Chapter 3

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

Thread a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking. But we use multithreading than multiprocessing because threads share a common 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 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 exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

•Each process has its own address in memory i.e. each process allocates separate memory area.

•Process is heavy weight.

•Cost of communication between the processes is high.

•Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

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

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight sub process, a 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 shares a common memory area. As shown in the above figure, 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.

Note: At a time one thread is executed only.

Life cycle of a Thread (Thread 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 Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Commonly used methods of Thread class:

1. public void run(): is used to perform action for a thread.
2. public void start(): starts the execution of the thread. JVM calls the run() method on the thread.

3. public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. public void join(): waits for a thread to die.
5. public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
6. public int getPriority(): returns the priority of the thread.
7. public int setPriority(int priority): changes the priority of the thread.
8. public String getName(): returns the name of the thread.
9. public void setName(String name): changes the name of the thread.
10. public Thread currentThread(): returns the reference of currently executing thread.
11. public int getId(): returns the id of the thread.
12. public Thread.State getState(): returns the state of the thread.
13. public boolean isAlive(): tests if the thread is alive.
14. public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. public void suspend(): is used to suspend the thread(depricated).
16. public void resume(): is used to resume the suspended thread(depricated).
17. public void stop(): is used to stop the thread(depricated).
18. public boolean isDaemon(): tests if the thread is a daemon thread.
19. public void setDaemon(boolean b): marks the thread as daemon or user thread.
20. public void interrupt(): interrupts the thread.
21. public boolean isInterrupted(): tests if the thread has been interrupted.
22. public static boolean interrupted(): tests if the current thread has been interrupted.
23. 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().

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 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();
 }         }
```

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();
 }         }
```

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

Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a

predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Sleep method in java Thread

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

Syntax of sleep() method in java

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 of sleep method in java

```
class TestSleepMethod1 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);
  }         }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();
  t1.start();
  t2.start();
 }         }
```

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.

Thread Priority

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name
is:"+Thread.currentThread().getName());
   System.out.println("running thread priority
is:"+Thread.currentThread().getPriority());
    }
 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
 }         }
```

Multithreads by anonymous class

Program of four mathematical operations by four Threads

```
class CalculatorThread extends Thread {
public static void main(String [] s){
     int x = 10;
        int y = 5;
     CalculatorThread ct1 = new CalculatorThread(){
          public void run(){
               System.out.println("Sum = " + (x+y));
          }          };
     CalculatorThread ct2 = new CalculatorThread(){
          public void run(){
               System.out.println("Difference = " + (x-y));
          }                       };
     CalculatorThread ct3 = new CalculatorThread(){
          public void run(){
     System.out.println("Product = " + (x*y));}
```

```
      };
      CalculatorThread ct4 = new CalculatorThread(){
            public void run(){
                  System.out.println("Division = " + (x/y));
      } };
      ct1.setName("Add");        ct1.start();
      ct2.setName("Sub");         ct2.start();
      ct3.setName("Mult");        ct3.start();
      ct4.setName("Div");         ct4.start();
}           }
```

Same example as above by annonymous class that implements Runnable interface:

Program of performing two tasks by two threads

```
class TestMultitasking5{
 public static void main(String args[]){
  Runnable r1=new Runnable(){
    public void run(){
      System.out.println("task one");
    }
  };
        Runnable r2=new Runnable(){
    public void run(){
      System.out.println("task two");
    }
  };
  Thread t1=new Thread(r1);
  Thread t2=new Thread(r2);
  t1.start();
  t2.start();
 }         }
```

Synchronization in Java

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.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    A. Synchronized method.
    B. Synchronized block.
    C. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. By synchronized method
2. By synchronized block
3. By static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them. From Java 5 the package java.util.concurrent.locks contains several lock implementations. Understanding the problem without Synchronization.

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

```
class Table{
void printTable(int n){//method not synchronized
   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);
}         }
class TestSynchronization1{
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();
}    }
```

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.

1. //example of java synchronized method

```
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();
}     }
```

Example of synchronized method by using annonymous class

In this program, we have created the two threads by annonymous class, so less coding is required.

//Program of synchronized method by using annonymous class

```
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);}      }
```

```
 }    }
public class TestSynchronization3{
public static void main(String args[]){
final Table obj = new Table();//only one object

Thread t1=new Thread(){
public void run(){
obj.printTable(5);
}           };
Thread t2=new Thread(){
public void run(){
obj.printTable(100);
}           };
t1.start();
t2.start();
}           }
```

Synchronized block in java

Synchronized block can be used to perform synchronization on any specific resource of the method. Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block. If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to remember for Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.

Syntax to use synchronized block

```
synchronized (object reference expression) {
  //code block
}
```

Example of synchronized block

Let's see the simple example of synchronized block.

Program of synchronized block

```
class Table{
 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
```

```
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

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 TestSynchronizedBlock1{
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();
}           }
```

Same Example of synchronized block by using annonymous class:

//Program of synchronized block by using annonymous class

```
class Table{

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

```
    }         }
}           }

public class TestSynchronizedBlock2{
public static void main(String args[]){
final Table obj = new Table();//only one object

Thread t1=new Thread(){
public void run(){
obj.printTable(5);
}           };
Thread t2=new Thread(){
public void run(){
obj.printTable(100);
}     };

t1.start();
t2.start();
}     }
```

Static synchronization

If you make any static method as synchronized, the lock will be on the class not on object.

Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

Example of static synchronization

In this example we are applying synchronized keyword on the static method to perform static synchronization.

```
class Table{
synchronized static void printTable(int n){
for(int i=1;i<=10;i++){
System.out.println(n*i);
try{
Thread.sleep(400);
}catch(Exception e){}
```

```
   }   }    }
   class MyThread1 extends Thread{
   public void run(){
   Table.printTable(1);
   }        }

   class MyThread2 extends Thread{
   public void run(){
   Table.printTable(10);
      }     }

   lass MyThread3 extends Thread{
   public void run(){
   Table.printTable(100);
   }        }
   class MyThread4 extends Thread{
   public void run(){
   Table.printTable(1000);
   }  }

      public class TestSynchronization4{
      public static void main(String t[]){
      MyThread1 t1=new MyThread1();
      MyThread2 t2=new MyThread2();
      MyThread3 t3=new MyThread3();
      MyThread4 t4=new MyThread4();
      t1.start();
      t2.start();
      t3.start();
      t4.start();
      }           }
```

Same example of static synchronization by anonymous class

In this example, we are using anonymous class to create the threads.

```
class Table{

 synchronized static  void printTable(int n){
   for(int i=1;i<=10;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }         }    }


public class TestSynchronization5 {
public static void main(String[] args) {

Thread t1=new Thread(){
```

```
public void run(){
Table.printTable(1);
}      };

Thread t2=new Thread(){
public void run(){
Table.printTable(10);
}      };
Thread t3=new Thread(){
public void run(){
Table.printTable(100);
}      };

Thread t4=new Thread(){
public void run(){
Table.printTable(1000);
}      };
t1.start();
t2.start();
t3.start();
t4.start();
}     }
```

Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference .class name .class. A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

```
static void printTable(int n) {
       synchronized (Table.class) {       // Synchronized block on class A
// ...
}              }
```

Inter-thread communication in Java

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()

- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed. The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method Description

public final void wait()throws InterruptedException waits until object is notified.

public final void wait(long timeout)throws InterruptedException waits for the specified amount of time.

2) notify() method

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. Syntax:

public final void notify()

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

Understanding the process of inter-thread communication

The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.

4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
```

```
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}     }

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();
  }         }
```