Chapter 14. Multithreaded Programming

1. Introduction

   - There are two types of multitasking one is process based and another is thread based. Process based multitasking is when computer runs two or three programmes concurrently ex. running a java compiler at the same time using a text editor or visiting a website. Thread based multitasking is when a single program can perform two or more task simultaneously. Ex. A text editor formats a text along with printing a page. Both tasks will be performed by two separate threads.

   - Process multitasking is heavy weight and thread based multitasking is light weight.

2. Thread Priorities :

   - Java assigns each thread a priority which defines how that thread should be treated respect to other threads.

   - Thread priorities are integers which specify relative priority of one thread to another.

   - Higher priority thread doesn't mean it runs faster than lower priority thread instead priorities are used to decide when to switch one running thread to the next. This is called **context switch.**

   - Rules that determine when a context switch takes place are simple

     i. A thread can voluntarily relinquish control : This is done by explicitly yielding, sleeping or blocking on pending I/O. In this scenario, all other threads are examined and the highest priority thread which is ready to run is given CPU.

     ii. A thread can be pre-empted by a higher priority thread : In this case a lower priority thread that doesn't yield the processor is simply pre-empted, no matter what its doing, by an higher priority thread. Basically, as soon as a higher priority thread wants to run, it does. This is called **pre-emptive multitasking**.

3. Synchronization :

   A method should be synchronized so that only one thread can use it at one time. Synchronization is very necessary in many cases example when two threads are sharing a data structure like linked list and one thread is updating it while other is reading it then it gives a wrong result. So methods should be synchronized and can be used by one thread at a time and other threads should wait.

4. Thread class and the Runnable interface :

   - To create a new thread a class must either extends **Thread class** or implements **Runnable interface**.

   - Thread class defines several methods, some of those are shown below

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

- Creating a thread by implementing Runnable

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

- One can create thread on any object which implements Runnable.
- Only one method run() needs to be implemented. Inside run() one will define code that constitute the thread. run() can call other methods, use other classes and declare variables just like main thread. The only difference is run() establishes the entry point for another concurrent thread of execution within the program.
- This thread will end when run() returns.
- Example 1 - TCR:

```
package multiThreading;

public class HelloRunnable implements Runnable
{

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}

Output:
Hello from a thread!
```

- Example 2 : Java complete reference

```
package multiThreading;

class ThreadWithRunnable1 implements Runnable
{
Thread t;
```

```
ThreadWithRunnable1()
{
	t = new Thread(this, "Demo thread");
	System.out.println("new thread "+t);
	t.start();
}
public void run()
{
	try
	{
		for(int j=5; j>0; j--)
		{
			System.out.println("child thread "+j);
			Thread.sleep(1000);

		}
	} catch (InterruptedException e)
		{
			System.out.println("child interrupted");
		}
	System.out.println("exiting child thread");
}
}

public class ThreadWithRunnable
{
public static void main(String[] args)
{
	new ThreadWithRunnable1();

		try {
			for(int i=5; i>0; i--)
			{
				System.out.println("main thread "+i);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			System.out.println("main thread");
		}
		System.out.println("exiting main thread");
}
}
Output:
new thread Thread[Demo thread,5,main]
main thread 5
child thread 5
main thread 4
child thread 4
main thread 3
child thread 3
child thread 2
main thread 2
child thread 1
main thread 1
exiting main thread
exiting child thread
```

- Example 3 : My program

```
package multiThreading;

public class RunnableThread implements Runnable
{
public static void main(String[] args)
{
	Thread t = new Thread(new RunnableThread());
	t.start();
	//or
	//new Thread(new RunnableThread()).start();
}

public void run()
{
	try {
		for (int i=5; i>0; i--)
		{
			System.out.println("new thread "+i);
			Thread.sleep(1000);
		}
	} catch (InterruptedException e) {
		// TODO Auto-generated catch block
		System.out.println("new thread catch");
	}
	System.out.println("exiting new thread");
}
}
```

```
Output:
new thread 5
new thread 4
new thread 3
new thread 2
new thread 1
exiting new thread
```

Note: Above we can see we can have run method both inside and outside main method.

- Extending Thread Class :
- Example 1

```
package multiThreading;

public class ThreadClassExtend extends Thread
{
public void run()
{
	try {

		for(int i=5; i>0; i--)
		{
			System.out.println("new thread class thread "+i);
			Thread.sleep(1000);
		}
	} catch (InterruptedException e)
	{
		System.out.println("thread class thread");
	}
```

```
	System.out.println("exiting thread class thread");
}

public static void main(String[] args)
{
	Thread t = new ThreadClassExtend();
	t.start();
}
}
Output:
new thread class thread 5
new thread class thread 4
new thread class thread 3
new thread class thread 2
new thread class thread 1
exiting thread class thread
```

- Note : Where run() method is called in above programme ? It is actually called through the start() method. If one even see source code for start() method he cannot find run() method called anywhere in it. Actually inside start() method start0() is called which is actually a native method. So in reality run() method will be called by JVM itself. The result is that two threads are running concurrently: the current thread (which returns from the call to the start method) and the other thread (which executes its run method). I.e. Start() method causes 'this' (i.e. the current thread) to begin execution and JVM will call run method for this thread. If thread was constructed using separate 'Runnable' run object then Runnable object's run() will be called else this method(Thread class 's run()) does nothing and returns. The subclasses of Thread should override this method. Also

- Example 2 :

```
package multiThreading;

class ThreadExtendSeperateClass1 extends Thread
{
public void run()
{
	try {
		for(int j=5; j>0; j--)
		{
			System.out.println("new thread class thread "+j);
			Thread.sleep(1000);
		}
	} catch (InterruptedException e) {
		// TODO Auto-generated catch block
		System.out.println("thread class thread");
	}
	System.out.println("exiting thread class");
}
}

public class ThreadExtendSeperateClass
{
public static void main(String[] args)
{
	Thread t = new ThreadExtendSeperateClass1();
	t.start();
```

```
	}
	}
	Output:
	new thread class thread 5
	new thread class thread 4
	new thread class thread 3
	new thread class thread 2
	new thread class thread 1
	exiting thread class
```

5. To conclude thread could be implemented in one of the two ways

```
package multiThreading;

class X implements Runnable
{
	public void run()
	{
		System.out.println("Inside runnable");
	}
}

class Y extends Thread
{
	public void run()
	{
		System.out.println("inside thread");
	}
}

public class ThreadnRunnableCheck {

	public static void main(String[] args)
	{
		Thread t1 = new Thread(new X());
		t1.start();

		Thread t2 = new Y();
		t2.start();

		t1.run();//It just a plain object not thread
	}
}
Output:
Inside runnable
inside thread
Inside runnable
```

6. Chosing an approach

Implementing Runnable should be preferred over extending Thread because

1. One is not really specializing the Thread behavior. He is just giving it something to run.
2. Interface gives clear separation between ones code and implementation of thread.
3. Implementing Runnable gives option to extend any other class.

7. Main Thread :

- Main thread can be controlled with thread object. To do so one should get reference to it by calling **currentThread()** method. Similarly other Thread methods are applicable too.
- Example

```
package multiThreading;

public class MainThreadClass
{
public static void main(String[] args)
{
	Thread t = Thread.currentThread();
	System.out.println("current thread "+t);

	t.setName("my main");

	System.out.println("main after name change "+t);

}
}
Output:
current thread Thread[main,5,main]
main after name change Thread[my main,5,main]
```

Above in the output `Thread[main,5,main]` represent in order : [Name of thread, priority, name of its group]. Thread group is a data structure that controls the state of a collection of threads as a whole. Next it can be seen that name of thread got changed to 'my main'.

8. Multiple Thread :

Multiple threads can be created in a single program.

Example

```
package multiThreading;

class ManyThreadClass1 implements Runnable
{
   String name;
   Thread t;

   ManyThreadClass1(String ThreadName)
   {
	name = ThreadName;
	t = new Thread(this, name);
	System.out.println("new thread "+t);
	t.start();
   }

   public void run()
   {
	try {
		for(int i = 5; i > 0; i--)
		{
			System.out.println(name+" "+i);
			t.sleep(1000);
		}
```

```
            } catch (InterruptedException e)
            {
                  System.out.println(name+" exception");
            }
            System.out.println(name+" exiting");
      }
}
public class ManyThreadClass
{
      public static void main(String[] args)
      {
            new ManyThreadClass1("one");
            new ManyThreadClass1("two");
            new ManyThreadClass1("three");

            try {
                  Thread.sleep(10000);
                  } catch (InterruptedException e) {

                        System.out.println("main exception");
            }
            System.out.println("main exiting");
      }
}
Output:
new thread Thread[one,5,main]
new thread Thread[two,5,main]
new thread Thread[three,5,main]
one 5
two 5
three 5
three 4
two 4
one 4
two 3
three 3
one 3
two 2
three 2
one 2
one 1
two 1
three 1
two exiting
one exiting
three exiting
main exiting
```

9. <u>isAlive() and join() methods</u> :

   - We always want main thread to finish last. In the preceding pgm it is accomplished by calling sleep() within main() with a long delay to ensure that all child threads terminate before main thread. But this is not always the solution because one cannot find out when other threads will terminate. Lets see what will happen if sleep() is not used in the above 'ManyThreadClass' programme and i=3.

     Output:

     ```
     new thread Thread[one,5,main]
     new thread Thread[two,5,main]
     new thread Thread[three,5,main]
     ```

```
main exiting
one 3
three 3
two 3
three 2
one 2
two 2
three 1
one 1
two 1
three exiting
one exiting
two exiting
```

Above it can be seen the main is exiting before child threads.

- One way to find out whether particular thread is still running is through **isAlive()** method. This method will return true if the thread on which it is called is still running else it will return false.
- One more method **join()** is very useful in this regards and when it is called on some thread lets say t1 then all the current threads will be paused till t1 will be terminated. Other thread will be terminated only after t1 got terminated.
  - Example 1 :

```
package multiThreading;

class JoinNIsAliveClass1 implements Runnable
{
  String name;
  Thread t;
  JoinNIsAliveClass1(String threadName)
  {
        name = threadName;
        t = new Thread(this, name);
        System.out.println(name+" "+t);
        t.start();
  }

  public void run()
  {
        try
        {
              for(int i=3; i>0; i--)
              {
                    System.out.println(name+" "+i);
                    Thread.sleep(1000);
              }
        } catch (InterruptedException e) {
              System.out.println(name+" exception");
        }
        System.out.println(name+" exiting");
  }
}

public class JoinNIsAliveClass
{
  public static void main(String[] args)
  {
        JoinNIsAliveClass1 jna1 = new JoinNIsAliveClass1("one");
```

```
            JoinNIsAliveClass1 jna2 = new JoinNIsAliveClass1("two");
            JoinNIsAliveClass1 jna3 = new JoinNIsAliveClass1("three");

            System.out.println("Thread one is alive "+jna1.t.isAlive());
            System.out.println("Thread two is alive "+jna2.t.isAlive());
            System.out.println("Thread three is alive "+jna3.t.isAlive());

            //wait for thread to finish
            try {
                  System.out.println("waiting for thread to finish");
                  jna1.t.join();
                  jna2.t.join();
                  jna3.t.join();

            } catch (InterruptedException e) {
                  System.out.println("main thread interrupted");
            }
            System.out.println("Thread one is alive "+jna1.t.isAlive());
            System.out.println("Thread two is alive "+jna2.t.isAlive());
            System.out.println("Thread three is alive "+jna3.t.isAlive());
            System.out.println("main thread exiting");

      }
}
Output:
one Thread[one,5,main]
two Thread[two,5,main]
three Thread[three,5,main]
Thread one is alive true
Thread two is alive true
Thread three is alive true
waiting for thread to finish
three 3
two 3
one 3
three 2
two 2
one 2
three 1
two 1
one 1
three exiting
two exiting
one exiting
Thread one is alive false
Thread two is alive false
Thread three is alive false
main thread exiting
```

- Above if one calls only join() on main thread then it will be terminated first before three child threads.

– Example 2

```
package multiThreading;

public class JoinMethodCls implements Runnable{

      String ThreadName;
```

```
	Thread t;
	JoinMethodCls(String name)
	{
		ThreadName = name;
		t = new Thread(this, name);
		System.out.println(t);
		try {
			t.start();
			t.join();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}

	public void run()
	{
		for(int i = 5; i>0; i--)
		{
			try {
				System.out.println(ThreadName+" "+i);
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		System.out.println("exiting "+ThreadName);
	}

	public static void main(String[] args)
	{
		JoinMethodCls jmc1 = new JoinMethodCls("one");
		JoinMethodCls jmc2 = new JoinMethodCls("two");
		JoinMethodCls jmc3 = new JoinMethodCls("three");

		System.out.println("exiting main");
	}
}
Output:
Thread[one,5,main]
one 5
one 4
one 3
one 2
one 1
exiting one
Thread[two,5,main]
two 5
two 4
two 3
two 2
two 1
exiting two
Thread[three,5,main]
three 5
three 4
three 3
three 2
three 1
exiting three
exiting main
```

10. Thread priorities :

- One can set the priority of a thread by using method

  final void setPriority(int level)

  The value of 'level' must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently these values are 1 and 10 respectively. The default priority can be specified using **NORMAL_PRIORITY** which is 5. These priorities are defined as static final variables within Thread.
- getPriority() can be used to get the priority of the thread.

11. Synchronization :

- When two or more thread used the same resource then one needs to ensure that resource will be used by one thread at a time. This is done through synchronization. There are two ways through which synchronization works in java. Synchronized methods(both static and non-static) and Synchronized statements(or code blocks both static and non-static).

– Lets say we have two methods inside a class

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

All objects automatically contain a single lock (also referred to as a *monitor)*. When you call any **synchronized** method, that object is locked and no other **synchronized** method of that object can be called until the first one finishes and releases the lock. For the preceding methods, if **f( )** is called for an object by one task, a different task cannot call **f( )** or **g( )** for the same object until **f( )** is completed and releases the lock. Thus, there is a single lock that is shared by all the **synchronized** methods of a particular object, and this lock can be used to prevent object memory from being written by more than one task at a time.

1. Synchronized methods :

```
package multiThreading;

class SynchWithMethod1
{
	void call(String msg)
	{
		System.out.println("[ "+msg);
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			System.out.println("SynchWithMethod1 exception");
		}
		System.out.println("]");
	}
```

```
}

class SynchWithMethod2 implements Runnable
{
	String str;
	SynchWithMethod1 sm1;
	Thread t;

	public SynchWithMethod2(SynchWithMethod1 sm, String str1)
	{
		str = str1;
		sm1 = sm;
		t = new Thread(this);
		t.start();
	}

	public void run()
	{
		sm1.call(str);
	}
}

public class SynchWithMethod
{
	public static void main(String[] args)
	{
		SynchWithMethod1 smm = new SynchWithMethod1();
		SynchWithMethod2 smm1 = new SynchWithMethod2(smm,
		"hello");
		SynchWithMethod2 smm2 = new SynchWithMethod2(smm,
		"synchranized");
		SynchWithMethod2 smm3 = new SynchWithMethod2(smm,
		"world");
		try
		{
			smm1.t.join();
			smm2.t.join();
			smm3.t.join();
		} catch (InterruptedException e) {
			System.out.println("main exception");
	}
}
}
Output:

[ hello
[ synchranized
[ world
]
]
]
```

- The expected output was the ‘msg’ should be printed inside a square bracket and a **sleep()** method occurs in between. but we can see the method **call()** allows execution to switch to another thread when sleep is called. Hence unexpected output occurs.
- Nothing stops three threads from calling a same object, same method at the same time. This is called as **Race Condition**. This means three threads racing each other to complete the method.

- To fix the problem, one should restrict access to the method one thread at a time. It is achieved through **synchronized** key word. If call() method defined in this way

```
synchronized void call(String msg)
{
```

Then output becomes

```
[ hello
]
[ world
]
[ synchranized
]
```

2. Synchronized statements :

- Sometime it may happen that the methods of a class are not synchronized and programme is written by a third party hence there is no access to actual code so that a developer can change it. In such a scenario synchronized statements or synchronized block can be used.
- Synchronized statements must be given an 'object' to synchronized upon. The preceding programme can be rewritten with synchronized statement is as shown below.
- (*) Synchronization only needed when two or more threads are dealing with the SAME object. Synchronization is not needed if two threads are dealing with two different objects.
- Example :

```
package multiThreading;

class SynchWithStatement1
{
   void call(String msg)
   {
         System.out.println("[ "+msg);
         try {
               Thread.sleep(1000);
         } catch (InterruptedException e) {
               System.out.println("SynchWithStatement1 exception");
         }
         System.out.println("]");
   }
}

class SynchWithStatement2 implements Runnable
{
   String str;
   SynchWithStatement1 sm1;
   Thread t;

   public SynchWithStatement2(SynchWithStatement1 sm, String str1)
   {
         str = str1;
         sm1 = sm;
         t = new Thread(this);
         t.start();
   }
```

```
   public void run()
   {
          synchronized(sm1)
          {
          sm1.call(str);
          }
   }
}

public class SynchWithStatement
{
   public static void main(String[] args)
   {
          SynchWithStatement1 smm = new SynchWithStatement1();
          SynchWithStatement2 smm1 = new SynchWithStatement2(smm,
          "hello");
          SynchWithStatement2 smm2 = new SynchWithStatement2(smm,
          "synchranized");
          SynchWithStatement2 smm3 = new SynchWithStatement2(smm,
          "world");

          try
          {
                smm1.t.join();
                smm2.t.join();
                smm3.t.join();
          } catch (InterruptedException e) {
                System.out.println("main exception");
   }
}
}
```

Output:

```
[ hello
]
[ world
]
[ synchranized
]
```

3. When synch work :

```
package multiThreading;

class SimpleClass
{
   synchronized void method1(String str)
   {
          System.out.println(str+" method1 starts");

          try {
                Thread.sleep(5000);
          } catch (InterruptedException e) {
                System.out.println("method1 interrupted");
```

```
            }

            System.out.println(str+" method1 completes");
      }
}

class Class1 implements Runnable
{
      String str1;
      SimpleClass sc1;
      Thread t;
      Class1(SimpleClass sc, String str)
      {
            str1 = str;
            sc1 = sc;
            t = new Thread(this);
            t.start();
      }
      public void run()
      {
            sc1.method1(str1);
      }
}

class Class2 implements Runnable
{
      String str2;
      SimpleClass sc2;
      Thread t;
      Class2(SimpleClass sc, String str)
      {
            str2 = str;
            sc2 = sc;
            t = new Thread(this);
            t.start();
      }
      public void run()
      {
            sc2.method1(str2);
      }
}

public class OneClassTwoObjSynch {

      public static void main(String[] args)
      {
            SimpleClass sc11 = new SimpleClass();
            SimpleClass sc12 = new SimpleClass();
            SimpleClass sc21 = new SimpleClass();
            SimpleClass sc22 = new SimpleClass();

            //same class, diff instance, same method instance- synch matters
            Class1 c11 = new Class1(sc11, "Class1c11SameMethodInst");
            Class1 c12 = new Class1(sc11, "Class1c12SameMethodInst");

            //same class, diff instance, separate method instance - synch
            doesnt matters
            Class1 c13 = new Class1(sc11, "Class1c13DiffMethodInst");
            Class1 c14 = new Class1(sc12, "Class1c14DiffMethodInst");
```

```
        //Diff class, diff instance, Same method instance - synch
        matters
        Class1 c15 = new Class1(sc11, "Class1c15SameMethodInst");
        Class2 c21 = new Class2(sc11, "Class2c21SameMethodInst");

        //Diff class, diff instance, diff method instance - synch doesnt
        matters
        Class1 c16 = new Class1(sc11, "Class1c16DiffMethodInst");
        Class2 c22 = new Class2(sc12, "Class2c22DiffMethodInst");
}

}
```

```
Output:
Class1c11SameMethodInst method1 starts
Class1c14DiffMethodInst method1 starts
Class1c11SameMethodInst method1 completes
Class1c16DiffMethodInst method1 starts
Class1c14DiffMethodInst method1 completes
Class2c22DiffMethodInst method1 starts
Class1c16DiffMethodInst method1 completes
Class1c13DiffMethodInst method1 starts
Class2c22DiffMethodInst method1 completes
Class1c13DiffMethodInst method1 completes
Class2c21SameMethodInst method1 starts
Class2c21SameMethodInst method1 completes
Class1c15SameMethodInst method1 starts
Class1c15SameMethodInst method1 completes
Class1c12SameMethodInst method1 starts
Class1c12SameMethodInst method1 completes
```

i. For clear understanding lets execute two cases at one go and comment out others.

```
1.//same class, diff instance, same method instance - synch matters
  Class1 c11 = new Class1(sc11, "Class1c11SameMethodInst");
  Class1 c12 = new Class1(sc11, "Class1c12SameMethodInst");
  Ouptut:
  Class1c11SameMethodInst method1 starts
  Class1c11SameMethodInst method1 completes
  Class1c12SameMethodInst method1 starts
  Class1c12SameMethodInst method1 completes

2.//same class, diff instance, separate method instance - synch doesnt
  matters
  Class1 c13 = new Class1(sc11, "Class1c13DiffMethodInst");
  Class1 c14 = new Class1(sc12, "Class1c14DiffMethodInst");
  Output:
  Class1c13DiffMethodInst method1 starts
  Class1c14DiffMethodInst method1 starts
  Class1c14DiffMethodInst method1 completes
  Class1c13DiffMethodInst method1 completes

3.//Diff class, diff instance, Same method instance - synch matters
  Class1 c15 = new Class1(sc11, "Class1c15SameMethodInst");
  Class2 c21 = new Class2(sc11, "Class2c21SameMethodInst");
  Output:
  Class1c15SameMethodInst method1 starts
  Class1c15SameMethodInst method1 completes
  Class2c21SameMethodInst method1 starts
  Class2c21SameMethodInst method1 completes
```

```
4.//Diff class, diff instance, diff method instance - synch doesnt
  matters
  Class1 c16 = new Class1(sc11, "Class1c16DiffMethodInst");
  Class2 c22 = new Class2(sc12, "Class2c22DiffMethodInst");
  Ouptut:
  Class1c16DiffMethodInst method1 starts
  Class2c22DiffMethodInst method1 starts
  Class1c16DiffMethodInst method1 completes
  Class2c22DiffMethodInst method1 completes
```

ii. This clearly explains Synchronization is only needed when two or more threads are working on the **same object** at a same time. Ex : Two instances of same class c11-12 or two instances of diff class c15-c21 works on same object sc11 through two different threads Synchronization is needed.

iii. When two objects of the same class c13-14 or diff class c16-c22 works on two different instance at a same time sc11, sc12 resp. then synchronization is not needed.

11. Interthread Communication through **wait()** and **notify()** :

- In many occasions we will find that there is a dependency of one thread on another. For example simple producer and consumer classes. There is a int value N where producer writes and consumer reads. We know that thread works mainly according to processors multithreading settings. It may happen that producer may be writing the integer value but consumer is not reading it and when consumer reads it will read only the latest value and haven't read the previous ones. This happens because there is no proper communication between these two threads. The output produced through this is shown below with an example.

- Example :

```
package multiThreading;

class Q
{
 int n;

 synchronized void put(int n)
 {

       this.n = n;
       try {
             Thread.sleep(1000);
       } catch (InterruptedException e) {
             System.out.println("inside put thread");
       }
       System.out.println("put: "+n);
 }

 synchronized int get()
 {
       System.out.println("get: "+n);
       return n;
 }
}

class Producer implements Runnable
{
```

```
 Q q;
 Producer(Q q)
 {
       this.q = q;
       new Thread(this, "producer").start();
 }
 public void run()
 {
       for(int i=0; i<5; i++)
       {
             q.put(i);
       }
 }
}

class Consumer implements Runnable
{
 Q q;
 Consumer(Q q)
 {
       this.q = q;
       new Thread(this, "consumer").start();
 }

 public void run()
 {
       while(q.get()< 4)
       {
             q.get();

       }
 }
}

public class ProblemWithoutWaitNNotify
{
 public static void main(String[] args)
 {
       Q q = new Q();
       new Producer(q);
       new Consumer(q);
 }
}
Output:
put: 0
put: 1
get: 1
get: 1
get: 1
get: 1
get: 1
get: 1
get: 1
get: 1
put: 2
put: 3
put: 4
get: 4
get: 4
```

```
Note : Above output may very from machine to machine depends on its
multithreading workings.
```

As seen above in the output when 'put' was 0, 'get' was not reading it. But when put became 1 get started reading it many times because put went to sleep. But again when put was 2 and after 3, get was not reading it. In programme we can see the condition while(q.get()<4) has been used but still get:4 got executed. This is not because of while loop was not working but when q.get was 3 get started reading the integer value but meanwhile it got updated with 4 by put.

- wait() and notify() method:

  wait():

```
public final void wait()throws InterruptedException
```

  Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object. In other words, this method behaves exactly as if it simply performs the call wait(0).

  The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the notify method or the notifyAll method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

  As in the one argument version, interrupts and spurious wakeups are possible, and this method should always be used in a loop:

```
synchronized (obj)
{
    while (<condition does not hold>)
        obj.wait();
    ... // Perform action appropriate to condition
}
```

  This method should only be called by a thread that is the owner of this object's monitor. See the notify method for a description of the ways in which a thread can become the owner of a monitor.

  IllegalMonitorStateException - if the current thread is not the owner of the object's monitor.
  InterruptedException - if any thread interrupted the current thread before or while the current thread was waiting for a notification. The *interrupted status* of the current thread is cleared when this exception is thrown.

  notify() :

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.

The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

- By executing a synchronized instance method of that object.
- By executing the body of a synchronized statement that synchronizes on the object.
- For objects of type Class, by executing a synchronized static method of that class.

Only one thread at a time can own an object's monitor.

- Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition which must be true before the block can proceed. It is resolved by using wait() and notify() or notifyAll() methods. Both of these methods are part of Object class. wait() suspends the current thread and the invocation of wait doesn't return until another thread has issued a notification.
- Note: Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.
- Shown below in improved version of preceding programme.
- Example 1 :

```
package multiThreading;

class Q1
{
 int n;
 boolean valueSet = false;

 synchronized void put(int n)
 {
       while(valueSet)
             try {
                   wait();
             } catch (InterruptedException e) {
                   System.out.println("inside put wait");
             }
       this.n = n;
       System.out.println("put: "+n);
       valueSet = true;
       notify();
```

```
 }

 synchronized int get()
 {
       while(!valueSet)
       try {
             wait();
       } catch (InterruptedException e) {
             System.out.println("inside get wait");
       }
       System.out.println("get: "+n);
       valueSet = false;
       notify();
       return n;
 }
}

class Producer1 implements Runnable
{
 Q1 q;
 Producer1(Q1 q)
 {
       this.q = q;
       new Thread(this, "producer1").start();
 }
 public void run()
 {
       for(int i=0; i<5; i++)
       {
             q.put(i);
       }
 }
}

class Consumer1 implements Runnable
{
 Q1 q;
 Consumer1(Q1 q)
 {
       this.q = q;
       new Thread(this, "consumer1").start();
 }

 public void run()
 {
       while(q.get()< 4)
       {
             q.get();
       }
 }
}

public class SolutionWithWaitNNotify
{
 public static void main(String[] args)
 {
       Q1 q = new Q1();
       new Producer1(q);
       new Consumer1(q);
 }
}
```

```
Output:
put: 0
get: 0
put: 1
get: 1
put: 2
get: 2
put: 3
get: 3
put: 4
get: 4
```

- It is very necessary to understand how the above program is working. When important point is when put() or get(), get into while loop and execute wait() then that thread got suspended and all instructions coming after while loop will not going to execute. Similarly when wait() is not invoked then only instructions written after while loop will be executed.

– Example 2:

```
package testProg;
public class ThreadA {
    public static void main(String[] args){
        ThreadB b = new ThreadB();
        b.start();

        synchronized(b){
           try
           {
                System.out.println("Waiting for b to complete...");
                b.wait();
           }catch(InterruptedException e)
           {
                e.printStackTrace();
           }
             System.out.println("Total is: " + b.total);
        }
    }
}

class ThreadB extends Thread{
    int total;
    @Override
    public void run(){
        synchronized(this){
            for(int i=0; i<100 ; i++){
                total += i;
            }
            notify();
        }
    }
}
Output:
Waiting for b to complete...
Total is: 4950
```

– If One removes wait() method from above programme then output would be

```
Output:
Waiting for b to complete...
Total is: 0
```

- One should keep in mind that wait and notify should always be used inside a synchronized block(or method) otherwise an `IllegalMonitorStateException` will be thrown. Why this condition has been created could be explained by following example.

Let's illustrate what issues we would run into if `wait()` could be called outside of a synchronized block with a **concrete example**.

Suppose we were to implement a blocking queue (I know, there is already one in the API :)

A first attempt (without synchronization) could look something along the lines below

```
class BlockingQueue {
    Queue<String> buffer = new LinkedList<String>();

    public void give(String data) {
        buffer.add(data);
        notify();                    // Since someone may be waiting in
take!
    }

    public String take() throws InterruptedException {
        while (buffer.isEmpty())     // don't use "if" due to spurious
wakeups.
            wait();
        return buffer.remove();
    }
}
```

This is what could potentially happen:

1. A consumer thread calls `take()` and sees that the `buffer.isEmpty()`.
2. Before the consumer thread goes on to call wait(), a producer thread comes along and invokes a full `give()`, that is, `buffer.add(data); notify();`
3. The consumer thread will now call `wait()` (and *miss* the `notify()` that was just called).
4. If unlucky, the producer thread won't produce more `give()` as a result of the fact that the consumer thread never wakes up, and we have a dead-lock.

Once you understand the issue, the solution is obvious: Always perform `give/notify` and `isEmpty/wait` atomically.

Without going into details: This synchronization issue is universal. w`ait/notify` is all about communication between threads, so you'll always end up with a race condition similar to the one described above. This is why the "only wait inside synchronized" rule is enforced.

12. DeadLock :

- Deadlock occurs when two threads have a circular dependency on a pair of synchronized objects.
- Deadlock mainly occurs due to synchronized methods are present and a thread accessing one synchronized method holds all other synchronized methods of same object for other thread as shown in the example below.
- Example:

```
package multiThreading;

class A
{
   synchronized void foo(B b)
   {
         String name = Thread.currentThread().getName();
         System.out.println(name+" entered A.foo");

         try {
               Thread.sleep(1000);
         } catch (InterruptedException e) {
               System.out.println("A Interrupted");
         }
         System.out.println(name+" trying to call B's last");
         b.last();
   }

   synchronized void last()
   {
         System.out.println("inside A.last");
   }
}

class B
{
   synchronized void bar(A a)
   {
         String name = Thread.currentThread().getName();
         System.out.println(name+" entered B.bar");
         try {
               Thread.sleep(1000);
         } catch (InterruptedException e) {
               System.out.println("B Interrupted");
         }
         System.out.println(name+" trying to call A's last");
         a.last();
   }

   synchronized void last()
   {
         System.out.println("inside B.last");
   }
}
public class DeadLockPgm implements Runnable
{
   A a = new A();
```

```
    B b = new B();

    DeadLockPgm()
    {
          Thread.currentThread().setName("MainThread");
          Thread t = new Thread(this, "Racing Thread");
          t.start();
          a.foo(b);
          System.out.println("back in main thread");
    }
    public void run()
    {
          b.bar(a);
          System.out.println("back in other thread");
    }

    public static void main(String[] args)
    {
          new DeadLockPgm();
    }
}
Output:
MainThread entered A.foo
Racing Thread entered B.bar
Racing Thread trying to call A's last
MainThread trying to call B's last
```

- Above programme never ends and A's last and B's last will be never called because of deadlock.

13. Suspending, Resuming and Stopping Threads :

- Sometime it is necessary to suspended(or paused) a thread and suspended thread can be resumed at any time. In **java 1.0** both of these tasks are achieved through **suspend()** and **resume()** method defined in thread class. Now both of these methods have been deprecated(will be scratched below in the pgm). But it is also necessary to know them as one may need to work on legacy classes for some old codes. Example for them is shown below.
- Example :

```
package multiThreading;

class SuspendResumeClass1 implements Runnable
{
    String str;
    Thread t;
    SuspendResumeClass1(String name)
    {
          str = name;
        t = new Thread(this, str);
        System.out.println("new thread :"+t);
        t.start();
    }
    public void run()
    {
          try {
                for(int i = 15; i>0 ; i--)
                {
                      System.out.println(str+" :"+i);
                      Thread.sleep(200);
```

```
                }
            } catch (InterruptedException e) {

                System.out.println(str+" interrupted");
            }
            System.out.println(str+" exiting");
    }
}

public class SuspendResumeClass
{
    public static void main(String[] args)
    {
            SuspendResumeClass1 src1 = new SuspendResumeClass1("one");
            SuspendResumeClass1 src2 = new SuspendResumeClass1("two");

            try {
                Thread.sleep(1000);
                src1.t.suspend();
                System.out.println("Suspending thread one");
                Thread.sleep(1000);
                src1.t.resume();
                System.out.println("resuming thread one");
                src2.t.suspend();
                System.out.println("Suspending thread two");
                Thread.sleep(1000);
                src2.t.resume();
                System.out.println("resuming thread two");

            } catch (InterruptedException e) {
                System.out.println("main interrupted");
            }
            try {
                System.out.println("wating for thread to finish");
                src1.t.join();
                src2.t.join();
            } catch (InterruptedException e) {
                System.out.println("main interrupted");
            }
            System.out.println("exiting main");
    }
}
Output:
new thread :Thread[one,5,main]
new thread :Thread[two,5,main]
one :15
two :15
one :14
two :14
one :13
two :13
one :12
two :12
one :11
two :11
one :10
Suspending thread one
two :10
two :9
two :8
two :7
```

```
two :6
resuming thread one
Suspending thread two
one :9
one :8
one :7
one :6
one :5
one :4
two :5
resuming thread two
wating for thread to finish
two :4
one :3
one :2
two :3
one :1
two :2
two :1
one exiting
two exiting
exiting main
```

- A thread can be stopped using method stop() and once stopped then thread cannot be resumed again.
- Example 2:

```
public static Object cacheLock = new Object();
  public static Object tableLock = new Object();
  ...
  public void oneMethod() {
    synchronized (cacheLock) {
      synchronized (tableLock) {
        doSomething();
      }
    }
  }
  public void anotherMethod() {
    synchronized (tableLock) {
      synchronized (cacheLock) {
        doSomethingElse();
      }
    }
  }
```

Now, imagine that thread A calls oneMethod() while thread B simultaneously callsanotherMethod(). Imagine further that thread A acquires the lock on cacheLock, and, at the same time, thread B acquires the lock on tableLock. Now the threads are deadlocked: neither thread will give up its lock until it acquires the other lock, but neither will be able to acquire the other lock until the other thread gives it up. When a Java program deadlocks, the deadlocking threads simply wait forever. While other threads might continue running, you will eventually have to kill the program, restart it, and hope that it doesn't deadlock again.

- After Java 2.0 :

  After java 2.0 it has been found that method suspend() can cause serious system failures. For example if a thread is accessing critical synchronized data structure and if it has been suspended then the lock for the method dont get released and other threads waiting for the methods get deadlocked. Method resume() has been deprecated because it is only useful with suspend().

- Method stop() too has been deprecated because it is too can cause problems. Lets say for example a thread is writing something to a critical synchronized data structure and in between it has been stopped. It will make it unlock but the other thread who are going to accessing the same data structure get corrupted version of it because of half work done by the previous thread.
- All suspend(), resume() and stop() has been deprecated in java 2.0.
- In place of them Object methods wait() and notify() will be used for the same purpose. Below programme explains how.

```
package multiThreading;

class SuspendResumeWithWaitNNotify1 implements Runnable
{
   String str;
   Thread t;
   boolean SuspendFlag;
   SuspendResumeWithWaitNNotify1(String name)
   {
         str = name;
       t = new Thread(this, str);
       System.out.println("new thread :"+t);
       SuspendFlag = false;
       t.start();
   }

   public void run()
   {
         try {
               for(int i = 15; i>0 ; i--)
               {
                     System.out.println(str+" :"+i);
                     Thread.sleep(200);
                     synchronized(this)
                     {
                           while(SuspendFlag)
                           {
                                 wait();
                           }
                     }
               }
         } catch (InterruptedException e) {

               System.out.println(str+" interrupted");
         }
         System.out.println(str+" exiting");
   }

   synchronized void mySuspend()
   {
         SuspendFlag = true;
   }

   synchronized void myResume()
   {
         SuspendFlag = false;
         notify();
   }
```

```
}

public class SuspendResumeWithWaitNNotify
{
   public static void main(String[] args)
   {
          SuspendResumeWithWaitNNotify1 src1 = new
SuspendResumeWithWaitNNotify1("one");
          SuspendResumeWithWaitNNotify1 src2 = new
SuspendResumeWithWaitNNotify1("two");

          try {
                Thread.sleep(1000);
                src1.mySuspend();
                System.out.println("Suspending thread one");
                Thread.sleep(1000);
                src1.myResume();
                System.out.println("resuming thread one");
                src2.mySuspend();
                System.out.println("Suspending thread two");
                Thread.sleep(1000);
                src2.myResume();
                System.out.println("resuming thread two");
                } catch (InterruptedException e) {
                System.out.println("main interrupted");
          }
          try {
                System.out.println("wating for thread to finish");
                src1.t.join();
                src2.t.join();
          } catch (InterruptedException e) {
                System.out.println("main interrupted");
          }
          System.out.println("exiting main");
   }
}
Output:
new thread :Thread[one,5,main]
new thread :Thread[two,5,main]
one :15
two :15
one :14
two :14
one :13
two :13
one :12
two :12
one :11
two :11
one :10
Suspending thread one
two :10
two :9
two :8
two :7
two :6
resuming thread one
Suspending thread two
one :9
one :8
one :7
```

```
one :6
one :5
resuming thread two
wating for thread to finish
two :5
one :4
two :4
one :3
two :3
one :2
two :2
one :1
two :1
one exiting
two exiting
exiting main
```

14. Obtaining Thread State's :

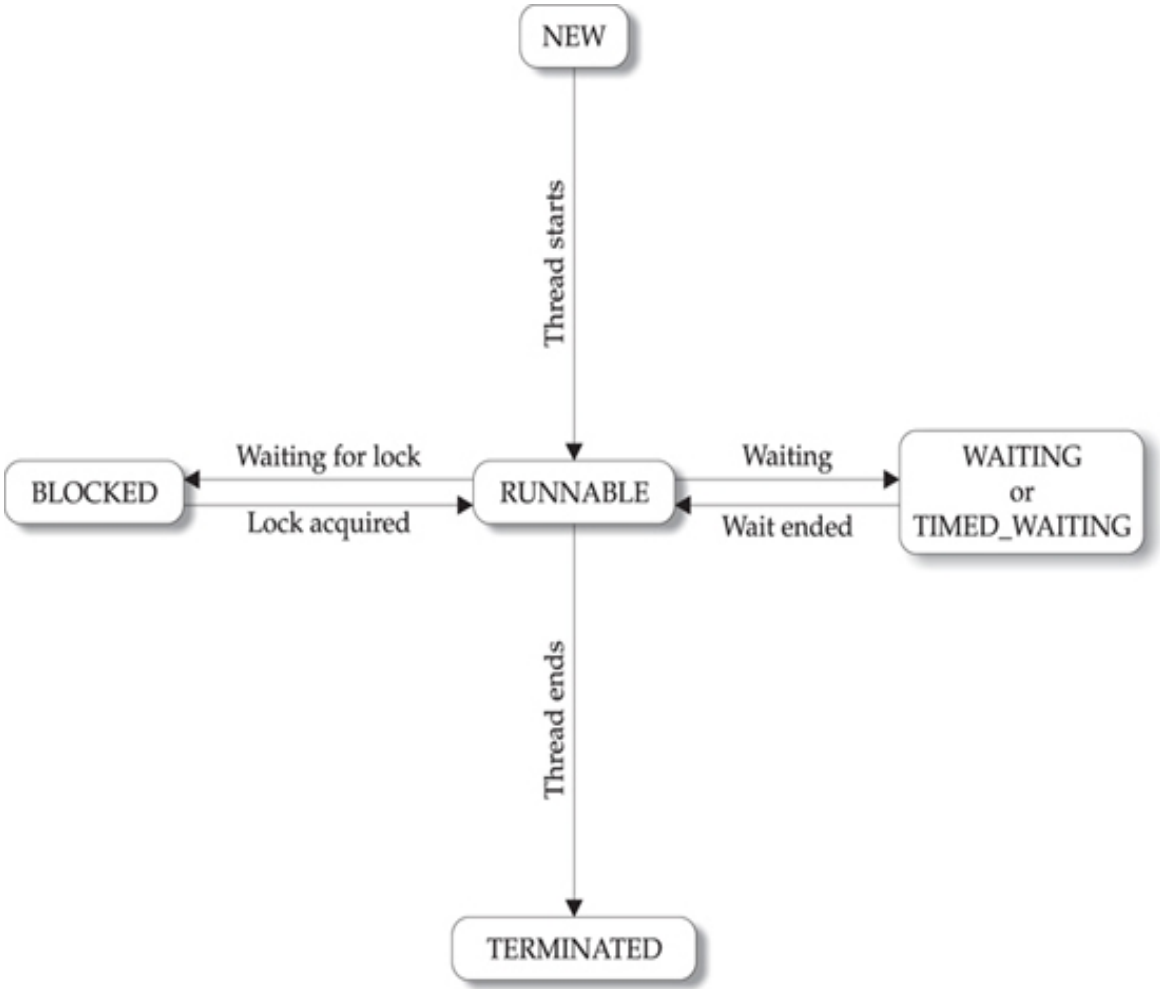
- A thread can exist in different States. One can get the current state of thread using **getState()** method defined by Thread. It is show here

  Thread.State getState()

- It returns a value of type Thread.State which indicate state of thread at the time the call was made.

  State is an enumeration defined by Thread. Below are values returned by getState().

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.



- Below example shows state of thread at different stages.
- Example :

```
package multiThreading;

class MethodClass1
{
   synchronized void method1()
   {
         System.out.println("inside method class");
   }
}

class ThreadStateClass1 implements Runnable
{
   MethodClass1 mc;
   String str;
   Thread t;


   ThreadStateClass1(MethodClass1 mc1, String name)
   {
         mc = mc1;
         str = name;
```

```
        t = new Thread(this, str);
        t.start();
        System.out.println("after start "+t.getName()+"
        :"+t.getState());
        System.out.println("after start "+t.getName()
        +" :"+t.getState());
   }
   public void run()
   {

        mc.method1();
        try {
              Thread.sleep(1000);
            } catch (InterruptedException e) {
                        System.out.println(str+" interrupted");
            }
        System.out.println("after run "+t.getName()+" :"+t.getState());
        System.out.println("after run "+t.getName()+" :"+t.getState());

   }
}
public class ThreadStateClass
{
   public static void main(String[] args)
   {
        MethodClass1 mc = new MethodClass1();
        ThreadStateClass1 tsc1 = new ThreadStateClass1(mc, "one");
        ThreadStateClass1 tsc2 = new ThreadStateClass1(mc, "two");

        System.out.println("from main "+tsc1.t.getName()
        +" :"+tsc1.t.getState());
        System.out.println("from main "+tsc2.t.getName()
        +" :"+tsc2.t.getState());

        try {
              tsc1.t.join();
              tsc2.t.join();
              System.out.println("after join "+tsc1.t.getName()
              +" :"+tsc1.t.getState());
              System.out.println("after join "+tsc2.t.getName()
              +" :"+tsc2.t.getState());
        } catch (InterruptedException e) {
              System.out.println("main interrupted");
        }
        System.out.println("end of main "+tsc1.t.getName()
        +" :"+tsc1.t.getState());
        System.out.println("end of main "+tsc2.t.getName()
        +" :"+tsc2.t.getState());
   }

}
Output:
after start one :RUNNABLE
inside method class
after start one :RUNNABLE
after start two :RUNNABLE
after start two :RUNNABLE
from main one :TIMED_WAITING
from main two :RUNNABLE
inside method class
after run two :RUNNABLE
```

```
after run one :RUNNABLE
after run two :RUNNABLE
after run one :RUNNABLE
after join one :TERMINATED
after join two :TERMINATED
end of main one :TERMINATED
end of main two :TERMINATED
```