MultiThreading

Module 7: Multi – Threading	
1.	MultiTasking, Multithreading
2.	Way to define a thread ➢ Extending Thread class ➢ Implementing Runnable interface
3.	Thread Name getter, setter
4.	Thread Priorities
5.	Methods to prevent Thread execution ➢ Yield() ➢ Join() ➢ Sleep()
6.	Synchronization
7.	InterThread communication
8.	Deadlock
9.	Deamon Threads
10.	Stop, Resume & Suspend thread
11.	Multithreading model
12.	Final lifecycle of Thread
13.	ThreadGroup
14.	Concurrent package: Lock, ReentrantLock
15.	Thread Pools (Executor Framework)
16.	Callable i/f & Future i/f
17.	ThreadLocal, InheritableThreadLocal

1. Multitasking, Multithreading

- **Multitasking** – Executing several tasks simultaneously is the concept of Multitasking.
- There are 2 types of Multitasking:
  a) Process – based Multitasking
  b) Thread – based Multitasking

a) Process – based Multitasking

- Executing several tasks simultaneously where each task is a separate independent Program/Process is called Process – based Multitasking.
- E.g., While typing a Java Program in the editor, we can listen audio songs from the same system & download a file from internet at the same time. All these tasks will be executed simultaneously & independent of one another. Hence it is **Process – based Multitasking.**
- It is best suitable at OS level.

b) Thread – based Multitasking

- Executing several tasks simultaneously where each task is a separate independent part of the same Program, is called Thread – based Multitasking.
- Each independent part is called a Thread.
- Thread – based Multitasking is best suitable at Programmatic level.

Whether its process – based Multitasking or Thread – based multitasking, the main objective of Multitasking is to reduce response time of the system & to improve performance.

- The main application areas of Multithreading are:
  - To develop multimedia graphics, animations, video games
  - To develop web servers & Application servers etc.

- When compared to old languages, developing Multithreaded application in Java is easy because Java provides inbuilt support for Multithreading with rich API (Thread, Runnable, ThreadGroup etc.)

2. Ways to define a Thread

- A Thread is a flow of execution & for every Thread, a separate independent Job is there.
- We can define a Thread in 2 ways
  a) By extending Thread class
  b) By implementing Runnable interface

a) By extending Thread class

```
// Define a Thread
Class MyThread extends Thread {                // Override run() method present in Thread class
   public void run () {                        // Job of Thread
         for (int i = 0; i < 10; i++) {        // Executed by Child Thread
             SoPln ("Child Thread");
         }
   }
}

class ThreadDemo {
  public static void main (String [] args) {
       MyThread t = new MyThread ();           // Thread instantiation
       t.start ();                             // Starting of child Thread
       for (int i = 0; i < 10; i++) {          // Executed by main Thread
           SoPln ("Child Thread");
       }
  }
}
```

Note: main () method & main thread are different. main () method is executed by main Thread.

Case 1: Thread Scheduler (Part of JVM)

- It is responsible to schedule Threads i.e., if multiple threads are waiting to get the chance of execution, then in which order threads will be executed is decided by Thread Scheduler.
- We can't expect exact algorithm followed by Thread Scheduler. It is varied from JVM to JVM. Hence, we can't expect Thread execution order & the exact output but we can provide several possible o/ps.

Case 2: Diff b/w t.start () & t.run ()

- In case of t.start (), a new Thread will be created which is responsible for the execution of run () method
- But in the case of t.run (), a new Thread won't be created & run () method will be executed just like a normal method call by main thread.

Case 3: Importance of Thread class start () method

- Thread class start () method is responsible to register the thread with Thread scheduler & all other mandatory activities. Hence, without executing Thread class start () method, there is no chance of starting a new Thread in Java. Due to this, Thread class start () method is considered as Heart of Multithreading.

```
start () {
    1.   Register this thread with Thread scheduler
    2.   Perform all other mandatory activities
    3.   Invoke run ();
}
```

Case 4: Overloading of run () method

- Overloading of run () method is always possible but Thread class start () method will invoke no – args run () method only.
- The other overloaded method, we have to call explicitly like a normal method call.

Case 5: If we're not overriding run () method

- If we're not overriding run () method then Thread class run method will be executed which has empty implementation, hence we won't get any output.
- It's highly recommended to override run () method otherwise don't go for multithreading concept.

Case 6: Overriding of start () method

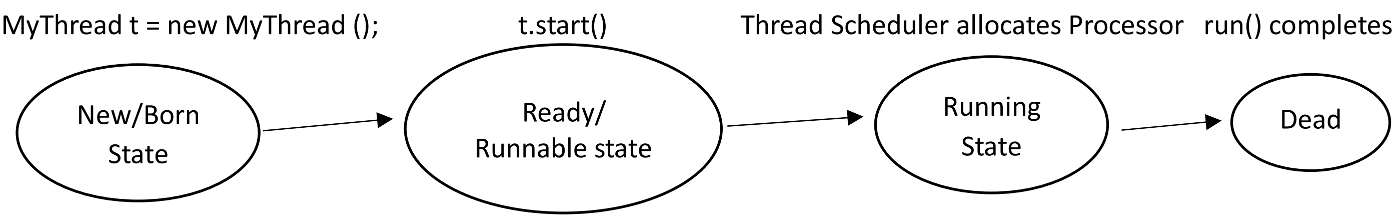
- If we override start () method then our start () method will be executed just like a normal method call & new thread won't be created.
- It's not recommended to override start () method otherwise don't go for multithreading concept

Case 7: Overriding of start () method but calling super () at first line then new thread will be created & run () method implementation will be executed by new thread & rest of start () method code will be executed by main thread only.

```
// Define a Thread
Class MyThread extends Thread {
  Public void start () {                    // Override start() method present in Thread class
      super.start();                        // new thread will be created here
      SoPln("start method");                // Executed by main thread
  }

   public void run () {
      for (int i = 0; i < 10; i++) {        // Executed by Child Thread
          SoPln ("Child Thread");
      }
   }
}

class ThreadDemo {
  public static void main (String [] args) {
      MyThread t = new MyThread ();         // Thread instantiation
      t.start ();                           // Starting of child Thread
      SoPln ("main method");                // Executed by main thread
  }
}
```

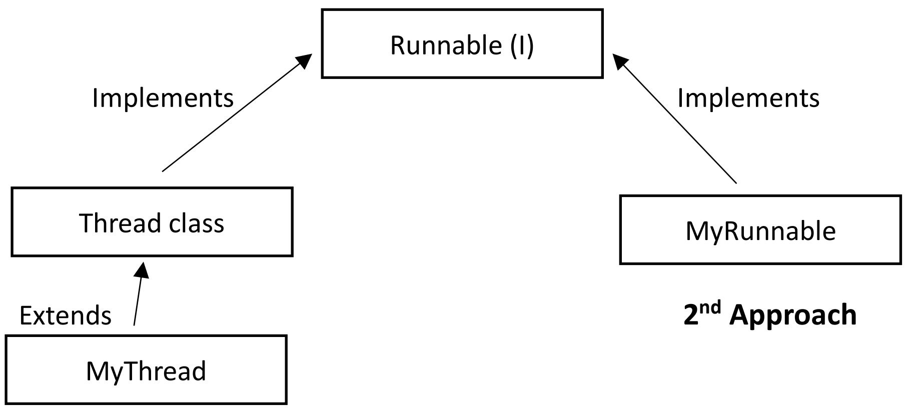
Case 8: After starting a thread, if we're trying to restart the same thread then we will get exception saying RE: IllegalThreadStateException.

Lifecycle of Thread



b) Defining a Thread by implementing Runnable (I)

- Runnable i/f is present in java.lang package & it contains only 1 method i.e., **public void run()**



2nd Approach

1st Approach

```
// Define a Thread by implementing Runnable i/f
Class MyRunnable implements Runnable {
    public void run () {
        for (int i = 0; i < 10; i++) {          // Executed by Child Thread
            SoPln ("Child Thread");
        }
    }
}

class ThreadDemo {
  public static void main (String [] args) {
      MyRunnable r = new MyRunnable ();
      MyThread t = new MyThread (r);       // Target Runnable
      t.start ();                         // Starting of child Thread
      SoPln ("main method");              // Executed by main thread
  }
}
```

Case Study:

```
Thread t1 = new Thread ();
MyRunnable r = new MyRunnable ();
Thread t2 = new Thread (r);
```

t1.start ()	A new Thread will be created for the execution of Thread class run () method, which has empty implementation.
t1.run ()	No new thread will be created & Thread class run () method will be executed just like a normal method call.
t2.start ()	A new Thread will be created for the execution of MyRunnable class run () method.
t2.run ()	No new Thread will be created & MyRunnable run () method will be executed just like a normal method call.
r.start ()	We will get CE: MyRunnable class doesn't have start capability
r.run ()	No new Thread will be created & MyRunnable run () method will be executed like a normal method call.

Q. Which approach is best to define a Thread
Ans: Among 2 ways of defining a Thread, Implements Runnable approach is recommended.

- In the 1st approach, our class always extends Thread class, there is no chance of extending any other class, hence we're missing Inheritance benefits.
- In the 2nd approach, while implementing Runnable i/f, we can extend any other class, hence we won't miss any Inheritance benefits.

Thread class Constructors

1. Thread t = new Thread ();
2. Thread t = new Thread (Runnable r);
3. Thread t = new Thread (String name);
4. Thread t = new Thread (Runnable r, String name);
5. Thread t = new Thread (ThreadGroup g, String name);
6. Thread t = new Thread (ThreadGroup g, Runnable r);
7. Thread t = new Thread (ThreadGroup g, Runnable r, String name);
8. Thread t = new Thread (ThreadGroup g, Runnable r, String name, long stackSize);

3. Thread Name Getter & Setter

- Every thread in Java has some name. It may have default name generated by JVM or customized name provided by Programmer.
- We can get & set Thread name by using following 2 methods of Thread class:
  a) **public final String getName ();**
  b) **public final void setName (String name);**

- We can get current executing Thread object by using **Thread.currentThread** () method (currentThread () is a static method present in Thread class)

```
class MyThread extends Thread {
    public void run () {
        SoPln (“run method executed by Thread: “ + Thread.currentThread().getName());
    }
}

Class Test {
    public static void main (String [] args) {
        MyThread t = new MyThread ();
        t.start ();
        SoPln (“main method executed by Thread: ” + Thread.currentThread().getName());
    }
}
```

```
O/p
run method executed by Thread: Thread-0
main method executed by Thread: main
```

4. Thread Priorities

- Every thread in Java has some priority. It may have default priority generated by JVM or customized priority provided by Programmer.
- The valid range of Thread Priorities is 1 to 10 where 1 is MIN_PRIORITY & 10 is MAX_PRIORITY
- Thread class defines the following constants to represent some standard priorities.
  a) Thread.**MIN_PRIORITY** = 1
  b) Thread.**NORM_PRIORITY** = 5
  c) Thread.**MAX_PRIORITY** = 10
- Thread scheduler will use priorities while allocating Processor. The thread that has highest priority will get chance first.
- If 2 Threads having same priority then we can’t expect exact execution order. It depends on Thread scheduler.
- Thread class defines the following methods to get & set Priority of a Thread:
  a) public final int getPriority ()
  b) public final void setPriority (): Allowed range 1 to 10 otherwise RE: IllegalArgumentException

- The Default Priority only for the main Thread is 5 but for all remaining threads default Priority will be inherited from Parent to child i.e., whatever priority parent Thread has the same priority will be there for the child thread.

```
class MyThread extends Thread {
    public void run () {
        SoPln (“run method executed by Thread: “ + Thread.currentThread().getPriority());
    }
}

Class Test {
    public static void main (String [] args) {
        MyThread t = new MyThread ();
        t.setPriority (10);
        t.start ();
        SoPln (“main method executed by Thread: ” + Thread.currentThread().getPriority());
    }
}

O/p
run method executed by Thread: 10
main method executed by Thread: 5
```

- Some platforms won’t provide proper support for Thread priorities.

5. Methods to prevent Thread execution [yield (), join (), sleep ()]

yield () method

- Yield () method pauses the current executing Thread to give the chance to waiting threads of same priority. If there is no waiting threading or all waiting threads having low priority then same thread will continue its execution.
- If Multiple threads are waiting with same priority, then which waiting thread will get the chance we can't expect. Also, when will the yielded thread get chance again, we can't expect. It depends on thread scheduler.
- **Use Case:** If a thread requires more processing or execution time then in b/w its recommended to call yield () method.
- **Prototype:**

```
public static native void yield ();
```

```
class MyThread extends Thread {
    public void run () {
        for (int i = 0; i< 10; i++) {
            SoPln ("Child Thread");
            Thread.yield();                    // Line1
        }
    }
}


Class ThreadYieldDemo {
    public static void main (String [] args) {
        MyThread t = new MyThread ();
        t.start ();
        for (int i = 0; i < 10; i++) {
            SoPln ("Main Thread");
        }
    }
}
```

If **we comment line1** then both threads will execute simultaneously & we can't expect which thread will complete first.

If **we don't comment line1** then child thread will call yield () method due to which main thread will first complete its execution then child thread will get chance to complete its execution.

join () method

- If a thread wants to wait until completing some other thread, then we should go for join () method.
- E.g., If a thread t1 wants to wait until completing t2 then t1 has to call t2.join ()
- When t1 executes t2.join () then immediately t1 will enter into waiting state until t2 completes. Once t2 completes, then t1 can continue its execution.
- **Prototype**

```
public final void join () throws InterruptedException;
public final void join (long ms) throws InterruptedException;
public final void join (long ms, int ns) throws InterruptedException;
```

- Every join () method throws InterruptedException which is checked exception hence compulsory we should handle this exception either by using try – catch or by throws keyword otherwise we will get Compile time error.

Case 1: Waiting of main thread until completion of child thread.

```
class MyThread extends Thread {
    public void run () {
        SoPln (“Child thread sleeping for 2 secs”);
        Try {
          Thread.sleep(2000);                    // Child thread sleeps for 2 secs.
        } catch (InterruptedException e) {}
        SoPln (“Child Thread”);
    }
}

Class ThreadYieldDemo {
    public static void main (String [] args) throws InterruptedException {
        MyThread t = new MyThread ();
        t.start ();
        t.join ();                  // main thread will wait for child thread to complete its execution
        SoPln (“Main Thread”);
    }
}
```

O/p:

```
Child thread sleeping for 2 secs
Child Thread
Main Thread
```

Case 2: Waiting of child thread until completion of main thread.

```
class MyThread extends Thread {
    static MyThread mt;
    public void run () {
        try {
          mt.join();                // child thread will wait for main thread  to complete its execution
        } catch (InterruptedException e) {}
        SoPln (“Child Thread”);
    }
}

class ThreadYieldDemo {
    public static void main (String [] args) throws InterruptedException {
        MyThread.mt = Thread.currentThread();          // main thread object
        MyThread t = new MyThread ();
        t.start ();
        SoPln (“main thread sleeping for 2 secs”);      // main thread sleeps for 2 secs
        t.sleep (2000);
        SoPln (“Main Thread”);                        // executed by main thread
    }
}
```

O/p:

```
main thread sleeping for 2 secs
Main Thread
Child Thread
```

- If main thread calls join () method on child thread object & child thread calls join () method on main thread object then both threads will wait forever & the program will be paused (like Deadlock)
- If a thread calls join () method on itself, then it will be in waiting state forever (like Deadlock)

sleep () method

- If a thread doesn't want to perform any operation for a particular amount of time then we should go for sleep () method.
- **Prototype:**

```
public static native void sleep (long millsecs) throws InterruptedException;
public static void sleep (long millsecs, int ns) throws InterruptedException;
```

- Every sleep () method throws InterruptedException which is checked exception hence compulsory we should handle this exception either by using try – catch or by throws keyword otherwise we will get Compile time error.

Q. How a Thread can interrupt another Thread?
Ans: A Thread can interrupt a sleeping Thread or waiting thread by using **interrupt** () method of Thread class.

```
public void interrupt ();
```

- Whenever we're calling interrupt () method if the target thread is not in sleeping/waiting state then there is no impact of interrupt call immediately. Interrupt call will wait until target Thread will enter into sleeping / waiting state.

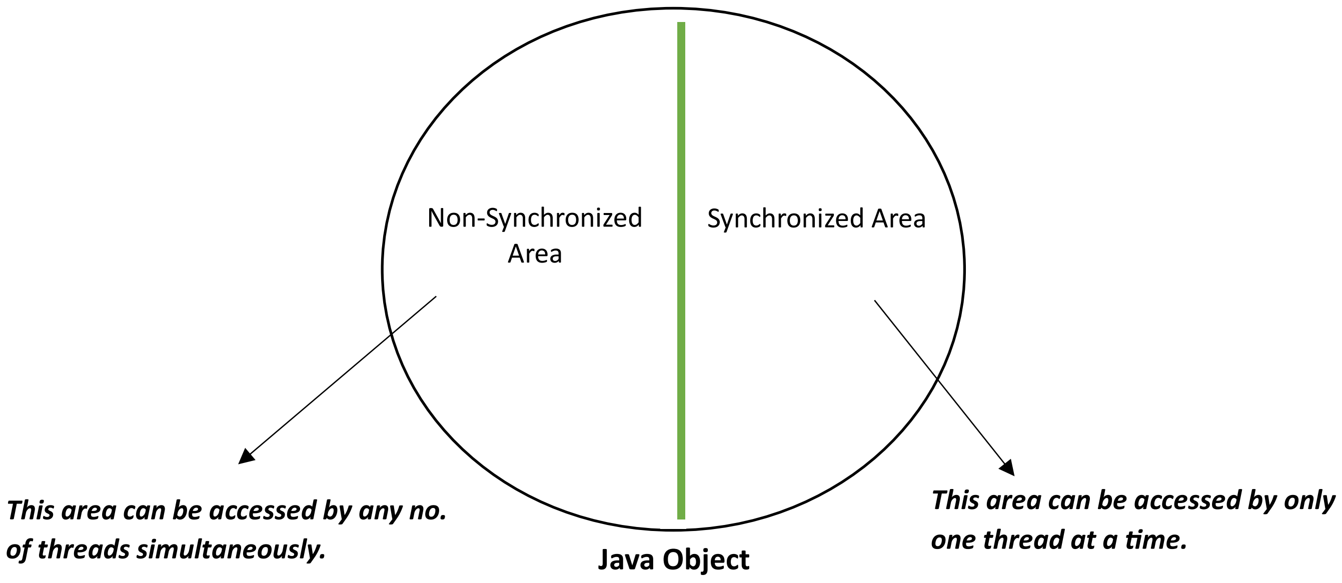
yield ()	join ()	sleep ()
If a thread wants to pass its execution to give the chance to remaining Threads of same Priority, then we should go for yield () method.	If a thread wants to wait until the completion of other thread, then we should go for join () method.	If a thread doesn't want to perform any operation for a particular amount of time, then we should go for sleep () method.

6. Synchronization

- synchronized is the modifier applicable only for methods & blocks but not for classes & variables.
- 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.
- 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 inconsistent problem will be resolved.
- The main adv. of synchronized keyword is we can resolve Data Inconsistency Problem but the main disadv. of synchronized keyword is it increases waiting time of threads & creates performance problem hence if there is no specific requirement then it's not recommended to use synchronized keyword.
- If multiple threads are operating on same java object, then synchronization is required.
- If multiple threads are operating on different java objects, then synchronization is not required.

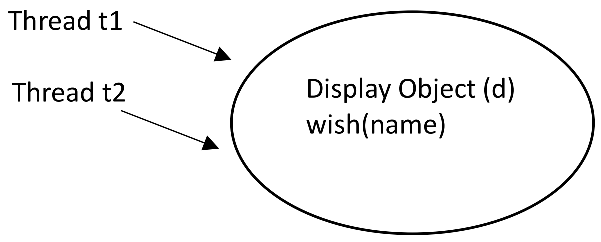
Internal working of synchronized keyword

- Internally, synchronization concept is implemented by using **lock**.
- Every object in Java has a unique lock & whenever we're using synchronized keyword then only lock concept will come into the pictures.
- If a thread wants to execute synchronized method on the given object. First it has to acquire lock of that object then it is allowed to execute any synchronized method on that object. Once method execution completes, automatically Thread releases the lock.
- Acquiring & releasing lock internally taken care by JVM & Programmer not responsible for this activity.
- **Lock concept is implemented based on Object but not based on Method.**


Non-Synchronized Area
Synchronized Area
This area can be accessed by any no. of threads simultaneously.
This area can be accessed by only one thread at a time.
Java Object

```
class X {
          synchronized Area {
              Update operation [e.g., Add / Remove / delete / Replace]
               i.e., Here state of object changes
          }

         Non – synchronized Area {
              Here Object state won't be changed like read() operation
          }

}
```

```
class Display {
     public synchronized void wish (String name) {
         SoPln (“Display sleeping for 2 secs”);
         try { Thread.sleep (2000) } catch (InterruptedException e){}
         SoPln (“Good morning: ” + name);
    }
}

class MyThread extends Thread {
    Display d;
    String name;
    MyThread (Display d, String name) {
      this.d = d; this.name = name;
    }

    public void run () {
       d.wish (name);
    }
}

class SynchronizedDemo {
    public static void main (String[] args) {
       Display d = new Display ();
       MyThread t1 = new MyThread (d, “Dhoni”);
       MyThread t2 = new MyThread (d, “Yuvi”);
       t1.start ();
       t2.start ();
    }
}
```



Output:
```
Display sleeping for 2 secs
Good morning: Yuvi
Display sleeping for 2 secs
Good morning: Dhoni
```

Class level lock

- Every class in java has a unique lock which is nothing but class level lock.
- If a thread wants to execute 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 the lock.
- While a thread executing static synchronized method, the remaining threads are not allowed to execute any static synchronized method of that class simultaneously but remaining threads are allowed to execute the following methods simultaneously i.e., Normal static methods, synchronized & non – synchronized instance methods.

Synchronized block

- If very few lines of the code required synchronization, then it is not recommended to declare entire method as synchronized, we can put this code in synchronized block.
- The main adv. of synchronized block over synchronized method is it reduces waiting time of Threads & improves performance of the application / system.
- We can declare synchronized block as follow
  1. To get lock of current object
  2. To get lock of particular object
  3. To get class level lock

1) To get Lock of current object

```
synchronized (this) {
      // If a thread gets lock of current object, then only its allowed to
         execute this area.
}
```

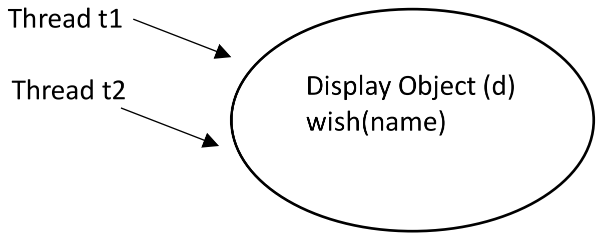
2) To get lock of particular object

```
synchronized (b) {
      // If a thread gets lock of particular object b, then only its allowed to
         execute this area.
}
```

3) To get class level lock

```
synchronized (Display.class) {
      // If a thread gets class level lock of Display class, then only its allowed
         to execute this area.
}
```

```
class Display {
     public void wish (String name) {
         synchronized (this) {
             SoPln (“Display sleeping for 2 secs”);
             try { Thread.sleep (2000) } catch (InterruptedException e){}
             SoPln (“Good morning: ” + name);
        }
     }
}

class MyThread extends Thread {
    Display d;
    String name;
    MyThread (Display d, String name) {
       this.d = d; this.name = name;
    }
    public void run () {
        d.wish (name);
    }
}
```



```
class SynchronizedBlockDemo {
    public static void main (String[] args) {
        Display d = new Display ();
        MyThread t1 = new MyThread (d, "Dhoni");
        MyThread t2 = new MyThread (d, "Yuvi");
        t1.start ();
        t2.start ();
    }
}
```

Output:

```
Display sleeping for 2 secs
Good morning: Yuvi
Display sleeping for 2 secs
Good morning: Dhoni
```

Note:

- Lock concept is applicable for Object types & class types but not for Primitives, hence we can't pass primitive type as argument to synchronized block otherwise we will get Compile time error.
- A Thread can acquire multiple locks simultaneously but from different objects.

7. Interthread communication

- Two Threads can communicate with each other by using wait (), notify (), & notifyAll () methods.
- The thread that is expecting updation is responsible to call wait () method then immediately the thread will enter into waiting state.

- The thread that is responsible to perform updation, after updation it is responsible to call notify () method so that waiting thread will get that notification & continue its execution with those updated items.

- wait (), notify () & notifyAll () methods are present in Object class not in Thread class.

- **Imp. Point:** To call wait (), notify () & notifyAll () methods on any object, thread should be owner of that object **i.e.,** the thread should have lock of that object **(the thread should be inside synchronized area)** otherwise we will get **RE: IllegalMonitorStateException**

- Except wait (), notify () & notifyAll (), there is no other method where thread releases the lock.

- The thread calling wait() method immediately releases the lock so that other thread calling notify() method will acquire the lock & will be able to notify the waiting thread.

Object class wait () method	Object class notify () method
If a thread calls wait () method on any object, it immediately releases the lock of that particular object & enter into waiting state.	If a thread calls notify () method on any object, it releases the lock of that object but may not immediately (because the thread notify first & then does the update operations)

- Every **wait () method throws InterruptedException** which is checked exception hence compulsory we should handle this exception either by using try – catch or by throws keyword otherwise we will get Compile time error.

```
class MyThread extends Thread {
    int total = 0;
    public void run () {
        synchronized (this) {        // child thread has acquired current object lock i.e., MyThread
            SoPln ("Child thread starts calculation");
            for (int i = 1; i < 10; i++) { total = total + i; }
            SoPln ("Child thread giving notification...");
            this.notify();       // child thread notifying the main thread but it may not release lock immediately
        }
    }
}

class SynchronizedBlockDemo {
    public static void main (String[] args) throws InterruptedException {
      MyThread t = new MyThread ();
      t.start ();
      synchronized (t) {                              // main thread has acquired MyThread object lock
          SoPln ("Main thread calling wait method...");
          t.wait();                  // main thread will immediately release lock & enters into waiting state
          SoPln ("Main thread got notification");
          SoPln ("Total: " + t.total);
      }
    }
}
```

Output:

```
Main thread calling wait method…
Child thread starts calculation
Child thread giving notification....
Main thread got notification
Total: 45
```

notify () vs notifyAll () method

Object class notify () method	Object class notifyAll () method
We can use notify () method to give the notification for only one waiting thread. If multiple threads are waiting then only one thread will be notified & the remaining threads have to wait for further notification. Also, which thread will be notified, we can't expect, it depends on JVM & Thread Scheduler.	We can use notifyAll () method to give the notification to all waiting threads of a particular object. Even though multiple threads notified but execution will be performed one by one because threads require lock & only one lock is available.

Best Example of Interthread communication (Producer Consumer Problem)

- Producer thread is responsible to produce items to the Queue & Consumer Thread is responsible to consume items from the Queue.
- If Queue is empty, then Consumer thread will call wait () method on Queue object & will immediately enter into waiting state & the Producer thread will acquire the Queue lock.
- After producing items to the Queue, Producer thread will call notify () method so that Consumer thread will be notified & continue its execution with updated items.

```
class Producer {                                   // ProducerThread
       produce () {
            synchronized (q) {                     // ProducerThread acquired Queue (q) object lock
                  // produce items to the Queue
                  q. notify ();                    // ProducerThread notified waiting ConsumerThread
            }
       }
}

class Consumer {                                   // ConsumerThread
       consume () {
            synchronized (q) {                     // ConsumerThread acquired Queue (q) object lock
                  if (q is empty) then q.wait ();
                  else consume items;
            }
       }
}
```

8. Deadlock

- If 2 threads are waiting for each other forever such type of infinite waiting is called Deadlock.
- Synchronized keyword is the only reason for Deadlock situation. Hence, while using synchronized keyword, we have to take special care. There are no resolution techniques for Deadlock but several prevention techniques are available.

```
class ObjectA {                                    // main thread will have lock of ObjectA
   public synchronized void method1(ObjectB objB) {
      SoPln (Thread.currentThread().getName () + " starts execution of ObjectA's method1()");
      try { Thread.sleep (3000); } catch(InterruptedException e) {};
      SoPln (Thread.currentThread ().getName () + " trying to call ObjectB's message()...");
      objB.message ();         // main thread is trying to call ObjectB message but Thread-0 has lock of ObjectA
   }

   public synchronized void message () {
      SoPln (Thread.currentThread().getName () + ", This is ObjectA's message ()");
   }
}

class ObjectB {                                  // Thread-0  will have lock of ObjectB
   public synchronized void method2(ObjectA objA) {
      SoPln (Thread.currentThread().getName () + " starts execution of ObjectB's method2()");
      try { Thread.sleep (5000); } catch(InterruptedException e) {};
      SoPln (Thread.currentThread().getName () + " trying to call ObjectA's message()...");
      objA.message ();         // Thread-0 is trying to call ObjectA message but main thread has lock of ObjectB
   }

   public synchronized void message () {
      SoPln (Thread.currentThread().getName () + ", This is ObjectB's message ()");
   }
}

class DeadLock extends Thread {
   ObjectA objA = new ObjectA ();
   ObjectB objB = new ObjectB ();

   public void m1() {
      this.start ();                        // new Thread (Thread-0) will be created here
      objA.method1(objB);                  // will be executed by main thread
   }
   public void run () {
      objB.method2(objA);                  // will be executed by Thread-0
   }
   public static void main (String [] args) {
      Deadlock d = new Deadlock ();
      d.m1();
   }
}
```

Output:

```
Thread-0 starts execution of ObjectB's method2 ()
main starts execution of ObjectA's method1 ()
main trying to call ObjectB's message ()...
Thread-0 trying to call ObjectA's message ()...
```

Deadlock Vs Starvation

- Long waiting of a thread where waiting never ends is called Deadlock while Long waiting of a thread where waiting ends at a certain point is called Starvation.

For E.g., Low Priority threads have to wait until completing all High Priority Threads. It may be long waiting but will end at a certain point which is nothing but Starvation.

9. Daemon Threads

- The threads which are executing in the background are called Daemon threads. E.g., Garbage Collector, Signal dispatcher etc.
- The main objective of Daemon threads is to provide support for Non – Daemon threads (main thread). For e.g., If main thread runs with low memory, then JVM will run Garbage collector to destroy useless objects so that free memory bytes will be improved.
- Usually, the Daemon threads have low priority but based on our requirement, Daemon threads can run with high priority also.
- We can check Daemon nature of a thread by using following methods but changing Daemon nature is possible only before starting of a thread.

```
public boolean isDaemon ()
public void setDaemon (boolean b)
```

- By default, main Thread is always Non – Daemon & for all remaining Threads, Daemon nature will be inherited from parent to child.
- **Imp. Point:** It is impossible to change Daemon nature of main thread because it's already started by JVM at beginning.
- **Imp. Point:** Whenever last non – Daemon thread terminates, automatically all Daemon threads will be terminated irrespective of their position.

```
class MyThread extends Thread {
   public void run () {
      SoPln ("run () MyThread isDaemon: " + this.isDaemon ());
      For (int i = 0; i < 10; i++) {
         SoPln ("Child thread");
         try{ Thread.sleep (2000); } catch(InterruptedException e) {}
      }
   }
}

class DaemonThreadDemo {
   public static void main (String [] args) {
      System.out.println("main thread isDaemon: " + Thread.currentThread().isDaemon ());
      //Thread.currentThread().setDaemon (true);      // RE: IllegalThreadStateException

      MyThread t = new MyThread ();
      System.out.println("main () MyThread isDaemon: " + Thread.currentThread().isDaemon ());
      t.setDaemon (true);
      t.start ();
      System.out.println("end of main thread");
      // main thread will end here since main thread is a non – daemon thread, it will terminate all daemon
      //   threads in the program.
   }
}
```

Output (correct order can't be expected)		
main thread isDaemon: false main() MyThread isDaemon: false end of main thread run() MyThread isDaemon: true Child thread	main thread isDaemon: false main() MyThread isDaemon: false end of main thread	main thread isDaemon: false main() MyThread isDaemon: false end of main thread run() MyThread isDaemon: true

10. Stop, Resume & Suspend thread

- We can stop a Thread execution by using stop() method of Thread class. Thread will immediately enter into dead state.

```
public void stop ()
```

- We can suspend a thread by using suspend() method of Thread class. Thread will immediately enter into suspended state & to resume thread we can use resume () method of Thread class.

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

- Anyway, these methods are deprecated & not recommended to use.

11. Multithreading Model

- Java multithreading concept is implemented by using the following 2 models
  a) Green Thread model
  b) Native OS model

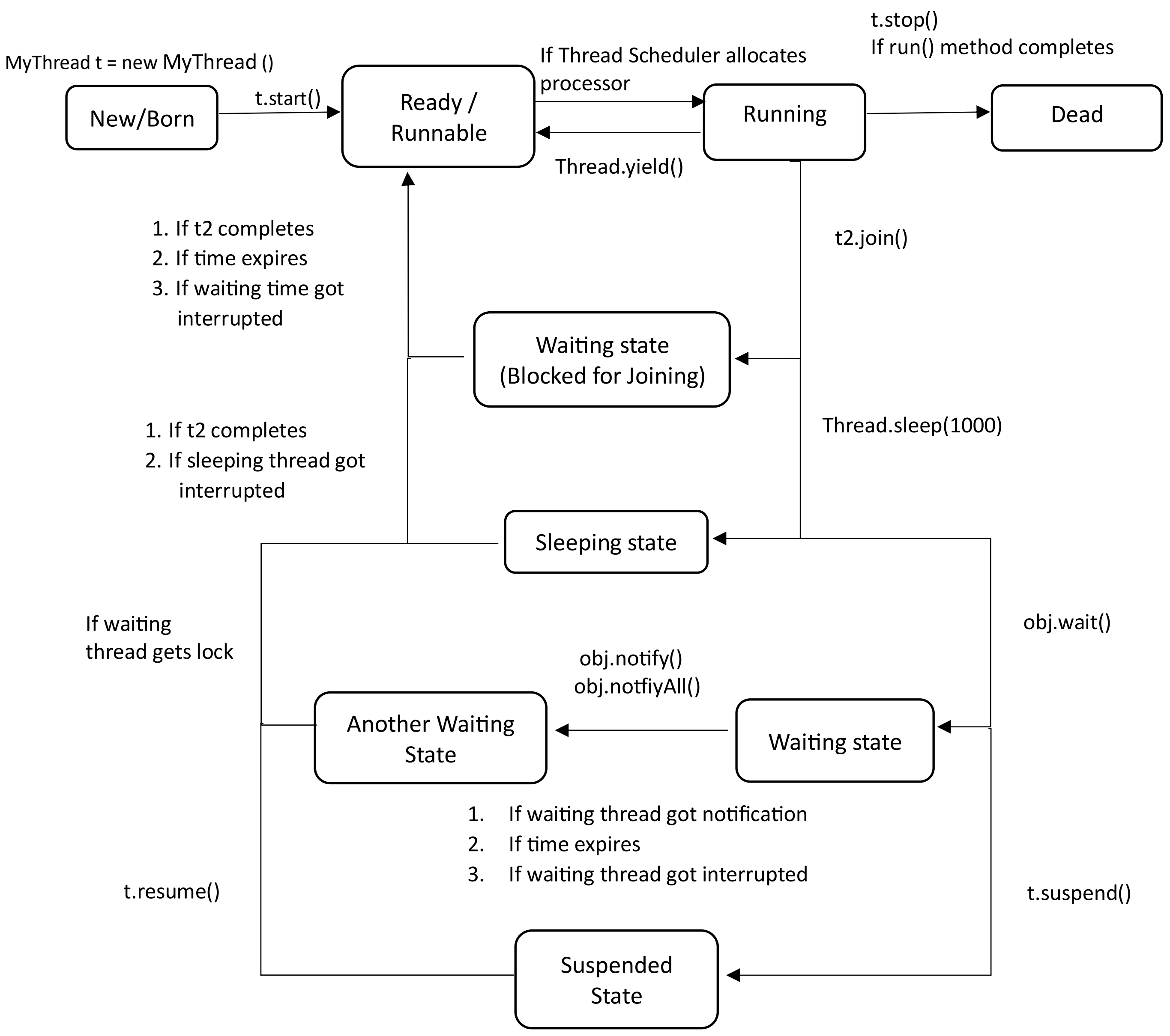
a) Green Thread model

- The thread which is managed completely by JVM without taking underlying OS support is called Green Thread.
- Very few OS like Sun Solaries provide support for Green thread model.
- Anyway, Green Thread model is deprecated & not recommended to use.

b) Native OS model

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

12. Final lifecycle of Thread

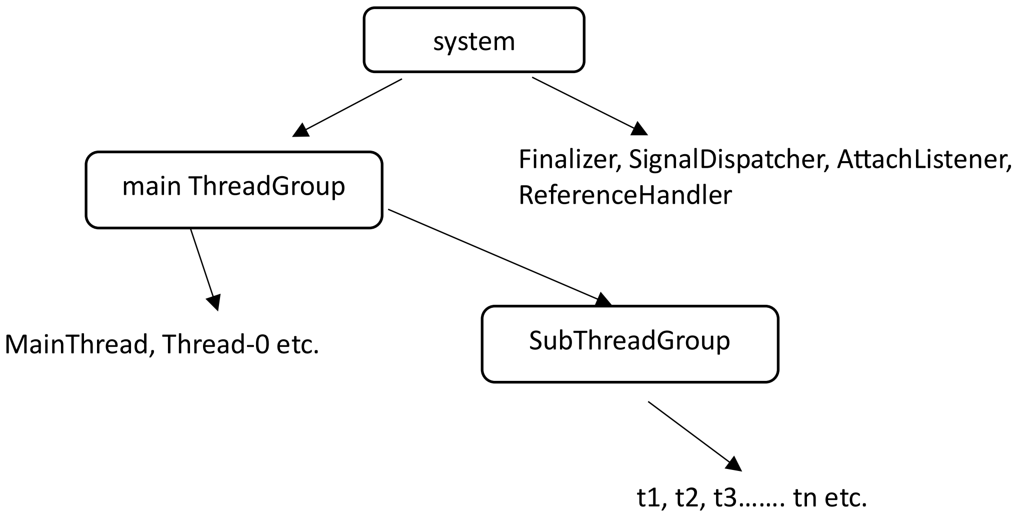


MyThread t = new MyThread ()
New/Born
t.start()
Ready / Runnable
If Thread Scheduler allocates processor
Running
Thread.yield()
t.stop()
If run() method completes
Dead
t2.join()
1. If t2 completes
2. If time expires
3. If waiting time got interrupted
Waiting state (Blocked for Joining)
Thread.sleep(1000)
1. If t2 completes
2. If sleeping thread got interrupted
Sleeping state
obj.wait()
If waiting thread gets lock
obj.notify()
obj.notfiyAll()
Another Waiting State
Waiting state
1. If waiting thread got notification
2. If time expires
3. If waiting thread got interrupted
t.resume()
t.suspend()
Suspended State

13. ThreadGroup

- Based on functionality, we can group Threads into a single unit called as ThreadGroup.
- In addition to threads, ThreadGroup can also contain subThreadGroups.
- The main adv. of maintaining threads in form of ThreadGroup is we can perform common operations very easily.
- Every thread in Java belongs to some Group. E.g., main thread belongs to main ThreadGroup.
- **Every ThreadGroup in java is the child group of system ThreadGroup either directly or indirectly. Hence, system ThreadGroup acts as root for all ThreadGroups in Java.**

- SystemGroup contains several system level Threads like Finalizer, ReferenceHandler, SignalDispatcher, AttachListener etc.


system
main ThreadGroup
Finalizer, SignalDispatcher, AttachListener, ReferenceHandler
MainThread, Thread-0 etc.
SubThreadGroup
t1, t2, t3……. tn etc.

```
class ThreadGroupDemo {
   public static void main (String [] args) {
      SoPln (Thread.currentThread().getThreadGroup());                 // java.lang.ThreadGroup [name=main]
      SoPln (Thread.currentThread().getThreadGroup().getName());             // main
      SoPln (Thread.currentThread().getThreadGroup().getParent());  // java.lang.ThreadGroup[name=system]
      SoPln (Thread.currentThread().getThreadGroup().getParent().getName());      // system
   }
}
```

- **ThreadGroup** is a java class present in java.lang package & it is direct child class of Object class.

- **Constructors**
  1) ThreadGroup g = new ThreadGroup (String groupName);
     - Creates a new ThreadGroup with specified groupName.
     - The parent of this new ThreadGroup is the currently executing Thread ThreadGroup.

  2) ThreadGroup g = new ThreadGroup (ThreadGroup parentGroup, String groupName);
     - Creates a new ThreadGroup with specified groupName.
     - The parent of this new ThreadGroup is the specified parentGroup.

- Imp. Methods of ThreadGroup class

No.	Methods	Description
1.	String getName()	Returns name of the ThreadGroup
2.	int getMaxPriority()	Returns max Priority of ThreadGroup
3.	void setMaxPriority()	To set max Priority of ThreadGroup (max = 10)
4.	ThreadGroup getParent()	Returns ParentGroup of current Thread
5.	void list()	Prints info about ThreadGroup to the console
6.	int activeCount()	Returns no. of active threads present in the ThreadGroup
7.	int activeGroupCount()	Returns no. of active ThreadGroups present in the current ThreadGroup

8.	int enumerate(Thread[] t)	To copy all active threads of this ThreadGroup into provided Thread array.
9.	int enumerate(ThreadGroup g)	To copy all active sub ThreadGroups into ThreadGroup array
10.	boolean isDaemon()	To check whether the ThreadGroup is Daemon or not
11.	void setDaemon(boolean b)	Change Daemon nature of ThreadGroup
12.	void interrupt()	To interrupt all waiting or sleeping threads present in the ThreadGroup
13.	void destroy()	To destroy ThreadGroup & its sub threadGroups

```
class MyThread extends Thread {
  MyThread (ThreadGroup g, String name) {
    super (g, name);
  }

  public void run () {
    SoPln (Thread.currentThread().getName() + " is executing...");
    try { Thread.sleep (3000); } catch(InterruptedException e) {}
    SoPln (Thread.currentThread().getName() + " executed!!");
  }
}

class ThreadGroupDemo {
  public static void main (String [] args) {
    ThreadGroup pg = new ThreadGroup ("Parent Group");
    ThreadGroup cg = new ThreadGroup (pg, "Child Group");

    MyThread t1 = new MyThread (pg, "Thread-1");
    MyThread t2 = new MyThread (pg, "Thread-2");

    t1.start();
    t2.start();
    SoPln ("Parent Group for main: " + Thread.currentThread().getThreadGroup().getParent().getName());
    SoPln ("Parent Group for pg: " + pg.getParent().getName());
    pg.list();
    SoPln ("No. of active Threads: " + pg.activeCount());
    SoPln ("No. of active ThreadGroup: " + pg.activeGroupCount());

  }
}
```

Output

```
Thread-1 is executing...
Thread-2 is executing...
Parent Group for main: system
Parent Group for pg: main
java.lang.ThreadGroup[name=Parent Group,maxpri=10]
  Thread[Thread-1,5,Parent Group]
  Thread[Thread-2,5,Parent Group]
  java.lang.ThreadGroup[name=Child Group,maxpri=10]
No. of active Threads: 2
No. of active ThreadGroup: 1
Thread-1 executed!!
Thread-2 executed!!
```

```
class ThreadGroupDemo {
   public static void main(String[] args) {
      ThreadGroup system = Thread.currentThread().getThreadGroup().getParent();
      Thread [] t = new Thread[system.activeCount()];
      system.enumerate (t);
      for (Thread t1: t) {
         System.out.println(t1.getName () + "--------" + t1.isDaemon());
      }
   }
}
```

Output:

```
Reference Handler--------true
Finalizer--------true
Signal Dispatcher--------true
main--------false
Common-Cleaner--------true
```

14. Concurrent package

The problems with traditional synchronized keyword:

a) We're not having any flexibility to try for a lock without waiting.
b) There is no way to specify maximum waiting time for a thread to get lock & due to this, thread has to wait to get lock that creates Performance problems / Deadlock.
c) It a thread releases a lock then we can't control which waiting thread will get that lock.
d) There is no API to list out all the waiting threads for a lock.
e) Synchronized keyword compulsory we have to use either at method level or within the method & it's not possible to use across multiple methods.

- To overcome these above problems, SUN people introduced java.util.concurrent.locks package in 1.5 version.
- This package provides several enhancements to have more control on concurrency.

Lock Interface

- Lock object is similar to implicit lock acquired by a thread to execute synchronized method or synchronized block.
- Lock implementation provide more extensive operations than traditional implicit locks.

Imp. Methods of Lock i/f

No.	Methods
1.	void lock ()
2.	boolean tryLock()
3.	boolean trylock(long_time, TimeUnit unit)
4.	void lockInterruptibly()
5.	void unlock()

1. void lock ()

- We can use this method to acquire a lock.
- If lock is already available, then immediately current thread will get that lock
- If lock is not available, then it will wait until getting the lock.
- It has exactly same behaviour of traditional synchronized keyword.

2. boolean tryLock ()

- We can use this method to acquire the Lock without waiting
- If lock is available, then the thread will acquire the lock & return true.
- If lock is not available, then this method returns false & can continue its execution without waiting.
- In this case, thread will never enter into waiting state.

```
If (lockObj.tryLock()) {
   // perform safe operation
} else {
  // perform alternative operations
}
```

3. boolean tryLock (long_time, TimeUnit unit)

- If lock is available then the thread will get the lock & continue its execution.
- If the lock is not available, then the thread will wait until specified amount of time, once time expires then thread can continue its execution.
- **TimeUnit**: It is an enum present in java.util.concurrent package

```
enum TimeUnit {
     NANOSECONDS, MICROSECONDS, MILLISECONDS, SECONDS, MINUTES, HOURS, DAYS
}

If (lockObj.tryLock(1000, TimeUnit.MILLISECONDS)){
       ………
}
```

4. void lockInterruptibly ()

- Acquires the lock if it is available & returns immediately. If lock is not available then it will wait.
- While waiting if the thread is interrupted then thread won't get the lock

5. void unlock ()

- We can use this method to release a lock.
- To call this method, compulsory current thread should be owner of the lock otherwise we will get Runtime exception saying **IllegalMonitorStateException**.

ReentrantLock

- It is the implementation class of Lock interface & it is direct child class of Object.
- Reentrant means a thread can acquire same lock multiple times without any issue.
- Internally, ReentrantLock increments thread's personal account whenever we call lock() method & decrements count value whenever thread calls unlock() method & lock will be released when count reaches zero.

Constructors

1. ReentrantLock rl = new ReentrantLock ();
   - Creates an instance of ReentrantLock.

2. ReentrantLock rl = new ReentrantLock (boolean fairness);
   - Creates ReentrantLock with a given Fairness policy.
   - If fairness is true, then longest waiting thread will 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 the chance we can't expect.
   - Default value of fairness is false.

Imp. Methods of ReentrantLock

No.	Methods	Description
1.	All Lock i/f methods i.e., lock(), tryLock(), lockInterruptibly(), unlock()	
2.	int getHoldCount ()	Returns no. of holds on the lock by current thread
3.	boolean isHeldByCurrentThread ()	Returns true iff lock is hold by current thread
4.	int getQueueLength ()	Returns no. of threads waiting for the lock
5.	Collection getQueuedThreads ()	Returns a collection of threads which are waiting to get the lock.
6.	boolean hasQueuedThreads ()	Return true if any thread waiting to get the lock.
7.	boolean isLocked ()	Returns true if the lock is acquired by some thread.
8.	boolean isFair ()	Return true if the fairness policy is set to "true" value
9.	Thread getOwner ()	Returns the thread which acquires the lock.

```
import java.util.concurrent.locks.*;

class ReentrantLockDemo {
    public static void main(String[] args) {
        ReentrantLock l = new ReentrantLock();
        l.lock ();
        l.lock ();
        System.out.println(l.isLocked ());                    // true
        System.out.println(l.isHeldByCurrentThread ());       //  true
        System.out.println(l.getQueueLength());               // 0

        l.unlock ();
        System.out.println(l.getHoldCount ());               // 1
        System.out.println(l.isLocked ());                   // true

        l.unlock ();
        System.out.println(l.getHoldCount ());               // 0
        System.out.println(l.isLocked ());                   // false

    }
}
```

Using ReentrantLock in place of traditional synchronized keyword

```
import java.util.concurrent.locks.*;

class Display {
   ReentrantLock l = new ReentrantLock ();

   public void wish (String name) {
      l.lock ();
      for (int i = 0; i < 5; i++) {
         SoPln ("Good morning: ");
         try { Thread.sleep (2000); } catch(InterruptedException e) {}
         SoPln (name);
      }
      l.unlock ();
   }
}

class MyThread extends Thread {
   Display d;
   String name;
   MyThread (Display d, String name) {
      this.d = d;
      this.name = name;
   }

   public void run () {
      d.wish(name);
   }
}

class ReentrantLockDemo {
   public static void main (String [] args) {
      Display d = new Display ();
      MyThread t1 = new MyThread (d, "Dhoni");
      MyThread t2 = new MyThread (d, "Yuvi");
      t1.start();
      t2.start();
   }
}
```

Output:

```
Good morning: Dhoni
Good morning: Dhoni
Good morning: Dhoni
Good morning: Dhoni
Good morning: Dhoni
Good morning: Yuvi
Good morning: Yuvi
Good morning: Yuvi
Good morning: Yuvi
Good morning: Yuvi
```

Note: Not much difference b/w using **synchronized** keyword or using **ReentrantLock with lock ()** & unlock ()

Demo Program for ReentrantLock tryLock () method

```
import java.util.concurrent.locks.*;

class MyThread extends Thread {
   static ReentrantLock l = new ReentrantLock ();
   MyThread (String name) {
      super(name);
   }

   public void run(){
      if(l.tryLock()) {
         SoPln (Thread.currentThread().getName() + "---- got lock & performing safe operations");
         try { Thread.sleep(3000); } catch(InterruptedException e) {}
         l.unlock ();
      }
      else {
         SoPln (Thread.currentThread().getName() + "---- unable to get lock & hence performing alternative
                    operations");
      }
   }
}

class ReentrantLockDemo {
   public static void main(String[] args) {
      MyThread t1 = new MyThread("First Thread");
      MyThread t2 = new MyThread("Second Thread");
      t1.start();
      t2.start();
   }
}
```

Output:

```
Second Thread---- got lock & performing safe operations
First Thread---- unable to get lock & hence performing alternative operations
```

Using tryLock () with Time Period

```
import java.util.concurrent.locks.*;
import java.util.concurrent.*;
import java.time.LocalTime;

class MyThread extends Thread {
   static ReentrantLock l = new ReentrantLock ();
   MyThread (String name) {
      super(name);
   }

   public void run (){
      do {
         try {
            if (l.tryLock(5000, TimeUnit.MILLISECONDS)) {
               SoPln ("[" + LocalTime.now() + "]: " + Thread.currentThread().getName() + "---- got lock");
               Thread.sleep(30000);
               l.unlock();
               SoPln ("[" + LocalTime.now() + "]: " + Thread.currentThread().getName() + "---- releases lock");
               break;
           }
           else {
              SoPln ("[" + LocalTime.now() + "]: " + Thread.currentThread().getName() + "---- unable to get lock
                    & will try again");
           }
         } catch (Exception e) {}
      } while(true);
   }
}

class ReentrantLock {
   public static void main (String [] args) {
      MyThread t1 = new MyThread ("First Thread");
      MyThread t2 = new MyThread ("Second Thread");
      t1.start();
      t2.start();
   }
}
```

Output:

```
[12:56:20]: Second Thread---- got lock
[12:56:25]: First Thread---- unable to get lock & will try again
[12:56:30]: First Thread---- unable to get lock & will try again
[12:56:35]: First Thread---- unable to get lock & will try again
[12:56:40]: First Thread---- unable to get lock & will try again
[12:56:45]: First Thread---- unable to get lock & will try again
[12:56:50]: First Thread---- unable to get lock & will try again
[12:56:50]: First Thread---- got lock
[12:56:50]: Second Thread---- releases lock
[12:57:20]: First Thread---- releases lock
```

15. Thread Pools (Executor Framework)

- Creating a new Thread for every Job may create performance & memory problems. To overcome this, we should go for ThreadPool.
- ThreadPool is a pool of already created Threads ready to do our Job.
- Java 1.5 version introduces ThreadPool Framework to implement Thread Pools.
- ThreadPool framework also known as Executor framework.
- While designing & developing web servers & application servers, we can use ThreadPool concept.
- We can create a ThreadPool as follows

```
ExecutorService service = Executors.newFixedThreadPool(3);
```

- We can submit a Runnable job by using submit () method:

```
service.submit(job);
```

- We can shutdown ExecutorService by using shutdown () method:

```
service.shutdown();
```

```
import java.util.concurrent.*;
import java.time.LocalTime;

class PrintJob implements Runnable {
   String job;
   PrintJob (String job) {
      this.job = job;
   }
   public void run () {
      System.out.println("[" + LocalTime.now() + "]: " + job + " started by Thread: " +
Thread.currentThread().getName());
      try { Thread.sleep(5000); } catch(InterruptedException e) {}
      System.out.println("[" + LocalTime.now() + "]: " + job +  " completed by Thread: " +
Thread.currentThread().getName());
   }
}

class ExecutorServiceDemo {
   public static void main (String [] args) {
      PrintJob [] jobs = {
         new PrintJob("Job-1"), new PrintJob("Job-2"),
         new PrintJob("Job-3"), new PrintJob("Job-4")};
      ExecutorService service = Executors.newFixedThreadPool(2);
      for (PrintJob job: jobs) {
         service.submit(job);
      }
      service.shutdown();
   }
}
```

Output:

```
[13:58:39]: Job-1 started by Thread: pool-1-thread-1
[13:58:39]: Job-2 started by Thread: pool-1-thread-2
[13:58:45]: Job-1 completed by Thread: pool-1-thread-1
[13:58:45]: Job-2 completed by Thread: pool-1-thread-2
[13:58:45]: Job-3 started by Thread: pool-1-thread-1
[13:58:45]: Job-4 started by Thread: pool-1-thread-2
[13:58:50]: Job-3 completed by Thread: pool-1-thread-1
[13:58:50]: Job-4 completed by Thread: pool-1-thread-2
```

16. Callable interface & Future interface

- In the case of Runnable Job, Thread won't return anything after completing the Job.
- If a thread is required to return some result after execution, then we should go for Callable interface.
- Callable interface contains only one method call () **public Object call () throws Exception**
- If we submit a **Callable** object to **Executors** then after completing the Job, Thread will return an object of type "**Future**" i.e., **Future** object can be used to retrieve the result from Callable Job.

```
import java.util.concurrent.*;
import java.time.LocalTime;

class MyCallable implements Callable {
   int num;
   MyCallable (int num) {
      this.num = num;
   }
   public Object call () throws Exception {
      SoPln (Thread.currentThread().getName() + " found sum of 1st " + num + " numbers.");
      int sum = 0;
      for(int i = 1; i <= num; i++) {
         sum = sum + i;
      }
      return sum;
   }
}

class CallableDemo {
   public static void main (String [] args) throws Exception {
      MyCallable [] jobs = {
         new MyCallable (10), new MyCallable (20),
         new MyCallable (30), new MyCallable (40)};
      ExecutorService service = Executors.newFixedThreadPool(2);
      for(MyCallable job: jobs) {
         Future f = service.submit(job);
         SoPln (f.get());
      }
      service.shutdown ();
   }
}
```

Output:

```
pool-1-thread-1 found sum of 1st 10 numbers.
55
pool-1-thread-2 found sum of 1st 20 numbers.
210
pool-1-thread-1 found sum of 1st 30 numbers.
465
pool-1-thread-2 found sum of 1st 40 numbers.
820
```

Runnable Vs Callable interface

No.	Runnable	Callable
1.	If a thread is not required to return anything after completing the Job then we should go for Runnable.	If a thread is required to return something after completing the Job then we should go for Callable interface.
2.	Runnable interface contains only one method i.e., run()	Callable interface contains only one method i.e., call()
3.	Return type of run() method is void	Return type of call() method is Object
4.	Within the run() method, if there is any chance of rising checked exception compulsory we should handle by using try-catch not with throws keyword	Within call() method, if there is any chance of rising checked exception, we're required to handle by using try-catch because call() method already throws Exception.
5.	Runnable interface present in java.lang package	Callable interface present in java.util.concurrent package.
6.	Introduced in 1.0 version	Introduced in 1.5 version.

17. ThreadLocal

- ThreadLocal class provides ThreadLocal variables. ThreadLocal class maintains values per Thread basis.
- Each ThreadLocal object maintains a separate value like userid, transaction_id etc. for each thread that accesses that object.
- Thread can access its local value, manipulate its value, & even remove its value.
- In every part of the code which is executed by the Thread, we can access its local variables.

e.g., Consider a Servlet which invokes some business methods.
We have a requirement to generate a unique transaction_id for each & every request & pass that transaction _id to the business methods.
For this Requirement, we can use ThreadLocal to maintain a separate transaction_id for every request/thread.

- ThreadLocal class introduced in 1.2 version & enhanced in 1.5 version.
- ThreadLocal can be associated with ThreadScope.
- Total code which is executed by the thread has access to the corresponding ThreadLocal variable.
- A thread can access its own local variables & can't access other thread's local variables.
- Once thread entered into dead state, all its local variables are by default eligible for Garbage Collection.

Constructors

1. ThreadLocal tl = new ThreadLocal ()

Methods

No.	Methods	Description
1.	Object get ()	Returns the value of ThreadLocal variable associated with current thread.
2.	Object initialValue ()	Returns initial value of ThreadLocal variable associated with current thread. The default implementation of this method returns null. To customize our own initialValue (), we have to override this method.
3.	void set (Object new_value)	To set a new_value
4.	void remove ()	To remove the value of ThreadLocal variable associated with currentThread. It is a newly added method in 1.5 version After removal, if we're trying to access, it will be reinitialized once again by invoking its initialValue () method.

```
class ThreadLocalDemo {
   public static void main (String [] args) {
      ThreadLocal tl = new ThreadLocal ();
      System.out.println(tl.get());                 // null
      tl.set("Yuvi");
      System.out.println(tl.get());                // Yuvi
      tl.remove();
      System.out.println(tl.get());               // null
      tl = new ThreadLocal() {
         public Object initialValue() {
            return "abc";
         }
      };
      System.out.println(tl.get());              // abc
      tl.set("Dhoni");
      System.out.println(tl.get());             // Dhoni
      tl.remove();
      System.out.println(tl.get());            // abc

   }
}
```

```
class CustomerThread extends Thread {
   static Integer custId = 0;
   private static ThreadLocal tl = new ThreadLocal() {
      protected Integer initialValue () {
         return ++custId;
      }
   };
   CustomerThread(String name) {
      super(name);
   }

   public void run() {
      SoPln (Thread.currentThread().getName() + " executing with customer id: " + tl.get());
   }
}

class ThreadLocalDemo {
   public static void main(String[] args) {
      CustomerThread ct1 = new CustomerThread("Customer-Thread-1");
      CustomerThread ct2 = new CustomerThread("Customer-Thread-2");
      CustomerThread ct3 = new CustomerThread("Customer-Thread-3");
      CustomerThread ct4 = new CustomerThread("Customer-Thread-4");
      ct1.start();
      ct2.start();
      ct3.start();
      ct4.start();
   }
}
```

```
Output:
Customer-Thread-3 executing with customer id: 4
Customer-Thread-1 executing with customer id: 1
Customer-Thread-4 executing with customer id: 3
Customer-Thread-2 executing with customer id: 2
```

ThreadLocal Vs Inheritance

- Parent Threads ThreadLocal variable by default not available to the child thread.
- If we want to make Parent Threads ThreadLocal variables value available to the Child Thread then we should go for **InheritableThreadLocal** class.
- **InheritableThreadLocal** is the child class of ThreadLocal & hence all methods present in ThreadLocal by default available to InheritableThreadLocal.
- By default, ChildThread value is exactly same as ParentThread's value but we can provide customized value for ChildThread by overriding childValue () method.

Constructors:

1. InheritableThreadLocal tl = new InheritableThreadLocal ();

Methods: In addition to these methods, it contains only one method

public Object childValue (Object parentValue)

```
class ChildThread extends Thread {
    public void run() {
      SoPln ("Child thread value: " + ParentThread.tl.get());
    }
}

class ParentThread extends Thread {
   public static InheritableThreadLocal tl = new InheritableThreadLocal () {
      public Object childValue (Object p) {
         return "cc";
      }
   };

   public void run() {
      tl.set("pp");
      System.out.println("Parent thread value: " + tl.get());
      ChildThread ct = new ChildThread ();
      ct.start();
   }
}

class InheritableThreadLocalDemo {
   public static void main (String [] args) {
      ParentThread pt = new ParentThread ();
      pt.start();
   }
}

Output:
Parent thread value: pp
Child thread value: cc
```