Multi-Threading

Module 7: Multi – Threading	
1.	Multitasking, Multithreading
2.	Way to define a thread ➢ Extending Thread class ➢ Implementing Runnable interface
3.	Thread 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.	MultiThreading enhancements

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

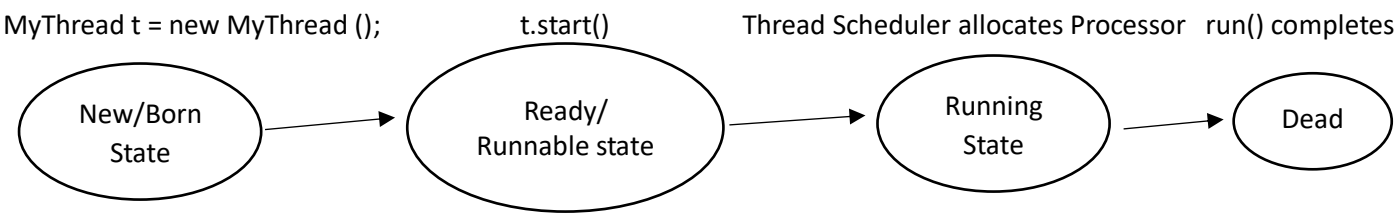
- If we override start () method then our start () method will be executed just like a normal method call & new thread won't be created.
- It's not recommended to override start () method otherwise don't go for multithreading concept

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
  }
}
```

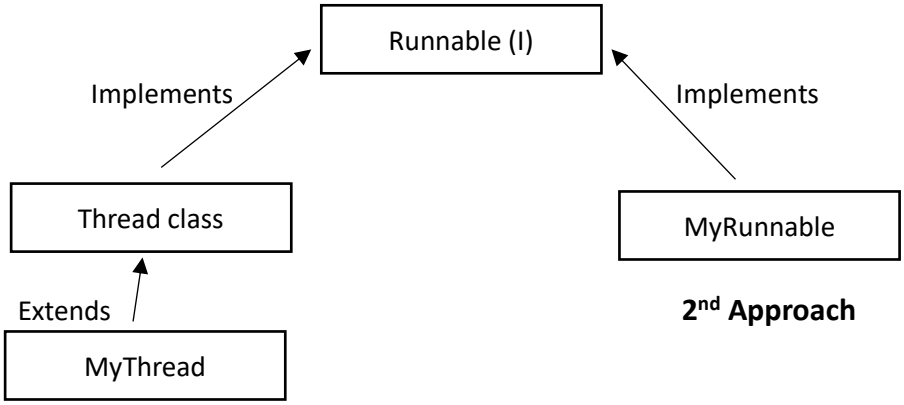
Case 8: After starting a thread, if we're trying to restart the same thread then we will get exception saying RE: IllegalThreadStateException.

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:**

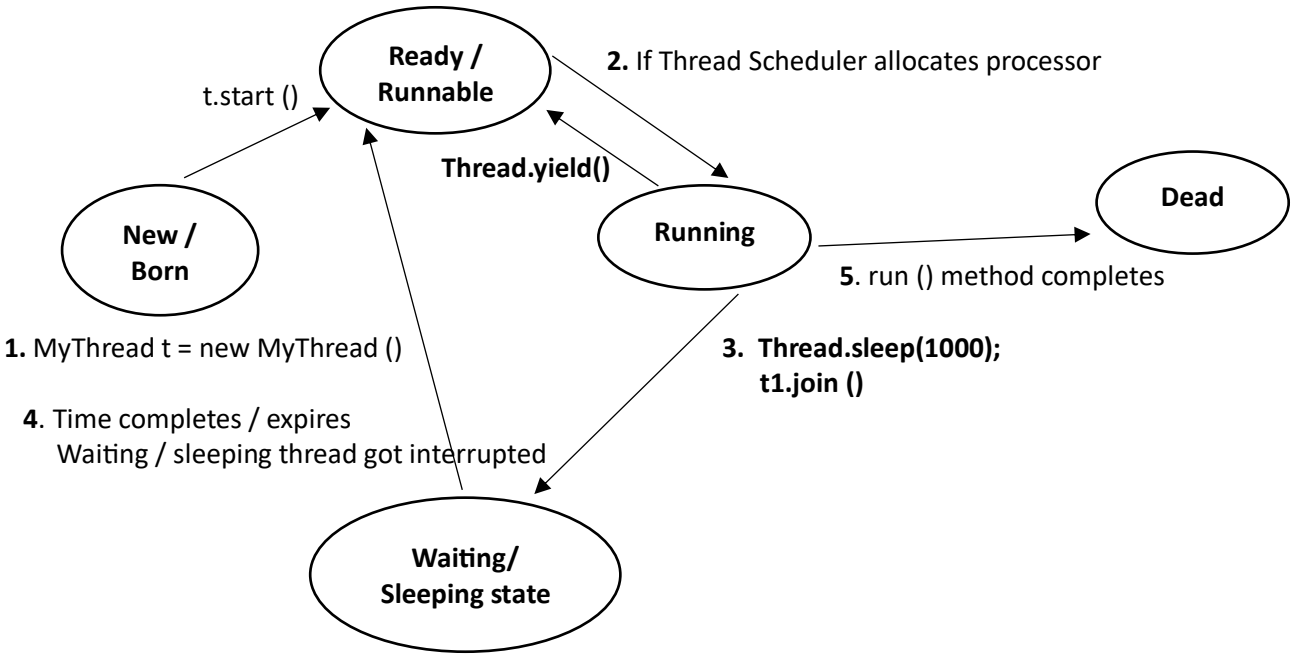
```
public static native void sleep (long millsecs) throws InterruptedException;
public static void sleep (long millsecs, int ns) throws InterruptedException;
```

- 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.



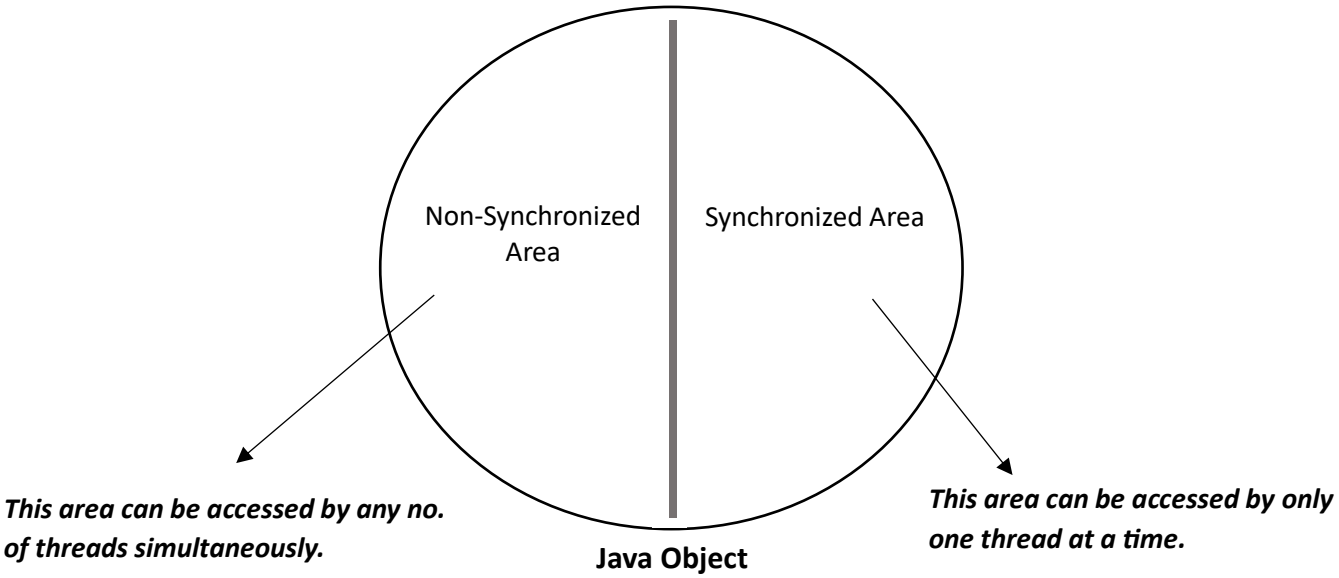
yield ()	join ()	sleep ()
If a thread wants to pass its execution to give the chance to remaining Threads of same Priority, then we should go for yield () method.	If a thread wants to wait until the completion of other thread, then we should go for join () method.	If a thread doesn't want to perform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

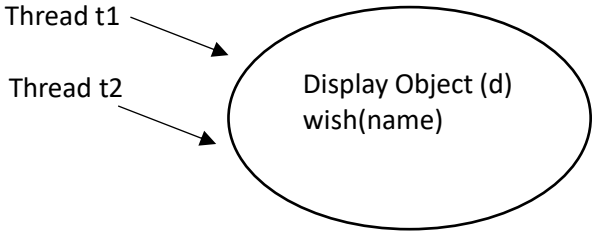
Internal working of synchronized keyword

- 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.**




```
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.
}
```

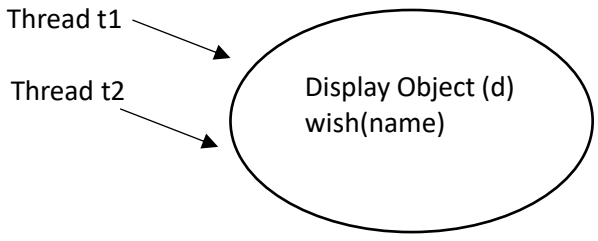
2) To get lock of particular object

```
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;
          }
      }
}
```