Multi Threading

Agenda

1. Introduction.
2. The ways to define, instantiate and start a new Thread.
1. By extending Thread class
2. By implementing Runnable interface
3. Thread priority
4. Getting and setting name of a Thread.
5. The methods to prevent(stop) Thread execution.
1. yield()
2. join()
3. sleep()
6. Synchronization.
7. Inter Thread communication.
8. Deadlock
9. Daemon Threads.
10. Life cycle of a Thread

Multitasking: Executing several tasks simultaneously is the concept of multitasking.

There are two types of multitasking's.

1. Process based multitasking.
2. Thread based multitasking.

Diagram:

Process based multitasking:

Executing several tasks simultaneously where each task is a separate independent

process such type of multitasking is called process based multitasking.

Example:

 While typing a java program in the editor we can able to listen mp3 audio songs

at the same time we can download a file from the net all these tasks are independent of each other and executing simultaneously and hence it is Process

based multitasking.

 This type of multitasking is best suitable at "os level".

Thread based multitasking:

Executing several tasks simultaneously where each task is a separate independent part

of the same program, is called Thread based multitasking.

And each independent part is called a "Thread".

1. This type of multitasking is best suitable for "programatic level".

2. When compared with "C++", developing multithreading examples is very easy

in java because java provides in built support for multithreading through a rich API (Thread, Runnable,...etc).

3. In multithreading on 10% of the work the programmer is required to do and 90% of the work will be down by java API.

4. *The main important application areas of multithreading are:*

1. To implement multimedia graphics.

2. To develop animations.

3. To develop video games etc.

4. To develop web and application servers

5. Whether it is process based or Thread based the main objective of multitasking

is to improve performance of the system by reducing response time.

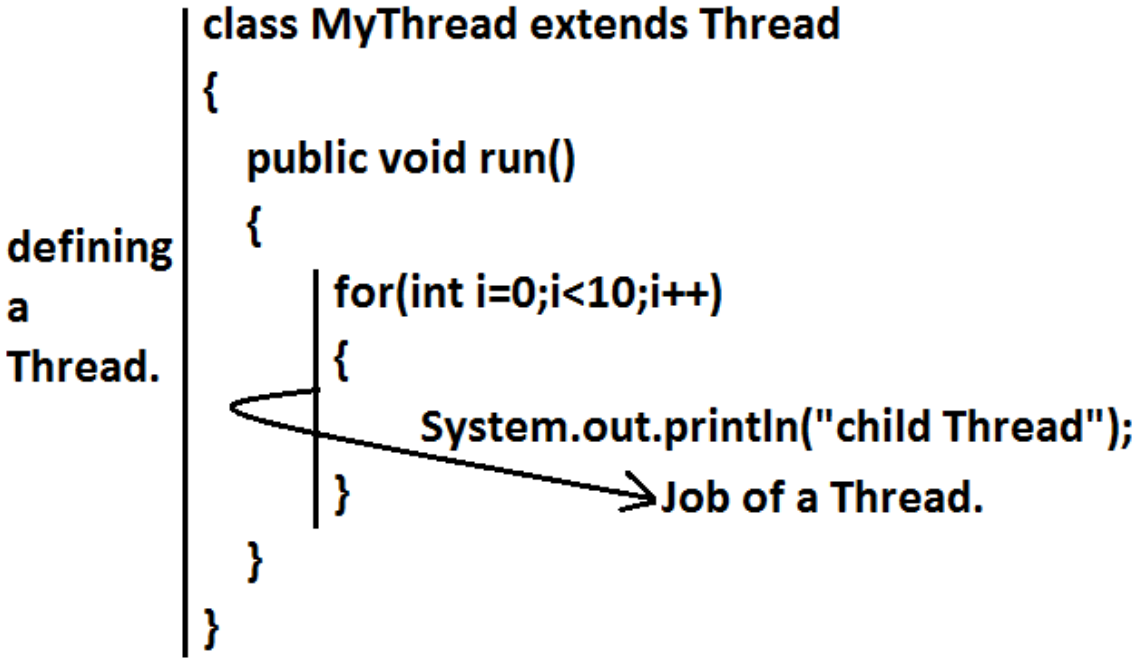
We can define a Thread in the following 2 ways.

1. By extending Thread class.

2. By implementing Runnable interface.

Defining a Thread by extending "Thread class"

Example:



```
class ThreadDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();//Instantiation of a Thread
t.start();//starting of a Thread
for(int i=0;i<5;i++)
{
System.out.println("main thread");
}
}
}
```

Case 1: Thread Scheduler:

 If multiple Threads are waiting to execute then which Thread will execute 1st is
decided by "Thread Scheduler" which is part of JVM.

 Which algorithm or behavior followed by Thread Scheduler we can't expect
exactly it is the JVM vendor dependent hence in multithreading examples we
can't expect exact execution order and exact output.

 The following are various possible outputs for the above program.

p1	p2	p3	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
main thread	main thread	main thread	
child thread	child thread	child thread	
child thread	child thread	child thread	----------
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	
child thread	child thread	child thread	

Case 2: Difference between t.start() and t.run() methods.

 In the 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() no new Thread will be created and run() method will be
executed just like a normal method by the main Thread.

 In the above program if we are replacing t.start() with t.run() the following is the
output.

Output:

child thread

child thread

child thread

child thread

child thread

child thread

child thread

child thread

child thread

child thread

main thread

main thread

main thread

main thread

main thread

Entire output produced by only main Thread.

Case 3: importance of Thread class start() method.

For every Thread the required mandatory activities like registering the Thread with

Thread Scheduler will takes care by Thread class start() method and programmer is

responsible just to define the job of the Thread inside run() method.

That is start() method acts as best assistant to the programmer.

Example:

start()

{

1. Register Thread with Thread Scheduler

2. All other mandatory low level activities.

3. Invoke or calling run() method.

}

We can conclude that without executing Thread class start() method there is no chance of starting a new Thread in java. Due to this start() is considered as heart of multithreading.

Case 4: If we are not overriding run() method:

If we are not overriding run() method then Thread class run() method will be executed.

which has empty implementation and hence we won't get any output.

Example:

```
class MyThread extends Thread
{}
class ThreadDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();
t.start();
}
}
```

It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

Case 5: Overloding of run() method.

We can overload run() method but Thread class start() method always invokes no

argument run() method the other overload run() methods we have to call explicitly then

only it will be executed just like normal method.

Example:

```
class MyThread extends Thread
{
public void run()
{
System.out.println("no arg method");
}
public void run(int i)
{
System.out.println("int arg method");
}
}

class ThreadDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();
t.start();
}
}
```

Output:

No arg method

Case 6: overriding of start() method:

If we override start() method then our start() method will be executed just like a normal

method call and no new Thread will be started.

Example:

```
class MyThread extends Thread
{
public void start()
{
System.out.println("start method");
}
public void run()
{
System.out.println("run method");
}
}
class ThreadDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();
t.start();
System.out.println("main method");
}
}
```

Output:

start method

main method

Entire output produced by only main Thread.

Note : It is never recommended to override start() method.

Case 7:

Example 1:

```
class MyThread extends Thread
{
    public void start()
    {
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
```

```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        System.out.println("main method");
    }
}
```

output:

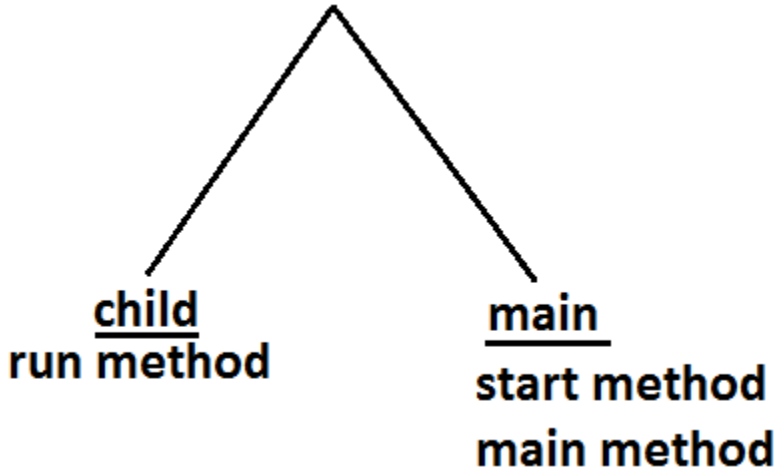
```
main thread
start method
main method
```

Example 2:

```
class MyThread extends Thread
{
    public void start()
    {
        super.start();
        System.out.println("start method");
    }
    public void run()
    {
        System.out.println("run method");
    }
}
```

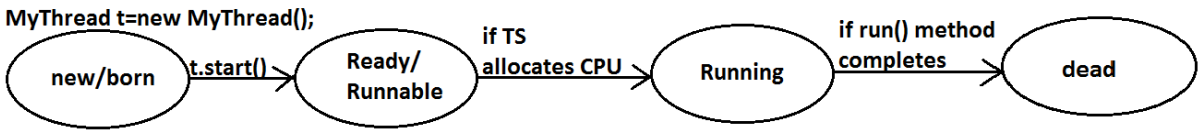
```
class ThreadDemo
{
    public static void main(String[] args)
    {
        MyThread t=new MyThread();
        t.start();
        System.out.println("main method");
    }
}
```

Output:



Case 8: life cycle of the Thread:

Diagram:



 Once we created a Thread object then the Thread is said to be in new state or
born state.
 Once we call start() method then the Thread will be entered into Ready or
Runnable state.
 If Thread Scheduler allocates CPU then the Thread will be entered into running
state.
 Once run() method completes then the Thread will entered into dead state.

Case 9:

After starting a Thread we are not allowed to restart the same Thread once again
otherwise we will get runtime exception saying
"IllegalThreadStateException".

Example:

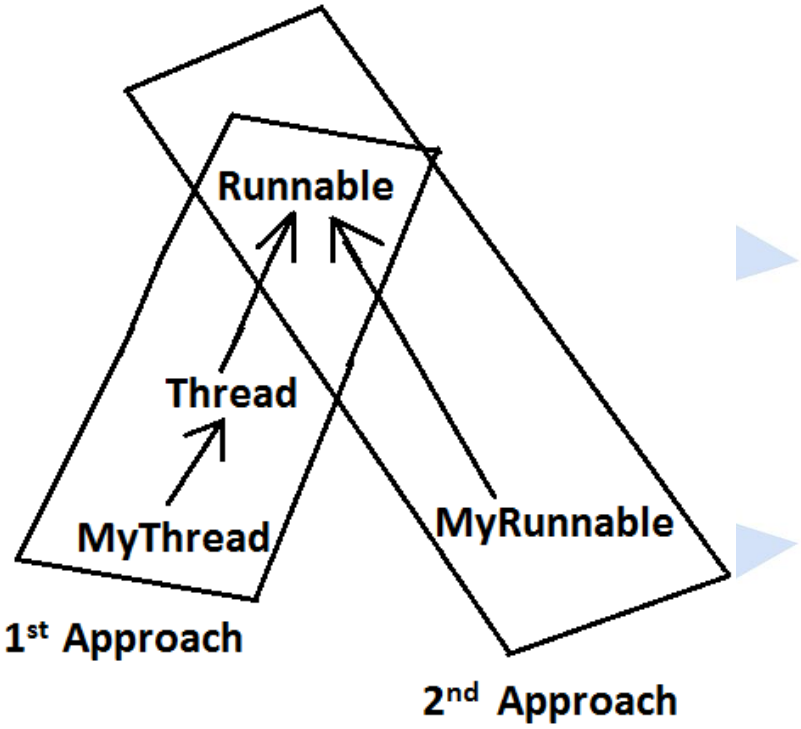
```
MyThread t=new MyThread();
t.start();//valid
;;;;;;;;
t.start();//we will get R.E saying: IllegalThreadStateException
```

Defining a Thread by implementing Runnable interface:

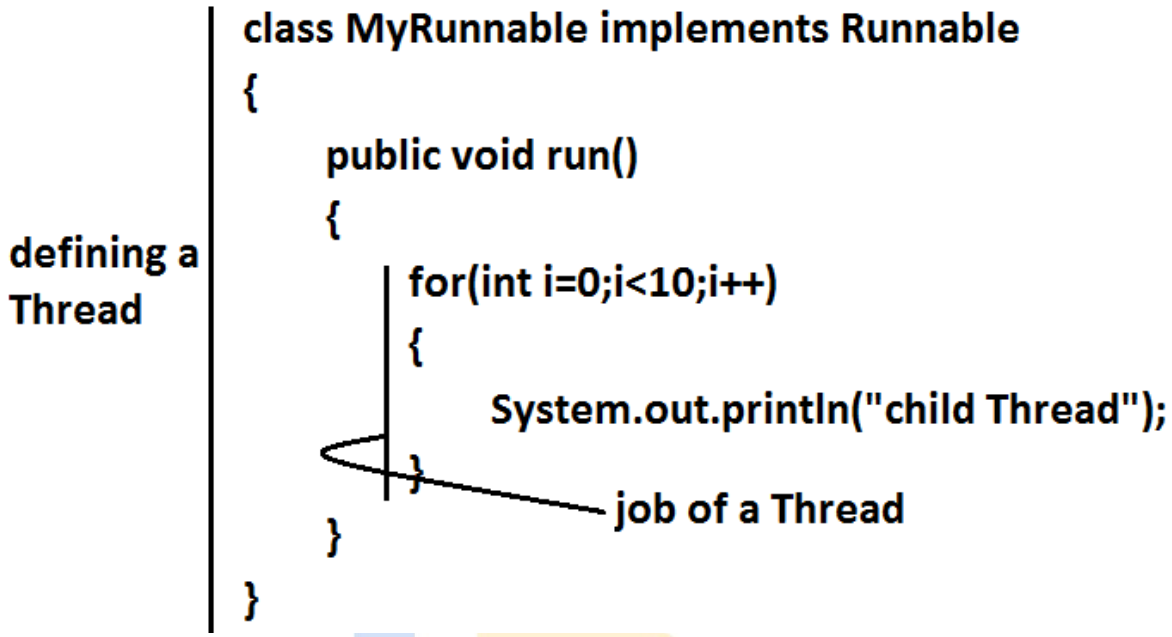
We can define a Thread even by implementing Runnable interface also.

Runnable interface present in java.lang.pkg and contains only one method run().

Diagram:



Example:




```
class ThreadDemo
{
public static void main(String[] args)
{
MyRunnable r=new MyRunnable();
Thread t=new Thread(r);//here r is a Target Runnable
t.start();
for(int i=0;i<10;i++)
{
System.out.println("main thread");
}
}
}
```

Output:

main thread

main thread

main thread

main thread

main thread

main thread

main thread

main thread

main thread

main thread

child Thread

child Thread

child Thread

child Thread

child Thread

child Thread

child Thread

child Thread

child Thread

child Thread

We can't expect exact output but there are several possible outputs.

Case study:

MyRunnable r=new MyRunnable();

Thread t1=new Thread();

Thread t2=new Thread(r);

Case 1: t1.start():

A new Thread will be created which is responsible for the execution of Thread class

run()method.

Output:

main thread

main thread

main thread

main thread

main thread

Case 2: t1.run():

No new Thread will be created but Thread class run() method will be executed just like

a normal method call.

Output:

main thread

main thread

main thread

main thread

main thread

Case 3: t2.start():

New Thread will be created which is responsible for the execution of MyRunnable run()

method.

Output:

main thread

main thread

main thread

main thread

main thread

child Thread

child Thread

child Thread

child Thread

child Thread

Case 4: t2.run():

No new Thread will be created and MyRunnable run() method will be executed just like

a normal method call.

Output:

child Thread

child Thread

child Thread

child Thread

child Thread

main thread

main thread

main thread

main thread

main thread

Case 5: r.start():

We will get compile time error saying start()method is not available in MyRunnable

class.

Output:

Compile time error

E:\SCJP>javac ThreadDemo.java

ThreadDemo.java:18: cannot find symbol

Symbol: method start()

Location: class MyRunnable

Case 6: r.run():

No new Thread will be created and MyRunnable class run() method will be executed

just like a normal method call.

Output:

child Thread

child Thread

child Thread

child Thread

child Thread

main thread

main thread

main thread

main thread

main thread

In which of the above cases a new Thread will be created which is responsible for the

execution of MyRunnable run() method ?

t2.start();

In which of the above cases a new Thread will be created ?

t1.start();

t2.start();

In which of the above cases MyRunnable class run() will be executed ?

t2.start();

t2.run();

r.run();

Best approach to define a Thread:

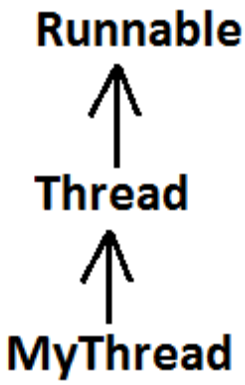
 Among the 2 ways of defining a Thread, implements Runnable approach is
always recommended.

 In the 1st approach our class should always extends Thread class there is no
chance of extending any other class hence we are missing the benefits of inheritance.

 But in the 2nd approach while implementing Runnable interface we can extend
some other class also. Hence implements Runnable mechanism is recommended
to define a Thread.

1. Thread t=new Thread();

Diagram:



Output:

main method

run method

Getting and setting name of a Thread:

 Every Thread in java has some name it may be provided explicitly by the programmer or automatically generated by JVM.

 Thread class defines the following methods to get and set name of a Thread.

Methods:

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

Example:

```
class MyThread extends Thread
{}
class ThreadDemo
{
public static void main(String[] args)
{
System.out.println(Thread.currentThread().getName());//main
MyThread t=new MyThread();
System.out.println(t.getName());//Thread-0
Thread.currentThread().setName("Bhaskar Thread");
System.out.println(Thread.currentThread().getName());//Bhaskar Thread
}
}
```

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

Thread Priorities

 Every Thread in java has some priority it may be default priority generated by JVM (or) explicitly provided by the programmer.

 The valid range of Thread priorities is 1 to 10[but not 0 to 10] where 1 is the least priority and 10 is highest priority.

 Thread class defines the following constants to represent some standard priorities.

1. Thread. MIN_PRIORITY----------1
2. Thread. MAX_PRIORITY----------10
3. Thread. NORM_PRIORITY--------5

 There are no constants like Thread.LOW_PRIORITY, Thread.HIGH_PRIORITY

 Thread scheduler uses these priorities while allocating CPU.

 The Thread which is having highest priority will get chance for 1st execution.

 If 2 Threads having the same priority then we can't expect exact execution order
it depends on Thread scheduler whose behavior is vendor dependent.

 We can get and set the priority of a Thread by using the following methods.

1. public final int getPriority()
2. public final void setPriority(int newPriority);//the allowed values are 1 to
10

 The allowed values are 1 to 10 otherwise we will get runtime exception saying
"IllegalArgumentException".

Default priority:

The default priority only for the main Thread is 5. But for all the remaining Threads

the default priority will be inheriting from parent to child. That is whatever the priority

parent has by default the same priority will be for the child also.

Example 1:

```
class MyThread extends Thread
{}
class ThreadPriorityDemo
{
public static void main(String[] args)
{
System.out.println(Thread.currentThread().getPriority());//5
Thread.currentThread().setPriority(9);
MyThread t=new MyThread();
System.out.println(t.getPriority());//9
}
}
```

Example 2:

```
class MyThread extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
System.out.println("child thread");
}
}
}
```

```
class ThreadPriorityDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();
//t.setPriority(10); //----> 1
t.start();
for(int i=0;i<10;i++)
{
System.out.println("main thread");
}
}
}
```

 If we are commenting line 1 then both main and child Threads will have the
same priority and hence we can't expect exact execution order.

 If we are not commenting line 1 then child Thread has the priority 10 and main
Thread has the priority 5 hence child Thread will get chance for execution and
after completing child Thread main Thread will get the chance in this the output
is:

Output:

```
child thread
child thread
child thread
child thread
```

child thread
child thread
child thread
child thread
child thread
child thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread
main thread

Some operating systems(like windowsXP) may not provide proper support for Thread
priorities. We have to install separate bats provided by vendor to provide support for
priorities.

The Methods to Prevent (Stop) Thread Execution:

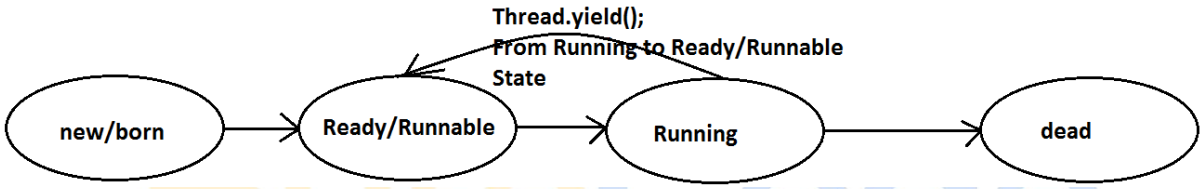
We can prevent(stop) a Thread execution by using the following methods.

1. yield();
2. join();
3. sleep();

yield():

1. yield() method causes "to pause current executing Thread for giving the chance
of remaining waiting Threads of same priority".
2. If all waiting Threads have the low priority or if there is no waiting Threads then
the same Thread will be continued its execution.
3. If several waiting Threads with same priority available then we can't expect
exact which Thread will get chance for execution.
4. The Thread which is yielded when it get chance once again for execution is
depends on mercy of the Thread scheduler.
5. public static native void yield();

Diagram:




```
class MyThread extends Thread
{
public void run()
{
for(int i=0;i<5;i++)
{
Thread.yield();
System.out.println("child thread");
```

```
}
}
}
class ThreadYieldDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();
t.start();
for(int i=0;i<5;i++)
{
System.out.println("main thread");
}
}
}
```

Output:

```
main thread
main thread
main thread
main thread
main thread
child thread
child thread
child thread
child thread
child thread
```

In the above program child Thread always calling yield() method and hence main

Thread will get the chance more number of times for execution.

Hence the chance of completing the main Thread first is high.

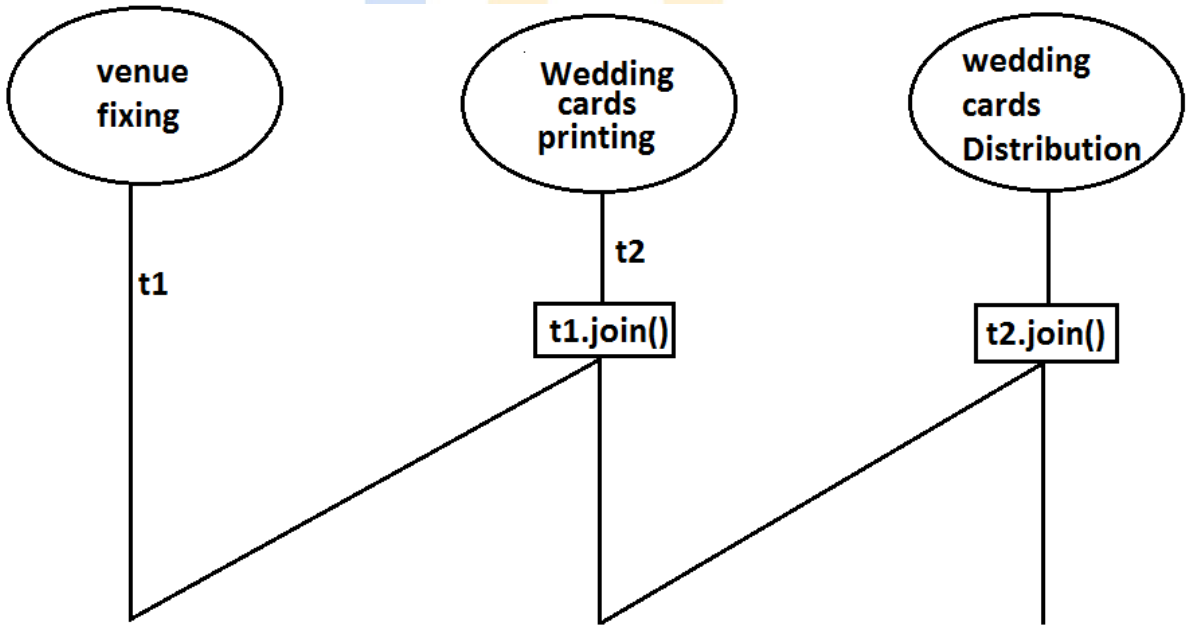
Note : Some operating systems may not provide proper support for yield() method.

Join():

If a Thread wants to wait until completing some other Thread then we should go for join() method.

Example: If a Thread t1 executes t2.join() then t1 should go for waiting state until completing t2.

Diagram:



1. public final void join()throws InterruptedException

Every join() method throws InterruptedException, which is checked exception hence compulsory we should handle either by try catch or by throws keyword. Otherwise we will get compiletime error.

Example:

```
class MyThread extends Thread
{
public void run()
{
for(int i=0;i<5;i++)
{
System.out.println("Sita Thread");
try
{
Thread.sleep(2000);
}
catch (InterruptedException e){}
}
}
}
class ThreadJoinDemo
{
public static void main(String[] args)throws InterruptedException
{
MyThread t=new MyThread();
t.start();
//t.join(); //--->1
for(int i=0;i<5;i++)
{
```

```
System.out.println("Rama Thread");
}
}
}
```

 If we are commenting line 1 then both Threads will be executed simultaneously
and we can't expect exact execution order.
 If we are not commenting line 1 then main Thread will wait until completing
child Thread in this the output is sita Thread 5 times followed by Rama Thread 5
times.

Waiting of child Thread untill completing main Thread :

Example:

```
class MyThread extends Thread
{
static Thread mt;
public void run() //throws InterruptedException
{
try
{
mt.join();
}
catch (InterruptedException e){}
for(int i=0;i<5;i++)
{
System.out.println("Child Thread");
```

```
}
}
}
class ThreadJoinDemo
{
public static void main(String[] args)throws InterruptedException
{
mt=Thread.currentThread();
MyThread t=new MyThread();
t.start();
for(int i=0;i<5;i++)
{
Thread.sleep(2000);
System.out.println("Main Thread");
}
}
}
```

Output :

```
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
Child Thread
```

Child Thread

Note :

If main thread calls join() on child thread object and child thread called join() on main

thread object then both threads will wait for each other forever and the program will be hanged(like **deadlock** if a Thread class join() method on the same thread itself then the

program will be hanged).

Example :

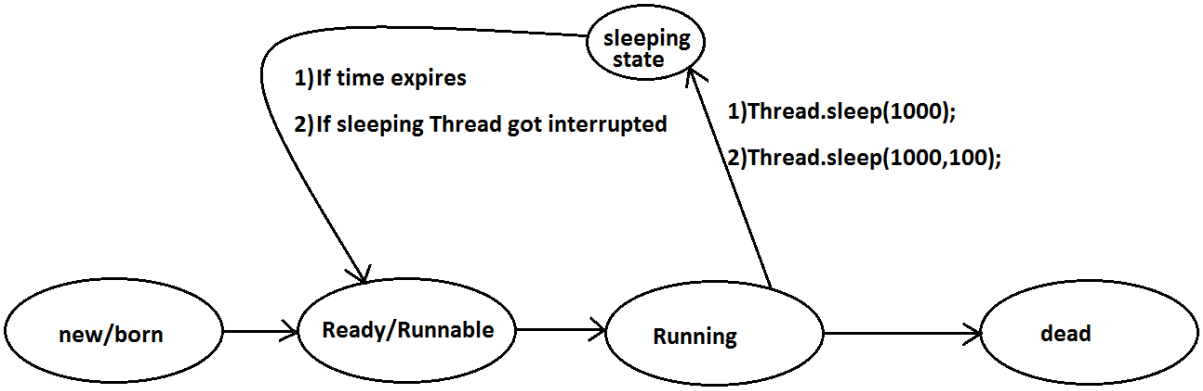
```
class ThreadDemo {
public static void main() throws InterruptedException {
Thread.currentThread().join();
---------- --------
main main
}
}
```

Sleep() method:

If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.

1. public static native void sleep(long ms) throws InterruptedException
2. public static void sleep(long ms,int ns)throws InterruptedException

Diagram:



Example:

```
class ThreadJoinDemo
{
public static void main(String[] args)throws InterruptedException
{
System.out.println("M");
Thread.sleep(3000);
System.out.println("E");
Thread.sleep(3000);
System.out.println("G");
Thread.sleep(3000);
System.out.println("A");
}
}
```

Output:

M

E

G

A

Interrupting a Thread:

How a Thread can interrupt another thread ?

If a Thread can interrupt a sleeping or waiting Thread by using interrupt()(break of method of Thread class.

public void interrupt();

Example:

```
class MyThread extends Thread
{
public void run()
{
try
{
for(int i=0;i<5;i++)
{
System.out.println("i am lazy Thread :"+i);
Thread.sleep(2000);
}
}
catch (InterruptedException e)
{
System.out.println("i got interrupted");
}
}
}
class ThreadInterruptDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();
```

```
t.start();
//t.interrupt(); //--->1
System.out.println("end of main thread");
}
}
```

 If we are commenting line 1 then main Thread won't interrupt child Thread and hence child Thread will be continued until its completion.

 If we are not commenting line 1 then main Thread interrupts child Thread and hence child Thread won't continued until its completion in this case the output is:

End of main thread

I am lazy Thread: 0

I got interrupted

Note:

 Whenever we are calling interrupt() method we may not see the effect immediately, if the target Thread is in sleeping or waiting state it will be interrupted immediately.

 If the target Thread is not in sleeping or waiting state then interrupt call will wait until target Thread will enter into sleeping or waiting state. Once target

Thread entered into sleeping or waiting state it will effect immediately.

 In its lifetime if the target Thread never entered into sleeping or waiting state then there is no impact of interrupt call simply interrupt call will be wasted.

Example:

```
class MyThread extends Thread
{
public void run()
```

```
{
for(int i=0;i<5;i++)
{
System.out.println("iam lazy thread");
}
System.out.println("I'm entered into sleeping stage");
try
{
Thread.sleep(3000);
}
catch (InterruptedException e)
{
System.out.println("i got interrupted");
}
}
}
class ThreadInterruptDemo1
{
public static void main(String[] args)
{
MyThread t=new MyThread();
t.start();
t.interrupt();
System.out.println("end of main thread");
}
}
```

 In the above program interrupt() method call invoked by main Thread will wait
until child Thread entered into sleeping state.
 Once child Thread entered into sleeping state then it will be interrupted immediately.

Compression of yield, join and sleep() method?

Property	Yield()	Join()	Sleep()
1)Purpose?	To pause current executing Thread for giving the chance of remaining waiting Threads of same priority.	If a Thread wants to wait until completing some other Thread then we should go for join.	If a Thread don't want to perform any operation for a particular amount of time then we should go for sleep() method.
2) Is it static?	Yes	No	Yes
3) Is it final?	No	Yes	No
4) Is it overloaded?	No	Yes	Yes
5) Is it throws InterruptedException?	No	Yes	Yes
6) Is it native method?	Yes	No	sleep(long ms) --

Synchronization

1. Synchronized is the keyword applicable for methods and blocks but not for
classes and variables.
2. If a method or block declared as the synchronized then at a time only one Thread is allow to execute that method or block on the given object.
3. The main advantage of synchronized keyword is we can resolve data inconsistency problems.
4. But the main disadvantage of synchronized keyword is it increases waiting time of the Thread and effects performance of the system.
5. Hence if there is no specific requirement then never recommended to use synchronized keyword.
6. Internally synchronization concept is implemented by using lock concept.
7. Every object in java has a unique lock. Whenever we are using **synchronized** keyword then only lock concept will come into the picture.

8. If a Thread wants to execute any synchronized method on the given object 1st it has to get the lock of that object. Once a Thread got the lock of that object then it's allow to execute any synchronized method on that object. If the synchronized method execution completes then automatically Thread releases lock.

9. While a Thread executing any synchronized method the remaining Threads are not allowed execute any synchronized method on that object simultaneously. But remaining Threads are allowed to execute any non-synchronized method simultaneously. [lock concept is implemented based on object but not based on method].

```
package com.bytecode.multi;

class MyAccount {
    static int availableBalance = 10000;
    int required;

    public synchronized  void withDraw(int
required) {
        if (required <= availableBalance) {
            System.out.println("Amount deducted by
"
        + Thread.currentThread().getName());
            System.out.println("Updating
database");
            availableBalance=availableBalance-
required;
        }
        else {
            System.out.println("Insufficient
balance for "
        +Thread.currentThread().getName());
        }
    }
```

```
}

public class JointAccountApp {

    public static void main(String[] args) {
        MyAccount obj1=new MyAccount();
        Runnable obj=()->{
            obj1.withDraw(8000);
        };
        Thread t1=new Thread(obj);
        Thread t2=new Thread(obj);
        t1.setName("Raja");
        t2.setName("Rani");
        t1.start();
        t2.start();
    }
}
```

Example:

```
class Display
{
public synchronized void wish(String name)
{
for(int i=0;i<5;i++)
{
System.out.print("good morning:");
try
{
Thread.sleep(1000);
}
catch (InterruptedException e)
{}
```

```
System.out.println(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 d1=new Display();
MyThread t1=new MyThread(d1,"dhoni");
MyThread t2=new MyThread(d1,"yuvaraj");
t1.start();
t2.start();
```

}

}

If we are not declaring wish() method as synchronized then both Threads will be

executed simultaneously and we will get irregular output.

Output:

good morning:good morning:yuvaraj

good morning:dhoni

good morning:yuvaraj

good morning:dhoni

good morning:yuvaraj

good morning:dhoni

good morning:yuvaraj

good morning:dhoni

good morning:yuvaraj

dhoni

If we declare wish()method as synchronized then the Threads will be executed one by one that is until completing the 1st Thread the 2nd Thread will wait in this case we will get regular output which is nothing but

Output:

good morning:dhoni

good morning:dhoni

good morning:dhoni

good morning:dhoni

good morning:dhoni

good morning:yuvaraj

good morning:yuvaraj

good morning:yuvaraj

good morning:yuvaraj

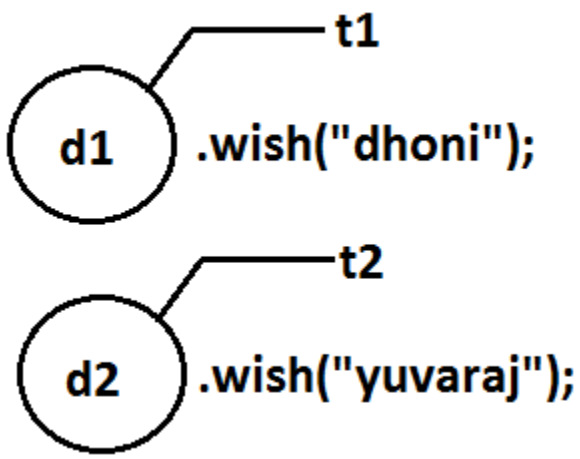
good morning:yuvaraj

Case study:

Case 1:

```
Display d1=new Display();
Display d2=new Display();
MyThread t1=new MyThread(d1,"dhoni");
MyThread t2=new MyThread(d2,"yuvaraj");
t1.start();
t2.start();
```

Diagram:



Even though we declared wish() method as synchronized but we will get irregular output in this case, because both Threads are operating on different objects.

Conclusion : If multiple threads are operating on multiple objects then there is no impact of Syncronization.

If multiple threads are operating on same java objects then syncronized concept is required(applicable).

Class level lock:

1. Every class in java has a unique lock. If a Thread wants to execute a static synchronized method then it required class level lock.

2. Once a Thread got class level lock then it is allow to execute any static synchronized method of that class.

3. While a Thread executing any static synchronized method the remaining Threads are not allow to execute any static synchronized method of that class simultaneously.

4. But remaining Threads are allowed to execute normal synchronized methods,normal static methods, and normal instance methods simultaneously.

5. Class level lock and object lock both are different and there is no relationship between these two.

```
package com.bytecode.multi;

class Ticket {
    static int available = 1;
    int wanted;

     static synchronized void process(int wanted) {
       if (wanted <= available) {
            String name =
Thread.currentThread().getName();
            System.out.println(name + " has been
booked " + wanted + " ticket  ");
            available = available - wanted;
            System.out.println("Updating
database");
        } else {
            String name =
Thread.currentThread().getName();
            System.out.println("sorry No tickets
available for " + name);
```

```
        }
    }
}

public class BookTicket {

    public static void main(String[] args) {
        //Ticket obj1 = new Ticket();
        Runnable obj = () -> {
            Ticket.process(1);
        };
        Thread t1 = new Thread(obj);
        t1.setName("First Person");
        t1.start();

        Thread t2 = new Thread(obj);
        t2.setName("second Person");
        t2.start();

    }

}
```

Synchronized block:

1. If very few lines of the code required synchronization then it's never recommended to declare entire method as synchronized we have to enclose those few lines of the code with in synchronized block.
2. The main advantage of synchronized block over synchronized method is it reduces waiting time of Thread and improves performance of the system.

```
package com.bytecode.multi;

class Reserve implements Runnable {
```

```
    int available = 1;
    int wanted;

    Reserve(int wanted) {
        this.wanted = wanted;
    }

    @Override
    public void run() {
        String name =
Thread.currentThread().getName();

        synchronized (this) {
            if (wanted <= available) {
                System.out.println(name + " have
been selected  "
            + wanted + " number of ticket  ");
                available = available - wanted;
                System.out.println("Updating
database");
            } else {
                System.out.println("sorry No
tickets available" +
            "for " + name);
            }
        }
    }
}

public class TSUsingBlock {

    public static void main(String[] args) {
        Reserve obj = new Reserve(1);
        Thread t1 = new Thread(obj);
        t1.start();
        t1.setName("you");
```

```
        Thread t2 = new Thread(obj);
        t2.start();
        t2.setName("your wife");

    }

}
```

Example 1: To get lock of current object we can declare synchronized block as follows.

If Thread got lock of current object then only it is allowed to execute this block.

Synchronized(this){}

Example 2: To get the lock of a particular object 'b' we have to declare a synchronized

block as follows.

If thread got lock of 'b' object then only it is allowed to execute this block.

Synchronized(b){}

Questions:

1. Explain about synchronized keyword and its advantages and disadvantages?
2. What is object lock and when a Thread required?
3. What is class level lock and when a Thread required?
4. What is the difference between object lock and class level lock?
5. While a Thread executing a synchronized method on the given object is the remaining Threads are allowed to execute other synchronized methods simultaneously on the same object?

Ans: No.

6. What is synchronized block and explain its declaration?

7. What is the advantage of synchronized block over synchronized method?

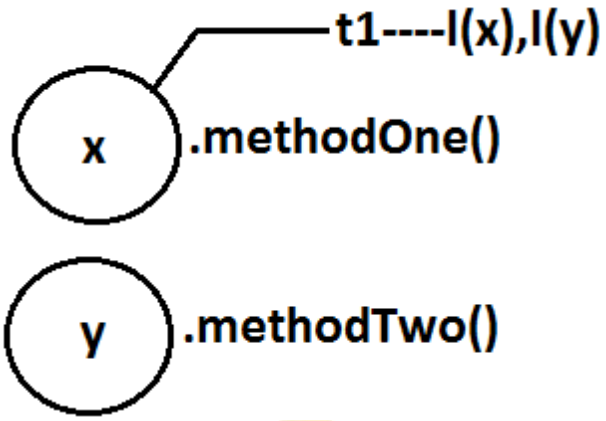
8. Is a Thread can hold more than one lock at a time?

Ans: Yes, of course from different objects. Example:

```
class X
{
    synchronized void methodOne()
    {
        Y y=new Y();
        y.methodTwo();
    }
}
```

```
class Y
{
    synchronized void methodTwo()
    {}
}
```

Diagram:



10. What is synchronized statement?

Ans: The statements which present inside synchronized method and synchronized block are called synchronized statements. [Interview people created terminology].

Inter Thread communication (wait(),notify(), notifyAll()):

 Two Threads can communicate with each other by using wait(), notify() and notifyAll() methods.

 The Thread which is required updation it has to call wait() method on the required object then immediately the Thread will entered into waiting state.

The Thread which is performing updation of object, it is responsible to give notification by calling notify() method.

After getting notification the waiting Thread will get those updations.

 wait(), notify() and notifyAll() methods are available in Object class but not in Thread class because Thread can call these methods on any common object.

 To call wait(), notify() and notifyAll() methods compulsory the current Thread should be owner of that object

i.e., current Thread should has lock of that object

i.e., current Thread should be in synchronized area. Hence we can call wait(), notify() and notifyAll() methods only from synchronized area otherwise we will get runtime exception saying IllegalMonitorStateException.

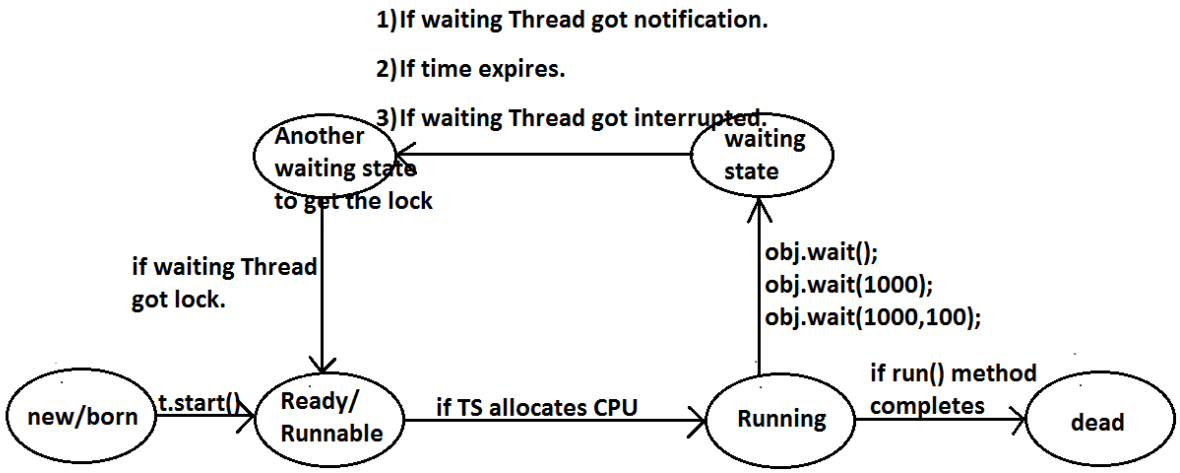
 Once a Thread calls wait() method on the given object 1st it releases the lock of that object immediately and entered into waiting state.

 Once a Thread calls notify() (or) notifyAll() methods it releases the lock of that object but may not immediately.

 Except these (wait(),notify(),notifyAll()) methods there is no other place(method) where the lock release will be happen.

50, Lakhanpur Housing Society, Near Lucky Restaurant, Vikas Nagar 208024
Contact: 9794687277, 8707276645
Web: Www.Bytecode.Co.In, Email: Bytecodeitsolutions@Gmail.Com

 Once a Thread calls wait(), notify(), notifyAll() methods on any object then it releases the lock of that particular object but not all locks ..



Example 1:

```
class ThreadA
{
public static void main(String[] args)throws InterruptedException
{
ThreadB b=new ThreadB();
b.start();
synchronized(b)
{
System.out.println("main Thread calling wait() method");//step-1
b.wait();
```

```
System.out.println("main Thread got notification call");//step-4
System.out.println(b.total);
}
}
}
class ThreadB extends Thread
{
int total=0;
public void run()
{
synchronized(this)
{
System.out.println("child thread starts calcuation");//step-2
for(int i=0;i<=100;i++)
{
total=total+i;
}
System.out.println("child thread giving notification call");//step-

this.notify();
}
}
}
```

Output:

main Thread calling wait() method

child thread starts calculation

child thread giving notification call

main Thread got notification call

5050

Example 2:

Notify vs notifyAll():

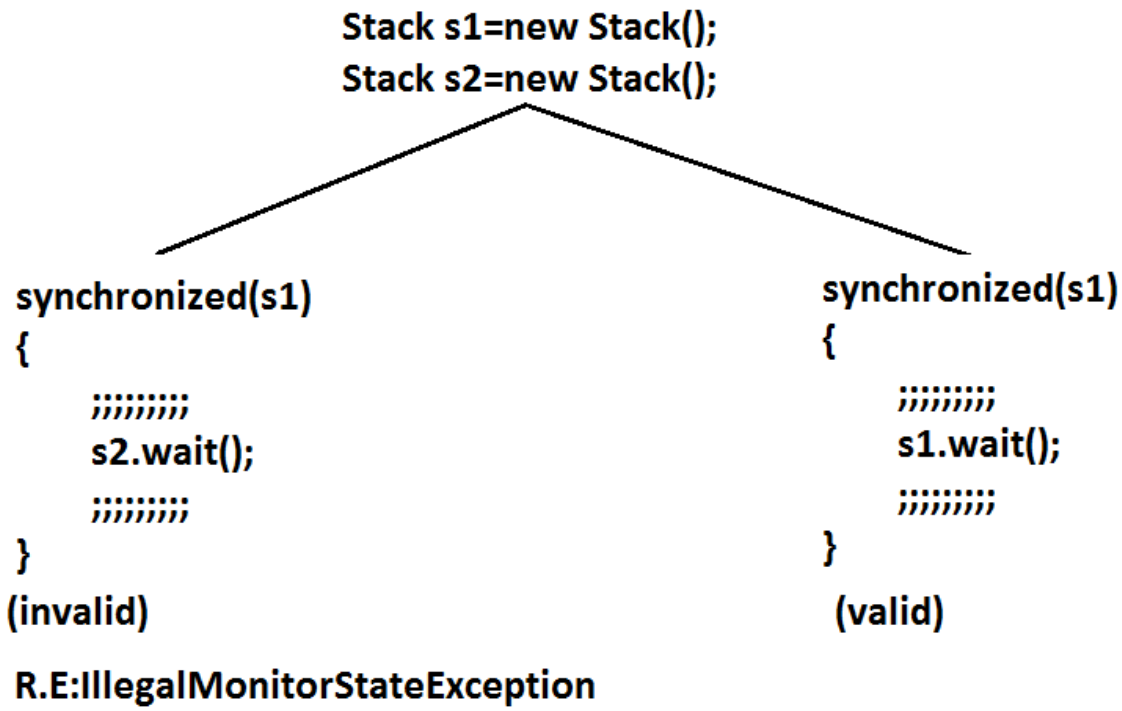
 We can use notify() method to give notification for only one Thread. If multiple Threads are waiting then only one Thread will get the chance and remaining Threads has to wait for further notification. But which Thread will be notify(inform) we can't expect exactly it depends on JVM.

 We can use notifyAll() method to give the notification for all waiting Threads. All waiting Threads will be notified and will be executed one by one, because they are required lock

Note: On which object we are calling wait(), notify() and notifyAll() methods that

corresponding object lock we have to get but not other object locks.

Example:



Which of the folowing statements are True ?

1. Once a Thread calls wait() on any Object immediately it will entered into waiting state without releasing the lock ?

NO

2. Once a Thread calls wait() on any Object it releases the lock of that Object but may not immediately ?

NO

3. Once a Thread calls wait() on any Object it immediately releases all locks whatever it has and entered into waiting state ?

NO

4. Once a Thread calls wait() on any Object it immediately releases the lock of that perticular Object and entered into waiting state ?

YES

5. Once a Thread calls notify() on any Object it immediately releases the lock of that Object ?

NO

6. Once a Thread calls notify() on any Object it releases the lock of that Object but may not immediately ?

YES

Daemon Threads:

The Threads which are executing in the background are called daemon Threads.

The main objective of daemon Threads is to provide support for non-daemon Threads

like main Thread.

Example:

Garbage collector

When ever the program runs with low memory the JVM will execute Garbage Collector

to provide free memory. So that the main Thread can continue it's execution.

 We can check whether the Thread is daemon or not by using isDaemon() method

of Thread class.

public final boolean isDaemon();

 We can change daemon nature of a Thread by using setDaemon () method.

public final void setDaemon(boolean b);

 But we can change daemon nature before starting Thread only. That is after

starting the Thread if we are trying to change the daemon nature we will get R.E

saying *IllegalThreadStateException*.

 Default Nature : Main Thread is always non daemon and we can't change its daemon nature because it's already started at the beginning only.

 Main Thread is always non daemon and for the remaining Threads daemon nature will be inheriting from parent to child that is if the parent is daemon child is also daemon and if the parent is non daemon then child is also non daemon.

 Whenever the last non daemon Thread terminates automatically all daemon Threads will be terminated.

Example:

```
class MyThread extends Thread
{
}
class DaemonThreadDemo
{
public static void main(String[] args)
{
System.out.println(Thread.currentThread().isDaemon());
MyThread t=new MyThread();
System.out.println(t.isDaemon()); 1
t.start();
t.setDaemon(true);
System.out.println(t.isDaemon());
}
}
```

Output:

```
false
false
RE:IllegalThreadStateException
```

Example:

```
class MyThread extends Thread
{
public void run()
{
for(int i=0;i<10;i++)
{
System.out.println("lazy thread");
try
{
Thread.sleep(2000);
}
catch (InterruptedException e)
{}
}
}


}
class DaemonThreadDemo
{
public static void main(String[] args)
{
MyThread t=new MyThread();
t.setDaemon(true); //-->1
t.start();
System.out.println("end of main Thread");
}
```

}

Output:

End of main Thread

 If we comment line 1 then both main & child Threads are non-Daemon , and

hence both threads will be executed untill there completion.

 If we are not comment line 1 then main thread is non-Daemon and child thread

is Daemon. Hence when ever main Thread terminates automatically child thread

will be terminated.

Lazy thread

 If we are commenting line 1 then both main and child Threads are non daemon

and hence both will be executed until they completion.

 If we are not commenting line 1 then main Thread is non daemon and child

Thread is daemon and hence whenever main Thread terminates automatically

child Thread will be terminated.

Life cycle of a Thread: