Contents

[**Multithreading in Java** 1](#_Toc503866330)

[Lifecycle and States of a Thread in Java 3](#_Toc503866331)

[Main thread in Java 5](#_Toc503866332)

[Java Concurrency – yield (), sleep () and join () methods 7](#_Toc503866333)

[Inter-thread Communication in Java 12](#_Toc503866334)

[Java.lang.Thread class in Java 17](#_Toc503866335)

[What does start() function do in multithreading in Java? 28](#_Toc503866336)

[Java Thread Priority in Multithreading 30](#_Toc503866337)

[Joining Threads in Java 33](#_Toc503866338)

[Naming a thread and fetching name of current thread in Java 37](#_Toc503866339)

[Synchronized in Java 41](#_Toc503866340)

[Method and Block Synchronization in Java 44](#_Toc503866341)

[Thread Pools in Java 52](#_Toc503866342)

[Semaphore in Java 59](#_Toc503866343)

[Java.util.concurrent.Semaphore class in Java 65](#_Toc503866344)

[CountDownLatch in Java 70](#_Toc503866345)

**Multithreading in Java**

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  
  
**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.

|  |  |
| --- | --- |
| |  | | --- | | class MultithreadingDemo extends Thread {      public void run()  {          try  {               System.out.println ("Thread " +  Thread.currentThread().getId() +  " is running");          } catch (Exception e) {              System.out.println ("Exception is caught");          }      }  }   public class Multithread{      public static void main(String[] args) {          int n = 8; // Number of threads          for (int i=0; i<8; i++)  {              MultithreadingDemo object = new MultithreadingDemo();              object.start();          }      }  } |   **Output :**  Thread 8 is running  Thread 9 is running  Thread 10 is running  Thread 11 is running  Thread 12 is running  Thread 13 is running  Thread 14 is running  Thread 15 is running |

 **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.

|  |  |
| --- | --- |
| |  | | --- | | class MultithreadingDemo implements Runnable{      public void run() {          try {              System.out.println ("Thread " + Thread.currentThread().getId() + " is running");          } catch (Exception e) {               System.out.println ("Exception is caught");          }      }  }   class Multithread{      public static void main(String[] args) {          int n = 8; // Number of threads          for (int i=0; i<8; i++) {              Thread object = new Thread(new MultithreadingDemo());              object.start();          }      }  } |   **Output:**  Thread 8 is running  Thread 9 is running  Thread 10 is running  Thread 11 is running  Thread 12 is running  Thread 13 is running  Thread 14 is running  Thread 15 is running |

**Thread Class vs Runnable Interface**  
  
1. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.  
  
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

# Lifecycle and States of a Thread in Java

A [thread](https://www.geeksforgeeks.org/multithreading-in-java/) 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

**Life Cycle of a thread**

1. **New Thread:** When a new thread is created, it is in the new state. The thread has not yet started to run when thread is in this state.
2. **Runnable State: In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run.**
3. **Blocked/Waiting state:** When a thread is temporarily inactive, then it’s in one of the following states:
   * Blocked
   * Waiting

A thread is in the **blocked state** when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the thread which is blocked for that section and moves it to the runnable state. Whereas, a thread is in the **waiting state** when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

1. **Timed Waiting:** 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 timed waiting state.
2. **Terminated State:** A thread terminates because of either of the following reasons:
   * Because it exists normally. This happens when the code of thread has entirely executed by the program.
   * Because there occurred some unusual erroneous event, like segmentation fault or an unhandled exception.

A thread that lies in terminated state does no longer consumes any cycles of CPU.

**Implementing Thread States in Java**

In Java, to get the current state of the thread, use **Thread.getState()** method to get the current state of the thread. Java provides **java.lang.Thread.State** class that defines the ENUM constants for the state of a thread, as summary of which is given below:

1. **Constant type:**New

Declaration: public static final Thread.State NEW

1. **Constant type:**Runnable

Declaration: public static final Thread.State RUNNABLE

1. **Constant type:**Blocked

Declaration: public static final Thread.State BLOCKED

1. **Constant type:**Waiting

Declaration: public static final Thread.State WAITING

A thread in the waiting state is waiting for another thread to perform a particular action. A thread is in the waiting state due to calling one of the following methods:

* + Object.wait with no timeout
  + [Thread.join](https://www.geeksforgeeks.org/joining-threads-in-java/) with no timeout
  + LockSupport.park

1. **Constant type:**Timed Waiting

Declaration: public static final Thread.State TIMED\_WAITING

A thread is in the timed waiting state due to calling one of the following methods with a specified positive waiting time:

* + Thread.sleep
  + Object.wait with timeout
  + Thread.join with timeout
  + LockSupport.parkNanos
  + LockSupport.parkUntil

1. **Constant type:**Terminated

Declaration: public static final Thread.State TERMINATED

|  |  |
| --- | --- |
| |  | | --- | | class thread implements Runnable{      public void run() {          try {              Thread.sleep(1500);          } catch (InterruptedException e) {              e.printStackTrace();          }           try {              Thread.sleep(1500);          } catch (InterruptedException e) {              e.printStackTrace();          }          System.out.println("State of thread1 while it called join() method on thread2 -"+ Test.thread1.getState());          try{              Thread.sleep(200);          } catch (InterruptedException e){              e.printStackTrace();          }      }  }   public class Test implements Runnable{      public static Thread thread1;      public static Test obj;       public static void main(String[] args) {          obj = new Test();          thread1 = new Thread(obj);           System.out.println("State of thread1 after creating it - " + thread1.getState());          thread1.start();           System.out.println("State of thread1 after calling .start() method on it - " + thread1.getState());      }       public void run(){          thread myThread = new thread();          Thread thread2 = new Thread(myThread);           System.out.println("State of thread2 after creating it - "+ thread2.getState());          thread2.start();         System.out.println("State of thread2 after calling .start() method on it - " + thread2.getState());          try {              Thread.sleep(200);          } catch (InterruptedException e){              e.printStackTrace();          }          System.out.println("State of thread2 after calling .sleep() method on it - "+ thread2.getState() );          try  {              thread2.join();          } catch (InterruptedException e){              e.printStackTrace();          }          System.out.println("State of thread2 when it has finished it's execution - " + thread2.getState());      }  }  **Output:**  State of thread1 after creating it - NEW  State of thread1 after calling .start() method on it - RUNNABLE  State of thread2 after creating it - NEW  State of thread2 after calling .start() method on it - RUNNABLE  State of thread2 after calling .sleep() method on it - TIMED\_WAITING  State of thread1 while it called join() method on thread2 -WAITING  State of thread2 when it has finished it's execution - TERMINATED | |

# 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.

**Properties:**

* 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**

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.

|  |  |
| --- | --- |
| |  | | --- | | public class Test extends Thread{      public static void main(String[] args) {          Thread t = Thread.currentThread();          System.out.println("Current thread: " + t.getName());          t.setName("Geeks");          System.out.println("After name change: " + t.getName());          System.out.println("Main thread priority: "+ t.getPriority());          t.setPriority(MAX\_PRIORITY);          System.out.println("Main thread new priority: "+ t.getPriority());          for (int i = 0; i < 5; i++) {              System.out.println("Main thread");          }           ChildThread ct = new ChildThread();           System.out.println("Child thread priority: "+ ct.getPriority());           ct.setPriority(MIN\_PRIORITY);           System.out.println("Child thread new priority: "+ ct.getPriority());           ct.start();      }  }   class ChildThread extends Thread{      @Override      public void run() {          for (int i = 0; i < 5; i++) {              System.out.println("Child thread");          }      }  }  **Output:**  Current thread: main  After name change: Geeks  Main thread priority: 5  Main thread new priority: 10  Main thread  Main thread  Main thread  Main thread  Main thread  Child thread priority: 10  Child thread new priority: 1  Child thread  Child thread  Child thread  Child thread  Child thread | |

**Relation between the main () method and main thread in Java**

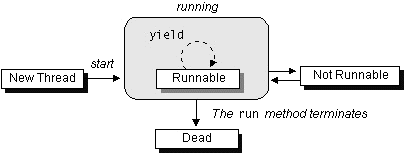
For each program, a Main thread is created by [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) (Java Virtual Machine). The “Main” thread first verifies the existence of the main () method, and then it initializes the class.

**Deadlocking with use of Main Thread (only single thread)**

We can create a deadlock by just using Main thread, i.e. by just using a single thread.

|  |
| --- |
| public class Test{      public static void main(String[] args) {          try{              System.out.println("Entering into Deadlock");              Thread.currentThread().join();              System.out.println("This statement will never execute");       } catch (InterruptedException e) {              e.printStackTrace();          }      }  }  **Output:**  Entering into Deadlock |

# Java Concurrency – yield (), sleep () and join () methods

1. **yield():** Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. Completion time for thread t1 is 5 hour and completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent execution of a thread in between if something important is pending. yeild() helps us in doing so.  
   **yield()**basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.  
   [](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/yield_method_java.gif)  
   **Use of yield method:**
   * Whenever a thread calls java.lang.Thread.yield method, it gives hint to the thread scheduler that it is ready to pause its execution. Thread scheduler is free to ignore this hint.
   * If any thread executes yield method , thread scheduler checks if there is any thread with same or high priority than this thread. If processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give processor to other thread and if not – current thread will keep executing.

**Syntax:**

public static native void yield()

|  |
| --- |
| // Java program to illustrate yield() method  // in Java  import java.lang.\*;    // MyThread extending Thread  class MyThread extends Thread  {      public void run()      {          for (int i=0; i<5 ; i++)              System.out.println(Thread.currentThread().getName()                                  + " in control");      }  }    // Driver Class  public class yieldDemo  {      public static void main(String[]args)      {          MyThread t = new MyThread();          t.start();            for (int i=0; i<5; i++)          {              // Control passes to child thread              Thread.yield();                // After execution of child Thread              // main thread takes over              System.out.println(Thread.currentThread().getName()                                  + " in control");          }      }  } |

Run on IDE

**Output:**

Thread-0 in control

Thread-0 in control

Thread-0 in control

Thread-0 in control

Thread-0 in control

main in control

main in control

main in control

main in control

main in control

Output may vary in machine to machine but chances of execution of yield() thread first is higher than the other thread because main thread is always pausing its execution and giving chance to child thread(with same priority).

**Note:**

* + Once a thread has executed yield method and there are many threads with same priority is waiting for processor, then we can't specify which thread will get execution chance first.
  + The thread which executes the yield method will enter in the Runnable state from Running state.
  + Once a thread pauses its execution, we can't specify when it will get chance again it depends on thread scheduler.
  + Underlying platform must provide support for preemptive scheduling if we are using yield method.

1. **sleep():** This method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.  
   **Syntax:**
2. // sleep for the specified number of milliseconds
3. public static void sleep(long millis) throws InterruptedException
4. //sleep for the specified number of milliseconds plus nano seconds
5. public static void sleep(long millis, int nanos)

throws InterruptedException

|  |
| --- |
| // Java program to illustrate  // sleep() method in Java  import java.lang.\*;    public class SleepDemo implements Runnable  {      Thread t;      public void run()      {          for (int i = 0; i < 4; i++)          {              System.out.println(Thread.currentThread().getName()                                                     + "  " + i);              try              {                  // thread to sleep for 1000 milliseconds                  Thread.sleep(1000);              }                catch (Exception e)              {                  System.out.println(e);              }          }      }        public static void main(String[] args) throws Exception      {          Thread t = new Thread(new SleepDemo());            // call run() function          t.start();            Thread t2 = new Thread(new SleepDemo());            // call run() function          t2.start();      }  } |

Run on IDE

Output:

Thread-0 0

Thread-1 0

Thread-0 1

Thread-1 1

Thread-0 2

Thread-1 2

Thread-0 3

Thread-1 3

Note:

* + Based on the requirement we can make a thread to be in sleeping state for a specified period of time
  + Sleep() causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, the CPU will be idle (and probably enter a power saving mode).

**yield() vs sleep()**

**yield:()** indicates that the thread is not doing anything particularly important and if any other threads or processes need to be run, they can. Otherwise, the **current thread will continue to run.**

**sleep()**: causes the thread to definitely stop executing for a given amount of time; if no other thread or process needs to be run, **the CPU will be idle** (and probably enter a power saving mode).

1. [**join():**](https://www.geeksforgeeks.org/joining-threads-in-java/) The join() method of a Thread instance is used to join the start of a thread’s execution to end of other thread’s execution such that a thread does not start running until another thread ends. If join() is called on a Thread instance, the currently running thread will block until the Thread instance has finished executing.  
   The join() method waits at most this much milliseconds for this thread to die. A timeout of 0 means to wait forever  
   Syntax:
2. // waits for this thread to die.
3. public final void join() throws InterruptedException
4. // waits at most this much milliseconds for this thread to die
5. public final void join(long millis)
6. throws InterruptedException
7. // waits at most milliseconds plus nanoseconds for this thread to die.

The java.lang.Thread.join(long millis, int nanos)

|  |
| --- |
| // Java program to illustrate join() method in Java  import java.lang.\*;    public class JoinDemo implements Runnable  {      public void run()      {          Thread t = Thread.currentThread();          System.out.println("Current thread: "                                 + t.getName());            // checks if current thread is alive          System.out.println("Is Alive? "                                 + t.isAlive());      }        public static void main(String args[]) throws Exception      {          Thread t = new Thread(new JoinDemo());          t.start();            // Waits for 1000ms this thread to die.          t.join(1000);            System.out.println("\nJoining after 1000"+                               " mili seconds: \n");          System.out.println("Current thread: " +                                      t.getName());              // Checks if this thread is alive          System.out.println("Is alive? " + t.isAlive());      }  } |

Run on IDE

Output:

Current thread: Thread-0

Is Alive? True

Joining after 1000 mili seconds:

Current thread: Thread-0

Is alive? False

Note:

* + If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.
  + Giving a timeout within join(), will make the join() effect to be nullified after the specific timeout.

# Inter-thread Communication in Java

**What is Polling and what are problems with it?**  
The process of testing a condition repeatedly till it becomes true is known as polling.

Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain action is taken. This waste many CPU cycles and makes the implementation inefficient.  
For example, in a classic queuing problem where one thread is producing data and other is consuming it.

**How Java multi threading tackles this problem?**  
To avoid polling, Java uses three methods, namely, wait(), notify() and notifyAll().  
All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

* **wait()-**It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
* **notify()-**It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.
* **notifyAll()-**It wakes up all the threads that called wait() on the same object.

**A simple Java program to demonstrate the three methods-**  
Please note that this program might only run in offline IDEs as it contains taking input at several points.

|  |
| --- |
| // Java program to demonstrate inter-thread communication  // (wait(), join() and notify()) in Java  import java.util.Scanner;  public class threadexample  {      public static void main(String[] args)                             throws InterruptedException      {          final PC pc = new PC();            // Create a thread object that calls pc.produce()          Thread t1 = new Thread(new Runnable()          {              @Override              public void run()              {                  try                  {                      pc.produce();                  }                  catch(InterruptedException e)                  {                      e.printStackTrace();                  }              }          });            // Create another thread object that calls          // pc.consume()          Thread t2 = new Thread(new Runnable()          {              @Override              public void run()              {                  try                  {                      pc.consume();                  }                  catch(InterruptedException e)                  {                      e.printStackTrace();                  }              }          });            // Start both threads          t1.start();          t2.start();            // t1 finishes before t2          t1.join();          t2.join();      }        // PC (Produce Consumer) class with produce() and      // consume() methods.      public static class PC      {          // Prints a string and waits for consume()          public void produce()throws InterruptedException          {              // synchronized block ensures only one thread              // running at a time.              synchronized(this)              {                  System.out.println("producer thread running");                    // releases the lock on shared resource                  wait();                    // and waits till some other method invokes notify().                  System.out.println("Resumed");              }          }            // Sleeps for some time and waits for a key press. After key          // is pressed, it notifies produce().          public void consume()throws InterruptedException          {              // this makes the produce thread to run first.              Thread.sleep(1000);              Scanner s = new Scanner(System.in);                // synchronized block ensures only one thread              // running at a time.              synchronized(this)              {                  System.out.println("Waiting for return key.");                  s.nextLine();                  System.out.println("Return key pressed");                    // notifies the produce thread that it                  // can wake up.                  notify();                    // Sleep                  Thread.sleep(2000);              }          }      }  } |

Run on IDE

Output:

producer thread running

Waiting for return key.

Return key pressed

Resumed

As monstrous as it seems, it really is a piece of cake if you go through it twice.

1. In the main class a new PC object is created.
2. It runs produce and consume methods of PC object using two different threads namely t1 and t2 and wait for these threads to finish.

Lets understand how our produce and consume method works.

* First of all, use of synchronized block ensures that only one thread at a time runs. Also since there is a sleep method just at the beginning of consume loop, the produce thread gets a kickstart.
* When the wait is called in produce method, it does two things. Firstly it releases the lock it holds on PC object. Secondly it makes the produce thread to go on a waiting state until all other threads have terminated, that is it can again acquire a lock on PC object and some other method wakes it up by invoking notify or notifyAll on the same object.
* Therefore we see that as soon as wait is called, the control transfers to consume thread and it prints -“Waiting for return key”.
* After we press the return key, consume method invokes notify(). It also does 2 things- Firstly, unlike wait(), it does not releases the lock on shared resource therefore for getting the desired result, it is advised to use notify only at the end of your method. Secondly, it notifies the waiting threads that now they can wake up but only after the current method terminates.
* As you might have observed that even after notifying, the control does not immediately passes over to the produce thread. The reason for it being that we have called Thread.sleep() after notify(). As we already know that the consume thread is holding a lock on PC object, another thread cannot access it until it has released the lock. Hence only after the consume thread finishes its sleep time and thereafter terminates by itself, the produce thread cannot take back the control.
* After a 2 second pause, the program terminates to its completion.

If you are still confused as to why we have used notify in consume thread, try removing it and running your program again. As you must have noticed now that the program never terminates.  
The reason for this is straightforward-When you called wait on produce thread, it went on waiting and never terminated. Since a program runs till all its threads have terminated, it runs on and on.  
There is a second way round this problem. You can use a second variant of wait().

void wait(long timeout)

This would make the calling thread sleep only for a time specified.

# Java.lang.Thread class in Java

[2.2](https://www.geeksforgeeks.org/easy/)

Thread a line of execution within a program. Each program can have multiple associated threads. Each thread has a priority which is used by thread scheduler to determine which thread must run first. Java provides a thread class that has various method calls inorder to manage the behaviour of threads.  
**Note:** Every class that is used as thread must implement Runnable interface and over ride it’s run method.

**Constructors:**

* **Thread()**: Allocates a new Thread object
* **Thread(Runnable target)**: Allocates a new Thread object
* **Thread(Runnable target, String name)**: Allocates a new Thread object
* **Thread(String name)**: Allocates a new Thread object
* **Thread(ThreadGroup group, Runnable target)**: Allocates a new Thread object
* **Thread(ThreadGroup group, Runnable target, String name)**: Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group
* **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**: Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group, and has the specified stack size
* **Thread(ThreadGroup group, String name)**: CAllocates a new Thread object

**Declaration:**

**public class Thread**

**extends Object**

**implements Runnable**

**Methods:**

1. **activeCount(): java.lang.Thread.activeCount()**Returns an estimate of the number of active threads in the current thread’s thread group and its subgroups

**Syntax:**

public static int activeCount()

**Returns:**

an estimate of the number of active threads in the current

thread's thread group and in any other thread group that has

the current thread's thread group as an ancestor

1. **checkAccess(): java.lang.Thread.checkAccess()**Determines if the currently running thread has permission to modify this thread

**Syntax:**

public final void checkAccess()

**Throws:**

SecurityException - if the current thread is not allowed to

access this thread.

1. **clone(): java.lang.Thread.clone()**Throws CloneNotSupportedException as a Thread can not be meaningfully cloned

**Syntax:**

protected Object clone() throws CloneNotSupportedException

**Throws:**

CloneNotSupportedException - always

**Returns:**

a clone of this instance

1. **currentThread(): java.lang.Thread.currentThread()**Returns a reference to the currently executing thread object

**Syntax:**

public static Thread currentThread()

**Returns:**

the currently executing thread

1. **dumpStack(): java.lang.Thread.dumpStack()**Prints a stack trace of the current thread to the standard error stream

**Syntax:**

public static void dumpStack()

**Description:**

Prints a stack trace of the current thread to the standard

error stream. This method is used only for debugging

1. **enumerate(Thread[] tarray): java.lang.Thread.enumerate(Thread[] tarray)**Copies into the specified array every active thread in the current thread’s thread group and its subgroups

**Syntax:**

public static int enumerate(Thread[] tarray)

**Parameters:**

tarray - an array into which to put the list of threads

**Returns:**

the number of threads put into the array

**Throws:**

SecurityException - if ThreadGroup.checkAccess()

determines that the current thread cannot

access its thread group

1. **getAllStackTraces(): java.lang.Thread.getAllStackTraces()**Returns a map of stack traces for all live threads

**Syntax:**

public static Map getAllStackTraces()

**Returns:**

a Map from Thread to an array of StackTraceElement that represents

the stack trace of the corresponding thread

**Throws:**

SecurityException - if a security manager exists and its checkPermission method

doesn't allow getting the stack trace of thread

1. **getContextClassLoader(): java.lang.Thread.getContextClassLoader()**Returns the context ClassLoader for this Thread

**Syntax:**

public ClassLoader getContextClassLoader()

**Returns:**

the context ClassLoader for this Thread, or null indicating the system class loader

(or, failing that, the bootstrap class loader)

**Throws:**

SecurityException - if the current thread cannot get the context ClassLoader

1. **getDefaultUncaughtExceptionHandler(): java.lang.Thread.getDefaultUncaughtExceptionHandler()**Returns the default handler invoked when a thread abruptly terminates due to an uncaught exception

**Syntax:**

public static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()

**Returns:**

the default uncaught exception handler for all threads

1. **getId(): java.lang.Thread.getId()**Returns the identifier of this Thread

**Syntax:**

public long getId()

**Returns:**

this thread's ID

1. **getName(): java.lang.Thread.getName()**Returns this thread’s name

**Syntax:**

public final String getName()

**Returns:**

this thread's name

1. **getPriority(): java.lang.Thread.getPriority()**Returns this thread’s priority

**Syntax:**

public final int getPriority()

**Returns:**

this thread's priority

1. **getStackTrace(): java.lang.Thread.getStackTrace()**Returns an array of stack trace elements representing the stack dump of this thread

**Syntax:**

public StackTraceElement[] getStackTrace()

**Returns:**

an array of StackTraceElement, each represents one stack frame

**Throws:**

SecurityException - if a security manager exists and its

checkPermission method doesn't

allow getting the stack trace of thread

1. **getState(): java.lang.Thread.getState()**Returns the state of this thread

**Syntax:**

public Thread.State getState()

**Returns:**

this thread's state

1. **getThreadGroup(): java.lang.Thread.getThreadGroup()**Returns the thread group to which this thread belongs

**Syntax:**

public final ThreadGroup getThreadGroup()

**Returns:**

this thread's thread group

1. **getUncaughtExceptionHandler(): java.lang.Thread.getUncaughtExceptionHandler()**Returns the handler invoked when this thread abruptly terminates due to an uncaught exception

**Syntax:**

public Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()

**Returns:**

the uncaught exception handler for this thread

1. **holdsLock(Object obj): java.lang.Thread.holdsLock(Object obj)**Returns true if and only if the current thread holds the monitor lock on the specified object

**Syntax:**

public static boolean holdsLock(Object obj)

**Parameters:**

obj - the object on which to test lock ownership

**Returns:**

true if the current thread holds the monitor lock

on the specified object.

**Throws:**

NullPointerException - if obj is null

1. **interrupt(): java.lang.Thread.interrupt()**Interrupts this thread

**Syntax:**

public void interrupt()

**Throws:**

SecurityException - if the current thread cannot modify this thread

1. **interrupted(): java.lang.Thread.interrupted()**Tests whether the current thread has been interrupted

**Syntax:**

public static boolean interrupted()

**Returns:**

true if the current thread has been interrupted; false otherwise

1. **isAlive(): java.lang.Thread.isAlive()**Tests if this thread is alive

**Syntax:**

public final boolean isAlive()

**Returns:**

true if this thread is alive; false otherwise

1. **isDaemon(): java.lang.Thread.isDaemon()**Tests if this thread is a daemon thread

**Syntax:**

public final boolean isDaemon()

**Returns:**

true if this thread is a daemon thread; false otherwise

1. **isInterrupted(): java.lang.Thread.isInterrupted()**Tests whether this thread has been interrupted

**Syntax:**

public boolean isInterrupted()

**Returns:**

true if this thread has been interrupted; false otherwise

1. **join(): java.lang.Thread.join()**Waits for this thread to die

**Syntax:**

public final void join() throws InterruptedException

**Throws:**

InterruptedException - if any thread has interrupted the

current thread. The interrupted status of the current thread is

cleared when this exception is thrown

1. **join(long millis): java.lang.Thread.join(long millis)**Waits at most millis milliseconds for this thread to die

**Syntax:**

public final void join(long millis) throws InterruptedException

**Parameters:**

millis - the time to wait in milliseconds

**Throws:**

IllegalArgumentException - if the value of millis is negative

InterruptedException - if any thread has interrupted the current thread.

The interrupted status of the current thread is cleared

when this exception is thrown.

1. **run(): java.lang.Thread.run()**If this thread was constructed using a separate Runnable run object, then that Runnable object’s run method is called; otherwise, this method does nothing and returns

**Syntax:**

public void run()

**Description:**

If this thread was constructed using a separate Runnable run object,

then that Runnable object's run method is called;

otherwise, this method does nothing and returns.

Subclasses of Thread should override this method.

1. **yield(): java.lang.Thread.yield()**A hint to the scheduler that the current thread is willing to yield its current use of a processor

**Syntax:**

public static void yield()

**Description:**

A hint to the scheduler that the current thread is willing

to yield its current use of a

processor. The scheduler is free to ignore this hint

1. **toString(): java.lang.Thread.toString()**Returns a string representation of this thread, including the thread’s name, priority, and thread group

**Syntax:**

public String toString()

**Returns:**

a string representation of this thread

1. **start(): java.lang.Thread.start()**Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread

**Syntax:**

public void start()

**Throws:**

IllegalThreadStateException - if the thread was already started.

1. **sleep(long millis): java.lang.Thread.sleep(long millis)**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers

**Syntax:**

public static void sleep(long millis) throws InterruptedException

**Parameters:**

millis - the length of time to sleep in milliseconds

**Throws:**

IllegalArgumentException - if the value of millis is negative

InterruptedException - if any thread has interrupted the

current thread. The interrupted status of the current thread

is cleared when this exception is thrown.

1. **setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh): java.lang.Thread.setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)**Set the handler invoked when this thread abruptly terminates due to an uncaught exception

**Syntax:**

public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)

**Parameters:**

eh - the object to use as this thread's uncaught exception handler.

If null then this thread has no explicit handler.

**Throws:**

SecurityException - if the current thread is not allowed to

modify this thread.

1. **setPriority(int newPriority): java.lang.Thread.setPriority(int newPriority)**Changes the priority of this thread

**Syntax:**

public final void setPriority(int newPriority)

**Parameters:**

newPriority - priority to set this thread to

**Throws:**

IllegalArgumentException- If the priority is not in the range

MIN\_PRIORITY to MAX\_PRIORITY

SecurityException - if the current thread cannot modify this thread.

1. **setName(String name): java.lang.Thread.setName(String name)**Changes the name of this thread to be equal to the argument name.

**Syntax:**

public final void setName(String name)

**Parameters:**

name - the new name for this thread.

**Throws:**

SecurityException - if the current thread cannot modify this thread.

1. **setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh): java.lang.Thread.setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)**Set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread

**Syntax:**

public static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)

**Parameters:**

eh - the object to use as the default uncaught exception handler.

If null then there is no

default handler.

**Throws:**

SecurityException - if a security manager is present and

it denies RuntimePermission ("setDefaultUncaughtExceptionHandler")

1. **setDaemon(boolean on): java.lang.Thread.setDaemon(boolean on)**Marks this thread as either a daemon thread or a user thread

**Syntax:**

public final void setDaemon(boolean on)

**Parameters:**

on - if true, marks this thread as a daemon thread

**Throws:**

IllegalThreadStateException - if this thread is alive

SecurityException - if checkAccess() determines that the current

thread cannot modify this thread

1. **setContextClassLoader(ClassLoader cl): java.lang.Thread.setContextClassLoader(ClassLoader cl)**Sets the context ClassLoader for this Thread

**Syntax:**

public void setContextClassLoader(ClassLoader cl)

**Parameters:**

cl - the context ClassLoader for this Thread, or null indicating the

system class loader (or, failing that, the bootstrap class loader)

**Throws:**

SecurityException - if the current thread cannot set the

context ClassLoader

**Methods inherited from class java.lang.Object**

* equals
* finalize
* getClass
* hashCode
* notify
* notifyAll
* toString
* wait

**Java program to demonstrate usage of Thread class**

|  |
| --- |
| // Java program to demonstrate  // method calls of Thread class  package generic;  class Helper implements Runnable  {      public void run()      {          try          {              System.out.println("thread2 going to sleep for 5000");              Thread.sleep(5000);          } catch (InterruptedException e)          {              System.out.println("Thread2 interrupted");}          }  }    public class Test implements Runnable  {      public void run()      {          //thread run() method      }      public static void main(String[] args)      {          Test obj = new Test();          Helper obj2 = new Helper();            Thread thread1 = new Thread(obj);          Thread thread2 = new Thread(obj2);            // moving thread to runnable states          thread1.start();          thread2.start();            ClassLoader loader = thread1.getContextClassLoader();          Thread thread3 = new Thread(new Helper());            // getting number of active threads          System.out.println(Thread.activeCount());          thread1.checkAccess();            // fetching an instance of this thread          Thread t = Thread.currentThread();            System.out.println(t.getName());            System.out.println("Thread1 name: "+thread1.getName());          System.out.println("Thread1 ID: " + thread1.getId());            // fetching the priority and state of thread1          System.out.println("Priority of thread1 = " + thread1.getPriority());            System.out.println(thread1.getState());            thread2 = new Thread(obj2);          thread2.start();          thread2.interrupt();          System.out.println("Is thread2 interrupted? " + thread2.interrupted() );          System.out.println("Is thread2 alive? " + thread2.isAlive());            thread1 = new Thread(obj);          thread1.setDaemon(true);          System.out.println("Is thread1 a daemon thread? " + thread1.isDaemon());          System.out.println("Is thread1 interrupted? " + thread1.isInterrupted());            // waiting for thread2 to complete its execution          System.out.println("thread1 waiting for thread2 to join");          try          {              thread2.join();          }          catch (InterruptedException e)          {              e.printStackTrace();          }            // setting the name of thread1          thread1.setName("child thread xyz");          System.out.println("New name set for thread 1" + thread1.getName());            // setting the priority of thread1          thread1.setPriority(5);            thread2.yield();            // fetching the string representation of thread1          System.out.println(thread1.toString());            // getting list of active thread in current thread's group          Thread[] tarray = new Thread[3];            Thread.enumerate(tarray);          System.out.println("List of active threads:");          System.out.printf("[");          for(Thread thread : tarray)          {              System.out.println(thread);          }          System.out.printf("]\n");            System.out.println(Thread.getAllStackTraces());            ClassLoader classLoader = thread1.getContextClassLoader();          System.out.println(classLoader.toString());          System.out.println(thread1.getDefaultUncaughtExceptionHandler());            thread2.setUncaughtExceptionHandler(thread1.getDefaultUncaughtExceptionHandler());          thread1.setContextClassLoader(thread2.getContextClassLoader());          thread1.setDefaultUncaughtExceptionHandler(thread2.getUncaughtExceptionHandler());            thread1 = new Thread(obj);          StackTraceElement[] trace = thread1.getStackTrace();          System.out.println("Printing stack trace elements for thread1:");          for(StackTraceElement e : trace)          {              System.out.println(e);          }            ThreadGroup grp = thread1.getThreadGroup();          System.out.println("ThreadGroup to which thread1 belongs " +grp.toString());          System.out.println(thread1.getUncaughtExceptionHandler());          System.out.println("Does thread1 holds Lock? " + thread1.holdsLock(obj2));              Thread.dumpStack();        }  } |

Run on IDE

**Output:**

3

main

Thread1 name: Thread-0

Thread1 ID: 10

Priority of thread1 = 5

RUNNABLE

Is thread2 interrupted? false

Is thread2 alive? true

Is thread1 a daemon thread? true

Is thread1 interrupted? false

thread1 waiting for thread2 to join

thread2 going to sleep for 5000 ms

thread2 going to sleep for 5000 ms

Thread2 interrupted

New name set for thread 1child thread xyz

Thread[child thread xyz, 5, main]

List of active threads:

[Thread[main, 5, main]

Thread[Thread-1, 5, main]

null

]

{Thread[Signal Dispatcher, 9, system]=[Ljava.lang.StackTraceElement;@33909752,

Thread[Thread-1, 5, main]=[Ljava.lang.StackTraceElement;@55f96302,

Thread[main, 5, main]=[Ljava.lang.StackTraceElement;@3d4eac69,

Thread[Attach Listener, 5, system]=[Ljava.lang.StackTraceElement;@42a57993,

Thread[Finalizer, 8, system]=[Ljava.lang.StackTraceElement;@75b84c92,

Thread[Reference Handler, 10, system]=[Ljava.lang.StackTraceElement;@6bc7c054}

sun.misc.Launcher$AppClassLoader@73d16e93

null

Printing stack trace elements for thread1:

ThreadGroup to which thread1 belongs java.lang.ThreadGroup[name=main, maxpri=10]

java.lang.ThreadGroup[name=main, maxpri=10]

Does thread1 holds Lock? false

java.lang.Exception: Stack trace

at java.lang.Thread.dumpStack(Unknown Source)

at generic.Test.main(Test.java:111)

# What does start() function do in multithreading in Java?

[2.3](https://www.geeksforgeeks.org/easy/)

We have discussed that [Java threads are typically created using one of the two methods : (1) Extending thread class. (2) Implementing Runnable](http://quiz.geeksforgeeks.org/multithreading-in-java/)

In both the approaches, we override the run() function, but we start a thread by calling the start() function. So why don’t we directly call the oveerridden run() function? Why always the start function is called to execute a thread?

**What happens when a function is called?**  
When a function is called the following operations take place:

1. The arguments are evaluated.
2. A new stack frame is pushed into the call stack.
3. Parameters are initialized.
4. Method body is executed.
5. Value is retured and current stack frame is popped from the call stack.

**The purpose of start() is to create a separate call stack for the thread. A separate call stack is created by it, and then run() is called by JVM.**

Let us see what happens if we don’t call start() and rather call run() directly. We have modified the first program discussed [here](http://quiz.geeksforgeeks.org/multithreading-in-java/).

|  |
| --- |
| // Java code to see that all threads are  // pushed on same stack if we use run()  // instead of start().  class ThreadTest extends Thread  {    public void run()    {      try      {        // Displaying the thread that is running        System.out.println ("Thread " +                  Thread.currentThread().getId() +                  " is running");        }      catch (Exception e)      {        // Throwing an exception        System.out.println ("Exception is caught");      }    }  }    // Main Class  public class Main  {    public static void main(String[] args)    {      int n = 8;      for (int i=0; i<n; i++)      {        ThreadTest object = new ThreadTest();          // start() is replaced with run() for        // seeing the purpose of start        object.run();      }    }  } |

Run on IDE

Output:

Thread 1 is running

Thread 1 is running

Thread 1 is running

Thread 1 is running

Thread 1 is running

Thread 1 is running

Thread 1 is running

Thread 1 is running

# Java Thread Priority in Multithreading

[1.5](https://www.geeksforgeeks.org/basic/)

In a Multi threading environment, thread scheduler assigns processor to a thread based on priority of thread. 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 programmer explicitly.  
Accepted value of priority for a thread is in range of 1 to 10. There are 3 static variables defined in Thread class for priority.

**public static int MIN\_PRIORITY:** This is minimum priority that a thread can have. Value for this is 1.  
**public static int NORM\_PRIORITY:** This is default priority of a thread if do not explicitly define it. Value for this is 5.  
**public static int MAX\_PRIORITY:** This is maximum priority of a thread. Value for this is 10.

**Get and Set Thread Priority:**

1. **public final int getPriority():** java.lang.Thread.getPriority() method returns priority of given thread.
2. **public final void setPriority(int newPriority):** java.lang.Thread.setPriority() 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.

**Examples of getPriority() and set**

|  |
| --- |
| // Java program to demonstrate getPriority() and setPriority()  import java.lang.\*;    class ThreadDemo extends Thread  {      public void run()      {          System.out.println("Inside run method");      }        public static void main(String[]args)      {          ThreadDemo t1 = new ThreadDemo();          ThreadDemo t2 = new ThreadDemo();          ThreadDemo t3 = new ThreadDemo();            System.out.println("t1 thread priority : " +                                t1.getPriority()); // Default 5          System.out.println("t2 thread priority : " +                                t2.getPriority()); // Default 5          System.out.println("t3 thread priority : " +                                t3.getPriority()); // Default 5            t1.setPriority(2);          t2.setPriority(5);          t3.setPriority(8);            // t3.setPriority(21); will throw IllegalArgumentException          System.out.println("t1 thread priority : " +                                t1.getPriority());  //2          System.out.println("t2 thread priority : " +                                t2.getPriority()); //5          System.out.println("t3 thread priority : " +                                t3.getPriority());//8            // Main thread          System.out.print(Thread.currentThread().getName());          System.out.println("Main thread priority : "                         + Thread.currentThread().getPriority());            // Main thread priority is set to 10          Thread.currentThread().setPriority(10);          System.out.println("Main thread priority : "                         + Thread.currentThread().getPriority());      }  } |

Run on IDE

Output:

t1 thread priority : 5

t2 thread priority : 5

t3 thread priority : 5

t1 thread priority : 2

t2 thread priority : 5

t3 thread priority : 8

Main thread priority : 5

Main thread priority : 10

**Note:**

* Thread with highest priority will get execution chance prior to other threads. Suppose there are 3 threads t1, t2 and t3 with priorities 4, 6 and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.
* Default priority for main thread is always 5, it can be changed later. Default priority for all other threads depends on the priority of parent thread.

**Example:**

|  |
| --- |
| // Java program to demonstrat ethat a child thread  // gets same priority as parent  import java.lang.\*;    class ThreadDemo extends Thread  {      public void run()      {          System.out.println("Inside run method");      }        public static void main(String[]args)      {          // main thread priority is 6 now          Thread.currentThread().setPriority(6);            System.out.println("main thread priority : " +                     Thread.currentThread().getPriority());            ThreadDemo t1 = new ThreadDemo();            // t1 thread is child of main thread          // so t1 thread will also have priority 6.          System.out.println("t1 thread priority : "                                    + t1.getPriority());      }  } |

Run on IDE

Output:

Main thread priority : 6

t1 thread priority : 6

* If two threads have same priority then we can’t expect which thread will execute first. It depends on thread scheduler’s algorithm(Round-Robin, First Come First Serve, etc)
* If we are using thread priority for thread scheduling then we should always keep in mind that underlying platform should provide support for scheduling based on thread priority.

# Joining Threads in Java

[3.4](https://www.geeksforgeeks.org/medium/)

**java.lang.Thread** class provides the join() method which allows one thread to wait until another thread completes its execution. If **t** is a Thread object whose thread is currently executing, then **t.join();** it causes the current thread to pause its execution until thread it join completes its execution.  
If there are multiple threads calling the join() methods that means overloading on join allows the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.  
There are three overloaded join functions.

1. **join():**It will put the current thread on wait until the thread on which it is called is dead. If thread is interrupted then it will throw InterruptedException.  
   **Syntax:**
2. public final void join()
3. **join(long millis)**:It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds).  
   **Syntax:**
4. public final synchronized void join(long millis)
5. **join(long millis, int nanos):**It will put the current thread on wait until the thread on which it is called is dead or wait for specified time (milliseconds + nanos).  
   **Syntax:**
6. public final synchronized void join(long millis, int nanos)

|  |
| --- |
| // Java program to explain the  // concept of joining a thread.  import java.io.\*;    // Creating thread by creating the  // objects of that class  class ThreadJoining extends Thread  {      @Override      public void run()      {          for (int i = 0; i < 2; i++)          {              try              {                  Thread.sleep(500);                  System.out.println("Current Thread: "                          + Thread.currentThread().getName());              }                catch(Exception ex)              {                  System.out.println("Exception has" +                                  " been caught" + ex);              }              System.out.println(i);          }      }  }    class GFG  {      public static void main (String[] args)      {            // creating two threads          ThreadJoining t1 = new ThreadJoining();          ThreadJoining t2 = new ThreadJoining();          ThreadJoining t3 = new ThreadJoining();            // thread t1 starts          t1.start();            // starts second thread after when          // first thread t1 is died.          try          {              System.out.println("Current Thread: "                    + Thread.currentThread().getName());              t1.join();          }            catch(Exception ex)          {              System.out.println("Exception has " +                                  "been caught" + ex);          }            // t2 starts          t2.start();            // starts t3 after when thread t2 is died.          try          {              System.out.println("Current Thread: "                   + Thread.currentThread().getName());              t2.join();          }            catch(Exception ex)          {              System.out.println("Exception has been" +                                      " caught" + ex);          }            t3.start();      }  } |

1. Run on IDE
2. output:
3. Current Thread: main
4. Current Thread: Thread-0
5. 0
6. Current Thread: Thread-0
7. 1
8. Current Thread: main
9. Current Thread: Thread-1
10. 0
11. Current Thread: Thread-1
12. 1
13. Current Thread: Thread-2
14. 0
15. Current Thread: Thread-2
16. 1

# Naming a thread and fetching name of current thread in Java

[1.6](https://www.geeksforgeeks.org/basic/)

**Naming a Thread**

In Java, each thread has a name i.e Thread-0, Thread-1, Thread-2,….so on. Java provides some methods to change the thread name. There are basically two methods to set the thread name. Both methods are defined in java.lang.Thread class.

1. **Setting the thread’s name directly:**We can set the thread name at time of creating the thread and by passing the thread’s name as an argument.

|  |
| --- |
| // Java program to illustrate  // how to set the name  // of a thread at time of  // thread creation  import java.io.\*;    // we can create thread by creating the  // objects of that class  class ThreadNaming extends Thread  {        ThreadNaming(String name)      {          // call to constructor of          // the Thread class.          super(name);      }        @Override      public void run()      {          System.out.println("Thread is running.....");      }  }    class GFG  {      public static void main (String[] args)      {          // creating two threads          ThreadNaming t1 = new ThreadNaming("geek1");          ThreadNaming t2 = new ThreadNaming("geek2");            // getting the above created threads names.          System.out.println("Thread 1: " + t1.getName());          System.out.println("Thread 2: " + t2.getName());            t1.start();          t2.start();      }  } |

1. Run on IDE
2. Output:
3. Thread 1: geek1
4. Thread 2: geek2
5. Thread is running.....
6. Thread is running.....
7. **Using setName(String threadName) method:** We can set(change) the thread’s name by calling the setName method on that thread object.  
   **Syntax:**
8. public final void setName(String name)
9. // it will change the name of a thread.

|  |
| --- |
| // Java program to illustrate  // how to get and change the  // name of a thread.  import java.io.\*;    // we can create thread by creating the  // objects of that class.  class ThreadNaming extends Thread  {        @Override      public void run()      {          System.out.println("Thread is running.....");      }  }    class GFG  {      public static void main (String[] args)      {          // creating two threads          ThreadNaming t1 = new ThreadNaming();          ThreadNaming t2 = new ThreadNaming();            // getting the above created threads names.          System.out.println("Thread 1: " + t1.getName());          System.out.println("Thread 2: " + t2.getName());            t1.start();          t2.start();            // Now changing the name of threads.          t1.setName("geeksforgeeks");          t2.setName("geeksquiz");            // again getting the new names          // of the threads.          System.out.println("Thread names after changing the "+          "thread names");          System.out.println("New Thread 1 name:  " + t1.getName());          System.out.println("New Thread 2 name: " + t2.getName());        }  } |

1. Run on IDE
2. Output:
3. Thread 1: Thread-0
4. Thread 2: Thread-1
5. Thread names after changing the thread names
6. New Thread 1 name: geeksforgeeks
7. New Thread 2 name: geeksquiz
8. Thread is running.....
9. Thread is running.....

**Fetching the name of Current thread**

We can fetch the current thread name at time of creating the thread and by passing the thread’s name as an argument.  
**Syntax:**

public static Thread currentThread()

// It is defined in java.langThread class.

// It returns reference to currently executing thread.

|  |
| --- |
| // Java program to illustrate  // how to get name of current  // executing thread.  import java.io.\*;    // we can create thread by creating the  // objects of that class  class ThreadNaming extends Thread  {        @Override      public void run()      {          // getting the current thread 's name.          System.out.println("Fetching current thread name..");          System.out.println(Thread.currentThread().getName());      }  }    class GFG  {      public static void main (String[] args)      {          // creating two threads          ThreadNaming t1 = new ThreadNaming();          ThreadNaming t2 = new ThreadNaming();            t1.start();          t2.start();      }  } |

Run on IDE

output:

Fetching current thread name..

Thread-0

Fetching current thread name..

Thread-1

Synchronized in Java

[3.4](https://www.geeksforgeeks.org/medium/)

[Multi-threaded](http://quiz.geeksforgeeks.org/multithreading-in-java/)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 of time.

Java provides a way of creating threads and synchronizing their task by 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 synchronized 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.

Following is the general form of a synchronized block:

// Only one thread can execute at a time.

// sync\_object is a reference to an object

// whose lock associates with the [monitor](http://quiz.geeksforgeeks.org/monitors/).

// The code is said to be synchronized on

// the monitor object

synchronized(sync\_object)

{

// Access shared variables and other

// shared resources

}

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.

Following is an example of multi threading with synchronized.

|  |
| --- |
| // A Java program to demonstrate working of  // synchronized.  import java.io.\*;  import java.util.\*;    // A Class used to send a message  class Sender  {      public void send(String msg)      {          System.out.println("Sending\t"  + msg );          try          {              Thread.sleep(1000);          }          catch (Exception e)          {              System.out.println("Thread  interrupted.");          }          System.out.println("\n" + msg + "Sent");      }  }    // Class for send a message using Threads  class ThreadedSend extends Thread  {      private String msg;      private Thread t;      Sender  sender;        // Recieves a message object and a string      // message to be sent      ThreadedSend(String m,  Sender obj)      {          msg = m;          sender = obj;      }        public void run()      {          // Only one thread can send a message          // at a time.          synchronized(sender)          {              // synchronizing the snd object              sender.send(msg);          }      }  }    // Driver class  class SyncDemo{      public static void main(String args[]){          Sender snd = new Sender();          ThreadedSend S1 =              new ThreadedSend( " Hi " , snd );          ThreadedSend S2 =              new ThreadedSend( " Bye " , snd );            // Start two threads of ThreadedSend type          S1.start();          S2.start();            // wait for threads to end          try{              S1.join();              S2.join();          }catch(Exception e){              System.out.println("Interrupted");          }      }  } |

Run on IDE

Output:

Sending Hi

Hi Sent

Sending Bye

Bye Sent

The output is same every-time we run the program.

In the above example, we chose to synchronize the Sender object inside the run () method of the ThreadedSend class. Alternately, we could define the whole send () block as synchronized and it would produce the same result. Then we don’t have to synchronize the Message object inside the run () method in ThreadedSend class.

|  |
| --- |
| **// An alternate implementation to demonstrate**  **// that we can use synchronized with method also.**  class Sender{      public synchronized void send(String msg){          System.out.println("Sending\t" + msg );          try{              Thread.sleep(1000);          } catch (Exception e){              System.out.println ("Thread interrupted.");          }          System.out.println("\n" + msg + "Sent");      }  } |

We do not always have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

|  |
| --- |
| **// One more alternate implementation to demonstrate**  **// that synchronized can be used with only a part of**  **// method**  class Sender {      public void send(String msg){          synchronized(this){              System.out.println("Sending\t" + msg );              try{                  Thread.sleep(1000);              }catch (Exception e){                  System.out.println ("Thread interrupted.");              }              System.out.println("\n" + msg + "Sent");          }      }  } |

# Method and Block Synchronization in Java

[**3.2**](https://www.geeksforgeeks.org/medium/)

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors. Some synchronization constructs are needed to prevent these errors. Following example shows a situation where we need synchronization.

**Need of Synchronization**

Consider the following Example:

|  |
| --- |
| // Java program to illustrate need  // of Synchronization  import java.io.\*;    class Multithread  {      private int i = 0;      public void increment()      {          i++;      }        public int getValue()      {          return i;      }  }    class GfG  {      public static void main (String[] args)      {          Multithread t = new Multithread();          t.increment();          System.out.println(t.getValue());      }  } |

Run on IDE

Output:

1

In above example three operations are performed:

1. Fetch the value of variable i.
2. Increment the fetched value.
3. And store the increased value of i to its location.

Here,

* 1st thread fetches the value of i. (Currently value i is 0) and increases it by one, so value of variable i becomes 1.
* Now 2nd thread accesses the value of i that would be 0 as 1st thread did not store it back to its location.  
  And 2nd thread also increment it and store it back to its location. And 1st also store it.
* Finally value of variable i is 1. But it should be 2 by the effect of both threads. That’s why we need to synchronize the access to shared variable i.

Java is multi-threaded language where multiple threads runs parallel to complete their execution. We need to synchronize the shared resources to ensure that at a time only one thread is able to access the shared resource.  
If an Object is shared by multiple threads then there is need of synchronization in order to avoid the Object’s state to be getting corrupted. Synchronization is needed when Object is mutable. If shared Object is immutable or all the threads which share the same Object are only reading the Object’s state not modifying then you don’t need to synchronize it.

Java programming language provide two synchronization idioms:

* Methods synchronization
* Statement(s) synchronization (Block synchronization)

**Method Synchronization**

Synchronized methods enables a simple strategy for preventing the thread interference and memory consistency errors. If a Object is visible to more than one threads, all reads or writes to that Object’s fields are done through the **synchronized** method.

It is not possible for two invocations for synchronized methods to interleave. If one thread is executing the synchronized method, all others thread that invoke synchronized method on the same Object will have to wait until first thread is done with the Object.

**Example: This shows if more than one threads accessing getLine() method without synchronization.**

|  |
| --- |
| // Example illustrates multiple threads are executing  // on the same Object at same time without synchronization.  import java.io.\*;    class Line  {      // if multiple threads(trains) will try to      // access this unsynchronized method,      // they all will get it. So there is chance      // that Object's  state will be corrupted.      public void getLine()      {          for (int i = 0; i < 3; i++)          {              System.out.println(i);              try              {                  Thread.sleep(400);              }              catch (Exception e)              {                  System.out.println(e);              }          }      }  }    class Train extends Thread  {      // reference to Line's Object.      Line line;        Train(Line line)      {          this.line = line;      }        @Override      public void run()      {          line.getLine();      }  }    class GFG  {      public static void main(String[] args)      {          // Object of Line class that is shared          // among the threads.          Line obj = new Line();            // creating the threads that are          // sharing the same Object.          Train train1 = new Train(obj);          Train train2 = new Train(obj);            // threads start their execution.          train1.start();          train2.start();      }  } |

Run on IDE

Output

0

0

1

1

2

2

There can be two trains (more than two) which need to use same at same time so there is chance of collision. Therefore to avoid collision we need to synchronize the the line in which multiple want to run.

**Example: Synchronized access to getLine() method on the same Object**

|  |
| --- |
| // Example that shows multiple threads  // can execute the same method but in  // synchronized way.  class Line  {        // if multiple threads(trains) trying to access      // this synchronized method on the same Object      // but only one thread will be able      // to execute it at a time.      synchronized public void getLine()      {          for (int i = 0; i < 3; i++)          {              System.out.println(i);              try              {                  Thread.sleep(400);              }              catch (Exception e)              {                  System.out.println(e);              }          }      }  }    class Train extends Thread  {      // Reference variable of type Line.      Line line;        Train(Line line)      {          this.line = line;      }        @Override      public void run()      {          line.getLine();      }  }    class GFG  {      public static void main(String[] args)      {          Line obj = new Line();            // we are creating two threads which share          // same Object.          Train train1 = new Train(obj);          Train train2 = new Train(obj);            // both threads start executing .          train1.start();          train2.start();      }  } |

Run on IDE

Output:

0

1

2

0

1

2

**Block Synchronization**

If we only need to execute some subsequent lines of code not all lines (instructions) of code within a method, then we should synchronize only block of the code within which required instructions are exists.  
For example, lets suppose there is a method that contains 100 lines of code but there are only 10 lines (one after one) of code which contain critical section of code i.e. these lines can modify (change) the Object’s state. So we only need to synchronize these 10 lines of code method to avoid any modification in state of the Object and to ensure that other threads can execute rest of the lines within the same method without any interruption.

|  |
| --- |
| import java.io.\*;  import java.util.\*;    public class Geek  {      String name = "";      public int count = 0;        public void geekName(String geek, List<String> list)      {          // Only one thread is permitted          // to change geek's name at a time.          synchronized(this)          {              name = geek;              count++;  // how many threads change geek's name.          }            // All other threads are permitted          // to add geek name into list.          list.add(geek);      }  }    class GFG  {      public static void main (String[] args)      {          Geek gk = new Geek();          List<String> list = new ArrayList<String>();          gk.geekName("mohit", list);          System.out.println(gk.name);        }  } |

Run on IDE

Output :

1

**Important points:**

* When a thread enters into synchronized method or block, it acquires lock and once it completes its task and exits from the synchronized method, it releases the lock.
* When thread enters into synchronized instance method or block, it acquires Object level lock and when it enters into synchronized static method or block it acquires class level lock.
* Java synchronization will throw null pointer exception if Object used in synchronized block is null. For example, If in **synchronized(instance)**, **instance** is null then it will throw null pointer exception.
* In Java, **wait(), notify() and notifyAll()** are the important methods that are used in synchronization.
* You can not apply java **synchronized** keyword with the variables.
* Don’t synchronize on the non-final field on synchronized block because the reference to the non-final field may change anytime and then different threads might synchronize on different objects i.e. no synchronization at all.

**Advantages**

* **Multithreading:** Since java is multithreaded language, synchronization is a good way to achieve mutual exclusion on shared resource(s).
* **Instance and Static Methods:** Both synchronized instance methods and synchronized static methods can be executed concurrently because they are used to lock different Objects.

**Limitations**

* **Concurrency Limitations:**Java synchronization does not allow concurrent reads.
* **Decreases Efficiency:**Java synchronized method run very slowly and can degrade the performance, so you should synchronize the method when it is absolutely necessary otherwise not and to synchronize block only for critical section of the code.

# Thread Pools in Java

[**4.3**](https://www.geeksforgeeks.org/hard/)

**Background**

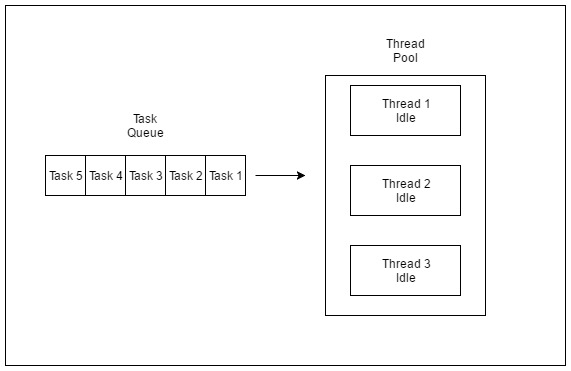
Server Programs such as database and web servers repeatedly execute requests from multiple clients and these are oriented around processing a large number of short tasks. An approach for building a server application would be to create a new thread each time a request arrives and service this new request 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 [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)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.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/tpinit.jpg)

*Thread Pool Initialization with size = 3 threads. Task Queue = 5 Runnable Objects*

**Executor Thread Pool Methods**

**Method** **Description**

newFixedThreadPool(int) Creates a fixed size thread pool.

newCachedThreadPool() Creates a thread pool that creates new

threads as needed, but will reuse previously

constructed threads when they are available

newSingleThreadExecutor() Creates a single thread.

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.

**Thread Pool Example**

In the following tutorial, we will look at a basic example of thread pool executor- FixedThreadPool.

**Steps to be followed**

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

|  |
| --- |
| // Java program to illustrate  // ThreadPool  import java.text.SimpleDateFormat;  import java.util.Date;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;    // Task class to be executed (Step 1)  class Task implements Runnable  {      private String name;        public Task(String s)      {          name = s;      }        // Prints task name and sleeps for 1s      // This Whole process is repeated 5 times      public void run()      {          try          {              for (int i = 0; i<=5; i++)              {                  if (i==0)                  {                      Date d = new Date();                      SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");                      System.out.println("Initialization Time for"                              + " task name - "+ name +" = " +ft.format(d));                      //prints the initialization time for every task                  }                  else                  {                      Date d = new Date();                      SimpleDateFormat ft = new SimpleDateFormat("hh:mm:ss");                      System.out.println("Executing Time for task name - "+                              name +" = " +ft.format(d));                      // prints the execution time for every task                  }                  Thread.sleep(1000);              }              System.out.println(name+" complete");          }            catch(InterruptedException e)          {              e.printStackTrace();          }      }  }  public class Test  {       // Maximum number of threads in thread pool      static final int MAX\_T = 3;        public static void main(String[] args)      {          // creates five tasks          Runnable r1 = new Task("task 1");          Runnable r2 = new Task("task 2");          Runnable r3 = new Task("task 3");          Runnable r4 = new Task("task 4");          Runnable r5 = new Task("task 5");            // creates a thread pool with MAX\_T no. of          // threads as the fixed pool size(Step 2)          ExecutorService pool = Executors.newFixedThreadPool(MAX\_T);            // passes the Task objects to the pool to execute (Step 3)          pool.execute(r1);          pool.execute(r2);          pool.execute(r3);          pool.execute(r4);          pool.execute(r5);            // pool shutdown ( Step 4)          pool.shutdown();      }  } |

Run on IDE

**Sample Execution**

Output:

Initialization Time for task name - task 2 = 02:32:56

Initialization Time for task name - task 1 = 02:32:56

Initialization Time for task name - task 3 = 02:32:56

Executing Time for task name - task 1 = 02:32:57

Executing Time for task name - task 2 = 02:32:57

Executing Time for task name - task 3 = 02:32:57

Executing Time for task name - task 1 = 02:32:58

Executing Time for task name - task 2 = 02:32:58

Executing Time for task name - task 3 = 02:32:58

Executing Time for task name - task 1 = 02:32:59

Executing Time for task name - task 2 = 02:32:59

Executing Time for task name - task 3 = 02:32:59

Executing Time for task name - task 1 = 02:33:00

Executing Time for task name - task 3 = 02:33:00

Executing Time for task name - task 2 = 02:33:00

Executing Time for task name - task 2 = 02:33:01

Executing Time for task name - task 1 = 02:33:01

Executing Time for task name - task 3 = 02:33:01

task 2 complete

task 1 complete

task 3 complete

Initialization Time for task name - task 5 = 02:33:02

Initialization Time for task name - task 4 = 02:33:02

Executing Time for task name - task 4 = 02:33:03

Executing Time for task name - task 5 = 02:33:03

Executing Time for task name - task 5 = 02:33:04

Executing Time for task name - task 4 = 02:33:04

Executing Time for task name - task 4 = 02:33:05

Executing Time for task name - task 5 = 02:33:05

Executing Time for task name - task 5 = 02:33:06

Executing Time for task name - task 4 = 02:33:06

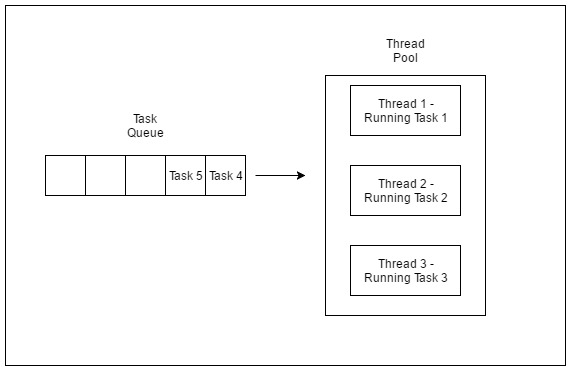
Executing Time for task name - task 5 = 02:33:07

Executing Time for task name - task 4 = 02:33:07

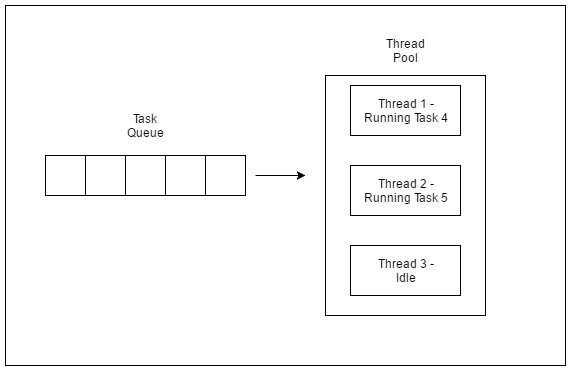
task 5 complete

task 4 complete

As seen in the execution of the program, the task 4 or task 5 are executed only when a thread in the pool becomes idle. Until then, the extra tasks are placed in a queue.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/tprun1.jpg)

*Thread Pool executing first three tasks*

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/tprun2.jpg)

*Thread Pool executing task 4 and 5*

**Risks in using Thread Pools**

* + 1. [**Deadlock**](https://www.geeksforgeeks.org/deadlock-in-java-multithreading/)**:**While deadlock can occur in any multi-threaded program, thread pools introduce another case of deadlock, one in which all the executing threads are waiting for the results from the blocked threads waiting in the queue due to the unavailability of threads for execution.
    2. **Thread Leakage :**Thread Leakage occurs if a thread is removed from the pool to execute a task but not returned to it when the task completed. As an example, if the thread throws an exception and pool class does not catch this exception, then the thread will simply exit, reducing the size of the thread pool by one. If this repeats many times, then the pool would eventually become empty and no threads would be available to execute other requests.
    3. **Resource Thrashing :**If the thread pool size is very large then time is wasted in context switching between threads. Having more threads than the optimal number may cause starvation problem leading to resource thrashing as explained.

**Important Points**

* + 1. Don’t queue tasks that concurrently wait for results from other tasks. This can lead to a situation of deadlock as described above.
    2. Be careful while using threads for a long lived operation. It might result in the thread waiting forever and would eventually lead to resource leakage.
    3. The Thread Pool has to be ended explicitly at the end. If this is not done, then the program goes on executing and never ends. Call shutdown() on the pool to end the executor. If you try to send another task to the executor after shutdown, it will throw a RejectedExecutionException.
    4. One needs to understand the tasks to effectively tune the thread pool. If the tasks are very contrasting then it makes sense to use different thread pools for different types of tasks so as to tune them properly.

**Tuning Thread Pool**

* + The optimum size of the thread pool depends on the number of processors available and the nature of the tasks. On a N processor system for a queue of only computation type processes, a maximum thread pool size of N or N+1 will achieve the maximum efficiency.But tasks may wait for I/O and in such a case we take into account the ratio of waiting time(W) and service time(S) for a request; resulting in a maximum pool size of N\*(1+ W/S) for maximum efficiency.

The thread pool is a useful tool for organizing server applications. It is quite straightforward in concept, but there are several issues to watch for when implementing and using one, such as deadlock, resource thrashing. Use of executor service makes it easier to implement.

# Semaphore in Java

[**3.8**](https://www.geeksforgeeks.org/medium/)

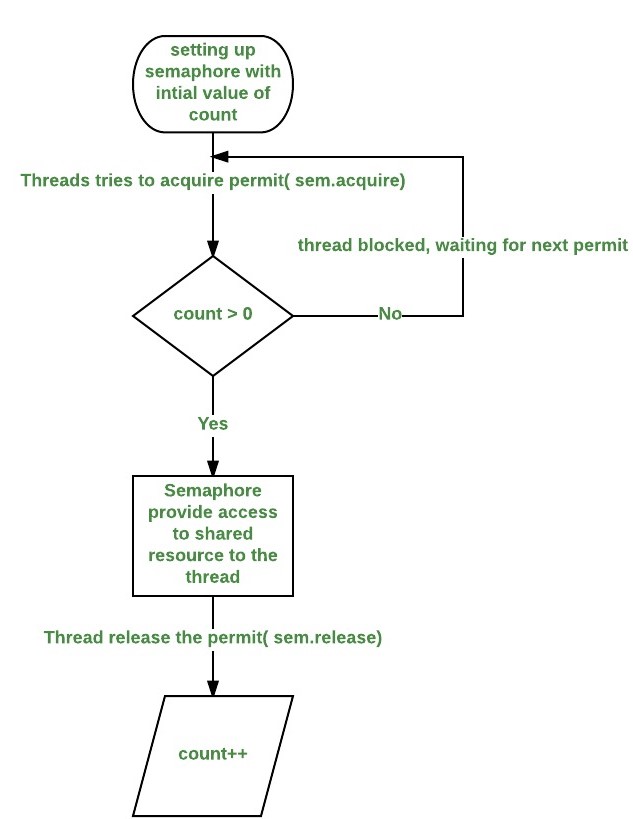
A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

**Working of semaphore**

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit.

* If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented.
* Otherwise, the thread will be blocked until a permit can be acquired.
* When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.
* If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

Java provide **Semaphore** class in java.util.concurrent package that implements this mechanism, so you don’t have to implement your own semaphores.

**Flow Diagram :**  
[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/d2.jpeg)

**Constructors in Semaphore class :** There are two constructors in Semaphore class.

Semaphore(int num)

Semaphore(int num, boolean how)

Here, num specifies the initial permit count. Thus, it specifies the number of threads that can access a shared resource at any one time. If it is one, then only one thread can access the resource at any one time. By default, all waiting threads are granted a permit in an undefined order. By setting how to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

**Using Semaphores as Locks([preventing race condition](https://practice.geeksforgeeks.org/problems/what-is-race-condition))**

We can use a semaphore to lock access to a resource, each thread that wants to use that resource must first call acquire( ) before accessing the resource to acquire the lock. When the thread is done with the resource, it must call release( ) to release lock. Here is an example that demonstrate this:

|  |
| --- |
| // java program to demonstrate  // use of semaphores Locks  import java.util.concurrent.\*;    //A shared resource/class.  class Shared  {      static int count = 0;  }    class MyThread extends Thread  {      Semaphore sem;      String threadName;      public MyThread(Semaphore sem, String threadName)      {          super(threadName);          this.sem = sem;          this.threadName = threadName;      }        @Override      public void run() {            // run by thread A          if(this.getName().equals("A"))          {              System.out.println("Starting " + threadName);              try              {                  // First, get a permit.                  System.out.println(threadName + " is waiting for a permit.");                    // acquiring the lock                  sem.acquire();                    System.out.println(threadName + " gets a permit.");                    // Now, accessing the shared resource.                  // other waiting threads will wait, until this                  // thread release the lock                  for(int i=0; i < 5; i++)                  {                      Shared.count++;                      System.out.println(threadName + ": " + Shared.count);                        // Now, allowing a context switch -- if possible.                      // for thread B to execute                      Thread.sleep(10);                  }              } catch (InterruptedException exc) {                      System.out.println(exc);                  }                    // Release the permit.                  System.out.println(threadName + " releases the permit.");                  sem.release();          }            // run by thread B          else          {              System.out.println("Starting " + threadName);              try              {                  // First, get a permit.                  System.out.println(threadName + " is waiting for a permit.");                    // acquiring the lock                  sem.acquire();                    System.out.println(threadName + " gets a permit.");                    // Now, accessing the shared resource.                  // other waiting threads will wait, until this                  // thread release the lock                  for(int i=0; i < 5; i++)                  {                      Shared.count--;                      System.out.println(threadName + ": " + Shared.count);                        // Now, allowing a context switch -- if possible.                      // for thread A to execute                      Thread.sleep(10);                  }              } catch (InterruptedException exc) {                      System.out.println(exc);                  }                  // Release the permit.                  System.out.println(threadName + " releases the permit.");                  sem.release();          }      }  }    // Driver class  public class SemaphoreDemo  {      public static void main(String args[]) throws InterruptedException      {          // creating a Semaphore object          // with number of permits 1          Semaphore sem = new Semaphore(1);            // creating two threads with name A and B          // Note that thread A will increment the count          // and thread B will decrement the count          MyThread mt1 = new MyThread(sem, "A");          MyThread mt2 = new MyThread(sem, "B");            // stating threads A and B          mt1.start();          mt2.start();            // waiting for threads A and B          mt1.join();          mt2.join();            // count will always remain 0 after          // both threads will complete their execution          System.out.println("count: " + Shared.count);      }  } |

Run on IDE

Output:

Starting A

Starting B

B is waiting for a permit.

B gets a permit.

A is waiting for a permit.

B: -1

B: -2

B: -3

B: -4

B: -5

B releases the permit.

A gets a permit.

A: -4

A: -3

A: -2

A: -1

A: 0

A releases the permit.

count: 0

**Note :**The output can be different in different executions of above program, but final value of count variable will always remain 0.

# Java.util.concurrent.Semaphore class in Java

[**4.7**](https://www.geeksforgeeks.org/hard/)

**Prerequisite :**[Semaphores in Java](https://www.geeksforgeeks.org/semaphore-in-java/)

public class Semaphore

extends Object

implements Serializable

Conceptually, a semaphore maintains a set of permits. Each acquire() blocks if necessary until a permit is available, and then takes it. Each release() adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly.  
**Methods:**

* 1. **void acquire()** : This method acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.If the current thread is interrupted while waiting for a permit then InterruptedException is thrown.
  2. **Syntax :**
  3. public void acquire() throws InterruptedException
  4. **Parameters :**
  5. NA
  6. **Returns :**
  7. NA
  8. **Throws:**
  9. InterruptedException - if the current thread is interrupted
  10. **void acquire(int permits)** : This method acquires the given number of permits, if they are available, and returns immediately, reducing the number of available permits by the given amount.If the current thread is interrupted while waiting for a permit then InterruptedException is thrown.
  11. **Syntax :**
  12. public void acquire(int permits) throws InterruptedException
  13. **Parameters :**
  14. permits - the number of permits to acquire
  15. **Returns :**
  16. NA
  17. **Throws:**
  18. InterruptedException - if the current thread is interrupted
  19. IllegalArgumentException - if permits is negative
  20. **void acquireUninterruptibly()** : This method acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.If the current thread is interrupted while waiting for a permit then it will continue to wait,
  21. **Syntax :**
  22. public void acquireUninterruptibly()
  23. **Parameters :**
  24. NA
  25. **Returns :**
  26. NA
  27. **void acquireUninterruptibly(int permits)** : This method the given number of permits, if they are available, and returns immediately, reducing the number of available permits by the given amount.If the current thread is interrupted while waiting for a permit then it will continue to wait,
  28. **Syntax :**
  29. public void acquireUninterruptibly(int permits)
  30. **Parameters :**
  31. permits - the number of permits to acquire
  32. **Returns :**
  33. NA
  34. **Throws:**
  35. IllegalArgumentException - if permits is negative
  36. **boolean tryAcquire()** : This method acquires a permit, if one is available and returns immediately, with the value true, reducing the number of available permits by one. If no permit is available then this method will return immediately with the value false.
  37. **Syntax :**
  38. public boolean tryAcquire()
  39. **Parameters :**
  40. NA
  41. **Returns :**
  42. true if a permit was acquired and false otherwise
  43. **boolean tryAcquire(int permits)** : This method acquires the given number of permits, if they are available, and returns immediately, with the value true, reducing the number of available permits by the given amount.If insufficient permits are available then this method will return immediately with the value false.
  44. **Syntax :**
  45. public boolean tryAcquire(int permits)
  46. **Parameters :**
  47. permits - the number of permits to acquire
  48. **Returns :**
  49. true if the permits were acquired and false otherwise
  50. **Throws:**
  51. IllegalArgumentException - if permits is negative
  52. **boolean tryAcquire(long timeout, TimeUnit unit)** : This method acquires a permit, if one is available and returns immediately, with the value true, reducing the number of available permits by one. If the specified waiting time elapses then the value false is returned. If the time is less than or equal to zero, the method will not wait at all.
  53. **Syntax :**
  54. public boolean tryAcquire(long timeout, TimeUnit unit)
  55. throws InterruptedException
  56. **Parameters :**
  57. timeout - the maximum time to wait for a permit
  58. unit - the time unit of the timeout argument
  59. **Returns :**
  60. true if a permit was acquired and
  61. false if the waiting time elapsed before a permit was acquired
  62. **Throws:**
  63. InterruptedException - if the current thread is interrupted
  64. **boolean tryAcquire(int permits, long timeout, TimeUnit unit)** : This method acquires the given number of permits, if they are available and returns immediately, with the value true, reducing the number of available permits by the given amount. If the specified waiting time elapses then the value false is returned. If the time is less than or equal to zero, the method will not wait at all.Any permits that were to be assigned to this thread, are instead assigned to other threads trying to acquire permits.
  65. **Syntax :**
  66. public boolean tryAcquire(int permits, long timeout, TimeUnit unit)
  67. throws InterruptedException
  68. **Parameters :**
  69. permits - the number of permits to acquire
  70. timeout - the maximum time to wait for a permit
  71. unit - the time unit of the timeout argument
  72. **Returns :**
  73. true if all permits were acquired and
  74. false if the waiting time elapsed before all
  75. permits were acquired
  76. **Throws:**
  77. InterruptedException - if the current thread is interrupted
  78. IllegalArgumentException - if permits is negative
  79. **void release()**: This method releases a permit, increasing the number of available permits by one. If any threads are trying to acquire a permit, then one is selected and given the permit that was just released.
  80. **Syntax :**
  81. public void release()
  82. **Parameters :**
  83. NA
  84. **Returns :**
  85. NA
  86. **void release(int permits)**: This method releases the given number of permits, increasing the number of available permits by that amount.If any threads are trying to acquire permits, then one is selected and given the permits that were just released. If the number of available permits satisfies that thread’s request then that thread is (re)enabled for thread scheduling purposes; otherwise the thread will wait until sufficient permits are available.
  87. **Syntax :**
  88. public void release(int permits)
  89. **Parameters :**
  90. permits - the number of permits to release
  91. **Returns :**
  92. NA
  93. **Throws :**
  94. IllegalArgumentException - if permits is negative
  95. **int availablePermits()**: This method returns the current number of permits available in this semaphore.This method is typically used for debugging and testing purposes.
  96. **Syntax :**
  97. public int availablePermits()
  98. **Parameters :**
  99. NA
  100. **Returns :**
  101. the number of permits available in this semaphore
  102. **int drainPermits()** : This method acquires and returns all permits that are immediately available.
  103. **Syntax :**
  104. public int drainPermits()
  105. **Parameters :**
  106. NA
  107. **Returns :**
  108. the number of permits acquired
  109. **void reducePermits(int reduction)** : This method shrinks the number of available permits by the indicated reduction. This method can be useful in subclasses that use semaphores to track resources that become unavailable. This method differs from acquire in that it does not block waiting for permits to become available.
  110. **Syntax :**
  111. protected void reducePermits(int reduction)
  112. **Parameters :**
  113. reduction - the number of permits to remove
  114. **Returns :**
  115. NA
  116. **Throws :**
  117. IllegalArgumentException - if reduction is negative
  118. **boolean isFair()** : This method returns true if this semaphore has fairness set true.
  119. **Syntax :**
  120. public boolean isFair()
  121. **Parameters :**
  122. NA
  123. **Returns :**
  124. true if this semaphore has fairness set true
  125. **final boolean hasQueuedThreads()** : This method queries whether any threads are waiting to acquire. Note that because cancellations may occur at any time, a true return does not guarantee that any other thread will ever acquire. This method is designed primarily for use in monitoring of the system state.
  126. **Syntax :**
  127. public final boolean hasQueuedThreads()
  128. **Parameters :**
  129. NA
  130. **Returns :**
  131. true if there may be other threads waiting to acquire the lock
  132. **final int getQueueLength()** : This method returns an estimate of the number of threads waiting to acquire. The value is only an estimate because the number of threads may change dynamically while this method traverses internal data structures. This method is designed for use in monitoring of the system state, not for synchronization control.
  133. **Syntax :**
  134. public final int getQueueLength()
  135. **Parameters :**
  136. NA
  137. **Returns :**
  138. the estimated number of threads waiting for this lock
  139. **Collection getQueuedThreads()** : This method returns a collection containing threads that may be waiting to acquire. Because the actual set of threads may change dynamically while constructing this result, the returned collection is only a best-effort estimate. The elements of the returned collection are in no particular order.
  140. **Syntax :**
  141. protected Collection getQueuedThreads()
  142. **Parameters :**
  143. NA
  144. **Returns :**
  145. the collection of threads
  146. **String toString()** : This method Returns a string identifying this semaphore, as well as its state. The state, in brackets, includes the String “Permits =” followed by the number of permits.
  147. **Syntax :**
  148. public String toString()
  149. **Parameters :**
  150. NA
  151. **Returns :**
  152. a string identifying this semaphore, as well as its state
  153. **Overrides:**

toString in class [Object](http://contribute.geeksforgeeks.org/object-class-in-java/)

# CountDownLatch in Java

[**2**](https://www.geeksforgeeks.org/easy/)

CountDownLatch is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all the required services have started.

**Working of CountDownLatch:**  
When we create an object of CountDownLatch, we specify the number if threads it should wait for, all such thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.

**Example of CountDownLatch in JAVA:**

|  |
| --- |
| /\* Java Program to demonstrate how to use CountDownLatch,     Its used when a thread needs to wait for other threads     before starting its work. \*/  import java.util.concurrent.CountDownLatch;    public class CountDownLatchDemo  {      public static void main(String args[]) throws InterruptedException      {          // Let us create task that is going to wait for four          // threads before it starts          CountDownLatch latch = new CountDownLatch(4);            // Let us create four worker threads and start them.          Worker first = new Worker(1000, latch, "WORKER-1");          Worker second = new Worker(2000, latch, "WORKER-2");          Worker third = new Worker(3000, latch, "WORKER-3");          Worker fourth = new Worker(4000, latch, "WORKER-4");          first.start();          second.start();          third.start();          fourth.start();            // The main task waits for four threads          latch.await();            // Main thread has started          System.out.println(Thread.currentThread().getName() +                             " has finished");      }  }    // A class to represent threads for which the main thread  // waits.  class Worker extends Thread  {      private int delay;      private CountDownLatch latch;        public Worker(int delay, CountDownLatch latch,                                       String name)      {          super(name);          this.delay = delay;          this.latch = latch;      }        @Override      public void run()      {          try          {              Thread.sleep(delay);              latch.countDown();              System.out.println(Thread.currentThread().getName()                                 + " finished");          }          catch (InterruptedException e)          {              e.printStackTrace();          }      }  } |

Run on IDE

**Output:**

WORKER-1 finished

WORKER-2 finished

WORKER-3 finished

WORKER-4 finished

main has finished

**Facts about CountDownLatch:**

1. Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.
2. The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await() proceed together once count down reaches to zero.
3. countDown() method decrements the count and await() method blocks until count == 0