Runnable Interface

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

void **run():** This method is used to perform action for a thread

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new call stack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) Java Thread Example by extending Thread class

```
class Multi extends Thread
{
    public void run()
    {
       System.out.println("thread is running...");
    }
    public static void main(String args[])
    {
        Multi t1=new Multi();
        t1.start();
    }
}
```

Output: thread is running...

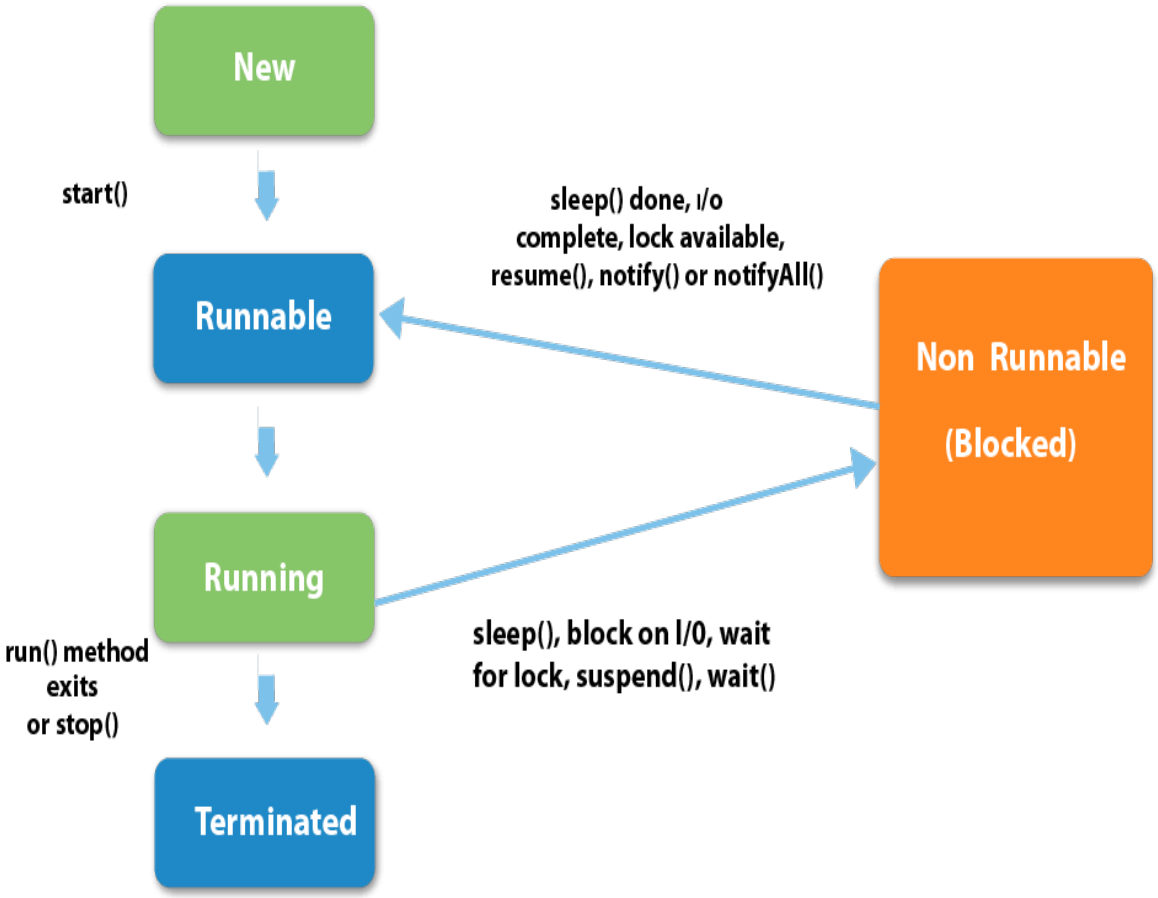
2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable
{
   public void run()
   {
       System.out.println("thread is running...");
   }
   public static void main(String args[])
   {
      Multi3 m1=new Multi3();
      Thread t1 =new Thread(m1);
```

```
        t1.start();
    }
}
```

Output:thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.



1. **New:** The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2. **Runnable:** The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3. **Running:** The thread is in running state if the thread scheduler has selected it.

4. **Non-Runnable (Blocked):** This is the state when the thread is still alive, but is currently not eligible to run.
5. **Terminated:** A thread is in terminated or dead state when its run() method exits

Sleep method

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax

The Thread class provides two methods for sleeping a thread:

public static void sleep(long miliseconds)throws InterruptedException

public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example

```
class ExSleep extends Thread
{
   public void run()
   {
      for( int i= 1;i< 5;i++)
      {
         try
         {
            Thread.sleep( 500);
         }
         catch(InterruptedException e)
         {
            System.out.println(e);
         }
         System.out.println(i);
       }
   }
}
class MEx
{
    public static void main(String args[])
    {
       ExSleep t1= new ExSleep();
       ExSleep t2= new ExSleep();
        t1.start();
        t2.start();
```

```
    }
}
```

Output:

1 1 2 2 3 3 4 4

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

stop() method

The stop() method of thread class terminates the thread execution. Once a thread is stopped, it cannot be restarted by start() method.

Syntax

```
public final void stop()

public final void stop(Throwable obj)
```

Parameter

obj : The Throwable object to be thrown.

Return

This method does not return any value.

Exception

SecurityException: This exception throws if the current thread cannot modify the thread.

Example

```
public class ExStop extends Thread
{
   public void run()
   {
     for( int i= 1; i< 5 ; i++)
     {
        try
        {
           // thread to sleep for 500 millisecond
          sleep( 500);
          System.out.println(Thread.currentThread().getName());
         }
        catch (InterruptedException e)
        {
          System.out.println(e);
        }
```

```
            System.out.println(i);
        }
    }
}
class ExStop
{
    public static void main(String args[])
    {
        // creating three threads
        ExStop t1= new ExStop();
        ExStop t2= new ExStop();
        ExStop t3= new ExStop();
        // call run() method
        t1.start();
        t2.start();
        // stop t3 thread
        t3.stop();
        System.out.println( "Thread t3 is stopped" );
    }
}
```

Disadvantages of a Multithreaded/Multicontexted Application

Multithreaded and multicontexted applications present the following disadvantages:

- **Difficulty of writing code:** Multithreaded and multicontexted applications are not easy to write. Only experienced programmers should undertake coding for these types of applications.
- **Difficulty of debugging:** It is much harder to replicate an error in a multithreaded or multicontexted application than it is to do so in a single-threaded, single-contexted application. As a result, it is more difficult, in the former case, to identify and verify root causes when errors occur.
- **Difficulty of managing concurrency:** The task of managing concurrency among threads is difficult and has the potential to introduce new problems into an application.
- **Difficulty of testing:** Testing a multithreaded application is more difficult than testing a single-threaded application because defects are often timing-related and more difficult to reproduce.
- **Difficulty of porting existing code:** Existing code often requires significant re-architecting to take advantage of multithreading and multicontexting. Programmers need to:

1. Remove static variables
2. Replace any function calls that are not thread-safe
3. Replace any other code that is not thread-safe

Because the completed port must be tested and re-tested, the work required to port a multithreaded and/or multicontexted application is substantial.

Synchronization in Java

In general, synchronization is used to protect access to resources that are accessed concurrently. One of the benefits of using

multiple threads in an application is that each thread executes asynchronously. There are many situations in which multiple threads must share access to common objects . For example, in a database system, you might not want one thread to be updating a database record while another thread is trying to read it. In these types of cases, we need to ensure that resource will be used by only one thread at a time. Otherwise, two or more threads could access the same resource at the same time, each unaware of the other's actions. Java enables you to coordinate the actions of multiple threads using synchronized methods and

synchronized statements . An object for which access is to be coordinated is accessed through the use of synchronized methods. These methods are declared with the synchronized keyword. Only one synchronized method can be invoked for an object at a given point in time. This keeps synchronized methods in multiple threads from conflicting with each other. Following is the general form of the synchronized statement:

Syntax

```
synchronized(objectidentifier)

{

  // Access shared variables and

other shared resources

}
```

The objectidentifier parameter is a reference to an object whose lock associates with the monitor that the synchronized statement represents. The Java programming language provides two basic synchronization idioms: synchronized methods and synchronized statements.

What are synchronized methods and synchronized statements?

Synchronized Methods

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods. It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object. To make a method synchronized, simply add the synchronized keyword to its declaration:

```
public synchronized void increament()
{
    count++;
}
```

Synchronized block

Synchronize block ensure atomicity of bunch of code statements. If you have to synchronize access to an object of a class or you only want a part of a method to be synchronized to an object then you can use synchronized block for it.

```
public void add(int value)
{
  synchronized(this)
{
     this.count += value;
  }
}
```

One significant difference between

synchronized method and block is that, Synchronized block generally reduce scope of lock. As scope of lock is inversely proportional to performance, its always better to lock only critical section of code. Also, Synchronized block can throw java.lang.NullPointerException if expression provided to block as parameter evaluates to null, which is not the case with synchronized methods.