Multi-Threading In Java

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".

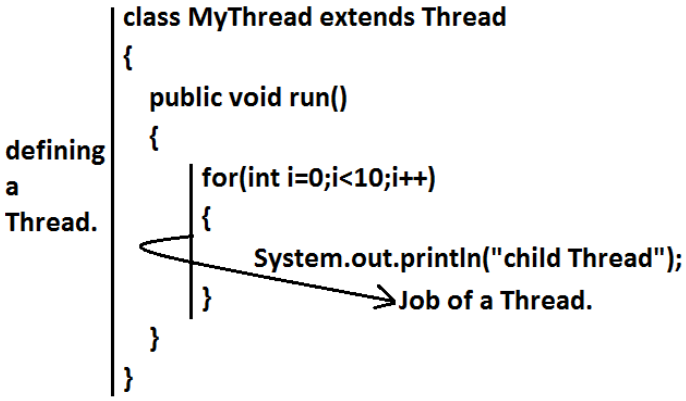
Thread based multitasking:
Executing several tasks simultaneously where each task is a separate independent part of the same program, is called Thread based multitasking.And each independent part is called a "Thread".

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

The ways to define instantiate and start a new Thread:

1. By extending Thread class.
2. By implementing Runnable interface.

Defining a Thread by extending "Thread class":




```java
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");
        }
    }
}
```

Thread Scheduler:

When multiple threads are awaiting execution, the determination of which thread will execute first is made by the "Thread Scheduler," an integral component of the JVM. The specific algorithm or behavior employed by the Thread Scheduler is not standardized and can vary depending on the JVM vendor. Consequently, in multithreading scenarios, the exact execution order and output cannot be anticipated with certainty.

Difference between t.start() and t.run() methods.

- When using **t.start()**, a new thread is created, and this new thread is responsible for executing the **run()** method.
- On the other hand, when using **t.run()**, no new thread is created. Instead, the **run()** method is executed like a regular method by the main thread.

If we were to replace **t.start()** with **t.run()** in the above program, the resulting output would be as follows:

```
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

Importance of Thread class start() method.

The Thread class's **start()** method takes care of essential activities for every thread, such as registering the thread with the Thread Scheduler. This relieves the programmer from these low-level tasks, allowing them to focus solely on defining the thread's job within the **run()** method.
In essence, the **start()** method serves as an invaluable assistant to the programmer, streamlining the thread creation process.
For instance, the **start()** method performs actions such as:

1. Registering the thread with the Thread Scheduler
2. Executing other necessary low-level activities
3. Invoking or calling the **run()** method

It's crucial to note that without executing the **start()** method of the Thread class, a new thread cannot be initiated in Java. Therefore, **start()** is rightfully regarded as the heart of multithreading, as it kickstarts the execution of a new thread and ensures its seamless integration with the JVM's threading mechanisms.

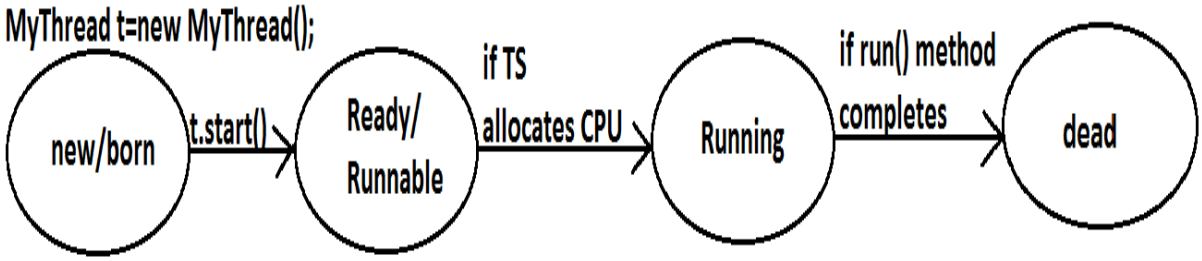
If we are not overriding run() method:

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();
    }
}
```

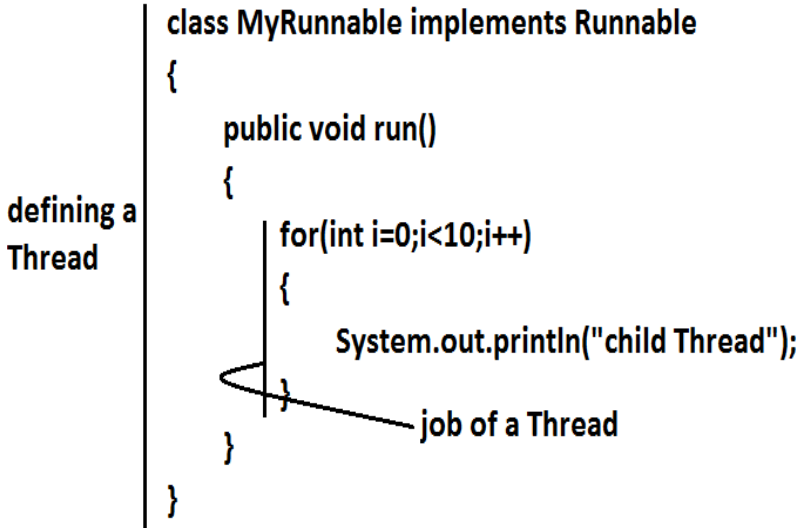
It is highly recommended to override run() method. Otherwise don't go for multithreading concept.

life cycle of the Thread:



- 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.

Thread by implementing Runnable interface:



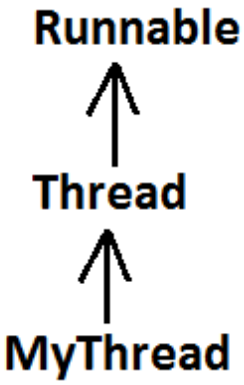

```
class ThreadDemo {
    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread t = new Thread(r);//here r is a Target Runnable t.start();
        for (int i = 0; i < 10; i++) {
            System.out.println("main thread");
        }
    }
}
```

o/p

```
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
```

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.



Getting and setting name of a Thread:

To get and set the name of a Thread in Java, you can use the following methods provided by the Thread class:

1. **Getting the Name**: You can retrieve the name of a Thread using the **getName()** method.
2. **Setting the Name**: You can set the name of a Thread using the **setName(String name)** method.

Here's an example demonstrating how to get and set the name of a Thread:

```
public class ThreadNameExample {
    public static void main(String[] args) {
        // Creating a new thread
        Thread thread = new Thread(new MyRunnable());

        // Getting the default name of the thread
        String defaultName = thread.getName();
        System.out.println("Default name of the thread: " + defaultName);

        // Setting a new name for the thread
        thread.setName("MyThread");

        // Getting the updated name of the thread
        String updatedName = thread.getName();
        System.out.println("Updated name of the thread: " + updatedName);
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            // Thread's job
            System.out.println("Thread is running...");
        }
    }
}
```

Thread Priorities

In Java, threads can have different priorities that indicate their importance to the thread scheduler. Thread priorities are represented by integer values ranging from 1 to 10, where 1 is the lowest priority and 10 is the highest priority. The default priority for a thread is usually inherited from its parent thread.

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

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

```
public class PriorityExample {
    public static void main(String[] args) {
        Thread thread1 = new Thread(new MyRunnable(), "Thread 1");
        Thread thread2 = new Thread(new MyRunnable(), "Thread 2");

        // Set priorities for the threads
        thread1.setPriority(Thread.MIN_PRIORITY); // Minimum priority
        thread2.setPriority(Thread.MAX_PRIORITY); // Maximum priority

        // Start the threads
        thread1.start();
        thread2.start();
    }

    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            Thread thread = Thread.currentThread();
            System.out.println("Thread: " + thread.getName() + ", Priority:
" + thread.getPriority());
        }
    }
}
```

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
    }
}
```

The Methods to Prevent a Thread from Execution:

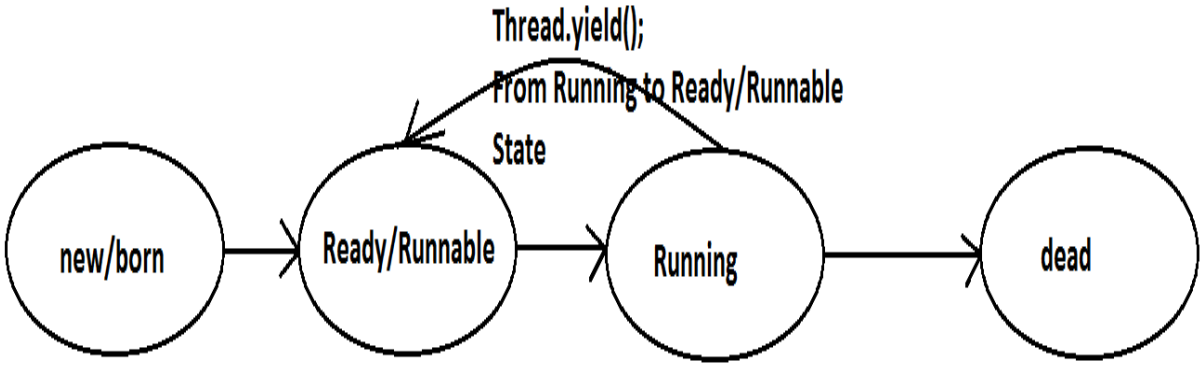
We can prevent(stop) a Thread execution by using the following methods.

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

The yield() method is a hint to the scheduler that the current thread is willing to yield its current use of the processor. It allows other threads of equal priority to run before the current thread resumes execution.

The join() method allows one thread to wait for the completion of another thread. When a thread calls join() on another thread, it will block until the joined thread completes its execution.

sleep() method causes the current thread to sleep for the specified number of milliseconds. It temporarily suspends the execution of the thread, allowing other threads to run.

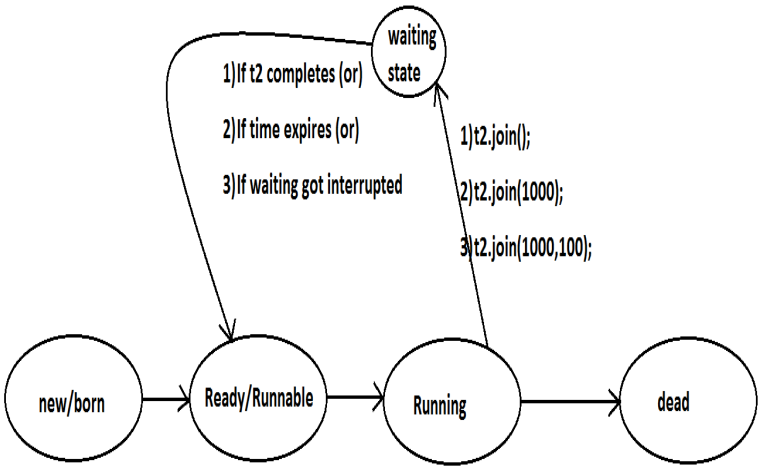



```java
public class YieldExample {
    public static void main(String[] args) {
        // Create two threads
        Thread thread1 = new MyThread("Thread 1");


        // Start the threads
        thread1.start();

    }

    static class MyThread extends Thread {


        public void run() {
            for (int i = 0; i < 5; i++) {
                // Print thread name and value of i
            System.out.println(Thread.currentThread().getName() + ": " +i);

                // Call yield to give chance to other threads
                Thread.yield();
            }
        }
    }
}
```

Note: **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.

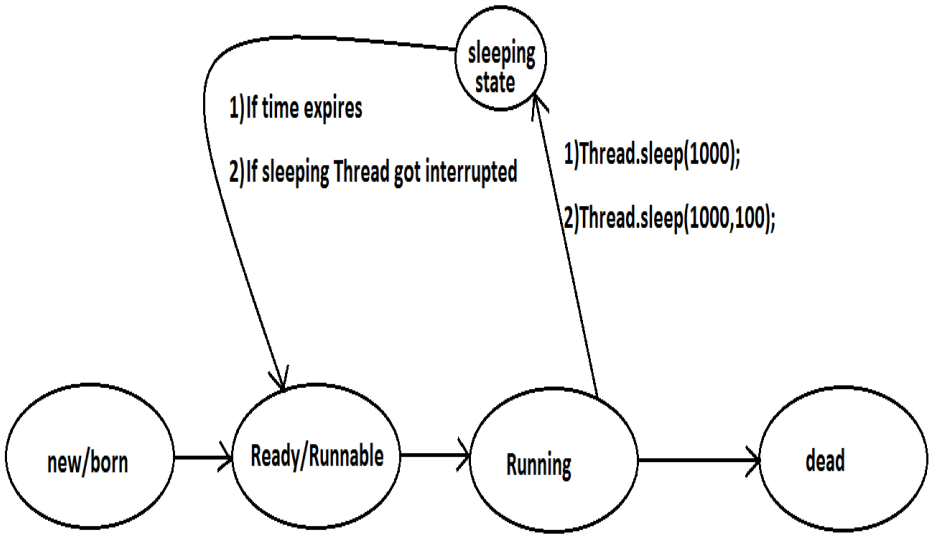
Join()




```
public class JoinExample {
    public static void main(String[] args) throws InterruptedException {
        // Create an instance of the Runnable class
        MyRunnable myRunnable = new MyRunnable();

        // Create a thread using the Runnable instance
        Thread thread = new Thread(myRunnable);

        // Start the thread
        thread.start();

        // Wait for the thread to finish
        thread.join();

        // Main thread resumes execution after thread has finished
        System.out.println("Thread has finished its execution.");
    }

    // Separate class for the run method
    static class MyRunnable implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 5; i++) {
                System.out.println(Thread.currentThread().getName() + ": "
+ i);
                try {
                    // Simulate some work
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

sleep()

```
public class SleepExample {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            System.out.println("Iteration " + i);
            try {
                // Sleep for 1 second (1000 milliseconds)
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



Synchronization

1. The synchronized keyword is applicable for methods and blocks but not for classes and variables.
2. When a method or block is declared as synchronized, only one thread is allowed to execute that method or block at a time on the given object.
3. The primary advantage of the synchronized keyword is that it helps resolve date inconsistency problems.
4. However, a major drawback of the synchronized keyword is that it increases the waiting time of the thread and negatively impacts the performance of the system.
5. Therefore, it is not recommended to use the synchronized keyword unless there is a specific requirement to do so.
6. Internally, the synchronization concept is implemented using locks.
7. Each object in Java has a unique lock. The usage of the synchronized keyword activates the lock concept.

8. If a thread wishes to execute any synchronized method on a given object, it must first acquire the lock of that object. Once the thread obtains the lock of the object, it is permitted to execute any synchronized method on that object. Upon completion of the synchronized method's execution, the thread automatically releases the lock.

```
class Table{
    synchronized void printTable(int n){//synchronized method
        for(int i=1;i<=5;i++){
            System.out.println(n*i);
            try{
                Thread.sleep(400);
            }catch(Exception e){System.out.println(e);}
        }

    }
}

class MyThread1 extends Thread{
    Table t;
    MyThread1(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(5);
    }

}
class MyThread2 extends Thread{
    Table t;
    MyThread2(Table t){
        this.t=t;
    }
    public void run(){
        t.printTable(100);
    }
}

public class TestSynchronization2{
    public static void main(String args[]){
        Table obj = new Table();//only one object
        MyThread1 t1=new MyThread1(obj);
        MyThread2 t2=new MyThread2(obj);
        t1.start();
        t2.start();
    }
}
```

synchronized block

```
public void increment() {
    synchronized (this) { // Synchronized block
        for (int i = 0; i < 10000; i++) {
            count++;
        }
    }
}
```

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

Inter-thread communication in Java is facilitated through the methods wait(), notify(), and notifyAll().

These methods are used to coordinate the execution of threads and allow them to synchronize their actions. Here's a brief overview of each method:

1. **wait():** This method is called on an object within a synchronized context to make the current thread wait until another thread invokes the notify() or notifyAll() method on the same object. When a thread calls wait(), it releases the lock on the object and enters the waiting state until it receives a notification. The wait() method can be called with a timeout parameter to specify the maximum time the thread should wait.
2. **notify():** This method is called on an object within a synchronized context to wake up a single thread that is waiting on the same object. When notify() is invoked, it notifies one of the threads that are waiting on the object to wake up. The choice of which thread to notify is not specified and depends on the JVM's implementation.
3. **notifyAll():** This method is similar to notify(), but it wakes up all threads that are waiting on the same object. It is often used when multiple threads are waiting for a particular condition to change. When notifyAll() is called, all threads that are waiting on the object become eligible to wake up, but which thread actually gets the lock first is determined by the thread scheduler.

```
4. 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-3
                    this.notify();
                }
            }
        }

    }
```

Output:

```
main Thread calling wait() method
child thread starts calculation
child thread giving notification call
main Thread got notification call 5050
```