MULTITHREADING IN JAVA

MIRACLE TECHIES:

Naznin Shafeek Sona Sudhakaran

Samyuktha Sudhakaran Athulya Vijayan

# Multithreading in Java- An Introduction

In [Java](https://www.mygreatlearning.com/academy/learn-for-free/courses/java-programming/?gl_blog_id=35562), Multithreading refers to a process of executing two or more threads simultaneously for maximum utilisation of the CPU. A thread in Java is a ***lightweight process*** requiring fewer resources to create and share the process resources.

Multithreading and Multiprocessing are used for multitasking in Java, but we prefer multithreading over multiprocessing. This is because the threads use a shared memory area which helps to save memory, and also, the content-switching between the threads is a bit faster than the process.

**Few more advantages of Multithreading are:**

* Multithreading saves time as you can perform multiple operations together.
* The threads are independent, so it does not block the user to perform multiple operations at the same time and also, if an exception occurs in a single thread, it does not affect other threads.

# Life Cycle of a Thread

There are five states a thread has to go through in its life cycle. This life cycle is controlled by JVM (Java Virtual Machine). These states are:

**1.New**

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

In this state, a new thread begins its life cycle. This is also called a ***born thread***. The thread is in the new state if you create an instance of Thread class but before the invocation of the ***start()*** method.

1. **Runnable**

A thread becomes runnable after a newly born thread is started. In this state, a thread would be executing its task.

1. **Running**

When the thread scheduler selects the thread then, that thread would be in a running state.

1. **Non-Runnable (Blocked)**

The thread is still alive in this state, but currently, it is not eligible to run

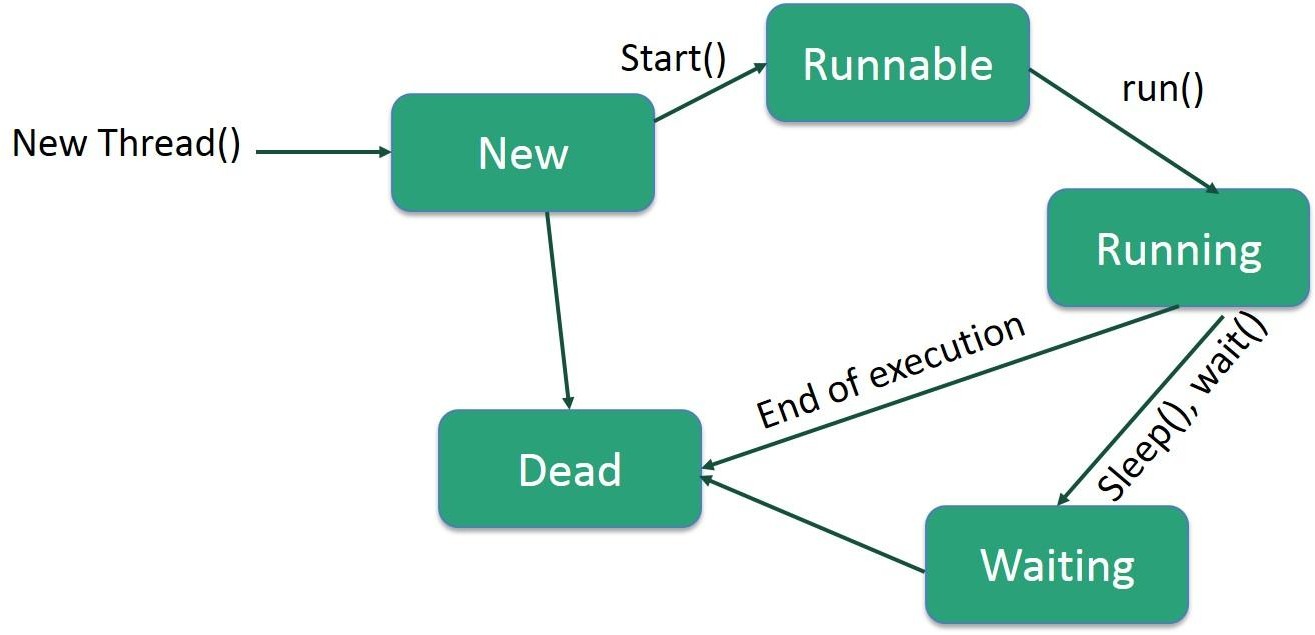
**5.Terminated**

A thread is terminated due to the following reasons:

* + Either its ***run()*** method exists normally, i.e., the thread’s code has executed the program.
  + Or due to some unusual errors like segmentation fault or an unhandled exception.

A thread that is in a terminated state does not consume ant cycle of the CPU.

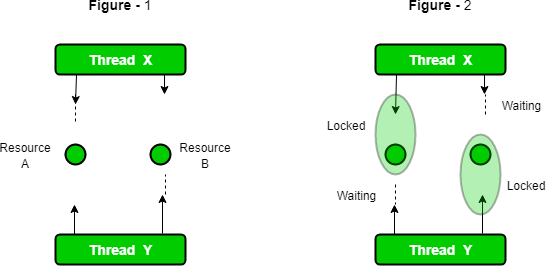
A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread



**Deadlock in Java Multithreading**

[**synchronized**](https://www.geeksforgeeks.org/synchronized-in-java/) keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronised method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.

It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called [**Deadlock**](https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/amp/). Below is a simple example of Deadlock condition.



**Example for Deadlock**

public class TestDeadlockExample1 {

public static void main(String[] args) {

final String resource1 = "java";

final String resource2 = "full stack";

// t1 tries to lock resource1 then resource2

Thread t1 = new Thread() {

public void run() {

synchronized (resource1) {

System.out.println("Thread 1: locked resource 1");

try { Thread.sleep(100);} catch (Exception e) {}

synchronized (resource2) {

System.out.println("Thread 1: locked resource 2");

}

}

}

};

// t2 tries to lock resource2 then resource1

Thread t2 = new Thread() {

public void run() {

synchronized (resource2) {

System.out.println("Thread 2: locked resource 2");

try { Thread.sleep(100);} catch (Exception e) {}

synchronized (resource1) {

System.out.println("Thread 2: locked resource 1");

}

}

}

};

t1.start();

t2.start();

}

}

## Output

Thread 1: locked resource 1

## Thread 2: locked resource 2

**How to Avoid Deadlock**

Although it is not possible to avoid deadlock condition but we can avoid it by using the following ways:

* **Avoid Unnecessary Locks:** We should use locks only for those members on which it is required. Unnecessary use of locks leads to a deadlock situation. We recommend you to use a **lock-free** da structure. If possible, keep your code free form locks. For example, instead of using synchronized **ArrayList**, use the **ConcurrentLinkedQueue**.
* **Avoid Nested Locks:** Another way to avoid deadlock is to avoid giving a lock to multiple threads if we have already provided a lock to one thread. Since we must avoid allocating a lock to multiple threads.
* **Using Thread.join() Method:** You can get a deadlock if two threads are waiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it's always best to use join with the maximum time you want to wait for the thread to finish.
* **Use Lock Ordering:** Always assign a numeric value to each lock. Before acquiring the lock with a higher numeric value, acquire the locks with a lower numeric value.
* **Lock Time-out:** We can also specify the time for a thread to acquire a lock. If a thread does not acquire a lock, the thread must wait for a specific time before retrying to acquire a lock.

**How to create a thread in Java**

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface

### **Thread class:**

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

### Commonly used Constructors of Thread class:

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

### Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long milliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long milliseconds):** waits for a thread to die for the specified milliseconds
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):**changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):**changes the name of the thread.

10.**public Thread currentThread():** returns the reference of currently executing thread.

11**. public int getId():** returns the id of the thread.

12.**public Thread.State getState():** returns the state of the thread.

13.**public boolean isAlive():** tests if the thread is alive.

14.**public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.

15.**public void suspend():** is used to suspend the thread(depricated).

16.**public void resume():** is used to resume the suspended thread(depricated).

17**.public void stop():** is used to stop the thread(depricated).

18.**public boolean isDaemon():** tests if the thread is a daemon thread.

19.**public void setDaemon(boolean b):**marks the thread as daemon or user thread.

20.**public void interrupt():** interrupts the thread.

21.**public boolean isInterrupted():** tests if the thread has been interrupted.

22.**public static boolean interrupted():** tests if the current thread has been interrupted.

### **Runnable Interface**:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.

### **Starting a thread:**

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

* A new thread starts(with new callstack).
* The thread moves from New state to the Runnable state.
* When the thread gets a chance to execute, its target run() method will run.

### 1)Java Thread Example by extending Thread class

1. **class** Multi **extends** Thread{
2. **public void** run(){
3. System.out.println("thread is running...");

4. }

1. **public static void** main(String args[]){
2. Multi t1=**new** Multi();
3. t1.start();

8. }

9. }

### Output:

thread is running…

### 2)Java Thread Example by implementing Runnable interface

* + 1. **class** Multi3 **implements** Runnable{
    2. **public void** run(){
    3. System.out.println("thread is running...");

4. }

5.

1. **public static void** main(String args[]){
2. Multi3 m1=**new** Multi3();
3. Thread t1 =**new** Thread(m1); // Using the constructor Thread(Runnable r)
4. t1.start();

10. }

11.}

### Output:

thread is running…

### 3)Using the Thread Class: Thread(String Name)

1**.public class** MyThread1

2. {

1. // Main method
2. **public static void** main(String argvs[])

5. {

1. // creating an object of the Thread class using the constructor Thread(String name)
2. Thread t= **new** Thread("My first thread");

8.

9. // the start() method moves the thread to the active state

10. t.start();

1. // getting the thread name by invoking the getName() method
2. String str = t.getName();
3. System.out.println(str);

14.}

15.}

### Output:

My ﬁrst thread

### 4)Using the Thread Class: Thread(Runnable r, String name)

1.**public class** MyThread2 **implements** Runnable

2. {

3. **public void** run()

4. {

5. System.out.println("Now the thread is running ...");

6. }

7.

1. // main method
2. **public static void** main(String argvs[])

10.{

1. // creating an object of the class MyThread2
2. Runnable r1 = **new** MyThread2();

13.

1. // creating an object of the class Thread using Thread(Runnable r, String nam e)
2. Thread th1 = **new** Thread(r1, "My new thread");

16.

1. // the start() method moves the thread to the active state
2. th1.start();
3. // getting the thread name by invoking the getName() method
4. String str = th1.getName();
5. System.out.println(str);

22.}

23.}

### Output:

My new thread

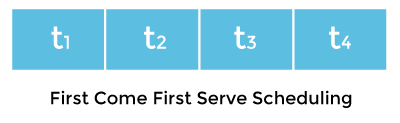
### Now the thread is running…

## **Thread Scheduler Algorithms**

First Come First Serve Scheduling:

In this scheduling algorithm, the scheduler picks the threads thar arrive first in the runnable queue. Observe the following table:

|  |  |
| --- | --- |
| **Threads** | **Time of Arrival** |
| t1 | 0 |
| t2 | 1 |
| t3 | 2 |
| t4 | 3 |

In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time.  


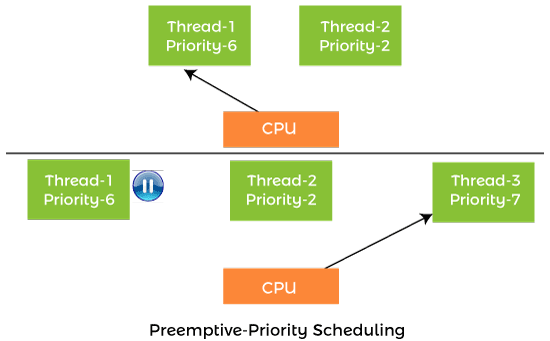
Hence, Thread t1 will be processed first, and Thread t4 will be processed last.

Time-slicing scheduling:

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

### **Preemptive-Priority Scheduling:**

The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.



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

**Thread Scheduler**

A component of Java that decides which thread to run or execute and which thread to wait is called a thread scheduler in Java.

In Java, a thread is only chosen by a thread scheduler if it is in the runnable state.

causes a performance drop if the time taken for one thread is too long. There are two factors for scheduling a thread i.e. Priority and Time of arrival.

#### Priority

Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

#### Time of Arrival

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

#### Working Of Thread Scheduler

●Suppose, there are five threads that have diﬀerent arrival times and diﬀerent priorities

●Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

●The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job.

●If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

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

**Synchronization**

●It is the capability to control the access of multiple threads to any shared Resource.

●Java Synchronization is better option where we want to allow only one thread to Access the shared resource.

●It is implemented in the program by using ‘synchronized‘ keyword.

Why use Synchronization?

1.To prevent thread interference.

2.To prevent consistency problem

3.Data Integrity

Disadvantage:

Performance issues will arise as during the execution of one thread all the other threads are put to a blocking state and do note they are not in waiting state. Thiscauses a performance drop if the time taken for one thread is too long.

**Types of Thread Synchronization**

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

1. **Mutual Exclusive-**helps keep threads from interfering with one another while sharing data.
   1. Synchronized method.
   2. Synchronized block.
   3. Static synchronization.
2. **Cooperation (Inter-thread communication in java)**

**Synchronized Method**

* If you declare any method as synchronized, it is known as synchronized method.
* Synchronized method is used to lock an object for any shared resource.
* When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Eg: public class SynchronizedCounter {

private int c = 0;

public synchronized void increment()

{ c++;

}

public synchronized void decrement() {

c--;

}

public synchronized int value() {

return c; } }

**Synchronized block**

Synchronized block can be used to perform synchronization on any specific resource of the method.If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

* Synchronized block is used to lock an object for any shared resource.
* Scope of synchronized block is smaller than the method.
* A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
* The system performance may degrade because of the slower working of synchronized keyword.

**Syntax:**

**synchronized** (object reference expression) {

//code block

 }

**Example of synchronized block**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    public class Geek  {      String name = "";      public int count = 0;        public void geekName(String geek, List<String> list)      {            synchronized(this)          {              name = geek;              count++;  }          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);        }  } |

Output :

Mohit

**Static Synchronization**

* If you make any static method as synchronized, the lock will be on the class not on object.
* To maintain the Synchronized behavior, we need a class-level lock rather than an instance-level lock such that only one thread can act on the static synchronized method.

**Syntax:**

synchronized static return type class name{}

Example:

**class** Display

{

**public** **static** **synchronized** **void** wish(String name)

    {

**for**(**int** i=0;i<3;i++)

        {

            System.out.print("Good Morning: ");

            System.out.println(name);

**try**{

                Thread.sleep(2000);

            }

**catch**(InterruptedException e)

            {

            }

        }

    }

}

**class** MyThread **extends** Thread{

    Display d;

    String name;

    MyThread(Display d,String name)

    {

**this**.d=d;

**this**.name=name;

    }

**public** **void** run()

    {

        d.wish(name);

    }

}

**class** Main{

**public** **static** **void** main(String arg[])

    {

        Display d1=**new** Display();

        Display d2=**new** Display();

        MyThread t1=**new** MyThread(d1,"Dhoni");

        MyThread t2=**new** MyThread(d2,"Yuvaraj");

        t1.start();

        t2.start();

    }

}

**Output:**

**First time of execution:**

Good Morning: Dhoni

Good Morning: Dhoni

Good Morning: Dhoni

Good Morning: Yuvaraj

Good Morning: Yuvaraj

Good Morning: Yuvaraj

**Second time of execution:**

Good Morning: Yuvaraj

Good Morning: Yuvaraj

Good Morning: Yuvaraj

Good Morning: Dhoni

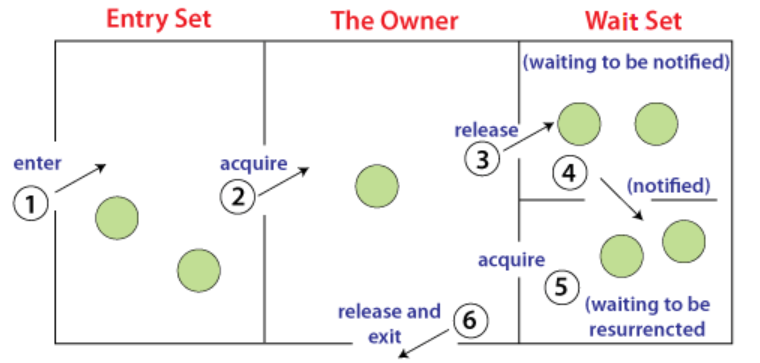
Good Morning: Dhoni

Good Morning: Dhoni

**Inter thread Communication**

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

* **wait()-**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 called wait() on the same object. It should be noted that calling notify() does not give up a lock on a resource.
* **notifyAll()**-It wakes up all the threads called wait() on the same object.



1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

**Example:**

class Customer{

int amount=10000;

synchronized void withdraw(int amount){

System.out.println("going to withdraw...");

if(this.amount<amount){

System.out.println("Less balance; waiting for deposit...");

try{wait();}

catch(Exception e){}

}

this.amount-=amount;

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

}

synchronized void deposit(int amount){

System.out.println("going to deposit...");

this.amount+=amount;

System.out.println("deposit completed... ");

notify();

}

}

class Test{

public static void main(String args[]){

final Customer c=new Customer();

new Thread(){

public void run(){c.withdraw(15000);}

}.start();

new Thread(){

public void run(){c.deposit(10000);}

}.start();

}}

**Output:**

going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed