**Multithreading**

🡪Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

🡪Threads can be created by using two mechanisms :

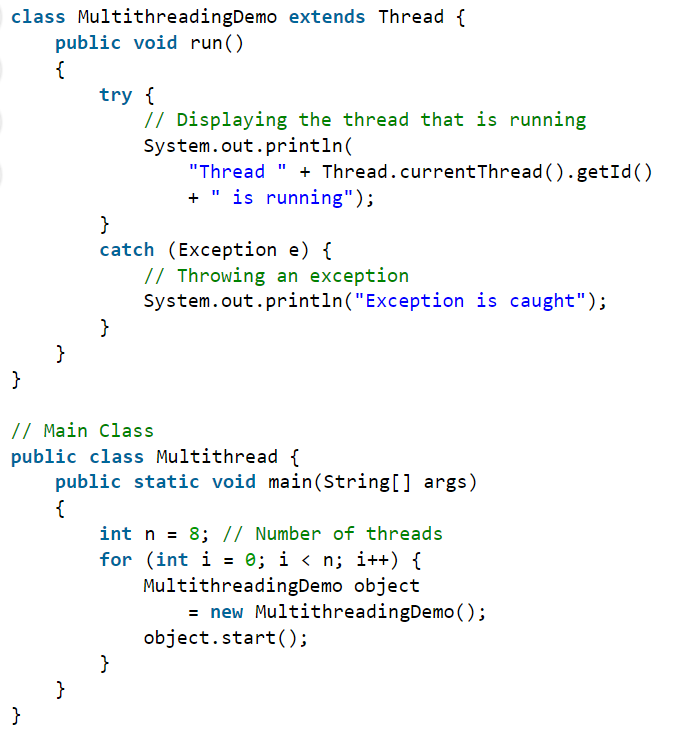
1. Extending the Thread class
2. Implementing the Runnable Interface

**T**

1. **Thread creation by extending the thread class :**

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

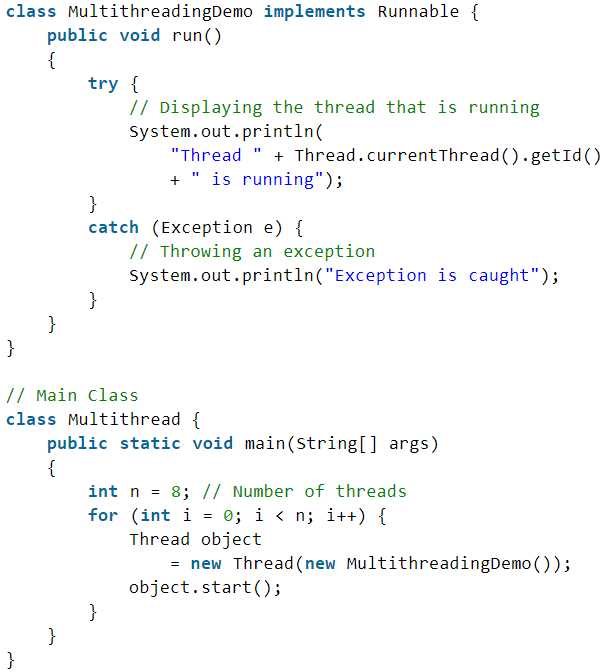
🡺Example Program :



1. **Thread creation by implementing the Runnable Interface**

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

🡺Example Program :



# Lifecycle and States of a Thread

A thread in Java at any point of time exists in any one of the following states. A thread lies only in one of the shown states at any instant:

1. New
2. Runnable
3. Blocked
4. Waiting
5. Timed Waiting
6. Terminated

**1.New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn’t started to execute.

**2.Runnable State:** A thread that is ready to run is moved to a runnable state. In this state, a thread might actually be running or it might be ready to run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.   
A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lie in a runnable state.

**3.Blocked/Waiting state:** When a thread is temporarily inactive, then it’s in one of the following states:

* 1. Blocked
  2. Waiting

**4.Timed Waiting:** A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to a timed waiting state.

**5.Terminated State:** A thread terminates because of either of the following reasons:

**🡺**Because it exits normally. This happens when the code of the thread has been entirely executed by the program.

🡺Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

**🡺Starting a new thread**

🡪The method can be made to run by first making the object of the class :

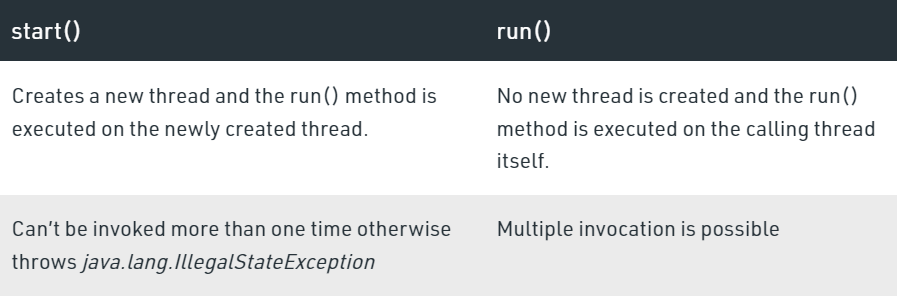
MyThreadClass aThread=new MyThreadClass();

aThread.run();

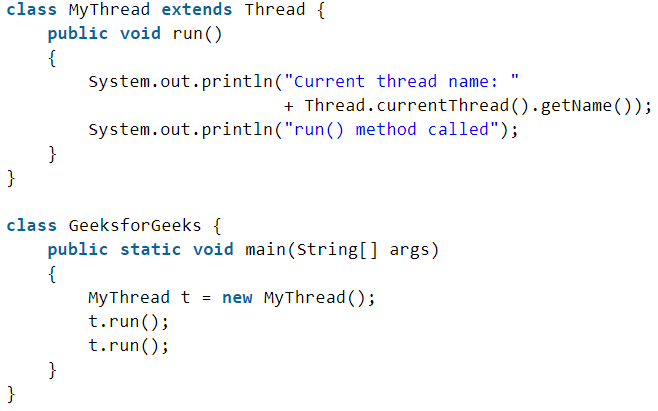
//OR

aThread.start();

**🡪Difference between the start() and run() methods**



**🡪Implementation on the run or start:**



🡪**join Function:** Whenever the join() method is called on a thread instance, the current thread executing that statement will wait for this thread to move to the Terminated state.

# Main thread in Java

When a Java program starts up, one thread begins running immediately. This is usually called the *main* thread of our program because it is the one that is executed when our program begins.

There are certain properties associated with the main thread which are as follows:

* It is the thread from which other “child” threads will be spawned.
* Often, it must be the last thread to finish execution because it performs various shutdown actions

**🡺How to control Main thread**

The main thread is created automatically when our program is started. To control it we must obtain a reference to it. This can be done by calling the method *currentThread()* which is present in Thread class. This method returns a reference to the thread on which it is called. The default priority of Main thread is 5 and for all remaining user threads priority will be inherited from parent to child.

**🡪Pseudo code for controlling the main thread**

**Methods Of Thread**

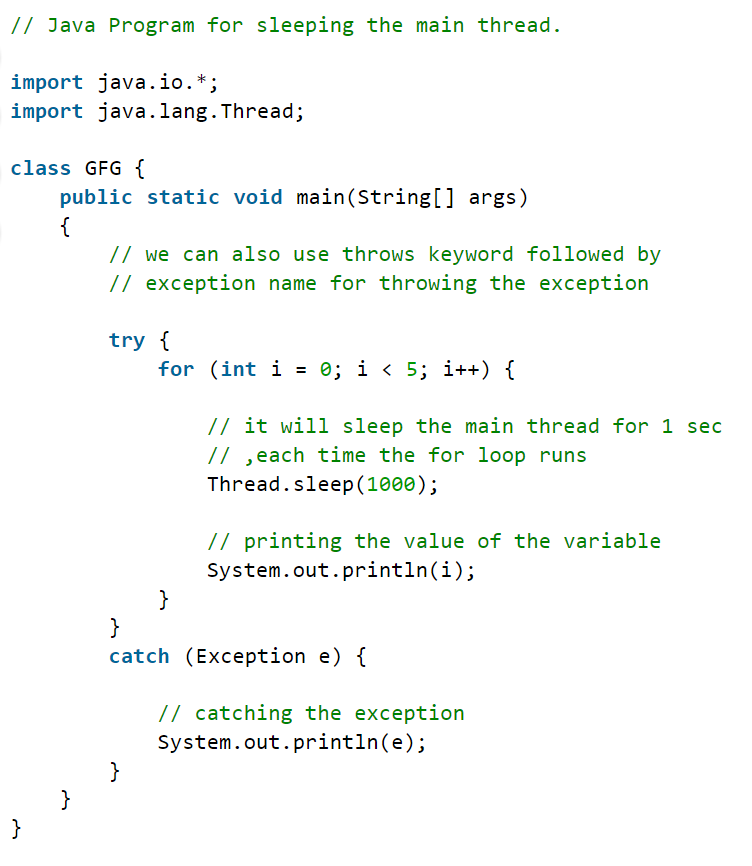
**🡺** **Yield method:** yield() method is a static method of Thread class and it can **stop the currently executing thread and will give a chance to other waiting threads of the same priority**. If in case there are no waiting threads or if all the waiting threads have low priority then the same thread will continue its execution.

**threadName.yield();**

**🡺sleep methods:** The sleep() method is used to stop the execution of the current thread(whichever might be executing in the system) for a specific duration of the time and after that time duration gets over, the thread which is executing earlier starts to execute again.

🡪If any other thread interrupts when the thread is sleeping, then InterruptedException will be thrown.

🡪The sleep time of the thread is in the mili second



**🡪Note: The class to sleep() method is enclosed in the try and catch block. This is necessary because the sleep() method throws an exception, which would be caught. If we failed to catch exception than the program will not compile**

**🡺Thread Priority**

Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by the programmer explicitly.

🡪Properties in threads is a concept where each thread is having a priority which in layman’s language one can say every object is having priority here which is represented by numbers ranging from 1 to 10.

* The default priority is set to 5 as excepted.
* Minimum priority is set to 1.
* Maximum priority is set to 10.

🡪Here 3 constants are defined in it namely as follows:

1. public static int NORM\_PRIORITY
2. public static int MIN\_PRIORITY
3. public static int MAX\_PRIORITY

🡪 Let us do discuss how to get and set priority of a thread in java.

1. **getPriority():** This method returns priority of given thread.
2. **setPriority(int newPriority):** This method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

# Thread Pools in Java

**Background**

Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request in the newly created thread. While this approach seems simple to implement, it has significant disadvantages. A server that creates a new thread for every request would spend more time and consume more system resources in creating and destroying threads than processing actual requests.

Since active threads consume system resources, a JMV creating too many threads at the same time can cause the system to run out of memory. This necessitates the need to limit the number of threads being created.

**What is ThreadPool in Java?**

**A thread pool reuses previously created threads to execute current tasks and offers a solution to the problem of thread cycle overhead and resource thrashing.** Since the thread is already existing when the request arrives, the delay introduced by thread creation is eliminated, making the application more responsive.

* Java provides the Executor framework which is centered around the Executor interface, its sub-interface –**ExecutorService** and the class-**ThreadPoolExecutor**, which implements both of these interfaces. By using the executor, one only has to implement the Runnable objects and send them to the executor to execute.
* They allow you to take advantage of threading, but focus on the tasks that you want the thread to perform, instead of thread mechanics.
* To use thread pools, we first create a object of ExecutorService and pass a set of tasks to it. ThreadPoolExecutor class allows to set the core and maximum pool size.The runnables that are run by a particular thread are executed sequentially.

🡪 In case of a fixed thread pool, if all threads are being currently run by the executor then the pending tasks are placed in a queue and are executed when a thread becomes idle.

**Steps to be followed for threadPooling**

1. Create a task(Runnable Object) to execute

2. Create Executor Pool using Executors

3. Pass tasks to Executor Pool

4. Shutdown the Executor Pool

🡪**One of the main advantages** of using this approach is when you want to process 100 requests at a time, but do not want to create 100 Threads for the same, so as to **reduce JVM overload**. You can use this approach to create a ThreadPool of 10 Threads and you can submit 100 requests to this ThreadPool.

ThreadPool will create maximum of 10 threads to process 10 requests at a time. After process completion of any single Thread,

ThreadPool will internally allocate the 11th request to this Thread

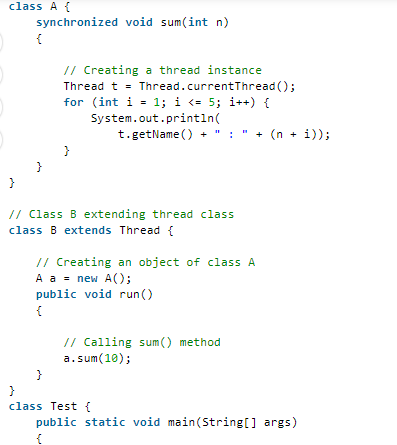
and will keep on doing the same to all the remaining requests.

# Synchronization

🡪multi threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

🡪So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point in time. Java provides a way of creating threads and synchronizing their tasks using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronize on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

🡪This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.



### **Using Volatile keyword**

A volatile keyword is a field modifier that ensures that the object can be used by multiple threads at the same time without having any problem. volatile is one good way of ensuring that the Java program is thread-safe. a volatile keyword can be used as an alternative way of achieving Thread Safety in Java.

**Inter-Thread Communication**

Inter-thread communication can be defined as the exchange of message between two or more threads. The transfer of message takes place before or after the change of state of a thread. For example, an active thread may notify to another suspended thread just before switching to the suspend state. Java implements inter-thread communication with the help of the following three methods:

* 1. **notify():** Resumes the first thread that went into the sleep mode. The object class declaration of notify() method is shown below.

**final void notify();**

* 1. **notifyall():** Resumes all the threads that are in sleep mode. The execution of these threads happens as per priority. The object class declaration of notifyall() method is shown below.

**final void notifyall();**

* 1. **wait():** Sends the calling thread into the sleep mode. This thread can now be activated only by notify() or notifyall() methods.

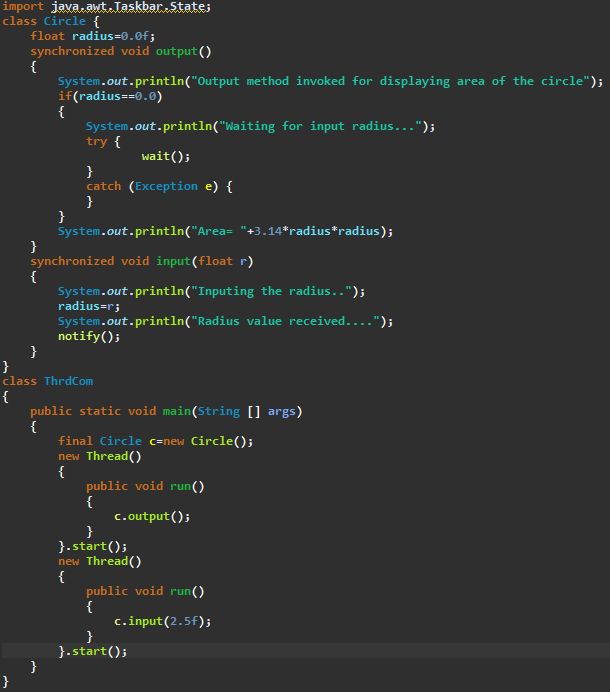
🡪One can also specify the time for which the thread has to wait.

🡪The desired waiting time period us specified as an argument to the wait() method.

**final void wait();**

**🡺All the above methods are declared in the root class, i.e., object. Since the methods are declared as final they cannot be overridden. All the three methods throw InterruptedException.**

**🡺Example Program for wait() and notify() for inter-thread communication**

****