**Multithreading in Java**

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

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.

**Multitasking**

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

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

**1) Process-based Multitasking (Multiprocessing)**

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

**2) Thread-based Multitasking (Multithreading)**

* Threads share the same address space.
* A 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 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.

**Note: At a time one thread is executed only.**

**Java Thread class**

Java provides **Thread class** to achieve thread programming. Thread class provides constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

**Java Thread Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Modifier and Type** | **Method** | **Description** |
| 1) | void | [start()](https://www.javatpoint.com/java-thread-start-method) | It is used to start the execution of the thread. |
| 2) | void | [run()](https://www.javatpoint.com/java-thread-run-method) | It is used to do an action for a thread. |
| 3) | static void | [sleep()](https://www.javatpoint.com/java-thread-sleep-method) | It sleeps a thread for the specified amount of time. |
| 4) | static Thread | [currentThread()](https://www.javatpoint.com/java-thread-currentthread-method) | It returns a reference to the currently executing thread object. |
| 5) | void | [join()](https://www.javatpoint.com/java-thread-join-method) | It waits for a thread to die. |
| 6) | int | [getPriority()](https://www.javatpoint.com/java-thread-getpriority-method) | It returns the priority of the thread. |
| 7) | void | [setPriority()](https://www.javatpoint.com/java-thread-setpriority-method) | It changes the priority of the thread. |
| 8) | String | [getName()](https://www.javatpoint.com/java-thread-getname-method) | It returns the name of the thread. |
| 9) | void | [setName()](https://www.javatpoint.com/java-thread-setname-method) | It changes the name of the thread. |
| 10) | long | [getId()](https://www.javatpoint.com/java-thread-getid-method) | It returns the id of the thread. |
| 11) | boolean | [isAlive()](https://www.javatpoint.com/java-thread-isalive-method) | It tests if the thread is alive. |
| 12) | static void | [yield()](https://www.javatpoint.com/java-thread-yield-method) | It causes the currently executing thread object to pause and allow other threads to execute temporarily. |
| 13) | void | [suspend()](https://www.javatpoint.com/java-thread-suspend-method) | It is used to suspend the thread. |
| 14) | void | [resume()](https://www.javatpoint.com/java-thread-resume-method) | It is used to resume the suspended thread. |
| 15) | void | [stop()](https://www.javatpoint.com/java-thread-stop-method) | It is used to stop the thread. |
| 16) | void | [destroy()](https://www.javatpoint.com/java-thread-destroy-method) | It is used to destroy the thread group and all of its subgroups. |
| 17) | Boolean | [isDaemon()](https://www.javatpoint.com/java-thread-isdaemon-method) | It tests if the thread is a daemon thread. |
| 18) | void | [setDaemon()](https://www.javatpoint.com/java-thread-setdaemon-method) | It marks the thread as daemon or user thread. |
| 19) | void | [interrupt()](https://www.javatpoint.com/java-thread-interrupt-method) | It interrupts the thread. |
| 20) | Boolean | [isinterrupted()](https://www.javatpoint.com/java-thread-isinterrupted-method) | It tests whether the thread has been interrupted. |
| 21) | static Boolean | [interrupted()](https://www.javatpoint.com/java-thread-interrupted-method) | It tests whether the current thread has been interrupted. |
| 22) | static int | [activeCount()](https://www.javatpoint.com/java-thread-activecount-method) | It returns the number of active threads in the current thread's thread group. |
| 23) | void | [checkAccess()](https://www.javatpoint.com/java-thread-checkaccess-method) | It determines if the currently running thread has permission to modify the thread. |
| 24) | static Boolean | [holdLock()](https://www.javatpoint.com/java-thread-holdlock-method) | It returns true if and only if the current thread holds the monitor lock on the specified object. |
| 25) | static void | [dumpStack()](https://www.javatpoint.com/java-thread-dumpstack-method) | It is used to print a stack trace of the current thread to the standard error stream. |
| 26) | StackTraceElement[] | [getStackTrace()](https://www.javatpoint.com/java-thread-getstacktrace-method) | It returns an array of stack trace elements representing the stack dump of the thread. |
| 27) | static int | [enumerate()](https://www.javatpoint.com/java-thread-enumerate-method) | It is used to copy every active thread's thread group and its subgroup into the specified array. |
| 28) | Thread.State | [getState()](https://www.javatpoint.com/java-thread-getstate-method) | It is used to return the state of the thread. |
| 29) | ThreadGroup | [getThreadGroup()](https://www.javatpoint.com/java-thread-getthreadgroup-method) | It is used to return the thread group to which this thread belongs |
| 30) | String | [toString()](https://www.javatpoint.com/java-thread-tostring-method) | It is used to return a string representation of this thread, including the thread's name, priority, and thread group. |
| 31) | void | [notify()](https://www.javatpoint.com/java-thread-notify-method) | It is used to give the notification for only one thread which is waiting for a particular object. |
| 32) | void | [notifyAll()](https://www.javatpoint.com/java-thread-notifyall-method) | It is used to give the notification to all waiting threads of a particular object. |
| 33) | void | [setContextClassLoader()](https://www.javatpoint.com/java-thread-setcontextclassloader-method) | It sets the context ClassLoader for the Thread. |
| 34) | ClassLoader | [getContextClassLoader()](https://www.javatpoint.com/java-thread-getcontextclassloader-method) | It returns the context ClassLoader for the thread. |
| 35) | static Thread.UncaughtExceptionHandler | [getDefaultUncaughtExceptionHandler()](https://www.javatpoint.com/java-thread-getdefaultuncaughtexceptionhandler-method) | It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception. |
| 36) | static void | [setDefaultUncaughtExceptionHandler()](https://www.javatpoint.com/java-thread-setdefaultuncaughtexceptionhandler-method) | It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception. |
| **Life cycle of a Thread (Thread States)**   1. [Life cycle of a thread](https://www.javatpoint.com/life-cycle-of-a-thread)    1. [New](https://www.javatpoint.com/life-cycle-of-a-thread#threadstatenew)    2. [Runnable](https://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunnable)    3. [Running](https://www.javatpoint.com/life-cycle-of-a-thread#threadstaterunning)    4. [Non-Runnable (Blocked)](https://www.javatpoint.com/life-cycle-of-a-thread#threadstateblocked)    5. [Terminated](https://www.javatpoint.com/life-cycle-of-a-thread#threadstateterminated)   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 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. |

**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();

 }

}

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. |

**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**

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();

 }

}

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 shedular picks up another thread and so on.

**Can we start a thread twice**

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

1. public class TestThreadTwice1 extends Thread{
2. public void run(){
3. System.out.println("running...");
4. }
5. public static void main(String args[]){
6. TestThreadTwice1 t1=new TestThreadTwice1();
7. t1.start();
8. t1.start();
9. }
10. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestThreadTwice1)

running

Exception in thread "main" java.lang.IllegalThreadStateException

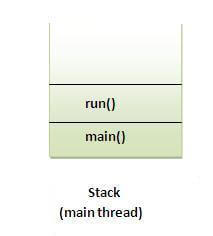
**What if we call run() method directly instead start() method?**

|  |
| --- |
| * Each thread starts in a separate call stack. * Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack. |

1. class TestCallRun1 extends Thread{
2. public void run(){
3. System.out.println("running...");
4. }
5. public static void main(String args[]){
6. TestCallRun1 t1=new TestCallRun1();
7. t1.run();//fine, but does not start a separate call stack
8. }
9. }

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestCallRun1)

Output:running...

***Problem if you direct call run() method***

class TestCallRun2 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[]){

  TestCallRun2 t1=new TestCallRun2();

  TestCallRun2 t2=new TestCallRun2();

  t1.run();

  t2.run();

 }

}

Output:1

2

3

4

5

1

2

3

4

5

|  |
| --- |
| As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object. |

**Priority of a Thread (Thread Priority):**

|  |
| --- |
| Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular 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();

 }

}

Output:running thread name is:Thread-0

running thread priority is:10

running thread name is:Thread-1

running thread priority is:1

**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
   1. Synchronized method.
   2. Synchronized block.
   3. 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 an 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();

}

}

Output: 5

100

10

200

15

300

20

400

25

500

**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();

}

}

Output: 5

10

15

20

25

100

200

300

400

500

**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.

1. //Program of synchronized method by using annonymous class
2. class Table{
3. synchronized void printTable(int n){//synchronized method
4. for(int i=1;i<=5;i++){
5. System.out.println(n\*i);
6. try{
7. Thread.sleep(400);
8. }catch(Exception e){System.out.println(e);}
9. }
11. }
12. }
14. public class TestSynchronization3{
15. public static void main(String args[]){
16. final Table obj = new Table();//only one object
18. Thread t1=new Thread(){
19. public void run(){
20. obj.printTable(5);
21. }
22. };
23. Thread t2=new Thread(){
24. public void run(){
25. obj.printTable(100);
26. }
27. };
29. t1.start();
30. t2.start();
31. }
32. }

Output: 5

10

15

20

25

100

200

300

400

500

# 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**

1. synchronized (object reference expression) {
2. //code block
3. }

**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();

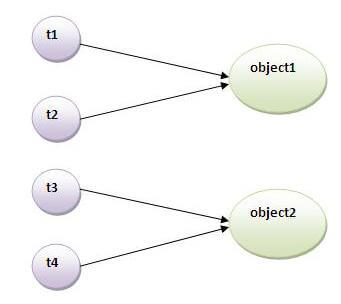
t2.start();

}

}

**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);

}

}

class 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();

}

}

[Test it Now](http://www.javatpoint.com/opr/test.jsp?filename=TestSynchronization4)

Output: 1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

**Same example of static synchronization by annonymous class**

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

1. class Table{
3. synchronized static  void printTable(int n){
4. for(int i=1;i<=10;i++){
5. System.out.println(n\*i);
6. try{
7. Thread.sleep(400);
8. }catch(Exception e){}
9. }
10. }
11. }
13. public class TestSynchronization5 {
14. public static void main(String[] args) {
16. Thread t1=new Thread(){
17. public void run(){
18. Table.printTable(1);
19. }
20. };
22. Thread t2=new Thread(){
23. public void run(){
24. Table.printTable(10);
25. }
26. };
28. Thread t3=new Thread(){
29. public void run(){
30. Table.printTable(100);
31. }
32. };
34. Thread t4=new Thread(){
35. public void run(){
36. Table.printTable(1000);
37. }
38. };
39. t1.start();
40. t2.start();
41. t3.start();
42. t4.start();
44. }
45. }

Output: 1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

**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:

1. static void printTable(int n) {
2. synchronized (Table.class) {       // Synchronized block on class A
3. // ...
4. }
5. }