dhoni");  MySyncThread t2 = **new** MySyncThread(**d2**, "Yuraj");  t1.start();  t2.start();  }  } | **Here first thread t1 is getting created by passing d1 and Dhoni and and then t1 calls wish() method available inside run()**  **And second thread t2 is getting created by passing obj d2 and yuraj and t2 calls wish() method available inside run()** |

So here even though wish method is synchronized we will get irregular output because thread are operating on two different java objects and synchronization works when multiple thread operate on same object not on different object.

**Conclusion:** If multiple threads are operating on same java object then synchronization is required, if multiple thread are operating on multiple java objects then synchronization is not required.

**So for example** if a couple have joint account and both (husband & wife) are trying to access simultaneously on the same account object then synchronization is required. But if both have different account I.e. husband and wife have their own account the synchronization is not required at all.

**One more example** three dogs are trying to access one plate biriyani then synchronization is required. But if all three dogs have their own one-one each plate of biriyani then synchronization is not required at all.

|  |  |
| --- | --- |
| Good Morning:Good Morning:Yuraj  Good Morning:Dhoni  Good Morning:Yuraj  Dhoni  Good Morning:Good Morning:Dhoni  Yuraj  Good Morning:Good Morning:Dhoni  Good Morning:Yuraj  Good Morning:Yuraj  Dhoni | So here both the object have their own space and do not required to be synchronized that is why we are getting irregular output, even though method is synchronized |

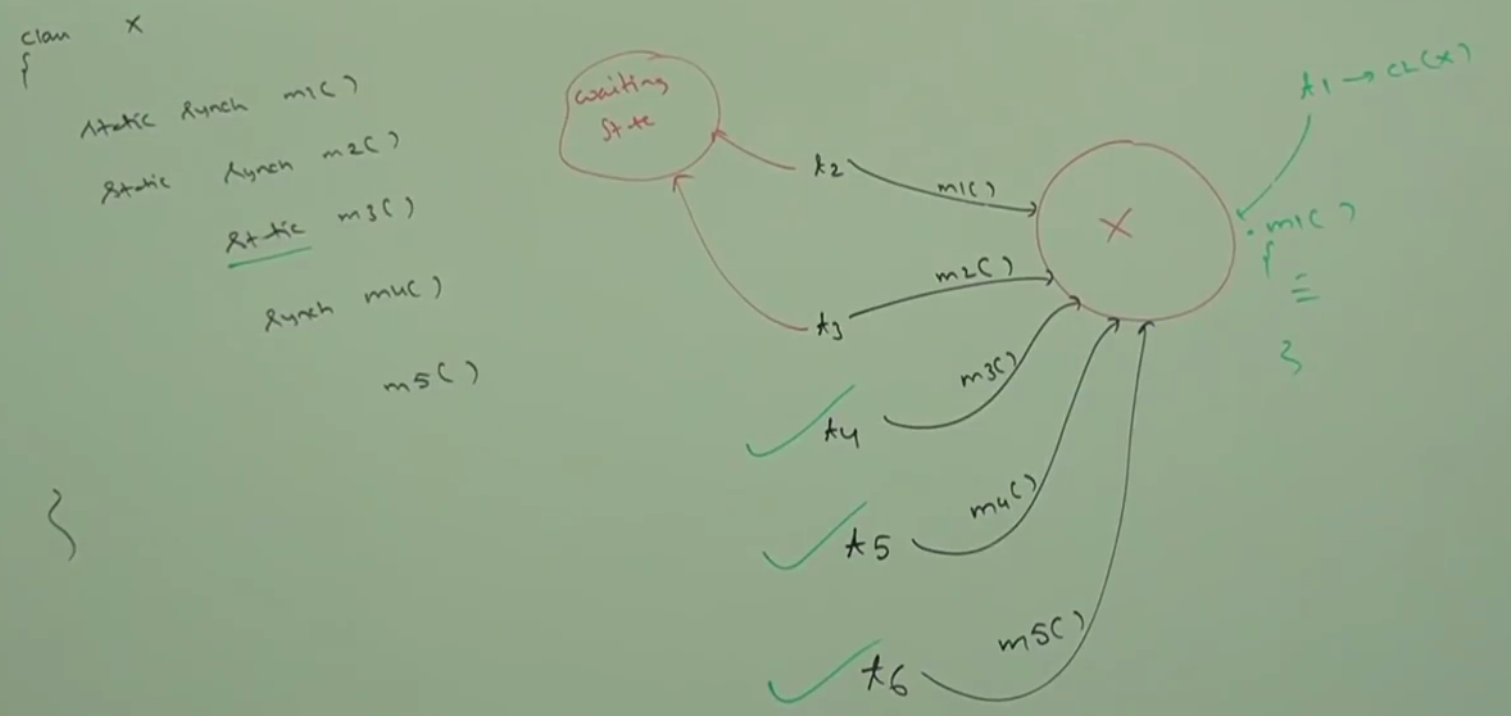
**Note: There are two level lock is available in java, 1- Class Level 2- Object Level**

**Class level lock:** Every class in java has unique lock which is nothing but class level lock. If a thread wants to execute a static synchronized method then thread required class level lock. Once thread got class level lock then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread releases a lock

While a thread executing static synchronized method the remaining threads are not allowed to execute static synchronized method of that class simultaneously, but remaining threads are allowed to execute the following methods simultaneously

1. Normal static method
2. Synchronized instance method
3. Normal instance method

|  |
| --- |
| **package** com.threadconcept.sync;  **public** **class** StaticSync {  **public** **static** **synchronized** **void** m1() { // Class level lock is required  }  **public** **static** **synchronized** **void** m2() { // Class level lock is required  }  **public** **static** **void** m3() { //It is not static synchronized (both) so class level lock is not required  }  **synchronized** **void** m4() { //It is not static synchronized(both) so class level lock is not required only object  level lock is required  }  **void** m5(){ // it is instance method Neither class level nor object level lock is required  }  } |



|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.sync;  **public** **class** MySynchronizedMethod {    **public** **static** **synchronized** **void** wish(String name) {  **for** (**int** i = 0; i < 5; i++) {  System.***out***.print("Good Morning:");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  }  System.***out***.println(name);  }  }  }  We have created **static** **synchronized method so class level lock is reaui** | **package** com.threadconcept.sync;  **public** **class** MySyncThread **extends** Thread {  MySynchronizedMethod ms;  String name;  **Public** MySyncThread (MySynchronizedMethod ms, String name) {  **super**();  **this**.ms = ms;  **this**.name = name;  }  @Override  **public** **void** run() {  ms.wish(name);  }  }  Created a thread class and in the run() called the wish() method | **package** com.threadconcept.sync;  **public** **class** TestSynchronization {  **public** **static** **void** main(String[] args) {  MySynchronizedMethod **d1** = **new** MySynchronizedMethod();  MySynchronizedMethod **d2** = **new** MySynchronizedMethod();  MySyncThread t1 = **new** MySyncThread(**d1**, "Dhoni");  MySyncThread t2 = **new** MySyncThread(**d2**, "Yuraj");  t1.start();  t2.start();  }  } |
| **Output:**  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Dhoni  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj  Good Morning:Yuraj | **Conclusion:** When multiple thread created from different objects and all the threads wants to operate on any synchronized method, we will get irregular output even though method is synchronized because here we have to different tread t1,t2 created by two different object so they will treat as two different thread of two different object and in this case both will be operating parallel on their own object and in this case we will get irregular output.  Solution: if we want to get regular output then apart from declaring synchronized, we will have to declare static along with synchronized as…  **public** **static** **synchronized** **void** wish(String name) | |

Another example

1. Create two synchronized method displayNum & displayChar
2. Create one thread class myNumThread for calling displayNum inside run() method
3. Create another thread class myCharThread for calling displayChar inside run() method
4. Create Test having main method, create one object and two thread on the same object

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.twothread;  **public** **class** **DisplayNumChar** {  **public** **synchronized** **void** **displayNum**() {  **for** (**int** i = 1; i <= 10; i++) {  System.***out***.print(i);  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  **public** **synchronized** **void** **displayChar**() {  **for** (**int** i = 65; i < 75; i++) {  System.***out***.print((**char**) i);  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  } | **package** com.threadconcept.twothread;  **public** **class** **MyThreadNum** **extends** Thread {  DisplayNumChar d;  **public** MyThreadNum(DisplayNumChar d) {  **super**();  **this**.d = d;  }  @Override  **public** **void** run() {  d.displayNum();  }  }  **package** com.threadconcept.twothread;  **public** **class** **MyThreadChar** **extends** Thread {  DisplayNumChar d;  **public** MyThreadChar(DisplayNumChar d) {  **this**.d = d;  }  @Override  **public** **void** run() {  d.displayChar();  }  } | **package** com.threadconcept.twothread;  **public** **class** TestNumCharThread {  **public** **static** **void** main(String[] args) {  DisplayNumChar d = **new** DisplayNumChar();  MyThreadChar t1 = **new** MyThreadChar(d);  MyThreadNum t2 = **new** MyThreadNum(d);  t1.start();  t2.start();  }  }  Output:  ABCDEFGHIJ12345678910  Or  12345678910ABCDEFGHIJ |

As here we can see that two threads are crated on same object. So once both the tread starts then it goes to run() method and from there it calls **displayNum and displayChar** synchronized method. So as we know that here multiple thread are trying to operate on the same object so to get regular output we have made both the method synchronized of class **DisplayNumChar.** So if one of the threads gets lock on the object then other thread will wait until first thread complete its execution.

So far we have discussed the following things

**Session-8**

**Synchronized blocks**

What is the purpose of synchronized bocks, how to define Synchronized blocks and what is the advantage over the synchronized method.

Example: Let’s say there is a bomb blast happen in the Whitefield area in Bangalore. So if we block entire Bangalore city then it would affect entire Bangalore city while blast happened only in Whitefield area. So instead of blocking entire Bangalore we should block only that particular critical area (Whitefield) where the blast happened. So that other part of the Bangalore people can move freely without any problem. In the same way--

If very few lines of code required synchronization then it is not recommended to declare entire method as synchronized, we have to enclose those few lines of code by using synchronized block.

The main advantage of synchronized block over synchronized method is it reduces waiting time of threads and improves performance of the system.

**Declaration**: **We can declare synchronized block as follows.**

|  |  |  |
| --- | --- | --- |
| **1-To Get lock of current object** | **2-To Get lock of particular object d** | **3-To Get class level lock** |
| **package** com.threadconcept.twothread;  **public** **class** SynchronizedBlock {  {  **synchronized** (**this**) {  // If a Thread got lock of current object(this) then only  //it is allowed to execute this area  }  }  } | **package** com.threadconcept.twothread;  **public** **class** SynchronizedBlock {  {  **DisplayNumChar d = new DisplayNumChar();**  **synchronized** (d) {  // If a Thread got lock of particular object(d) then only  //it is allowed to execute this area  }  }  } | **package** com.threadconcept.twothread;  **public** **class** SynchronizedBlock {  {  **synchronized** (DisplayNumChar.**class**) {  // If a Thread got lock of class -DisplayNumChar.class then only  // it is allowed to execute this area  }  }  } |

Example: Current object level lock:

|  |  |
| --- | --- |
| **package** com.threadconcept.twothread;  **public** **class** DisplayNumChar {  **public** **void** displayNum() {  System.***out***.println("Start Synchronized block of displayNum sexectuion");  **synchronized** (**this**) {  **for** (**int** i = 1; i <= 10; i++) {  System.***out***.print(i);  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) { e.printStackTrace();  }  }  }  System.***out***.println("End Synchronized block of displayNum sexectuion");  }  **public** **void** displayChar() {  System.***out***.println("Synchronized block of displayChar sexectuion");  **synchronized** (**this**) { // this means current object  **for** (**int** i = 65; i < 75; i++) {  System.***out***.print((**char**) i);  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  System.***out***.println("End Synchronized block of displayChar sexectuion");  }  } | **package** com.threadconcept.twothread;  **public** **class** **MyThreadNum** **extends** Thread {  DisplayNumChar d;  **public** MyThreadNum(DisplayNumChar d) {  **super**();  **this**.d = d;  }  @Override  **public** **void** run() {  d.displayNum();  }  }  **package** com.threadconcept.twothread;  **public** **class** **MyThreadChar** **extends** Thread {  DisplayNumChar d;  **public** MyThreadChar(DisplayNumChar d) {  **this**.d = d;  }  @Override  **public** **void** run() {  d.displayChar();  }  } |

So here instead of declaring a method as synchronized we have declared synchronized at block level having certain numbers of code.

|  |  |
| --- | --- |
| **package** com.threadconcept.twothread;  **public** **class** TestNumCharThread {  **public** **static** **void** main(String[] args) {  DisplayNumChar d = **new** DisplayNumChar();  MyThreadChar t1 = **new** MyThreadChar(d);  MyThreadNum t2 = **new** MyThreadNum(d);  t1.start();  t2.start();  }  } | Output:  Synchronized block of displayChar sexectuion  Start Synchronized block of displayNum sexectuion  **ABCDEFGHIJ** End Synchronized block of displayChar sexectuion  **12345678910** End Synchronized block of displayNum sexectuion  **Note**: Here we have created two thread on the same object and we have created the block as synchronized so only one thread will have lock on the current object and will be allowed first to execute and then second thread will start their execution. Hence we will get regular output as shown above. |

Note: Let’s create two tread (t1 & t2) on two different objects and then see the result.

|  |  |
| --- | --- |
| **package** com.threadconcept.twothread;  **public** **class** TestNumCharThread {  **public** **static** **void** main(String[] args) {  DisplayNumChar d1 = **new** DisplayNumChar();  DisplayNumChar d2 = **new** DisplayNumChar();  MyThreadChar t1 = **new** MyThreadChar(**d1**);  MyThreadNum t2 = **new** MyThreadNum(**d2**);  t1.start();  t2.start();  }  } | Output:  Synchronized block of displayChar sexectuion  Start Synchronized block of displayNum sexectuion  **A1B2C34DE5F6G7H89I10J**  End Synchronized block of displayNum sexectuion  End Synchronized block of displayChar sexectuion  **Note**: So we can see that threads got created on two different object so each thread have their own object to operate and hence both the thread will execute simultaneously and hence we will get irregular output as shown above.  In this case when two thread have their own object to operate then in case object level lock will not work to get regular output and in this case threads required class level lock instead of locking current lock let’s put class level locl in synchronized block as shown below. |
| **Output:**  Synchronized block of displayChar sexectuion  Start Synchronized block of displayNum sexectuion  **ABCDEFGHIJ**  End Synchronized block of displayChar sexectuion  **12345678910**  End Synchronized block of displayNum sexectuion  **So we can see that once we put lock on class level then only one thread will get lock at a time and will be allowed to operate on that particular class level lock and hence we have got regular output as shown above.**  **Note: Both the synchronized must have class level lock otherwise we will get irregular output** | **package** com.threadconcept.twothread;  **public** **class** DisplayNumChar {  **public** **void** displayNum() {  System.***out***.println("Start Synchronized block of displayNum sexectuion");  **synchronized** (DisplayNumChar.**class**) {// class level lock  **for** (**int** i = 1; i <= 10; i++) {  System.***out***.print(i);  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) { e.printStackTrace();  }  }  }  System.***out***.println("End Synchronized block of displayNum sexectuion");  }  **public** **void** displayChar() {  System.***out***.println("Synchronized block of displayChar sexectuion");  **synchronized** (DisplayNumChar.**class**) { // Class level lock  **for** (**int** i = 65; i < 75; i++) {  System.***out***.print((**char**) i);  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  System.***out***.println("End Synchronized block of displayChar sexectuion");  }  } |

So we have got to know how we can define synchronized block on current object lock and class level lock.

Lock concept applicable for object types and class types but not for primitive (int x =0) hence we cannot pass primitive type as argument to synchronized block otherwise we will get compile time error saying unexpected type found int: required reference.

|  |  |
| --- | --- |
| **package** com.threadconcept.twothread;  **public** **class** SynchronizedBlock {  {  DisplayNumChar d = **new** DisplayNumChar();  **synchronized** (d) { //  // If a Thread got lock of particular object(d) then only  // it is allowed to execute this area  }  **synchronized** (**this**) {  // If a Thread got lock of current object(this) then only  // it is allowed to execute this area  }  **synchronized** (DisplayNumChar.**class**) {  // If a Thread got lock of class -DisplayNumChar.class then only  // it is allowed to execute this area  }  **int** x=0;  **synchronized** (x) { // Not possible to pass primitive type in sync block  // not for primitive (int x =0) hence we cannot pass primitive type as argument to synchronized block otherwise we will get compile time error saying unexpected type found int: required reference.  }  }  } |  |

FAQs:

1. What is synchronized keyword and where we can apply? //**Ans:** It is modifier or keyword and can apply where data inconsistency problem coms
2. Explain advantage of synchronization keyword? // Ans: it Resolve data inconsistency problem
3. Disadvantage of synchronization keyword // **Ans**: It increases waiting time of thread and increase performance time.
4. What is race condition in multithreading.// **Ans**: If multiple threads operating on same object simultaneously then there may be a chance of inconsistency in data, this is nothing but race condition, and we can resolve this by using synchronization modifier or keyword.
5. What is object lock and when it is required? // **Ans:** it is a unique lock on object and whenever a thread wants to execute instance synchronized method
6. What is class level lock and when it is required? it is a unique lock on class whenever a thread wants to execute static synchronized method then this lock is required
7. What is the difference between class level lock and object level lock?
8. While a thread executing synchronization method on the given object is the remaining threads are allowed to execute any other synchronization method simultaneously on the same object// **Ans** : No
9. What is synchronized block?
10. How declare a bock to get a lock current object or to get class level lock?
11. What is the advantage of synchronized block over synchronization method? // **Ans:** Waiting will be reduces for thread and performance will be improved
12. What is synchronized statement? Ans: As per java specification there is no such type of terminology, it has got asked by interviewer only. The statements present in synchronized method and block are called synchronized statements.
13. Is a thread can acquire multiple lock simultaneously // **Ans:** yes of-course from different objects

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.twothread.multi.lock;  **public** **class** X {  **public** **synchronized** **void** m1() {  System.***out***.println("Lock on x-" + Thread.*currentThread*());  Thread.*currentThread*();  Y y = **new** Y();  **synchronized** (y) {  System.***out***.println("Lock on x and Y-" + Thread.*currentThread*());  Z z = **new** Z();  **synchronized** (z) {  System.***out***.println("Lock on X, Y, Z-" + Thread.*currentThread*());  }  }  }  } | **public** **class** Y {  }  **public** **class** Z {  } | **package** com.threadconcept.twothread.multi.lock;  **public** **class** MyMultiLockThread **extends** Thread {  X x;  **public** MyMultiLockThread(X x) {  **this**.x = x;  }  @Override  **public** **void** run() {  x.m1();  }  } |

|  |  |
| --- | --- |
| **package** com.threadconcept.twothread.multi.lock;  **public** **class** MultipleLockOnOneThread {  **public** **static** **void** main(String[] args) {  X x = **new** X();  MyMultiLockThread t = **new** MyMultiLockThread(x);  t.start();  }  } | Output  Lock on x-Thread[Thread-0,5,main]  Lock on x and Y-Thread[Thread-0,5,main]  Lock on X, Y, Z-Thread[Thread-0,5,main] |

**Session-9**

**Inter Thread Communication (Part-1)**

**(1)Wait (), (2) notify (), (3) notifyAll ()**

How two threads communicate with each other?

What is need of communication to each other?

The communication between two threads is nothing but thread communication.

Analogy: Let’s say someone send a letter to me whom has to drop into my home latter box. Now there could be possibility to check whether we have received letter in my letter box or not.

The first approach is i will go myself and check in the letter box periodically and when letter would be available then I will take it from letter box and read happily. But here I myself have to waste lot of effort and time to check the letter in the letter box, and this approach is not at all suitable.

Another approach is I will page one message on letter box that I am waiting for letter and please notify me if you are dropping letter in the letter box. Now when the postman will come to drop letter in my letter box then I will see that I am waiting for letter and then he will notify me that now letter has been dropped and now you can take and read letter happily.

So here the person (myself) who is expecting or waiting for letter will call wait () method on letter box (object) and the person (postman) who will drop letter in the box and will update letter box (object) will call notify () method on letter box (object). Once the person gets the notification happily he can continue read the letter. So here using wait() and notify() both the person or threads can communicate to each other.

The same analogy is applicable for two threads also when they want to communicate with other. In the above analogy there are two thread (me and postman) and letter box would be object on which the wait () and notify () would be applied.

**Point-1:**

Two threads can communicate with each other by using wait (), notify () and notifyAll () method. The thread (waiting thread) which is expecting updating is responsible to call wait () method then immediately the thread will enter into waiting state.

The thread (performing thread) which is performing updating, after performing updating it is responsible to call notify() method then waiting thread will get the notification and continue its execution with those updated items. So here both the thread can communicate to each other using wait (), notify (), notifyAll () methods.

All the three method [wait (), notify (), notifyAll ()] is present in Object class but not in Thread class.

**Point-2:**

Q: The above thee method is applicable in multithreading only then why these method is present in Object class not in Thread class?

Ans: Reason: All the above method presents in Object class but not in Thread class because thread can call these method on any java object.

Explanation:

|  |  |  |
| --- | --- | --- |
| **public** **class** Y {  **public** **void** y1() {  }  **public** **void** y2() {    }  } | **public** **class** Z {  **public** **void** z1() {  }  **public** **void** z2() {    }  } | **public** **class** X {  **public** **void** x1() {  }  **public** **void** x2() {    }  } |

As here created three classes (X,Y,Z) and their methods respectively

|  |
| --- |
| **package** com.threadconcept.twothread.multi.lock;  **public** **class** MultipleLockOnOneThread {  **public** **static** **void** main(String[] args) **throws** InterruptedException { |

|  |  |  |  |
| --- | --- | --- | --- |
| X x = **new** X();  // X class method  x.x1();  x.x2();  x.wait(2000);  x.notify();  x.notifyAll();  x.y1();  The method y1() is undefined for the type X | Y y = **new** Y();  // Y class method  y.y1();  y.y2();  y.wait(2000);  y.notify();  y.notifyAll();  y.x1();  The method x1() is undefined for the type Y | Z z = **new** Z();  // Z class method  z.z1();  z.z2();  z.wait(2000);  z.notify();  z.notifyAll();  z.start();  The method start() is undefined for the type Z | // For Thread method  Thread t1= **new** Thread();  t1.start();  t1.wait(2000);  t1.notify();  t1.notifyAll();  t1.x1();  The method x1() is undefined for the type Thread |

As we know that we can call a method of class using its object when it belongs to that particular class only. And if try to call the methods if it does not belongs to that particular class then we get compile time error like method above in case (x.y1();).

Here we are trying to call **y1 ()** on X class object and we are getting compile time error saying-The method y1() is undefined for the type X. Similarly for Class Y, Class Z and Class Thread.

But, all three methods [wait (), notify (), notifyAll ()] is available in all the classes X, Class Y, Class Z, Class Thread. Hence in case of multithreading environment a thread can call these methods on any java object.

For example let’s say if a thread wants to do some modification on object x or y or z, so thread can call these methods on x or y or z object, that is why these methods are available in Object class so that these methods can be called on any java objects for modification.

These methods are being used in thread communication which can be called on any class object just to convey the message that a thread waiting for your (another thread which is performing update) update on this particular java object. And once update is done another thread can call these methods on the same updated object just to inform their update.

**Point-3:**

If a thread wants to call wait () on any object then compulsory that **thread should be owner of that object**. For example I can call wait () on my letter box object. I cannot call wait () method on my neighbor letter box. If a thread try to call wait () on another object which it is not owning then we will IllegealMonitorStateException.

**Now Q: When we can say that a thread is the owner of the object.**

So when **a thread has lock on that objec**t then it means that thread I owner of that object. But again question comes that when a thread will get lock on that object. So for a thread having lock on the object, **the thread compulsory should be inside synchronized area**.

So three things should satisfy if a thread wants to call wait (), notify (), notifyAll () on any object.

1. **The thread should be owner of that object**
2. **The thread has lock on that object**
3. **The thread compulsory should be inside synchronized area**

And if we are calling these methods outside of the synchronized area, immediately we will get IllegealMonitorStateException.

To call wait (), notify(), notifyAll() methods on any object, thread should be owner of that object i.e. the thread should has lock of that object i.e. the thread should be inside synchronized area.

Hence we can call these methods only from synchronized area otherwise we will get run time exception saying IllegealMonitorStateException.

**Point-4:**

If a thread calls wait () method on any object it immediately release the lock of that particular object and entered into waiting state. For example if my thread call wait () method on letter box object then immediately my thread should release lock on letter box object and enter into waiting state so that other thread (postman) can get lock and update the letter in the letter box.

Now when another thread (postman) call notify () or notifyAll () method on letter box object (or any object) then it release the object but May not immediately because he will have to do some update work (like dropping letter into box) and after update thread should release the lock on the object (letter box object) so that that the waiting thread can continue their operation (reading letter)

**Note**: Except wait (), notify (), notifyAll () methods there is no other method where threads release the lock.

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** | **Methods in multithreading** | **Is thread release the lock?** | **Class** |
| 1 | yield() | No | Thread |
| 2 | join() | No | Thread |
| 3 | sleep() | No | Thread |
| 4 | wait() | Yes | Object |
| 5 | notify() | Yes | Object |
| 6 | notifyAll() | Yes | Object |

Which of the following is valid?

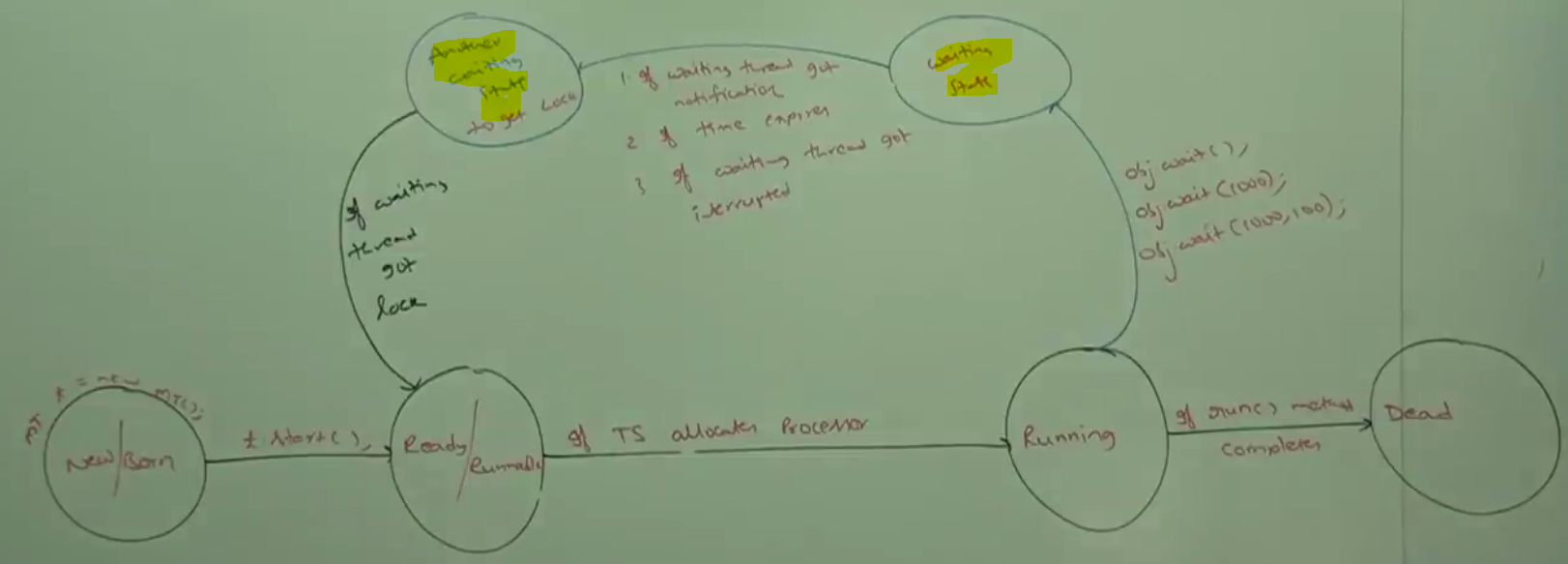
1. If a thread calls wait () method immediately it will enter into waiting state without releasing any lock- **False**
2. If a thread calls wait () method it release the lock of that object but may not immediately – **False**
3. If a thread calls wait () method on any object, it releases all locks acquired by that thread and immediately entered into waiting state- **False**
4. If a thread calls wait () method on any object if immediately releases the lock of that particular object and entered into waiting state – **True**
5. If a thread calls notify() method on any object, it immediately the lock of that particular object – **False**
6. If a thread calls notify () method on any object, it realizes the lock of that object but may not immediately- **True**

Overloaded Constructor of wait ()

|  |  |
| --- | --- |
| 1. Public final void wait() throws InterruptedException | Will wait until other thread complete its execution or operation |
| 1. Public final native void wait(long ms) throws IE | Will wait till the given time limit( not written in java) |
| 1. Public final void wait(long ms, int ns) throws IE | Will wait till the given time limit |
|  | |
| 1. Public final native void notify() throws IE |  |
| 1. Public final native void notifyAll() throws IE |  |

Every wait() throws IE which is checked exception hence whenever we are using compulsory we should hand this exception either by try-catch or by throws keyword otherwise we will get compile time error.

Q: What is the impact on our Thread life cycle?



**Explanation**:

**Inter Thread Communication (Part-2)**

|  |  |
| --- | --- |
| **package** com.threadconcept.communication;  **public** **class** ThreadOne {  **public** **static** **void** main(String[] args) {  ThreadTwo t = **new** ThreadTwo();  t.start();  // at this point now two thread will run  System.***out***.println("Total-" + t.total);  }  } | **package** com.threadconcept.communication;  **public** **class** ThreadTwo **extends** Thread {  **int** total = 0;  @Override  **public** **void** run() {  **for** (**int** i = 1; i < 100; i++) {  total = total + i;  }  }  } |

So as per the above program in the main method, child thread (t) got created and started. And in the child class run method child thread is doing the job of calculating 100 numbers.

As we know that in the main class once t.start () executes then at this point we have two threads (main and child). As per the program we are expecting sum of (100) number. Now it depend on thread scheduler that which thread will get first chance to run. After having two thread, If main thread get first chance then output **total=0** will come and if child thread get first chance then we will get total=5050. But it completely dependent on JVM and programmer cannot do anything. So here we are getting irregular output.

Solution:

1. Introduce Thread.sleep(10000) method after t.start().
2. Introduce t.join() method after t.start().
3. Introduce t.wait() method after t.start()

|  |  |  |
| --- | --- | --- |
| **public** **class** ThreadOne {  **public** **static** **void** main(String[] args) {  ThreadTwo t = **new** ThreadTwo();  t.start();  **// at this point now two thread will run**  Thread.*sleep*(2000);  **System.*out*.println("Total-" + t.total);**  }  }  So in this case when sleep method will be executed by main thread then main thread will go into sleep mode and by that time child thread will gets executed and hence we will get expected output. | **public** **class** ThreadOne {  **public** **static** **void** main(String[] args) {  ThreadTwo t = **new** ThreadTwo();  t.start();  **// at this point now two thread will run**  **t.join();**  **System.*out*.println("Total-" + t.total);**  }  }  So in this case when main thread will execute **t.join()** then main thread will wait until child thread complete its execution and hence we will get expected output every time. | **public** **class** ThreadOne {  **public** **static** **void** main(String[] args) {  ThreadTwo t = **new** ThreadTwo();  t.start();  **// at this point now two thread will run**  **t.wiat();**  **System.*out*.println("Total-" + t.total);**  }  }  So in this case when main thread will execute **t.wait()** then main thread will wait until child thread complete its update and **notify()** the main thread and hence we will get the expected output every time.  The full code is given below. |

Thread.*sleep*(2000);- Not recommended

But sleep(2000) is not recommended at all because let’s say child thread has completed its execution very early before the time mentioned in the sleep () method still main thread has to wait to complete sleep time mentioned in the sleep method, so unnecessary it will reduce the performance of the system.

t.join();- Not Recommended

But join() method is also not recommended, because until child thread complete its execution , the main thread is not allowed to print total value. Here we are expecting sum of 100 numbers which take less than nano second, let’s say after this calculation in the child thread have 1000 more line to run then main thread will have to wait until child thread complete its execution so in this case even though child thread is ready with update (result sum of 100 number) still main thread will have to wait to print total until child thread complete its execution. So unnecessary main thread waiting time is increasing and system performance will also reduce.

|  |  |
| --- | --- |
| **public** **class** ThreadOne {  **public** **static** **void** main(String[] args) {  ThreadTwo t = **new** ThreadTwo();  t.start();  **// at this point now two thread will run**  **t.wiat();**  **System.*out*.println("Total-" + t.total);**  }  } | **package** com.threadconcept.communication;  **public** **class** ThreadTwo **extends** Thread {  **int** total = 0;  @Override  **public** **void** run() {  **for** (**int** i = 1; i <= 100; i++) {  total = total + i;  }  **this**.notify();  }  } |

So as per the wait-notify, the main thread is waiting for update from child thread so main thread called wait() on child ThreadTwo object and main thread has lock on this object. So as per the wait()-notify() method calling satisfying criteria

1. **The thread should be owner of that object: So main thread has created ThreadTwo class object so it is the owner- satisfied**
2. **The thread has lock on that object: Since main thread is executing t.wait() so it has lock also- Satisfied**
3. **The thread compulsory should be inside synchronized area- the wait and notify method is not getting called in synchronized area so this is not being satisfied**

So to satisfied 3rd criteria let’s put the code in synchronized block. So the correct code is given below.

|  |
| --- |
| **public** **class** ThreadOne {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ThreadTwo t = **new** ThreadTwo();  t.start();  //Thread.*sleep*(10000);  **synchronized** (t) {  System.***out***.println("Cuurent Thread in main-" + Thread.*currentThread*().getName());  System.***out***.println("Main thread is trying to call wait-method");  **//t.wait(); Main thread waiting for update from child thread and keep waiting until child complete its execution**    **t.wait(10000);// Main thread waiting for update from child thread till given time (10 sec)**  System.***out***.println("Main thread got notification from child thread");  System.***out***.println("Total-" + t.total);  }  }  } |
| **public** **class** ThreadTwo **extends** Thread {  **int** total = 0;  @Override  **public** **void** run() {  System.***out***.println("Current thread in run-" + Thread.*currentThread*().getName());  **synchronized** (**this**) {// this means current object child class object  System.***out***.println ("Child thread start calculation.");  **for** (**int** i = 1; i <= 100; i++) {  total = total + i;  }  System.***out***.println("Child thread giving notification to main thread");  **this**.notify();// Sends notification to main thread  }  }  } |
| Current thread in run-Thread-0  Current Thread in main-main  Main thread is trying to call wait-method  Child thread start calculation.  Child thread giving notification to main thread  Main thread got notification from child thread  Total-5050 |

**Note**: If we keep Thread.sleep (10000); after t.start () along with wait() as mentioned above then what will happen?

As it is written but commented in the program because once main method execute sleep(10000) method then it will go into sleep mode for 10 second and then in that case child will get the chance to execute, so now child thread will be executed and sends notification to main thread child thread will go to dead state after completing its life-cycle.

Now after 10 second main thread will get the change and start executing remaining code, and once again in the synchronized block wait() method will be executed by main thread it means now main thread will wait until child thread gets executed and get notification from child . But now the problem is child thread already has completed its execution and life-cycle and went into dead state and cannot born again so in this case main thread will not get any notification from child thread because now there is no child thread exist and hence main thread will wait for infinite time and hence the remaining code wait () will never going to execute.

So sleeping of main thread becomes problem in executing the remaining code.

**Analogy**: It is very similar to analogy that I was waiting for letter and I got sleep for few hours and by that time postman came and dropped letter into letter box and notify me and complete his task but now as I was in sleeping mode so I could not get any notification and when I woke up then I am still waiting for notification from postman. But postman already dropped and notify me and complete his job, but due to sleep mode I could not get notification. Hence I will be waiting for notification from postman forever.

**Solution of above problem**

So the solution of the above problem is instead of waiting for forever i.e. instead of using wait(), now we will wait for few second to get notification and if main thread does not get notification then start executing the remaining code i.e. then will use wait( long time) method instead of using wait(). So the correct code is given below

|  |  |
| --- | --- |
| **public** **class** ThreadOne {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ThreadTwo t = **new** ThreadTwo();  t.start();  **Thread.*sleep*(10000);**  **synchronized** (t) {  System.***out***.println("Current Thread in main-" + Thread.*currentThread*().getName());  System.***out***.println("Main thread is trying to call wait-method");  **t.wait(10000);**  System.***out***.println("Main thread got notification from child thread");  System.***out***.println("Total-" + t.total);  }  }  } | |
| Ouput:  Current object in run-Thread-0  Child thread start calculation.  Child thread giving notification to main thread  Current Thread in main-main  Main thread is trying to call wait-method  Main thread got notification from child thread  Total-5050 |  |

So these how two threads communicate to each other?

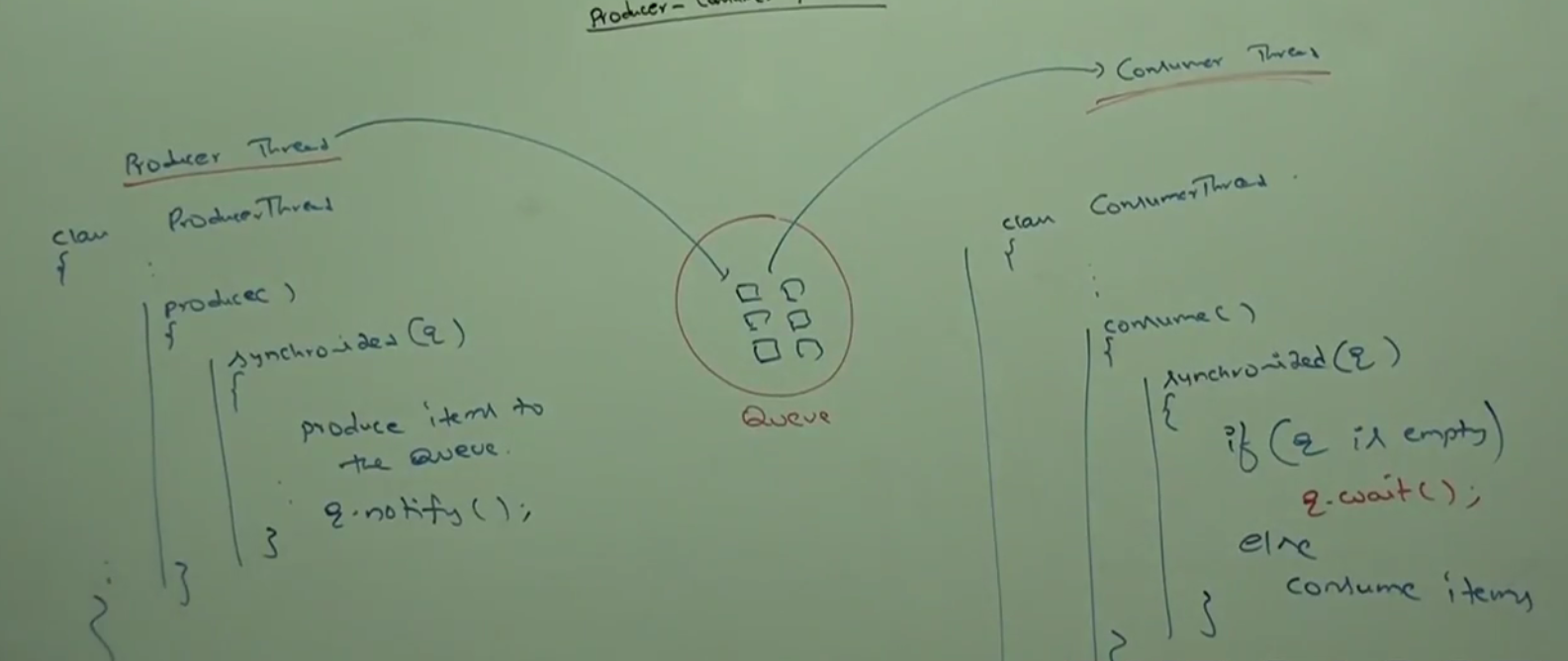
**Producer-Consumer Problem:**

Producer thread is responsible to produce items to queue and Consumer thread is responsible to consume items from the Queue.

If Queue is empty then consumer thread will call wait () method and entered into waiting state.

After producing items to the queue producer thread is responsible to call notify () method then waiting consumer will that notification and continue its execution with updated items.

|  |  |  |
| --- | --- | --- |
| Producer Thread | Queue | Consumer Thread |
| Class ProducerThread {  Produce(){  Synchronized(q){   1. Produce Items to queue 2. **q.notify();**   }  }  } | This queue contains items form producer  Items of the queue is being consumed by consumer. | Class ConusmnerThread{  Consume(){  Synchronized(q){  If(q is empty)  **q.wait()**  else  consume itms;  }  }  } |



Producer-Consumer is the best example for thread communication and using wait(), notify(), notifyAll() methods.

**Difference between notify() and notifyAll()**

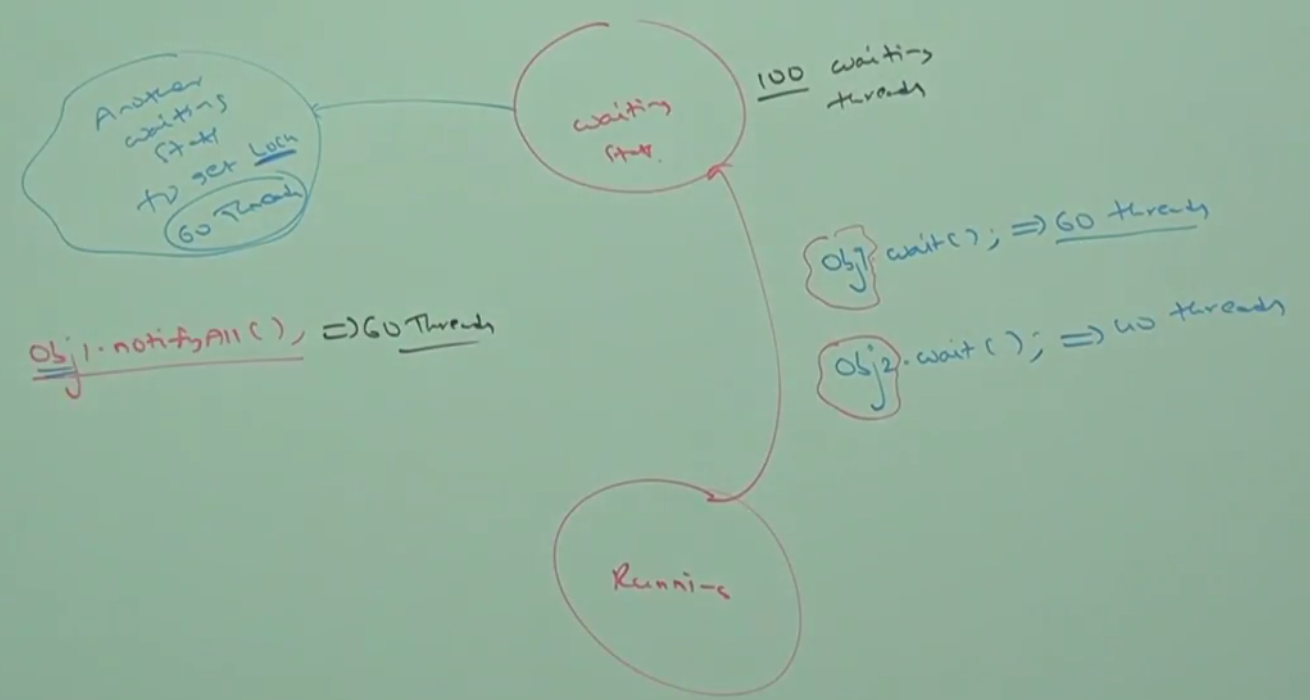
We can use notify () method to give notification for only one thread. If multiple threads are waiting then only one thread will be notified and the remaining threads will have to wait for further notifications but, which thread will be notified, we cannot tell, it depends on JVM (thread scheduler)

We can use notifyAll() to give the notification for all waiting thread of a particular object, even though multiple thread will be notified but execution will be performed one by one, because threads required lock and only one lock is available.

**Analogy**: Let’s say in the bus travel branch where buses from several destinations used to go. So all the people will gather on that branch who are travelling to different destinations. Now here in the branch, all the people are waiting for bus in a common waiting area. Now let’s say one bus which is going to vijaywada came and announced, so this announcement will be notified to all the people waiting for bus in the branch but only those people who are waiting to vijaywada will be gathered into another waiting area, and form this waiting area one by one people will go inside the bus.

Same this will be applied in case of notifyAll (). Let’s we have 100 waiting threads and 60 threads are belongs to obj1 and 40 threads are belongs to obj2. Now let’s say we have called notifyAll() on obj1 then the threads (60 threads) belongs to obj1 will be notified and they will be entered into another waiting state to get lock.

Since the lock on the object can be acquired by one thread at a time so all the 60 threads will be executing one by one whenever they will get lock on the obj1.



**Note**: On which object we are calling wait () method, thread required the lock of that particular object. For example if we are calling wait () on s1 then we have to get lock of s1 objet but not s2 object.

|  |  |
| --- | --- |
| Stack s1 = new Stack(); | Stack s2 = new Stack(); |
| Synchronized(s1){  **S2.wait()**  }  Invalid and will get IllegalMonitorStateException | Synchronized(s2){  **S2.wait()**  }  Valid |

**Invalid**: Here s2 is not the owner of synchronized means does not have lock on s2 object so wait () method cannot be called on s2

**Session-10**

**1-Dead- lock**

**2-Dead lock Vs Starvation**

If two threads are waiting for each-other forever, such type of infinite waiting is called deadlock.

So far we have learned that when two threads wants to be synch up to each-other, then in that case only one thread will have lock of the object at a time and to achieve this thread should execute their job into synchronized area (like synchronized method or block). So this raises one question.

If we use synchronized method or block then two threads will never come into deadlock?

**Ans**: So answer is just opposite. Synchronized keyword is the only the program is entered into deadlock situation. So it is not solution, it is the problem creator. So if there is not specific requirement then it is never recommended to use synchronized keyword otherwise we have to be very cautious while using synchronized keyword in the program.

So again the question comes, once the program entered into deadlock then what a programmer can do?

Ans: Being a programmer we cannot do anything, we just have to see that’s all but is operating system level to resolve the deadlock situation, for the programmer there is no resolution technique it there but several prevention technique is available.

Synchronized is the only reason for deadlock situation. Hence while using synchronized keyword we have to take special care and there is no resolution technique for deadlock but several prevention techniques are available.

Write a program to get deadlock or what are the possibilities when the program can enter into deadlock.

Program to be written

|  |  |
| --- | --- |
| **package com.threadconcept.deadlock;**  **public class A {**  **public synchronized void d1(B b) throws InterruptedException {**  **System.*out*.println("Thread 1 starts execution of d1() method..");**  **Thread.*sleep*(5000);**  **System.*out*.println("Thread 1 try to call B's last() method..");**  **b.last();**  **}**  **public synchronized void last() {**  **System.*out*.println("Inside A's Last method..");**  **}**  **}** | **package com.threadconcept.deadlock;**  **public class B {**  **public synchronized void d2(A a) throws InterruptedException {**  **System.*out*.println("Thread 2 starts execution of d2() method..");**  **Thread.*sleep*(5000);**  **System.*out*.println("Thread 2 try to call A's last() method..");**  **a.last();**  **}**  **public void last() { // Did not make synchronized**  **System.*out*.println("Inside B's Last method..");**  **}**  **}** |
| **package com.threadconcept.deadlock;**  **public class DeadLock extends Thread {**  **A a = new A();**  **B b = new B();**  **public void m1() throws InterruptedException {**  **this.start();**  **a.d1(b);**  **}**  **@Override**  **public void run() {**  **try {**  **b.d2(a);**  **} catch (InterruptedException e) {e.printStackTrace();}**  **}**  **public static void main(String[] args) throws InterruptedException {**  **DeadLock t = new DeadLock();**  **t.m1();**  **}**  **}** | **Output:**  Thread 1 starts execution of d1() method..  Thread 2 starts execution of d2() method..  Thread 2 try to call A's last() method..  Thread 1 try to call B's last() method..  Inside A's Last method..  Inside B's Last method.. |

In the above program if remove at least one synchronized keyword then program won’t entered into deadlock, hence synchronized keyword is the only reason for dead lock situation. So due to this while using synchronized keyword, we have to take special care.

|  |  |
| --- | --- |
| **package com.threadconcept.deadlock;**  **public class A {**  **public synchronized void d1(B b) throws InterruptedException {**  **System.*out*.println("Thread 1 starts execution of d1() method..");**  **Thread.*sleep*(5000);**  **System.*out*.println("Thread 1 try to call B's last() method..");**  **b.last();**  **}**  **public synchronized void last() {**  **System.*out*.println("Inside A's Last method..");**  **}**  **}** | **package com.threadconcept.deadlock;**  **public class B {**  **public synchronized void d2(A a) throws InterruptedException {**  **System.*out*.println("Thread 2 starts execution of d2() method..");**  **Thread.*sleep*(5000);**  **System.*out*.println("Thread 2 try to call A's last() method..");**  **a.last();**  **}**  **public synchronized void last() {**  **System.*out*.println("Inside B's Last method..");**  **}**  **}** |
| **package com.threadconcept.deadlock;**  **public class DeadLock extends Thread {**  **A a = new A();**  **B b = new B();**  **public void m1() throws InterruptedException {**  **this.start();**  **a.d1(b);// Main thread is having lock on Class B object**  **}**  **@Override**  **public void run() {**  **try {**  **b.d2(a); // Child thread is having lock on Class B object**  **} catch (InterruptedException e) {e.printStackTrace();}**  **}**  **public static void main(String[] args) throws InterruptedException {**  **DeadLock t = new DeadLock();**  **t.m1();**  **}**  **}** | **Output:**  Thread 1 starts execution of d1() method..  Thread 2 starts execution of d2() method..  Thread 2 try to call A's last() method..  Thread 1 try to call B's last() method..  **Here all four methods are synchronized.**  **In method m1- main thread trying to call Class A d1(b) method by having lock on Class B (b) object. In-turn d1(b) method trying to call Class B’s b.last() method and this method is locked by child thread.**  **Similarly in the run () method child thread trying to call Class B d2 (a) method by having lock on Class A (a) object and in-turn this d2(a) method trying to call Class A’s a.last() method and this method is locked by main-thread.**  **So main thread has the lock on Class B object and child thread has the lock on Class A object and waiting to each- other to release their locked object so that they can call last() method and execute further. And they will wait() forever to each-other and hence deadlock situation occurs.** |
| Analogy: This is something like two people have got each-other’s house lock key but both of them are not agreeing to release the lock key. So they have to wait to release each-other’s key other-wise they will have to wait forever to proceed further into their house. | |

**Deadlock Vs Starvation (Both are almost similar with a small difference.**

Long waiting of a thread where waiting never ends is called deadlock, where as long waiting of thread where waiting ends at certain point is called starvation. For example low priority thread has to wait until completing all high priority threads; it may be long waiting but ends at certain points, which is nothing but starvation.

So in case of deadlock the point from where two threads start waiting each other after that there will be no further execution of the program and program will never come to end of execution.

While in case of the program execution may take long time for a particular thread but program execution will come to end at a certain point. For example let’s say we have 1 Cr thread and one thread is have priority 1 and the remaining thread have priority 10 then in this case priority -1 thread will be executed at very last and hence it may take time in their execution due to other high priority thread, this is nothing but starvation.

**Session-11**

**Daemon Thread**

The thread which are executing in the back-ground is called Daemon thread.

Example: Garbage Collector (GC), Signal Dispatcher, Attach Listener etc.

What is advantage or objective of Daemon thread?

The main object of daemon thread is **to provide support for non-daemon thread** (like main thread). For example if main thread runs with low memory then JVM runs garbage collector to destroy useless objects so that number of bytes of free memory will be improved, with this free memory main thread can continue its execution.

**Analogy**: Let’s there is film small shooting in Bangalore, where hero just have to say few words to heroine. Now to shoot small scene a lot of people (Producer, Director, Makeup man etc) working in the background so that these two main people (hero and heroine) can perform without any problem.

Similarly in case of threading concept several threads like (garbage collector) is working in the background so that other threads like main threads can run without any problem.

Here the threads running in the background are called daemon threads.

Usually daemon thread having low priority but based on our requirement daemon can run with high priority also. For example let’s say main thread with priority-5 and daemon thread GC with priority-1 is running, as per the priorit main thread will get the change to run first, but let’s say suddenly memory is out of running then in this case JVM will assign GC with high priority-10. Now daemon thread GC will start executing and destroy useless object from the memory once memory gets increased again JVM will assign low priority-1 to daemon thread GC.

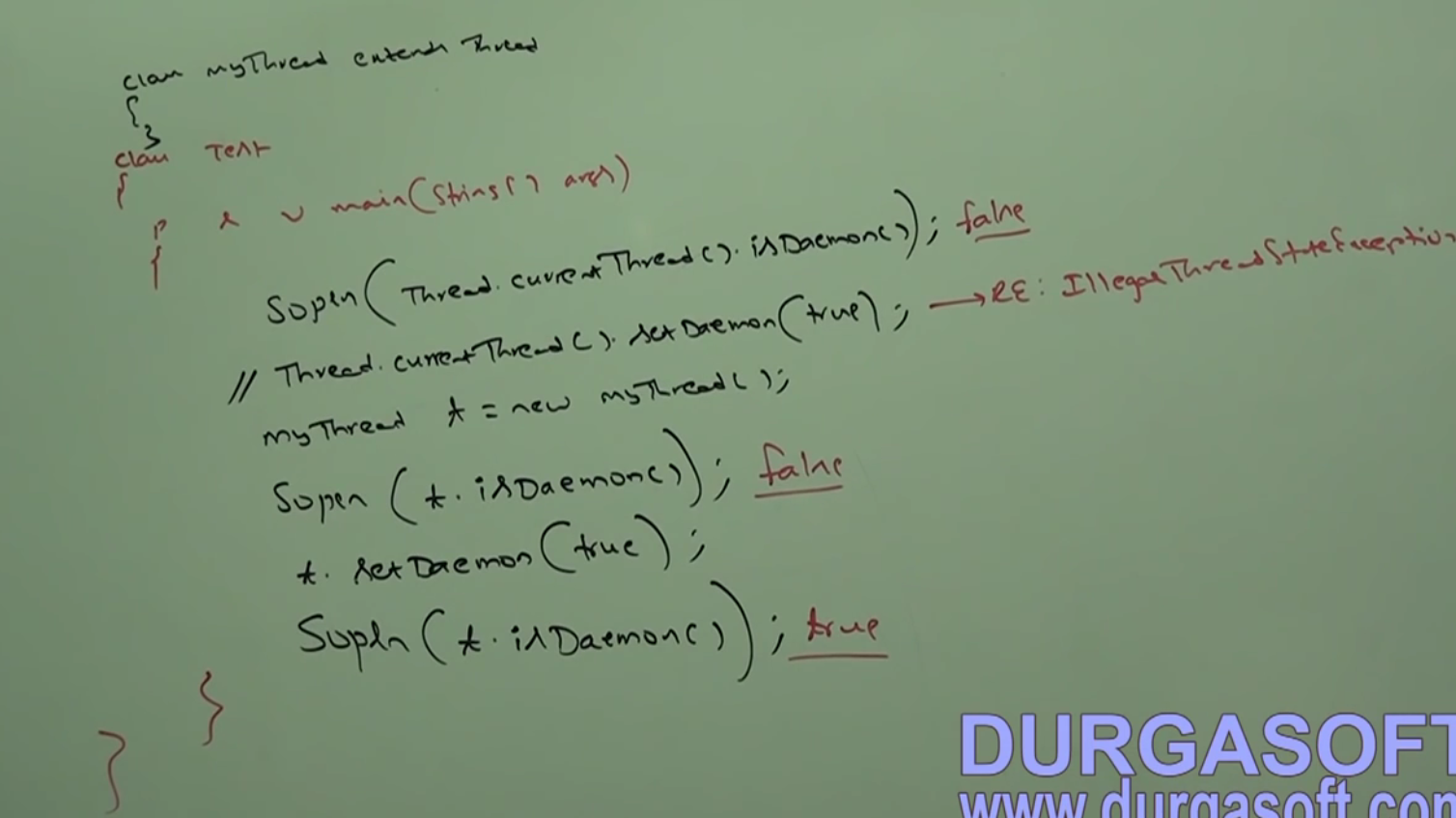
1. We can check daemon nature of a thread by using isDaemon () method of Thread class. [public boolean isDaemon()]
2. We can change daemon nature of a thread by using setDaemon (boolean b) of Thread Class.

[public void setDaemon(boolean b) ] but changing daemon nature is possible before starting of thread only, after starting a thread if we are trying to change daemon nature then we will get runtime exception illegalThreadStateException.

**Default Nature of Thread:** By default main thread always is non-daemon and all remaining thread daemon nature will be inherited from parent to child i.e. if the parent thread id daemon then automatically child thread is also daemon and if the parent thread is non-daemon then child thread is also non-daemon.

**Is it possible to change daemon nature of main thread?**

It is impossible to change daemon nature of main thread because it is already started by JVM at beginning and once threads starts then we cannot change daemon nature. But for the thread created by programmer, we can change the daemon nature of the thread because it is in our control and before starting thread we will just setDaemon(false) to make daemon thread into non-daemon thread and setDaemon(True) to make non-daemon thread into daemon thread.



|  |
| --- |
| **package** com.threadconcept;  **public** **class** DemonThread **extends** Thread {  **public** **static** **void** main(String[] args) {  System.***out***.println("Current thread is:-" + Thread.*currentThread*().getName());  System.***out***.println("Current thread is:-" + Thread.*currentThread*().isDaemon());  DemonThread t = **new** DemonThread();  System.***out***.println("Child thread is:-" + DemonThread.*currentThread*().getName());  t.setDaemon(**true**);  System.***out***.println("Child class is-" + t.isDaemon());  }  }  Explaination: |

Whenever last non-daemon thread terminates, then automatically all daemon threads will be terminated irrespective of their position.

Analogy: Once film completes then there is no required for Director, producer, makeup man and other supporting people.

|  |  |
| --- | --- |
| **public** **class** DemonThread **extends** Thread {  **public** **void** run() {  **for**(**int** i=0;i<10;i++) {  System.***out***.println("Child thread");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  }  } | **package** com.threadconcept.daemon;  **public** **class** TestDaemon {  **public** **static** **void** main(String[] args) {  DemonThread t = **new** DemonThread();  t.setDaemon(**true**);  t.start();  System.***out***.println("End of main program”);  }  }  **Output**:  Child class is-true  Child thread |

Explanation:

If we are commenting line-1 then both main and child threads are non-daemon and hence both the thread will be executed until their completion. If we are not commenting line-1 then main thread is non-daemon and child thread is daemon. Hence whenever main thread terminates then automatically child thrad will be terminated. In this case output is as given below.

|  |  |  |
| --- | --- | --- |
| End of main program  Child thread | End of main program | Child thread  End of main program |

So here child thread got executed only once or not at all.

**Session-12**

**Green Thread, stop (), suspend (), resume ()**

Java multithreading concept is implemented by using the following two models.

1. Green Thread Model
2. Native Operating System (OS) model.

**Green Thread Model:**

The thread which is managed completely by JVM without taking underlying OS support is called Green thread. Very few OS like Sun- Solaris provide support for Green Thread Model. Anyway Green Thread Model is deprecated and not remanded to use.

**Native OS Model:**

The thread which is managed by the JVM with the help of underlying OS, is called Native OS Model. All windows based OS; provide support for Native OS Model.

**How to stop a thread:**

We can stop a thread execution by using stop () method of Thread© class. [public void stop()].

If we call stop method then immediately the thread will enter into dead state. Anyway stop () method is deprecated and not recommended to use.

For example: Let’s say in the run (), we have open database connection and start reading data from database and finally after reading all the data we have closed the database connection. Let’s while thread was executing read () for reading data, same time we have called stop () method which was executing child thread (i.e. main thread called stop method on child class object) then immediately child thread will be stopped in between only and then child thread cannot proceed further for closing the database and database connection will be opened, so this is one of the disadvantage of stop() method of Thread class.

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.stop;  **public** **class** DBConnection **extends** Thread {  @Override  **public** **void** run() {  System.***out***.println("Open DB Connection..");  System.***out***.println("Start reading data from DB..");  **for** (**int** i = 0; i < 5; i++) {  System.***out***.println("Data-" + i);  }  System.***out***.println("Closed DB Connection..");  }  } | | **package** com.threadconcept.stop;  **public** **class** TestStopMethodOfThread {  @SuppressWarnings("deprecation")  **public** **static** **void** main(String[] args) {  DBConnection t = **new** DBConnection();  t.start();  System.***out***.println("Main thrad killing child thread by calling stop()");  t.~~stop~~();  }  }  main thread execute t.stop() line so once it gets executed child thread will be stop immediately |
| Main thrad killing child thread by calling stop()  Open DB Connection.. | Main thrad killing child thread by calling stop() | Main thrad killing child thread by calling stop()  Open DB Connection..  Start reading data from DB.. |

So we have got the multiple possible output and sometime when main thread get first chance it kills child tread first and then in this case child thread never get chance to execute. Sometime during child thread execution stop () methods gets executed and child thread gets killed in between and then not able to execute further.

**How to suspend and resume a thread:**

We can suspend a thread by using suspend () method of Thread class then immediately the thread will be entered into suspended state.[public void suspend()] [t.stop()]

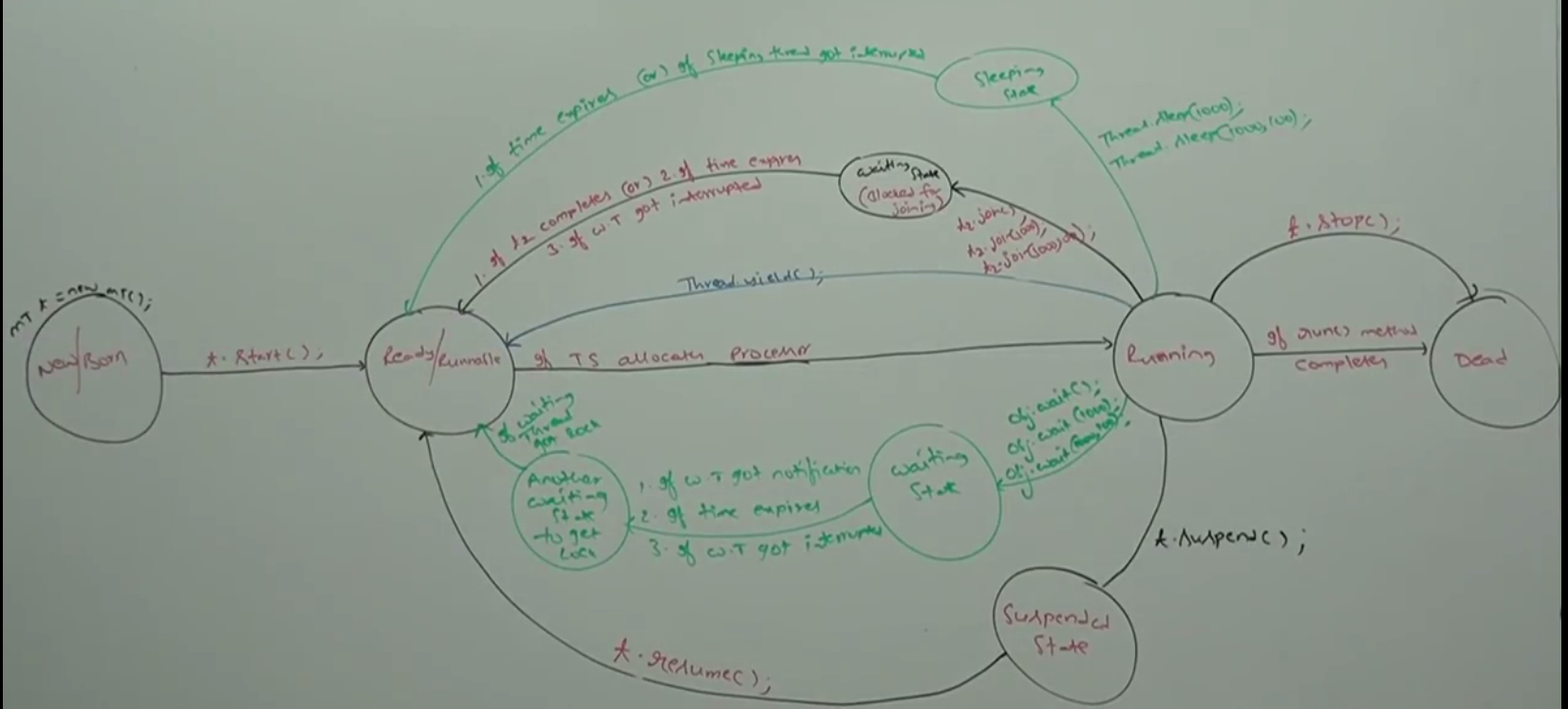
We can resume a suspended thread by using resume method of thread class, and then suspended thread can continue its execution. [public void resume ()][t.resume ()]

Anyways these methods are deprecated and not recommended to use

Analogy: Generally in an government office when an employee found in any legal activity then immediately he/she gets suspended temporarily and after investigation if found not guilty then again he/she can resume his/her service to continue work.

Same thing when a thread call suspend method [t.suspend ()] then thread immediately goes into suspended state. And after some time when call resume method [t.resume ()] then thread goes to ready/Runnable state to continue its execution.

Thread life-cycle including all methods.



1. [MyThread t = new MyThread ();] or [Thread t = new Thread ()] is nothing but **New/Born/create** **state** of any thread.
2. When calls t.start () then thread goes to **Ready/Runnable** **state**: Means now thread is ready to execute
3. If TS allocates processor then thread enters into **Running state**:
4. Now if run () method execution completes sucessfully then thread enters into **dead/stop state**.

The above from 1-4 steps is the execution state of thread from born to dead if everything goes well. But generally it does not happens because generally in complex programming when a thread enters into running state then some time it has to go from several phases. So we have the following sub-steps from **running state (step-3)** and before **dead state (Step-4)** which is given as below

|  |  |
| --- | --- |
| 3.1- | If the running thread calls **[Thread.yield ()]** method to give chance to other thread having same priority then thread directly enters into [ready/Runnable] state. And when TS allocates processor then tread again enter into [Running state] |
| 3.2- | If the running thread calls **[t2.join () / t2.join (1000) / t2.join (1000, 100)]** method then the running thread enters into waiting state or blocked state until the other thread complete its execution. Now if thread t2 complete its execution or if time expires or waiting thread got interrupted then the waiting thread entered into [Ready/Runnable state] and when TS allocates processor then thread once again enters into [Running state]. Note: The waiting thread can never enter into running state from waiting state, first the WT has to go into [Ready/Runnable] state and when TS allocates processor then only WT will enter into [Running state]. |
| 3.3- | If the running thread calls **[Thread.sleep (1000) or Thread.sleep (1000, 100)]** method then running thread immediately enters into sleeping state until the time given inside sleep () method expires or sleeping thread got interrupted in between. When time given inside sleep () method expires or sleeping thread got interrupted in between then thread will enter into [Ready/Runnable state] from sleeping state and when TS allocates processor then tread again enter into [Running state]. |
| 3.4- | If a running thread calls **[obj.wait () or obj.wait (1000) or obj.wait (1000, 100)]** method then thread enter into [waiting state] and when the waiting thread [ **got notified** or **the time given inside wait() expires** or **interrupted in between**] then thread enters into another waiting state to get the lock (chance) and when the thread get lock then it enters into [Ready/Runnable] state and when TS allocates processor then tread again enter into [Running state]. |
| 3.5- | If the running thread calls **[t.suspend ()]** method then thread enters into [suspended state] and from there when thread calls **[t.resume()]** then thread enters into [Ready/Runnable] state and when TS allocates processor then tread again enter into [Running state]. But these two method got deprecated and never recommended to use in programming |
| 3.6- | If the running thread calls **[t.stop ()]** method then thread enters into dead state directly. But this method got deprecated and never recommended to use in programming |

**Session-12**

**Multithreading Enhancement [Part-1]**

**Thread Group**

Thread Group is nothing but the collection of group which can be grouped based on their functionality. A Thread group can have its sub group. ThreadGroup(C) is a java class, present in java.lang package and which it is direct child of Object class.

|  |  |
| --- | --- |
|  | In addition to thread, thread group can also contain, thread sub group. |

**Advantage**: The main advantage of maintaining thread in the form of thread group is we can perform common operation very easily based on their functionality.

|  |
| --- |
| **public** **class** TestStopMethodOfThread {  **public** **static** **void** main(String[] args) {  **System.*out*.println("---" + Thread.*currentThread*().getThreadGroup().getName());**  **System.*out*.println("main thread group parent -"+ Thread.*currentThread*().getThreadGroup().getParent().getName().getClass());**  **System.*out*.println("main thread group parent-" + Thread.*currentThread*().getThreadGroup().getParent().getClass());**  }  }  **Output**:  ---main  main thread group parent-class java.lang.String  main thread group parent-class java.lang.ThreadGroup |

**System Thread group**:

As we know that every class in java is the child of Object class either directly or indirectly. Similarly every thread in java is child of **System Thread group** either directly or indirectly. Main thread group is the direct child of System Thread group.

**Note**: Every thread in java belongs to some group. Main thread belongs to **Main Thread Group**; every thread group in java is the child group of system either directly or indirectly. Hence System Thread Group act as root for all thread groups in java.

|  |  |
| --- | --- |
|  | System group contains several system level threads like   * Finalizer Garbage-Collector (GC), * Reference Handler * Signal Dispatcher * Attach Listener   ThreadGroup(C) is a java class, present in java.lang package and which it is direct child class of Object class. |

**Creation of Thread Group:**

Constructors: **1- ThreadGroup g = new ThreadGroup (String TGName);**

|  |  |
| --- | --- |
| **public** **class** TestStopMethodOfThread {  **public** **static** **void** main(String[] args) {  ThreadGroup g = **new** ThreadGroup("First Group");  System.***out***.println("Newly Created Thread group Belongs to" + **g.getParent().getName()**);  }  } | Output:  Newly Created Thread group Belongs to-main |

It creates a new thread group with the specified group name. The parent of this new group is the thread group of current executing thread group (**main thread group**) so the newly created thread group also belongs to main thread group.

**2-ThreadGroup cg = new (ThreadGroup g, String name);**

|  |
| --- |
| **public** **class** TestStopMethodOfThread {  **public** **static** **void** main(String[] args) {    ThreadGroup g = **new** ThreadGroup("First Group");  System.***out***.println("Newly Created Thread group Belongs to-" + g.getParent().getName()); //main  ThreadGroup cg = **new** ThreadGroup(g, "Second group");  System.***out***.println("**Newly Created child Thread group Belongs to**-"+cg.getParent().getName());// First Group  System.***out***.println("Newly Created child Thread group name-" + cg.getName());// Second Group  } } |
| **Output**:  Newly Created Thread group Belongs to-**main**  Newly Created child Thread group Belongs to-**First Group**  Newly Created child Thread group name-**Second group** |

It creates a new thread group with the specified group name; the parent of this new thread group is specified parent group in the constructor.

So hierarchy of the thread group will look like

**System🡨main🡨First Group🡨Second Group** [System is the root TG, which contains main TG, which contains First TG, which contains Second TG]

**Some important method in the Thread Group(TG**

|  |  |  |
| --- | --- | --- |
| 1 | String getName() | Returns the name of ThreadGroup |
| 2 | Int getMaxPriority() | Returns the max priority of TG & Default max priority of TG is 10 like Thread priority |
| 3 | Void setMaxPriority(int p) | Set the max priority of TG |
| 4 | ThreadGroup getParent() | Returns the parent group of current thread |
| 5 | void list() | It print information about the thread grout to the consol |
| 6 | Int activeCount() | Returns the number of active threads present in the thread group |
| 7 | Int activeGroupCount() | Returns the number of active groups present in the current thread group |
| 8 | Int enumerate( Thread[] t) | To copy all active threads of this thread group into provided thread array, in this case sub thread group threads also will be considered |
| 9 | **Int enumerate( ThreadGroup[] t)** | To copy all active sub-thread groups into thread group array |
| 9- | boolean isDaemon() | To check whether the TG is daemon or not |
| 10 | void setDaemon(boolan d) | To change the daemon |
| 11 | void interrupt() | To interrupt all waiting or sleeping thread present in the TG |
| 12 | void destroy() | To destroy and its sub-thread groups |

Threads in the TG that have already higher priority won’t be affected but for newly added thread this max priority is applicable

|  |  |
| --- | --- |
| **package** com.threadconcept.thread.group;  **public** **class** ThreadGroupDemo {  **public** **static** **void** main(String[] args) {  ThreadGroup g1 = **new** ThreadGroup("tg");  System.***out***.println("Default MAX P of TG g1-" + g1.getMaxPriority());  Thread t1 = **new** Thread(g1, "Thread-1");  Thread t2 = **new** Thread(g1, "Thread-2");  g1.setMaxPriority(3);  Thread t3 = **new** Thread(g1, "Thread-3");  Thread t4 = **new** Thread(g1, "Thread-4");  System.***out***.println("FirstThread Before set Pri-" + t1.getPriority());  System.***out***.println("SconThread Before set Pri-" + t2.getPriority());  System.***out***.println("ThirdThread after set Pri-" + t3.getPriority());  System.***out***.println("FourthThread after set Pri-" + t4.getPriority());  }} | Output:  Default MAX P of TG g1-10  FirstThread Befor set Pri-5  SconThread Befor set Pri-5  ThirdThread after set Pri-3  FourthThread after set Pri-3  So as we have seen that the  -Default Max Priority of TG is -10  -Default Max Priority of Thread- 5  -Once we set Max Priority of TG-3  -Then new created thread default  priority is also becomes – 3 |

So the once we set Max priority of any TG then the newly created thread default priority will be equal to newly set TG priority only.

As here the set Max priority TG g1 is 3 so the default priority of any thread created in this group will be 3 only and if we try to set high priority for this new thread of group g1 then it will automatically set to TG priority only but we can set thread priority lower than TG priority

|  |
| --- |
| **package** com.threadconcept.thread.group;  **public** **class** ThreadGroupDemo {  **public** **static** **void** main(String[] args) {  ThreadGroup g1 = **new** ThreadGroup("tg");  System.***out***.println("Default MAX P of TG g1-" + g1.getMaxPriority());  Thread t1 = **new** Thread(g1, "Thread-1");  Thread t2 = **new** Thread(g1, "Thread-2");  g1.setMaxPriority(3);  Thread t3 = **new** Thread(g1, "Thread-3");  Thread t4 = **new** Thread(g1, "Thread-4");  t3.setPriority(8);// Trying to set higher than TG p  t4.setPriority(2);// Trying to set lower than TG p  System.***out***.println("FirstThread Before set Pri-" + t1.getPriority());  System.***out***.println("SconThread Before set Pri-" + t2.getPriority());  System.***out***.println("ThirdThread after set Pri-" + t3.getPriority());  System.***out***.println("FourthThread after set Pri-" + t4.getPriority());  }  }  **Output**:  Default MAX P of TG g1-10  FirstThread Before set Pri-5  SconThread Before set Pri-5  ThirdThread after set Pri-3 // Did not change  FourthThread after set Pri-2 |

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.thread.group;  **public** **class** MyGroupThread **extends** Thread {  **public** MyGroupThread(ThreadGroup g, String name) {  **super**(g, name);  }  @Override  **public** **void** run() {  System.***out***.println("Child Thread");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  }  } | **package** com.threadconcept.thread.group;  **public** **class** ThreadGroupDemo2 {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ThreadGroup pg = **new** ThreadGroup("ParentGroup");  ThreadGroup cg = **new** ThreadGroup(pg, "ChildGroup");  MyGroupThread t1 = **new** MyGroupThread(pg, "ChildThread-1");  MyGroupThread t2 = **new** MyGroupThread(pg, "ChildThread-2");  t1.start();  t2.start();  System.***out***.println("PG Active thread count-" + pg.activeCount());//2  System.***out***.println("PG Active thread group count-"+ pg.activeGroupCount()); // 1  pg.list();  Thread.*sleep*(5000);  System.***out***.println("-----After sleep of 5 sec----");  System.***out***.println("PG Active thread count-" + pg.activeCount());//0  System.***out***.println("PG Active thread group count-" + pg.activeGroupCount());//1  pg.list();  }  } | |
| Child Thread  Child Thread  PG Active thread count-2  PG Active thread group count-1  and Grp Name-ParentGroup  java.lang.ThreadGroup[name=ParentGroup,maxpri=10]  Thread[ChildThread-1,5,ParentGroup]  Thread[ChildThread-2,5,ParentGroup]  java.lang.ThreadGroup[name=ChildGroup,maxpri=10] | | -----After sleep of 5 sec----  PG Active thread count-0  PG Active thread group count-1  java.lang.ThreadGroup[name=ParentGroup,maxpri=10]  java.lang.ThreadGroup[name=ChildGroup,maxpri=10] |

So here we done as follows

1. Created one TG with the name of ParentGroup
2. Created another TG sub-group inside ParentGroup
3. Created two thread t1 and t2 inside ParentGroup with the name ChildThread-1 & ChildThread-2
4. Started threads t1 & t2
5. Get the number of active thread(running threads) -2
6. Get the number of active group -1
7. pg.list() provides information about the Thread Group(ParentGroup ), like [TG Name, MaxPri] [Number of active thread name, Its priority, and belonging TG Name] [Name of sub group and pri]
8. Put the main thread on sleep for 5 sec, Now by that time child thread will get time and will complete its execution and went into dead state after 5 sec again
9. Get the number of active thread = 0 because child thread already complete its execution and went into dead state
10. Get the number of active Group=1
11. pg.list() so as here thread has already reached into dead state so it will not print thread information but it will print information about active Thread Group and its sub-TC name, and pri.

So the hierarchy of TG and T

|  |  |
| --- | --- |
|  | System🡨Main🡨ParentGroup🡨Child Group  ParentGroup 🡨[ChildThread-1,ChildThread-2] |
| **public** MyGroupThread(ThreadGroup g, String name) {  **super**(g, name);  } | here we have used [**super**(g, name);] class constructor which in-turn called Thread Class constructer to associate the thread into mentioned group |
| **public** Thread(ThreadGroup group, String name) {  init(group, **null**, name, 0);  } | This constructor associate thread name (ChildThread-1) and to which TG pg it belongs to. |

WAP to display all active thread names belongs to system group and its child groups.

|  |
| --- |
| **package** com.threadconcept.thread.group;  **public** **class** DisplayAllSystemGPThread {  **public** **static** **void** main(String[] args) {  ThreadGroup system = Thread.*currentThread*().getThreadGroup().getParent();  **// Get the system TG reference using above line**  Thread[] t = **new** Thread[system.activeCount()];  **//system.activeCount() returns number of active thread in integer and hence will create a Thread[] array with same number like: Thread[] t= new Thread[4];**    system.enumerate(t);  **// It copies every active thread from the system thread group and its subgroups to given array [] t**  **// Now iterate using for each loop**  **for** (Thread t1 : t) {  System.***out***.println(t1.getName() + "--" + t1.isDaemon());  }  }  } |
| **Output:**  Reference Handler--true  Finalizer--true  Signal Dispatcher--true  Attach Listener--true  main--false |

**Session-12**

**Multithreading Enhancement [Part-1]**

**Java.util.concurrent package**

In the last discussion we have covered synchronized keyword and we have faced n-number of problem with synchronized like performance problem, if don’t use synchronized keyword properly then we may have deadlock problem.

The problems with traditional synchronized keyword:

1. We are not having any flexibility to try for a lock without waiting i.e. if a thread wants to get lock to complete its execution then thread has to wait for lock in waiting state and once TS allocates processor then only we get lock, so completely dependent on JVM, programmer cannot do anything instead of waiting.
2. There is no way to specify maximum waiting time for a thread to get lock so that thread will wait until getting the lock, which may create performance problems and which may cause deadlock.
3. If a thread releases a lock then which waiting thread will get lock we are not having any control on this, only TS or JVM decides to allocate the lock to the waiting thread.
4. There is no API to list out all waiting thread for a lock.
5. The synchronized keyword compulsory we have to use either at method level or inside the method and it is not possible to use across multiple methods.

To overcome these problem sun people introduced Java.util.concurrent.locks package in 1.5 versions.

It also provides several enhancements to the programmer to provide more control on concurrency.

**Lock (I) Interface:**

Lock object (t) is similar to implicit lock acquired by a thread to execute synchronized method or block. Lock implementations provide more extensive operation than traditional implicit locks.

**Important methods of Lock (I):**

|  |  |
| --- | --- |
| **void lock ()** | This method used to acquire a lock; if lock is already available then immediately current thread will get that lock. If lock is not already available then it will wait until getting the lock. It has exactly same behavior of traditional synchronized keyword. |
| **boolean tryLock()** | To acquire the lock without waiting, if the lock is available then thread acquires that lock and returns true, **if the lock is not available then this method returns false and can continue its execution without waiting**. In this case thread never is entered into waiting state.  **If(l.tryLock() // if lock is available then perform safe operation otherwise perform else**  Perform safe operation  **else**  Perform alternative operation |
| **boolean tryLock(long time, TimeUnit unit)**  **This is the tryLock with time period.** | If lock is available then the thread will get the lock and continue its execution. If the lock is not available then thread will wait until specified amount of time and still if the lock is not available then thread can continue its execution  TimeUnit: It is an enum present in java.util.concurrent.timeunit package  enum TimeUnit{  NANOSECOND,  MICROSECOND,  MILISECON,  SECOND,  MINUTS,  HOURS,  DAYS  }  Example: if (l.tryLock (1000, TimeUnit.HOURS)){…….} |
| **void lockInterruptibly()** | It acquires the lock it is available and returns immediately. If the lock is not available then it will wait and during waiting if the thread is interrupted then thread won’t get lock. |
| **void unlock()** | To releases a lock. To call this method compulsory current thread should be owner of the lock i.e. without locking first how you can unlock otherwise we will get runtime exception: IllegealMonitorStateException |

**So these are the five methods are present in the lock (I) interface**

**Session-12**

**Multithreading Enhancement [Part-2]**

**Java.util.concurrent.locks package**

Now let’s discuss about lock (I) implementation class: Now here will discuss about **java.util.concurrent.locks** package

**ReentrantLock(c): (it means get lock again and again)**

It is the implementation class of lock (I) interface and it is direct child of Object class.

Reentrant means a thread can acquire same lock multiple times without any issue. Internally Reentrant lock increments thread’s personal count whenever we call lock method, and decrement count value whenever thread calls unlock() and lock will be released whenever count reaches zero(0)

**Constructor:**

|  |  |
| --- | --- |
| Reentrant l = new Reentrant(); | Creates an instance of Reentrant lock. |
| Reentrant l = new Reentrant(boolean fairness) | It creates Reentrant lock with the given fairness policy. If the fairness is true then longest waiting thread can acquire the lock if it is available i.e. it follows First-Come-First-Serve policy. If fairness is false then which waiting thread will get chance we cannot expect.  The default value for fairness is false. |
| Reentrant l = new Reentrant();  Reentrant l = new Reentrant(true)  Reentrant l = new Reentrant(false)  All the above | **Which of the following declarations are equal**  So first and third are equal to each other |

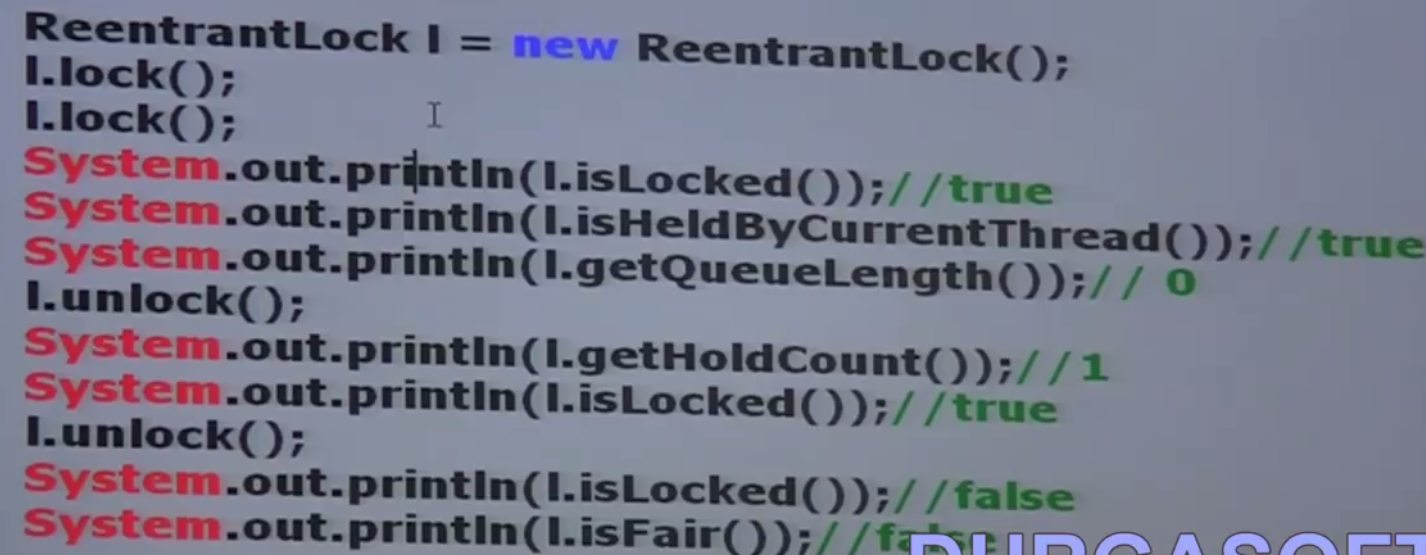
**Important methods of Reentrant lock:**

Since Reentrant class is the implementation of the Lock (I) interface so whatever method (five) is declared in the Lock- Interface would be implemented in Reentrant class also. So the Reentrant class methods are given as below. And apart from these five methods Reentrant has some other methods as given below

1. void lock()
2. boolean tryLock()
3. boolean tryLock(long I, TimeUnit t)
4. void lockInterruptibly()
5. void unlock()

**Other methods in Reentrant class:**

|  |  |
| --- | --- |
| **int getHoldCount()** | Returns number of holds on this lock by current thread |
| **boolean isHeldByCurrentThread()** | Returns true iff lock is hold by current thread |
| **Int getQueueLength()** | Returns number of threads waiting for the lock |
| **Collection getQueuedThreads()** | Returns a collection of threads which are waiting to get the lock |
| **boolean hasQueuedThreads()** | Returns True if any thread waiting to get the lock |
| **boolean isLocked()** | Returns true if the lock is acquired by some thread |
| **boolean isFair()** | Returns true if the fairness policy is set with the true value |
| **Thread getOwner()** | Returns the thread which acquirers the lock |

****

**Let’s WAP to use above method:**

|  |  |
| --- | --- |
| **package** com.threadconcept.reentarant;  **import** java.util.concurrent.locks.\*;  **public** **class** ReentrantLockDemo {  **public** **static** **void** main(String[] args) {    ReentrantLock lc = **new** ReentrantLock();  lc.lock();// First Lock  lc.lock();// Second Lock  System.***out***.println(lc.getHoldCount()); // 2  System.***out***.println(lc.isHeldByCurrentThread());// ture  System.***out***.println(lc.getQueueLength()); // 0  System.***out***.println(lc.hasQueuedThreads());// false  System.***out***.println(lc.isLocked()); // ture  System.***out***.println(lc.isFair()); // false  lc.unlock();  System.***out***.println(lc.getHoldCount());// 1  }  } | Output:  2  true  0  false  true  false  1 |

Example to show the difference of using synchronized keyword and ReentrantLock .

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.sync;  **public** **class** MySynchronizedMethod {  //synchronized  **public** **void** wish(String name) {  **for** (**int** i = 0; i < 3; i++) {  System.***out***.print("Good Morning:");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  }  System.***out***.println(name);  }  }  }  Created a **non** **synchronized** wish method and include sleep () for 2 sec. | **package** com.threadconcept.sync;  **public** **class** MySyncThread **extends** Thread {  MySynchronizedMethod ms;  String name;  **Public** MySyncThread (MySynchronizedMethod ms, String name) {  **super**();  **this**.ms = ms;  **this**.name = name;  }  @Override  **public** **void** run() {  ms.wish(name);  }  }  Created a thread class and in the run() called the wish() method | **package** com.threadconcept.sync;  **public** **class** TestSynchronization {  **public** **static** **void** main(String[] args) {  MySynchronizedMethod ms = **new** MySynchronizedMethod();  **MySyncThread t1 = new MySyncThread(ms, "Dhoni");**  **MySyncThread t2 = new MySyncThread(ms, "Yuraj");**  t1.start();  t2.start();  }}  **Ouput:**  Good Morning: Good Morning: **Dhoni**  Good Morning: **Dhoni**  Good Morning: **Yuraj**  Good Morning: **Dhoni**  Good Morning: **Yuraj**  Good Morning: **Dhoni** |

So without Synchronized method: we have got irregular output: Now let’s check with synchronized method:

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.sync;  **public** **class** MySynchronizedMethod {  //synchronized  **public** **synchronized void** wish(String name) {  **for** (**int** i = 0; i < 3; i++) {  System.***out***.print("Good Morning:");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  }  System.***out***.println(name);  }  }  }  Created a **non** **synchronized** wish method and include sleep () for 2 sec. | **package** com.threadconcept.sync;  **public** **class** MySyncThread **extends** Thread {  MySynchronizedMethod ms;  String name;  **Public** MySyncThread (MySynchronizedMethod ms, String name) {  **super**();  **this**.ms = ms;  **this**.name = name;  }  @Override  **public** **void** run() {  ms.wish(name);  }  }  Created a thread class and in the run() called the wish() method | **package** com.threadconcept.sync;  **public** **class** TestSynchronization {  **public** **static** **void** main(String[] args) {  MySynchronizedMethod ms = **new** MySynchronizedMethod();  **MySyncThread t1 = new MySyncThread(ms, "Dhoni");**  **MySyncThread t2 = new MySyncThread(ms, "Yuraj");**  t1.start();  t2.start();  }  }  Here we have created two child threads and each calls Child class run() method which in turn calls non synchronized method wish(). Since here two thread are trying to call one non synchronized method simultaneously so let’s see the output |

Output:

|  |  |
| --- | --- |
| Good Morning: **Dhoni**  Good Morning: **Dhoni**  Good Morning: **Dhoni**  Good Morning: **Yuraj**  Good Morning: **Yuraj**  Good Morning: **Yuraj** | Good Morning: **Yuraj**  Good Morning: **Yuraj**  Good Morning: **Yuraj**  Good Morning: **Dhoni**  Good Morning: **Dhoni**  Good Morning: **Dhoni** |

So we have got regular reault

Now same example by using Reentrant class of java.util.concurrent.lock package:

|  |  |  |
| --- | --- | --- |
| **package** com.threadconcept.sync;  **import java.util.concurrent.locks.\*;**  **public** **class** MySynchronizedMethod {  **ReentrantLock l = new ReentrantLock();**  //synchronized  **public** **void** wish(String name) {  **l.lock()-----🡪 line1**  **for** (**int** i = 0; i < 3; i++) {  System.***out***.print("Good Morning:");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  }  System.***out***.println(name);  }  **l.unlock() ----🡪line2**  }  }  Created a **non** **synchronized** wish method and include sleep () for 2 sec. | **package** com.threadconcept.sync;  **public** **class** MySyncThread **extends** Thread {  MySynchronizedMethod ms;  String name;  **Public** MySyncThread (MySynchronizedMethod ms, String name) {  **super**();  **this**.ms = ms;  **this**.name = name;  }  @Override  **public** **void** run() {  ms.wish(name);  }  }  Created a thread class and in the run() called the wish() method | **package** com.threadconcept.sync;  **public** **class** TestSynchronization {  **public** **static** **void** main(String[] args) {  MySynchronizedMethod ms = **new** MySynchronizedMethod();  **MySyncThread t1 = new MySyncThread(ms, "Dhoni");**  **MySyncThread t2 = new MySyncThread(ms, "Yuraj");**  t1.start();  t2.start();  }  }  Here we have created two child threads and each calls Child class run() method which in turn calls non synchronized method wish(). Since here two thread are trying to call one non synchronized method simultaneously so let’s see the output |

So three line of code we have included in the below program

1. **import** java.util.concurrent.locks.\*;
2. **Created object as ReentrantLock l = new ReentrantLock();**
3. **l.lock();[ put lock from which point the thread should lock the code**
4. **l.unlock();[ finally unlock the from where threads are required to get lock]**

**output**

|  |  |
| --- | --- |
| Good Morning: **Dhoni**  Good Morning: **Dhoni**  Good Morning: **Dhoni**  Good Morning: **Yuraj**  Good Morning: **Yuraj**  Good Morning: **Yuraj** | Good Morning: **Yuraj**  Good Morning: **Yuraj**  Good Morning: **Yuraj**  Good Morning: **Dhoni**  Good Morning: **Dhoni**  Good Morning: **Dhoni** |

**So without using synchronized keyword we have achieve synchronization in a very easy way just by using l.lock () & l.unlock () method in the proper position.**

If we comment line-2 and line-2 then the threads will be executed simultaneously and we will get irregular output.

If we are not commenting line1 & 2 then the threads will be executed one by one and we will get regular output.

The above program is just replacement for synchronized keyword but still so many things (like waiting thread can get lock immediately) is remains to see.

**Demo program for tryLock () method:**

|  |
| --- |
| **package** com.threadconcept.reentarant;  **import** java.util.concurrent.locks.\*;  **public** **class** MyReentrantThread **extends** Thread {  **static** ReentrantLock *l* = **new** ReentrantLock();// Created static so that only one copy will be available  **public** MyReentrantThread(String name) {  **super**(name);// Allocates a new Thread object with the name of the new thread.  }  @Override  **public** **void** run() {  **if** (*l*.tryLock()) {  System.***out***.println(Thread.*currentThread*().getName() + "-Got lock and performing safe operation");  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  *l*.unlock();  } **else** {  System.***out***.println(Thread.*currentThread*().getName() + "-unable to get lock and performing alternative operation");  }  }  } |

|  |
| --- |
| **package** com.threadconcept.reentarant;  **public** **class** TryLockDemo {  **public** **static** **void** main(String[] args) {  MyReentrantThread t1 = **new** MyReentrantThread("FirstThread");  MyReentrantThread t2 = **new** MyReentrantThread("SecondThread");  t1.start();  t2.start();  }  } |

**Output**

|  |  |
| --- | --- |
| FirstThread-Got lock and performing safe operation  SecondThread-unable to get lock and performing alternative operation | SecondThread-unable to get lock and performing alternative operation  FirstThread-Got lock and performing safe operation |

So here we can see that the thread which gets the lock first, executes if block and other thread does not wait to get the lock and simultaneously execute else block. Sometime FirstThread will get first chance to get lock and sometime SecondThread get the lock first, so we have got two possible output as shown above.

tryLock with TimeUnit:

|  |  |
| --- | --- |
| **package** com.threadconcept.reentarant;  **import** java.util.concurrent.locks.\*;  **import** java.util.concurrent.\*;  **public** **class** TryLockWithTimeUnit **extends** Thread {  **static** ReentrantLock *l* = **new** ReentrantLock();  **public** TryLockWithTimeUnit(String name) {  **super**(name);  }  @Override  **public** **void** run() {  **do** {  **try** {  **if** (*l*.tryLock(5000, TimeUnit.***MILLISECONDS***)) {  System.***out***.println(Thread.*currentThread*().getName()+ "-Got lock and going to sleep for 30-Sec and then unlock");  Thread.*sleep*(30000);  *l*.unlock();  System.***out***.println(Thread.*currentThread*().getName() + "-Released lock after 30-Sec");  **break**;  } **else** {  System.***out***.println(Thread.*currentThread*().getName() + "-Unable to get lock and trying after each 5 sec");  }  } **catch** (InterruptedException e) {  e.printStackTrace();  }  } **while** (**true**);  }  } | |
| **package** com.threadconcept.reentarant;  **public** **class** TestTryLockDemo {  **public** **static** **void** main(String[] args) {  TryLockWithTimeUnit t1 = **new** TryLockWithTimeUnit("FirstThread");  TryLockWithTimeUnit t2 = **new** TryLockWithTimeUnit("SecondThread");  t1.start();  t2.start();  }  } | |
| SecondThread-Got lock and going to sleep for 30-Sec and then unlock  FirstThread-Unable to get lock and trying after each 5 sec  FirstThread-Unable to get lock and trying after each 5 sec  FirstThread-Unable to get lock and trying after each 5 sec  FirstThread-Unable to get lock and trying after each 5 sec  FirstThread-Unable to get lock and trying after each 5 sec  SecondThread-Released lock after 30-Sec  FirstThread-Got lock and going to sleep for 30-Sec and then unlock  FirstThread-Released lock after 30-Sec | FirstThread-Got lock and going to sleep for 30-Sec and then unlock  SecondThread-Unable to get lock and trying after each 5 sec  SecondThread-Unable to get lock and trying after each 5 sec  SecondThread-Unable to get lock and trying after each 5 sec  SecondThread-Unable to get lock and trying after each 5 sec  SecondThread-Unable to get lock and trying after each 5 sec  FirstThread-Released lock after 30-Sec  SecondThread-Got lock and going to sleep for 30-Sec and then unlock  SecondThread-Released lock after 30-Sec |

So When FirstThread get the lock in first chance then goes to sleep for 30 sec and at the same time SecondThread keep trying to get lock as per the defined TimeUnit(after each 5 sec). Once the FirstThread wakeup then release the lock and complete its execution. Now SecondThread gets the lock and sleep for 30 second and releases the lock after 30 second.

Note: Sometime FirstThread will get first lock and Sometime SecondThread will get the lock first and hence we have got two possible outputs as shown above.

This concept is more better than traditional Multithreading concept like (synchronized keyword) as in the traditional threading concept there is a chance to get deadlock if synchronized method or block have not used carefully and apart from that if one thread get the lock first then another thread hast to wait until get the lock.

But such type of problem is there in this Threading enhancement concept and we will never get deadlock situation and thread are not required to wait until gets lock to continue its execution.

**Session-12**

**Multithreading Enhancement [Part-2] Java Threads Pools (Executer framework)**

**We have used the connection pool in database concept. What is the need of connection pool in DB?**

Ans: To improve performance, because if don’t’ use Connection pools for DB then it may create performance and memory problem because on every request every time, create an object for database connection. So instead of creating object every time for establishing DB-connection, a pool of connection object will created at the bingeing only. Now whenever we required getting a new connection object then we request to connection pool and use that object in communicating with database and once communication with DB gets completed then connection pool take back that connection objet into its pool so that same connection object can be used multiple times and hence performance by-default will be improved and memory also will be saved.

Similarly same thing happens in case of thread also, so every time for every independent job creates a separate thread, which executes its job and finally gets terminated automatically once job gets executed successfully. So if we have 10 independent job is there, then 10 threads required to create which executes job separately and then will be terminated. So here, we will get performance and memory problem. So now instead of creating 10 independent threads we will create a pool of 5 threads and submit all the 10 jobs to thread pool and now let’s say first 5 jobs gets executed by 5 threads and after completing 5 job same thread will complete the remaining 5 jobs. Hence a single thread can perform multiple independent jobs so the performance of the system by-default will be improved.

So thread pool means, create a bunch of threads in the pool and reuse the same threads for multiple independent jobs.

**Q: How to create Thread Pool? How we can submit jobs to the thread pool?**

Creating new threads for every job may create performance and memory problems. To overcome this we should go for thread-pool. Thread Pool is a pool of already created threads ready to do our job.

Java 1.5V introduces Thread Pool framework to implement Thread Pool. TP framework also known as executer framework.

Creating Thread Pool

|  |  |
| --- | --- |
| **ExecuterService** **service** = Executors.**newFixedThreadPool**(3); | **So here three threads are created into service Thread-Pool** |

**Q: Now after creating TP having three threads, we have Runnable jobs which should be executed by these three threads, so how we can submit Runnable jobs to the Executer Framework?**

Ans: We can submit Runnable jobs by using submit () method as: [**service.submit(job)]**

**Q: Once the job gets executed by Executer framework then we need to shutdown the executer framework. How to shutdown?**

**Ans:** We can shutdown the executer framework service by using shutdown () method as: [**service.shutdown()**]

Demo:

|  |
| --- |
| **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **public** **class** ThreadPoolExecuterDemo {  **public** **static** **void** main(String[] args) {  PrintJobThreadPool[] jobs = { **new** PrintJobThreadPool("Arun"),  **new** PrintJobThreadPool("Tarun"),  **new** PrintJobThreadPool("Bunty"),  **new** PrintJobThreadPool("Junty"),  **new** PrintJobThreadPool("Atharv"),  **new** PrintJobThreadPool("Radhika"), };  ExecutorService service = Executors.*newFixedThreadPool*(3);  **for** (PrintJobThreadPool job : jobs) {  service.submit(job);  }  service.shutdown();  }  } |

**So here we have created 6 jobs and 3 threads in Thread Pool using Executer Framework and submit all the 6 jobs in the pool and once every job executed successfully then shutdown the executer framework service.**

|  |  |
| --- | --- |
| **package** com.threadconcept.thread.pool;  **public** **class** PrintJobThreadPool **implements** Runnable {  String name;  **public** PrintJobThreadPool(String name) {  **this**.name = name;  }  @Override  **public** **void** run() {  System.***out***.println(name + "- Job started by-" + Thread.*currentThread*().getName());  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  System.***out***.println(name + "- Job completed by-" + Thread.*currentThread*().getName());  }  } | |
| Output:  Arun- Job started by-pool-1-thread-1  Arun- Job completed by-pool-1-thread-1  Atharv- Job started by-pool-1-thread-1  Atharv- Job completed by-pool-1-thread-1  Tarun- Job started by-pool-1-thread-2  Tarun- Job completed by-pool-1-thread-2 | Junty- Job started by-pool-1-thread-2  Junty- Job completed by-pool-1-thread-2  Bunty- Job started by-pool-1-thread-3  Bunty- Job completed by-pool-1-thread-3  Radhika- Job started by-pool-1-thread-3  Radhika- Job completed by-pool-1-thread-3 |

**So here in the above program 3- threads are responsible to execute 6 jobs so that a single thread can be reused for multiple jobs.**

**Q: Where Thread Pool concept is applicable or required?**

**A:** Let’s say we have one web application, and multiple requests are coming to this Web Application. So for every request a new thread will be allocated by web-server for processing the request. So in every web-application or web-server a pool of threads is there and whenever a request come then a thread from the pool is allocated to that request or job and after completing request processing, same thread will come back to the pool.

The default thread pool size for every web server/web-application is- 60. So while designing or developing web-server or web-application, internally Thread Pool concept is used.

Callable- Interface:

When after completing the job, thread has to return something to the caller (t.call) then we should go for Callable (I) interface instead of Runnable (I) interface. This Callable interface contains one method [**public Object call ()**] whose return type is object.

|  |  |
| --- | --- |
| **package** com.threadconcept;  **import** java.util.concurrent.Callable;  **public** **class** MyCallable **implements** Callable<Object> {  @Override  **public** Object call() **throws** Exception {  **return** "Callable returned- Object";  }  } | **package** com.threadconcept;  **public** **class** TestRunnableDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  MyCallable c = **new** MyCallable();  Object r = c.call();  // String call = c.call();  System.***out***.println(r.toString());  System.***out***.println("Main Thread");  }  } |

Callable & Future: In the case of Runnable job, thread won’t return anything after completing job. If a thread is required to return some result after execution then we should go for callable. Callable interface contain only one method

[public Object call () throws Exception]

If we submit a callable object to execute then after completing the job thread returns an object of the type Future i.e. Future object can be used to retrieve the result from callable job.

In case of Runnable Interface since run() does not return anything so nothing is required hold the result from Runnable job, while in case of callable interface call() return the value so to hold the returned value from service executer we need Future.

Q: WAP for service Executer framework using callable interface instead of using Runnable

|  |
| --- |
| **package** com.threadconcept.thread.pool;  **import** java.util.concurrent.Callable;  **public** **class** ThreadPoolWithCallable **implements** Callable<Object> {    **int** num;  **public** ThreadPoolWithCallable(**int** num) {  **this**.num = num;  }  @Override  **public** Object call() **throws** Exception {  System.***out***.println(Thread.*currentThread*().getName() + "- is doing addition of first" + num + "numbers");  **int** sum = 0;  **for** (**int** i = 1; i < num; i++) {  sum = sum + i;  }  **return** sum;  }  } |
| **package** com.threadconcept.thread.pool;  **import** java.util.concurrent.\*;  **public** **class** ThreadPoolCallableDemo {  **public** **static** **void** main(String[] args) **throws** Exception {  ThreadPoolWithCallable[] jobs = {    **new** ThreadPoolWithCallable(10), **new** ThreadPoolWithCallable(20),  **new** ThreadPoolWithCallable(30), **new** ThreadPoolWithCallable(40),  **new** ThreadPoolWithCallable(50), **new** ThreadPoolWithCallable(60),  };  ExecutorService service = Executors.*newFixedThreadPool*(3);  **for** (ThreadPoolWithCallable job : jobs) {  Future<Object> f = service.submit(job);  System.***out***.println(f.get());  }  service.shutdown();  }  }  Output:  pool-1-thread-1- is doing addition of first10numbers  45  pool-1-thread-2- is doing addition of first20numbers  190  pool-1-thread-3- is doing addition of first30numbers  435  pool-1-thread-1- is doing addition of first40numbers  780  pool-1-thread-2- is doing addition of first50numbers  1225  pool-1-thread-3- is doing addition of first60numbers  1770 |
| [--> [Open Declaration](eclipse-open:%E2%98%82=ThreadingConcept/C:\/Program%20Files\/Java\/jdk1.8.0_211\/jre\/lib\/rt.jar%3cjava.util.concurrent(Future.class%E2%98%83Future)](eclipse-open:%E2%98%82=ThreadingConcept/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util.concurrent(Future.class%E2%98%83Future)[java](eclipse-javadoc:%E2%98%82=ThreadingConcept/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava).[util](eclipse-javadoc:%E2%98%82=ThreadingConcept/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util).[concurrent](eclipse-javadoc:%E2%98%82=ThreadingConcept/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util.concurrent).Future<[Object](eclipse-javadoc:%E2%98%82=ThreadingConcept/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util.concurrent(Future.class%E2%98%83Future%E2%98%82java.lang.Object)> **A Future represents the result of an asynchronous computation**. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled. If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.  **Sample Usage** (Note that the following classes are all made-up.)    interface ArchiveSearcher { String search(String target); }  class App {  ExecutorService executor = ...  ArchiveSearcher searcher = ...  void showSearch(final String target)  throws InterruptedException {  Future future  = executor.submit(new Callable() {  public String call() {  return searcher.search(target);  }});  displayOtherThings(); // do other things while searching  try {  displayText(future.get()); // use future  } catch (ExecutionException ex) { cleanup(); return; }  }  }}  The [FutureTask](eclipse-javadoc:%E2%98%82=ThreadingConcept/C:%5C/Program%20Files%5C/Java%5C/jdk1.8.0_211%5C/jre%5C/lib%5C/rt.jar%3Cjava.util.concurrent(Future.class%E2%98%83Future%E2%98%82FutureTask) class is an implementation of Future that implements Runnable, and so may be executed by an Executor. For example, the above construction with submit could be replaced by:  FutureTask<String> future =  new FutureTask<String>(new Callable<String>() {  public String call() {  return searcher.search(target);  }});  executor.execute(future);}  Memory consistency effects: Actions taken by the asynchronous computation [*happen-before*](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility) actions following the corresponding Future.get() in another thread. |

**Difference between Runnable & Callable Interface:**

|  |  |
| --- | --- |
| **Runnable** | **Callable** |
| If a thread is not required to return anything after completing job then we should go for Runnable | If a thread is required to return something then we should go for Callable |
| Runnable (I) contains only one method run() | Callable (I) contains only one method call() |
| The return type is void because it does not return anything | The return type is Object because it return something |
| Inside run() if there is any chance to rise checked exception then we can handle by using try-catch block because we cannot use throws keyword for run() | But here inside call() method if checked exception is raised then not required to handle using try-catch block we can handle just by using **throws** Exception because call() already throws Exception |
| Runnable (I) present in java.lang package & Introduced in 1.0 version | Callable (I) present in java.util.concurrent package & Introduced in 1.5 version |

Note: If parent class does not throw any checked exception then child class cannot throw checked exception and in that case we can handle exception within the try-catch block while checked exception can be handled just by using [throws Exception]

**Session-12**

**Multithreading Enhancement [Part-2]-java thread local**

What is the meaning of local person?

As we know that we have some attributes in the Servlet that we used to set in the request (request.setAttribute (“Arun)) and this attribute is available during the request process only and once request process complete the attribute also automatically gone. Similarly the attribute which added in the session scope by default available throughout the session until we perform logout. For example when we entering email id and password in the Gmail once we enter into the Gmail we can perform any operation like sending email getting email. Similarly the data which is stored in application context by default will be available throughout application.

Similarly let’s we have some data and this data is required to thread then we will add that attribute for this thread wherever this thread is executing happily we can get the data from that thread attribute and for this process thread local can be used to define thread scope. I.e. wherever this thread will execute this local thread attribute value will be available. Thread local class provide thread local variables, Thread local class maintains values per thread bases.

Each thread local object maintains a separate value like userId, transactionId etc for each thread that access that object (thread local object). Thread can access its local value, manipulate its value and even can remove its value.

In every part of the code which is executed by the thread, we can access its local variable.

Example: Consider a Servlet which invokes some business methods, we have a requirement to generate a unique transactionId for each and every request and we have to pass this transactionId to the business method. For this requirement we can use thread local to maintain a separate transactionId for every request i.e. for every thread.

Note:

1. Thread local class introduced in 1.2 versions and enhanced in 1.5 versions.
2. Thread local can be associated with thread scope
3. Total code which is executed by the thread has access to the corresponding thread local variable
4. A thread can access its own local variables and can’t access other thread’s local variables.
5. Once thread entered into dead states, all its local variable are by default eligible for garbage collection

How to create Thread local variable (Constructor)

|  |  |
| --- | --- |
| ThreadLocal tl = new ThreadLocal() | Creates a thread local variable |
| Methods: Object get() | Returns the value of thread local variable associated with current thread |
| Object initialValue() | Returns initial value of thread local variable associated with current thread. The default implementation of this method returns null. To customize our own initial value we have to override this method |
| void set(Object newValue) | To set a new value |
| Void remove() | To remove the value of thread local variable associated with current thread. After removal if we are trying to access then it will reinitialized once again by invoking its initialValue() method |

Example:

|  |  |
| --- | --- |
| **package** com.threadconcept.thread.pool;  **public** **class** ThreadLocalDemo {  **public** **static** **void** main(String[] args) {  ThreadLocal tl = **new** ThreadLocal();  System.***out***.println(tl.get());// null  tl.set("ThreadLocalValueSet");  System.***out***.println(tl.get());  tl.remove();  System.***out***.println(tl.get()); // null  }  }  Output:  null  ThreadLocalValueSet  Null  Here we have not set any initivalue so the default get local thread value for main thread is – null | Overriding of initialValue() method:  **public** **class** ThreadLocalDemo {  **public** **static** **void** main(String[] args) {  ThreadLocal tl = **new** ThreadLocal() {  **public** Object initialValue() {  **return** "InitialValue-Arun";  }  };  System.***out***.println(tl.get());  tl.set("ThreadLocalValueSet");  System.***out***.println(tl.get());  tl.remove();  System.***out***.println(tl.get());  }  }  Created on anonymous inner class and override initialValue() method and provided the initial value InitialValue-Arun for the main thread. |
| **Output: after overriding initialValue() method in the anonymous class**  InitialValue-Arun  ThreadLocalValueSet  InitialValue-Arun |  |

So in the previous example we have seen, how to create ThreadLocal, how to get value, how to set value and how to override initial default value.

Now we will create multiple threads and will check it is possible for every thread to maintain a separate value or not.

|  |  |
| --- | --- |
| **package** com.threadconcept.thread.pool;  **public** **class** CustomerThreadLocal **extends** Thread {  **static** Integer *custId* = 0;  **private** **static** ThreadLocal *tl* = **new** ThreadLocal() {  **protected** Integer initialValue() {  **return** ++*custId*;  }  };  **public** CustomerThreadLocal(String name) {  **super**(name);  }  @Override  **public** **void** run() {  System.***out***.println(Thread.*currentThread*().getName() + "-executing with customer id-" + *tl*.get());  }  } | |
| **package** com.threadconcept.thread.pool;  **public** **class** ThreadLocalDemo {  **public** **static** **void** main(String[] args) {  CustomerThreadLocal t1 = **new** CustomerThreadLocal("Customer-Thread-1");  CustomerThreadLocal t2 = **new** CustomerThreadLocal("Customer-Thread-2");  CustomerThreadLocal t3 = **new** CustomerThreadLocal("Customer-Thread-3");  CustomerThreadLocal t4 = **new** CustomerThreadLocal("Customer-Thread-4");  t1.start();  t2.start();  t3.start();  t4.start();  }  } | |
| **Output:**  Customer-Thread-2-executing with customer id-1  Customer-Thread-3-executing with customer id-2  Customer-Thread-4-executing with customer id-3  Customer-Thread-1-executing with customer id-4 | **In the above program for every customer thread a separate customer id will be maintained by ThreadLocal object** |

ThreadLocal behavior with respect to inheritance:

ThreadLocal Vs Inheritance:

Parent Thread’s thread local variable by default not available to the child thread. If we want to make Parent Thread’s thread local variable value available to the child thread then we should go for InheritableThreadLocal class.

By default Child Thread’s value is exactly same as Parent Thread’s value, but we can provided customized value for Child Thread by overriding childValue() method of InheritableThreadLocal class.

**Constructor:**

InheritableThreadLocal tl = new InheritableThreadLocal ()

**Methods:**

InheritableThreadLocal is the child class of ThreadLocal class and hence all method present in ThreadLocal class by default available to InheritableThreadLocal class. In addition to these methods it contains only method.

[**Public Object childValue (Object parentValue)**]

Example to mention:

|  |
| --- |
| **package** com.threadconcept.thread.local;  **public** **class** ParentThreadLocal **extends** Thread {  **public** **static** InheritableThreadLocal *tl* = **new** InheritableThreadLocal() {  **public** Object childValue(Object p) {  **return** "CC";  }  };  @Override  **public** **void** run() {  *tl*.set("SetByParentThreadLocal");  System.***out***.println("ParentThreadValue- " + *tl*.get());  ChildThreadLocal t = **new** ChildThreadLocal();  t.start();  }  } |
| **package** com.threadconcept.thread.local;  **public** **class** ChildThreadLocal **extends** Thread {  @Override  **public** **void** run() {  System.***out***.println("Parent Thread Value-" + ParentThreadLocal.*tl*.get());  }  } |
| **package** com.threadconcept.thread.local;  **public** **class** InheritableThreadLocalDemo {  **public** **static** **void** main(String[] args) {  ParentThreadLocal pt = **new** ParentThreadLocal();  pt.start();  }  } |
| **Output:**  ParentThreadValue- SetByParentThreadLocal  Parent Thread Value-CC |

**Explanation**:

1. Created anonymous class which extend **InheritableThreadLocal** class and here we are override its childValue (Object p) and set the ThreadLocal value for child thread created by Parent thread.