UNIT-12 Concurrency

Introduction to Multithreading

Multithreading enables multiple tasks in a program to be executed concurrently.

A program may consist of many tasks that can run concurrently. A thread is the flow of execution, from beginning to end, of a task.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

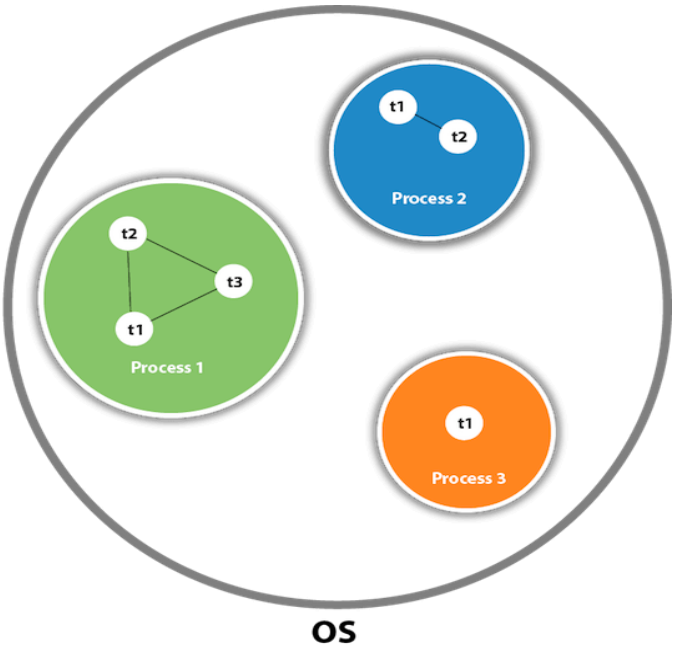
1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

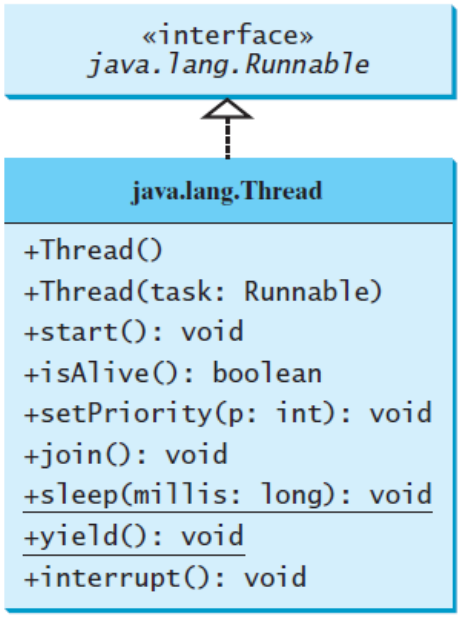


A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS, and one process can have multiple threads.

The **Thread** *class contains the constructors for creating threads for tasks and the methods for controlling threads.*



Creates an empty thread.	
Creates a thread for a specified task.	
Starts the thread that causes the `run()` method to be invoked by the JVM.	
Tests whether the thread is currently running.	
Sets priority `p` (ranging from 1 to 10) for this thread.	
Waits for this thread to finish.	
Puts a thread to sleep for a specified time in milliseconds.	
Causes a thread to pause temporarily and allow other threads to execute.	
Interrupts this thread.	

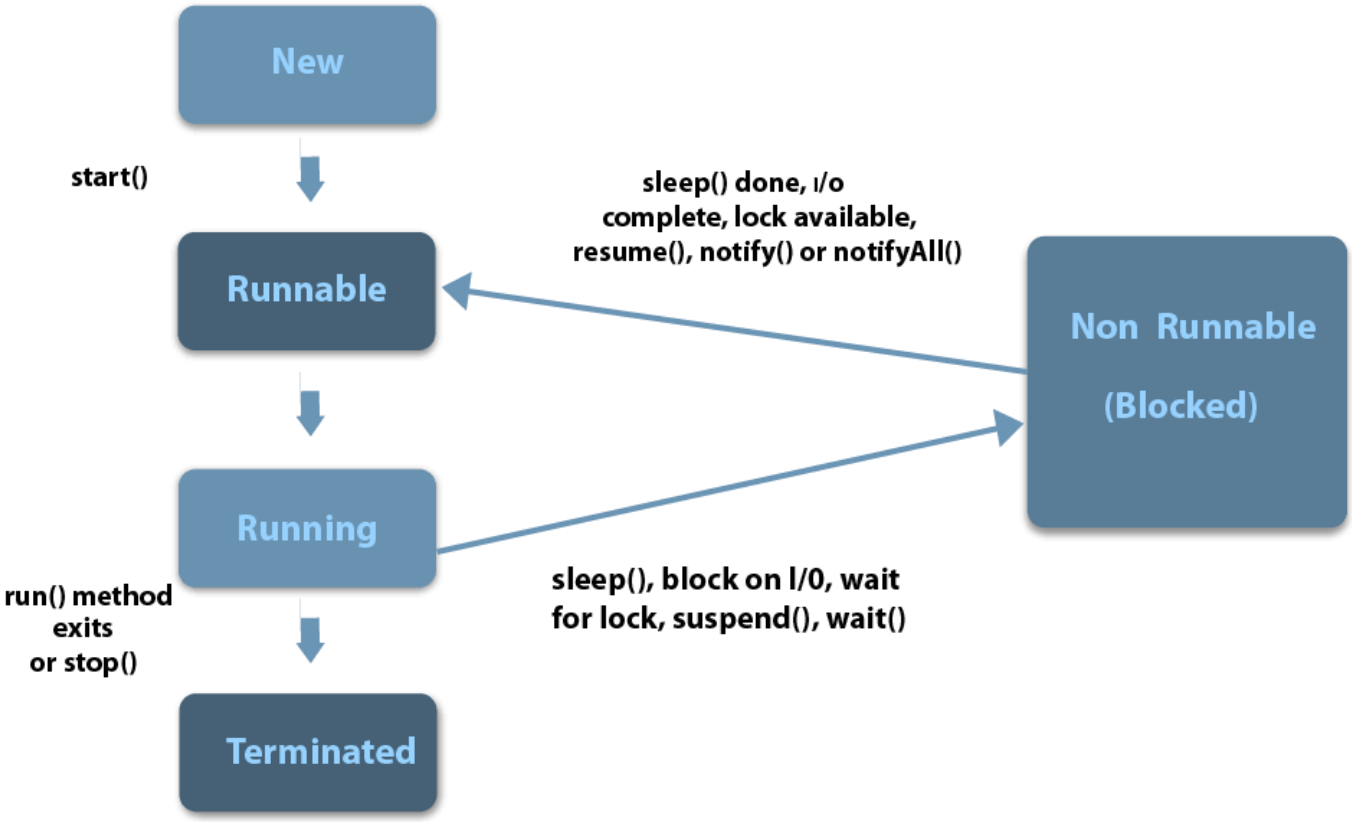
12.1 Life cycle of a Thread (Thread States)

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



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.

How to create thread

There are two ways to create a thread:

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

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends implements Runnable interface.

Runnable interface:

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

1. **public void run():** is used to perform action for a thread.

Starting a thread:

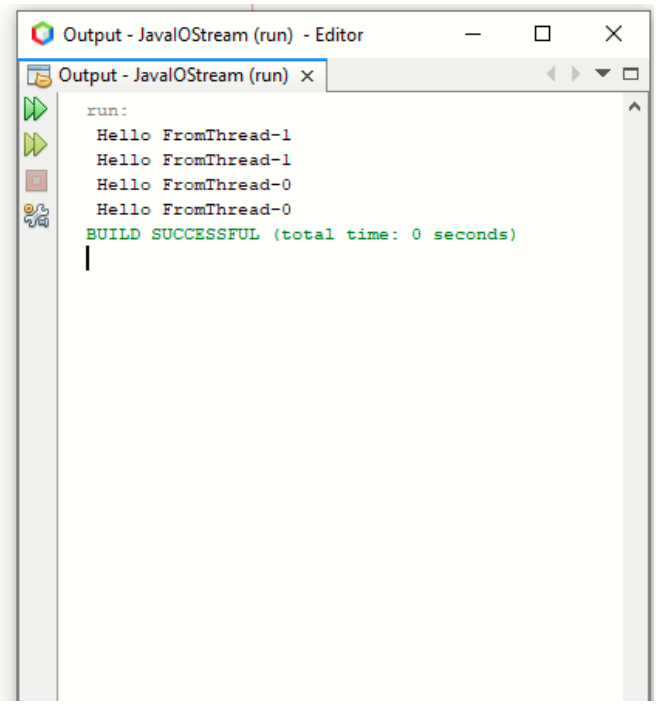
start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- 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 Multi-Threading Example by **extending Thread class**

```
public class Main
{
    public static void main(String args[])
    {

        for(int i=1;i<3;i++) {
            MultiThreading t=new MultiThreading();
            t.start();
        }
    }
}
class MultiThreading extends Thread
{
    public void run()
    {
        for(int i=1;i<3;i++)        {
            String test = Thread.currentThread().getName();
            System.out.println(" Hello From" +test);
           }
    }
}
```



2) Java Multi-Threading Example by **implementing Runnable interface**

We need to explicitely create Thread class object in which the object of our class implementing Runnable interface will be passes.

```java
package javaiostream;


public class Main
{
    public static void main(String args[])
    {

        for(int i=1;i<3;i++) {
            MultiThreading r=new MultiThreading();
            Thread t = new Thread(r);
            t.start();
        }
    }
}
class MultiThreading implements Runnable
{
    public void run()
    {
        for(int i=1;i<3;i++)        {
            String test = Thread.currentThread().getName();
            System.out.println(" Hello From" +test);
          }
    }
}
```

```
Output - JavaIOStream (run) - Editor
Output - JavaIOStream (run)
run:
 Hello FromThread-1
 Hello FromThread-1
 Hello FromThread-0
 Hello FromThread-0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Example: Create 2 threads in which First thread prints Good Morning and Second prints Good Afternoon

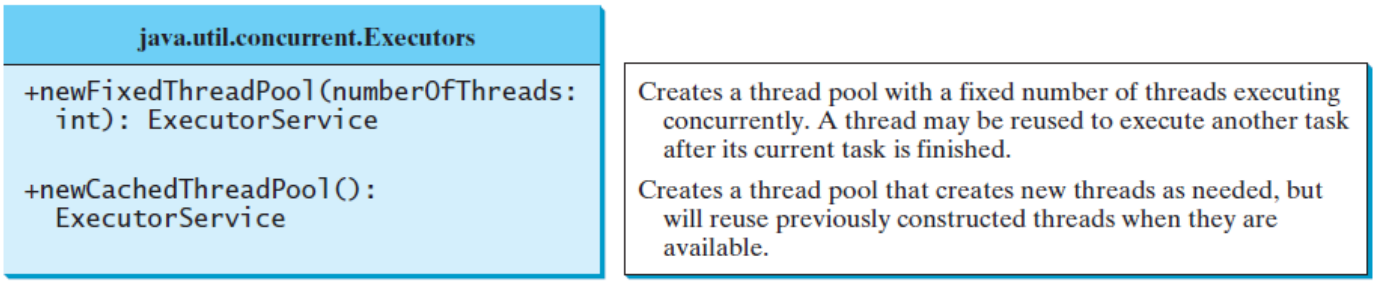
```java
public class Main
{
    public static void main(String args[])
    {
        clock1 obj1=new clock1();
        clock2 obj2=new clock2();
        obj1.start();
        obj2.start();
    }
}
class clock1 extends Thread
{
    public void run()
    {
        for(int i=0;i<3;i++)        {
            String test = Thread.currentThread().getName();
            System.out.println(test+" :- Good Morning");
          }
    }
}
class clock2 extends Thread
{
    public void run()
    {
        for(int i=1;i<3;i++)        {
            String test = Thread.currentThread().getName();
            System.out.println(test+":- Good Afternoon");
        }
    }
}
```

```
Output - JavaIOStream (run) - Editor
Output - JavaIOStream (run)
run:
Thread-1:- Good Afternoon
Thread-1:- Good Afternoon
Thread-0 :- Good Morning
Thread-0 :- Good Morning
Thread-0 :- Good Morning
BUILD SUCCESSFUL (total time: 0 seconds)
```

12.2 Creating and Executing threads with the Executor Framework

Thread Pools
A thread pool can be used to execute tasks efficiently.

- Java provides the **Executor** interface for executing tasks in a thread pool and the **ExecutorService** interface for managing and controlling tasks.
- **ExecutorService** is a subinterface of **Executor**
- The **Executor** interface executes threads, and the **ExecutorService** subinterface manages threads.
- The **Executors** class provides static methods for creating **Executor** objects.




```
1 import java.util.concurrent.*;
2
3 public class ExecutorDemo {
4 public static void main(String[] args) {
5 // Create a fixed thread pool with maximum three threads
6 ExecutorService executor = Executors.newFixedThreadPool(3);
7
8 // Submit runnable tasks to the executor
9 executor.execute(new PrintChar('a', 100));
10 executor.execute(new PrintChar('b', 100));
11 executor.execute(new PrintNum(100));
12
13 // Shut down the executor
14 executor.shutdown();
15 }
16 }
```

Suppose that you replace line 6 with
ExecutorService executor = Executors.newFixedThreadPool(**1**);
The three runnable tasks will be executed sequentially because there is only one thread in the pool.

Suppose you replace line 6 with
ExecutorService executor = Executors.newCachedThreadPool();
New threads will be created for each waiting task, so all the tasks will be executed concurrently.

The **shutdown()** method in line 14 tells the executor to shut down. No new tasks can be accepted, but any existing tasks will continue to finish.

12.3 Thread Synchronization

- **The process of allowing only a single thread to access the shared data or resource at a particular point of time** is known as Synchronization. This helps us to protect the data from the access by multiple threads. Java provides the mechanism of synchronization using the synchronized blocks.
- ***Synchronized*** blocks in ***Java*** are marked with the ***synchronized keyword***. A ***synchronized*** block in ***Java*** is ***synchronized*** on some object.
- Thread synchronization is to coordinate the execution of the dependent threads.
- Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.
- Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Understanding the problem without Synchronization

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

```
class counter
{
      int count;
      public void incr()
      {
            count++;
      }
}
public class Main
{
      public static void main(String args[])
      {
            counter c = new counter();
            Thread S1 =
                  new Thread( () -> {
                                    for(int i=0;i<1000;i++){
                                     c.incr();
                              }
                   });
            Thread S2 =
                  new Thread( () -> {
                                    for(int i=0;i<1000;i++){
                                     c.incr();
                              }
                   });
            // Start two threads of ThreadedSend type
            S1.start();
            S2.start();
            // wait for threads to end
            try
            {
                  S1.join();
                  S2.join();
```

```
            }
            catch(Exception e)
            {
                  System.out.println("Interrupted");
            }
                   System.out.println("count" + c.count);
      }
}
```

```
Count 1998
```

Java synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Make **incr()** method **synchronized** in **counter** class (in above example) as follows:

```
class counter
{
      int count;
      public synchronized void incr()
      {
            count++;
      }
}
```