**Multithreading**

1. **Introduction**

Multithreading is a process of executing multiple threads simultaneously. Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

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

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

1. **Advantages of Java Multithreading**
2. It **doesn't block the user** because threads are independent, and you can perform multiple operations at the same time.
3. You **can perform many operations together, so it saves time**.
4. 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)

* In process-based multitasking, each process allocates a separate memory area.
* A process is heavyweight.
* The Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading [registers](https://www.javatpoint.com/register-memory), memory maps, updating lists, etc.

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

* Threads share the same address space. Threads uses a shared memory area.
* 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.**

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.

There is context-switching between the threads. There can be multiple processes inside the [OS](https://www.javatpoint.com/os-tutorial), 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](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

The method of thread class are: -

|  |  |  |  |
| --- | --- | --- | --- |
| **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. |

**Lifecycle of thread**

In Java, a thread always exists in one of the following states. The states are.

1. New
2. Active
3. Blocked / waiting.
4. time waiting
5. terminated.

***New: -***

Whenever thread is created it is always in a new state. For a thread in the new state, the code has not run yet, and thus has not begun its execution.

***Active: -***

When a thread invokes the start () method, it moves from the new state to the active state. The active state contains two states within it: one is runnable, and the other is running.

***Runnable*** - A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.

***Running:*** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

***Blocked or Waiting:***

Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

***Timed Waiting:***

Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

***Terminated:***

A thread reaches the termination state because of the following reasons:

* When a thread has finished its job, then it exists or terminates normally.
* **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

## **Implementation of Thread States**

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

1. **public** **static** **final** Thread.State NEW

It represents the first state of a thread that is the NEW state.

1. **public** **static** **final** Thread.State RUNNABLE

It represents the runnable state.It means a thread is waiting in the queue to run.

1. **public** **static** **final** Thread.State BLOCKED

It represents the blocked state. In this state, the thread is waiting to acquire a lock.

1. **public** **static** **final** Thread.State WAITING

It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

1. **public** **static** **final** Thread.State TIMED\_WAITING

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

* sleep
* join with timeout
* wait with timeout
* parkUntil
* parkNanos

1. **public** **static** **final** Thread.State TERMINATED

It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

# **How to create a thread in Java**

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.

***Example of thread class.***

**public class ThreadClass1 extends Thread{**

**public void run()**

**{**

**for(int i=10; i>=1; i--)**

**{**

**System.*out*.println(i + " \n");**

**try**

**{**

**Thread.*sleep*(1000);**

**}catch(Exception e)**

**{**

**}**

**}**

**}**

**}**

**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:**

The **start() method** of Thread class is used to start a newly created thread. It performs the 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.

# **Thread scheduler in java**

Thread Scheduler is a component of java that decides which thread to run or execute and which thread to wait.

If there are more than one thread in runnable state, it is up to thread scheduler to pick one of the threads and ignore the other ones.

There are two factors for scheduling a thread: -

1. **Priority**
2. **Time of arrival**

***Priority: -*** Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

***Time of arrival: -*** When two threads of same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such case, arrival of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

***Thread scheduler algorithms***

The scheduling algorithm is followed by a Java thread: -

1. First come first serve scheduler.

In this scheduling algorithm, the scheduler picks the threads that arrive first in the runnable queue.

1. Time-scaling scheduling.

The first come first server algorithm is non-pre-emptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.

1. Pre-emptive-Priority Scheduling.

Suppose there are multiple threads available in the runnable state. The thread scheduler picks the thread that has the highest priority. Since the algorithm is also preemptive, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

***Working of the java thread scheduler.***

Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

# **Sleeping of a thread (Thread.sleep())**

The java thread class provides two variants of the sleep() method. The first one accepts only one argument, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping states its execution form where it has left.

***The sleep() method syntax.***

Following are the syntax of the sleep() method: -

1. **public** **static** **void** sleep(**long** mls) **throws** InterruptedException
2. **public** **static** **void** sleep(**long** mls, **int** n) **throws** InterruptedException

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

Parameters

* 1. mls: The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().
  2. n: It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

***Important point about the sleep() method.***

* Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.
* Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.
* The method does not return anything.

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.

class thread\_thread extends Thread{

    public void run()

    {

        System.out.println("This is demo.....");

    }

    public static void main(String[] args)

    {

        thread\_thread th = new thread\_thread();

        Thread t= new Thread(th);

        t.start();;

        t.start();

    }

}

**If we call Java run() method directly instead start() method.**

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

**Naming of thread and current thread name**

We can get the and also change the name of thread.

getName() : - is used to get the name of thread.

setName() : - is used to set the name of thread.

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

Public final int getPriority(): The java.lang.Thread.getPriority() method returns the priority of the given thread.

public final void setPriority(int newPriority): The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

***3 constants defined in Thread class:***

public static int MIN\_PRIORITY

public static int NORM\_PRIORITY

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.

**Daemon Thread in java**

Daemon thread in Java is a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

***Why JVM terminates the daemon thread if there is no user thread?***

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

***Methods for Java Daemon thread by Thread class***

The java.lang.Thread class provides two methods for java daemon thread.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread as daemon thread or user thread. |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

# **Interview Questions**

* 1. What is multitasking?
  2. How can you identify the process?
  3. How do you see a thread?
  4. What is Multithreading and How it is Different from Multitasking?

- Multithreading is a specialized form of multitasking. Process-based multitasking refers to executing several tasks simultaneously where each task is a separate independent process is Process-based multitasking.

Thread-based multitasking refers to executing several tasks simultaneously where each task is a separate independent part of the same program known as a thread. For example, JUnits uses threads to run test cases in parallel. Henceforth, process-based multitasking is a bigger scenario handling process where threads handle the details. It is already discussed to deeper depth already with visual aids.

* 1. What is a thread?
  2. Which Kind of Multitasking is Better and Why?
  3. What are the different states of a thread, or what is thread lifecycle?
  4. What is the task of the main thread?

- All Java programs have at least one thread, known as the main thread which is created by JVM at the program start when the main() method is invoked with the main thread

* 1. What are the Different Types of threads in Java?

- There are two types of threads in Java as follows:

1. User thread
2. Daemon thread
   1. How to Create a user thread?

- All other threads created in main acts as child threads of the “Main” thread.

1. User thread can be implemented in two ways listed below:
2. Using Thread class by extending java.lang.Thread class.
3. Using Runnable Interface by implementing it
   1. How to set the name of the thread?
   2. What is thread priority?
   3. How deadlock plays a important role in multithreading?
   4. Why output is not ordered?

- Scheduling of threads involves two boundary scheduling,

Scheduling of user-level threads (ULT) to kernel-level threads (KLT) via lightweight process (LWP) by the application developer.

Scheduling of kernel-level threads by the system scheduler to perform different unique os functions.

If multiple threads are waiting to execute then thread execution is decided by “ThreadScheduler” which is a part of JVM hence its vendor dependent resulting in unexpected execution of output order.

Note:

In multithreading, the guarantee of order is very less where we can predict possible outputs but not exactly one.

Also, note that synchronization when incorporated with multithreading does affect our desired output simply by using the keyword ‘synchronized’.

* 1. What is Daemon Thread in Java and explain their properties?
  2. How to Make User Thread to Daemon Thread?
  3. What are the tasks of the start() method?
  4. What is the difference between the start() and run() method?
  5. Can we Override the start() method?

# **Coding…….**

## **Using run method.**

* 1. **public** **class** ThreadClass1 **extends** Thread{
  2. **public** **void** run()
  3. {
  4. **for**(**int** i=10; i>=1; i--)
  5. {
  6. System.***out***.println(i + " \n");
  7. **try**
  8. {
  9. Thread.*sleep*(1000);
  10. }**catch**(Exception e)
  11. {
  13. }
  14. }
  15. }
  16. }
  17. }

## **Example of sleep method.**

* 1. public class sleepthread extends Thread {
  2. public void run()
  3. {
  4. System.out.println("This is demo program....");
  5. try
  6. {
  7. Thread.sleep(1000);
  8. }
  9. catch(Exception e)
  10. {
  11. }
  12. System.out.println("There is a hold for 1000 milisecond ......");
  13. }
  14. public static void main(String[] args) {
  15. sleepthread sp = new sleepthread();
  16. Thread th = new Thread(sp);
  17. th.start();
  18. }
  19. }

## **Program to demonstrate the implementation of setPriority() in thread.**

public class priority\_thread {

    public static void main(String[] args) {

        //creating the object of thread the class

        thread t1 = new thread();

        thread2 t2 = new thread2();

        thread3 t3 = new thread3();

        //printing the priority of threads

        System.out.println("t1 thread Priority is  .."+ t1.getPriority() );

        System.out.println("t2 thread Priority is  .."+ t2.getPriority() );

        System.out.println("t3 thread Priority is  .."+ t3.getPriority() );

        //setting  the priority of threads

        t1.setPriority(1);

        t2.setPriority(2);

        t3.setPriority(3);

        System.out.println("t1 thread Priority is  .."+ t1.getPriority() );

        System.out.println("t2 thread Priority is  .."+ t2.getPriority() );

        System.out.println("t3 thread Priority is  .."+ t3.getPriority() );

        t1.start();

        t2.start();

        t3.start();

    }

}

class thread extends Thread {

    public void run()

    {

        for(int i=0; i<=10; i++)

        {

            System.out.println(i);

            try{

                Thread.sleep(1000);

            }catch(Exception e)

            {

                e.printStackTrace();

            }

        }

    }

}

class thread2 extends Thread {

    public void run()

    {

        for(int i=10000; i<=10010; i++)

        {

            System.out.println(i);

            try{

                Thread.sleep(1000);

            }catch(Exception e)

            {

                e.printStackTrace();

            }

        }

    }

}

class thread3 extends Thread

{

    public void run()

    {

        for(int i=1000; i<=1010; i++)

        {

            System.out.println(i);

            try

            {

                Thread.sleep(1000);

            }catch(Exception e)

            {

                e.printStackTrace();

            }

        }

    }

}

## **Create multiple thread using threa class**

import javafx.scene.media.Track;

public class multiple\_thread2 {

    public static void main(String[] args) {

        //creating  the object of our thread class

        thread1 t1 = new thread1();

        thread2 t2 = new thread2();

        t1.start();

        t2.start();

    }

}

class thread1 extends Thread{

    public void  run()

    {

        System.out.println("Thread 1.....");

        try

        {

            Thread.sleep(5000);

            Thread.currentThread().setPriority(2);

        }catch(Exception e)

        {

            e.printStackTrace();

        }

    }

}

class thread2 extends Thread {

    public void run()

    {

        System.out.println("Thread 2............");

        try

        {

            Thread.sleep(5000);

            Thread.currentThread().setPriority(1);

        }

        catch(Exception e)

        {

            e.printStackTrace();

        }

    }

}

## **Implementing the priority method .**

public class priority\_thread {

    public static void main(String[] args) {

        //creating the object of thread the class

        thread t1 = new thread();

        thread2 t2 = new thread2();

        thread3 t3 = new thread3();

        //printing the priority of threads

        System.out.println("t1 thread Priority is  .."+ t1.getPriority() );

        System.out.println("t2 thread Priority is  .."+ t2.getPriority() );

        System.out.println("t3 thread Priority is  .."+ t3.getPriority() );

        //setting  the priority of threads

        t1.setPriority(1);

        t2.setPriority(2);

        t3.setPriority(3);

        System.out.println("t1 thread Priority is  .."+ t1.getPriority() );

        System.out.println("t2 thread Priority is  .."+ t2.getPriority() );

        System.out.println("t3 thread Priority is  .."+ t3.getPriority() );

        t1.start();

        t2.start();

        t3.start();

    }

}

class thread extends Thread {

    public void run()

    {

        for(int i=0; i<=10; i++)

        {

            System.out.println(i);

            try{

                Thread.sleep(1000);

            }catch(Exception e)

            {

                e.printStackTrace();

            }

        }

    }

}

class thread2 extends Thread {

    public void run()

    {

        for(int i=10000; i<=10010; i++)

        {

            System.out.println(i);

            try{

                Thread.sleep(1000);

            }catch(Exception e)

            {

                e.printStackTrace();

            }

        }

    }

}

class thread3 extends Thread

{

    public void run()

    {

        for(int i=1000; i<=1010; i++)

        {

            System.out.println(i);

            try

            {

                Thread.sleep(1000);

            }catch(Exception e)

            {

                e.printStackTrace();

            }

        }

    }

}

## **Using run method.**

## **Using run method.**

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.

## Using run method.