Multithreading in Java

1. [Multithreading](https://www.javatpoint.com/multithreading-in-java)
2. [Multitasking](https://www.javatpoint.com/multithreading-in-java#multitasing)
3. [Process-based multitasking](https://www.javatpoint.com/multithreading-in-java#multiprocessing)
4. [Thread-based multitasking](https://www.javatpoint.com/multithreading-in-java#multithreading)
5. [What is Thread](https://www.javatpoint.com/multithreading-in-java#thread)

**Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

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

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

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

## **What is Thread in java**

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



As shown in the above figure, a thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the [OS](https://www.javatpoint.com/os-tutorial), and one process can have multiple threads.

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

## **Java Thread class**

Java provides **Thread class** to achieve thread programming. Thread class provides [constructors](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

# Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

New :

-> Whenever a new thread is created, it is always in the new state.

Active :

-> When a thread invokes the start() method, it moves from the new state to the active state

->The active state contains two states within it: one is runnable, and the other is running.

Runnable –

-> A thread, that is ready to run is then moved to the runnable state.

Running –

-> When the thread gets the CPU, it moves from the runnable to the running state.

When the run() method has been invoked and the thread starts its execution.

Blocked/Waiting:

-> Whenever a thread is inactive for a span of time

-> Thread A wants to enter synchronized code but it is unable to as Thread B thread is operating in that synchronized block on the same object.

The thread A has to wait until the thread B exits the synchronized block.

A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU).

Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

-> When the main thread invokes the join() method then, it is said that the main thread is in the waiting state.

The main thread then waits for the child threads to complete their tasks.

When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

-> when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state.

After the time runs out, the thread wakes up and start its execution from when it has left earlier.

Terminated :

-> Once the run() method execution is completed, the thread is said to enter the TERMINATED step and is considered to not be alive.

-> It occurs when some unusual events such as an unhandled exception or segmentation fault.

or

## **Explanation of Different Thread States**

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

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

* **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.  
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
* **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

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

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

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

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

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

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

Java Threads | 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.

Runnable interface:

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

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

Starting a thread:

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

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

# Thread Scheduler in Java

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. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. 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.

**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. A thread that arrived first gets the preference over the other threads.

## **Thread Scheduler Algorithms**

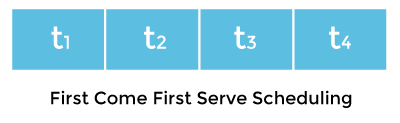
On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

### 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 |

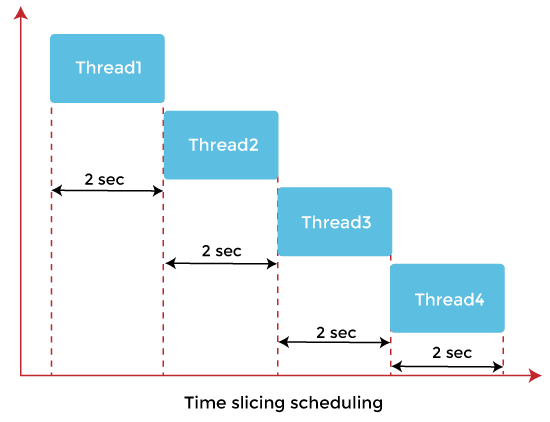
In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.



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

### Time-slicing scheduling:

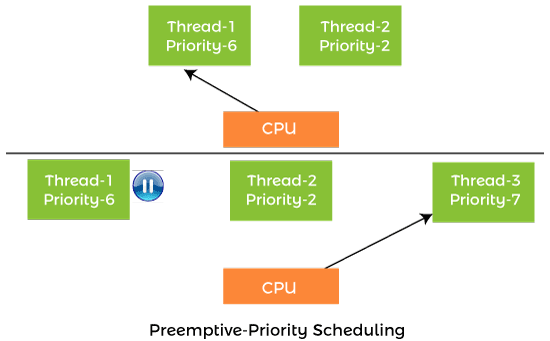
Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads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



In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

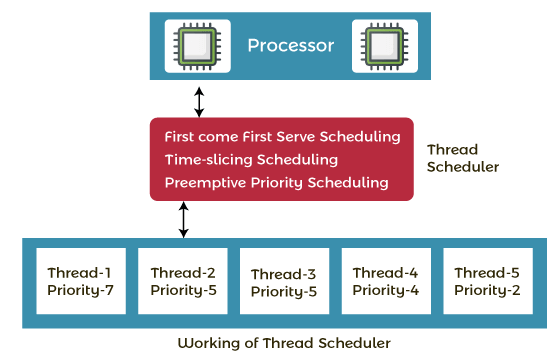
### Preemptive-Priority Scheduling:

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.

## **Working of the Java Thread Scheduler**



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

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

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

Thread.sleep() in Java with Examples

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

The sleep() Method Syntax:

Following are the syntax of the sleep() method.

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

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

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

Parameters:

The following are the parameters used in the sleep() method.

**mls:** The time in milliseconds is represented by the parameter mls. The duration for which the thread will sleep is given by the method sleep().

**n:** It shows the additional time up to which the programmer or developer wants the thread to be in the sleeping state. The range of n is from 0 to 999999.

The method does not return anything.

Important Points to Remember About the Sleep() Method

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

Can we start a thread twice

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

Let's understand it by the example given below:

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

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

**Output:**

running

Exception in thread "main" java.lang.IllegalThreadStateException

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

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

**FileName:** TestCallRun1.java

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

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

**Output:**

running...

# Java join() method

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution. Suppose th be the object the class Thread whose thread is doing its execution currently, then the th.join(); statement ensures that th is finished before the program does the execution of the next statement. When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the developer or programmer to mention the waiting period. However, similar to the sleep() method in Java, the join() method is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters. The following are the three overloaded join() methods.

## **Description of The Overloaded join() Method**

**join():** When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

# Naming Thread and Current Thread

## **Naming Thread**

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name, i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using the setName() method. The syntax of setName() and getName() methods are given below:

1. **public** String getName(): is used to **return** the name of a thread.
2. **public** **void** setName(String name): is used to change the name of a thread.

We can also set the name of a thread directly when we create a new thread using the constructor of the class.

# Priority of a Thread (Thread Priority)

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

## **Setter & Getter Method of Thread Priority**

Let's discuss the setter and getter method of the thread priority.

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

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

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

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

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

# Daemon Thread in Java

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

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

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

## **Points to remember for Daemon Thread in Java**

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

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

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

### Methods for Java Daemon thread by Thread class

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

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

### Simple example of Daemon thread in java

*File: MyThread.java*

1. **public** **class** TestDaemonThread1 **extends** Thread{
2. **public** **void** run(){
3. **if**(Thread.currentThread().isDaemon()){//checking for daemon thread
4. System.out.println("daemon thread work");
5. }
6. **else**{
7. System.out.println("user thread work");
8. }
9. }
10. **public** **static** **void** main(String[] args){
11. TestDaemonThread1 t1=**new** TestDaemonThread1();//creating thread
12. TestDaemonThread1 t2=**new** TestDaemonThread1();
13. TestDaemonThread1 t3=**new** TestDaemonThread1();
15. t1.setDaemon(**true**);//now t1 is daemon thread
17. t1.start();//starting threads
18. t2.start();
19. t3.start();
20. }
21. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread1)

**Output:**

daemon thread work

user thread work

user thread work

#### **Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.**

*File: MyThread.java*

1. **class** TestDaemonThread2 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("Name: "+Thread.currentThread().getName());
4. System.out.println("Daemon: "+Thread.currentThread().isDaemon());
5. }
7. **public** **static** **void** main(String[] args){
8. TestDaemonThread2 t1=**new** TestDaemonThread2();
9. TestDaemonThread2 t2=**new** TestDaemonThread2();
10. t1.start();
11. t1.setDaemon(**true**);//will throw exception here
12. t2.start();
13. }
14. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread2)

**Output:**

exception in thread main: java.lang.IllegalThreadStateException

# Java Thread Pool

Java Thread pool represents a group of worker threads that are waiting for the job and reused many times.

In the case of a thread pool, a group of fixed-size threads is created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, the thread is contained in the thread pool again.

# Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

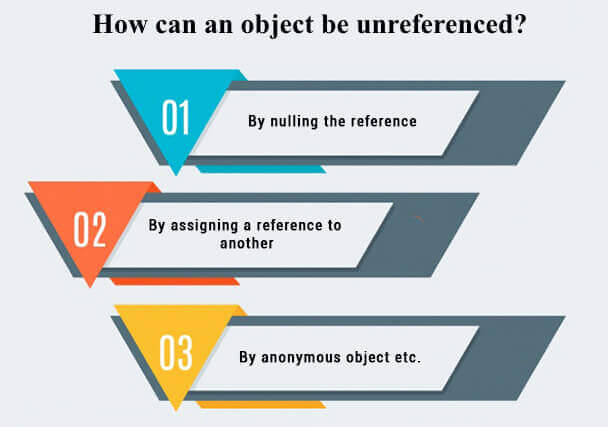
### Advantage of Garbage Collection

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## **How can an object be unreferenced?**

There are many ways:

* By nulling the reference
* By assigning a reference to another
* By anonymous object etc.



### 1) By nulling a reference:

1. Employee e=**new** Employee();
2. e=**null**;

### 2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

### 3) By anonymous object:

1. **new** Employee();

## **finalize() method**

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. **protected** **void** finalize(){}

#### **Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).**

## **gc() method**

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. **public** **static** **void** gc(){}

#### **Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.**

### Simple Example of garbage collection in java

1. **public** **class** TestGarbage1{
2. **public** **void** finalize(){System.out.println("object is garbage collected");}
3. **public** **static** **void** main(String args[]){
4. TestGarbage1 s1=**new** TestGarbage1();
5. TestGarbage1 s2=**new** TestGarbage1();
6. s1=**null**;
7. s2=**null**;
8. System.gc();
9. }
10. }

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestGarbage1)

object is garbage collected

object is garbage collected

#### **Note: Neither finalization nor garbage collection is guaranteed.**

Synchronization in Java

Synchronization in Java is the capability to control the access of multiple threads to any shared resource.

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

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

Thread Synchronization

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

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. Static synchronization.
2. Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Synchronized Block in Java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

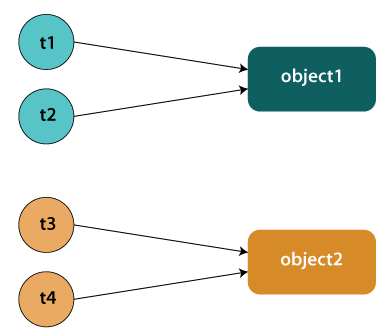
If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to Remember

* 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.
* Java synchronized block is more efficient than Java synchronized method.

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



Problem without static synchronization

Suppose there are two objects of a shared class (e.g. Table) named object1 and object2. In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock. But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Example of Static Synchronization

In this example we have used **synchronized** keyword on the static method to perform static synchronization.

**TestSynchronization4.java**

1. **class** Table
2. {
3. **synchronized** **static** **void** printTable(**int** n){
4. **for**(**int** i=1;i<=10;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){}
9. }
10. }
11. }
12. **class** MyThread1 **extends** Thread{
13. **public** **void** run(){
14. Table.printTable(1);
15. }
16. }
17. **class** MyThread2 **extends** Thread{
18. **public** **void** run(){
19. Table.printTable(10);
20. }
21. }
22. **class** MyThread3 **extends** Thread{
23. **public** **void** run(){
24. Table.printTable(100);
25. }
26. }
27. **class** MyThread4 **extends** Thread{
28. **public** **void** run(){
29. Table.printTable(1000);
30. }
31. }
32. **public** **class** TestSynchronization4{
33. **public** **static** **void** main(String t[]){
34. MyThread1 t1=**new** MyThread1();
35. MyThread2 t2=**new** MyThread2();
36. MyThread3 t3=**new** MyThread3();
37. MyThread4 t4=**new** MyThread4();
38. t1.start();
39. t2.start();
40. t3.start();
41. t4.start();
42. }
43. }

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

**Output:**

1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

Example of static synchronization by Using the anonymous class

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

**TestSynchronization5.java**Hello

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

[**Test it Now**](https://www.javatpoint.com/opr/test.jsp?filename=TestSynchronization5)

**Output:**

1

2

3

4

5

6

7

8

9

10

10

20

30

40

50

60

70

80

90

100

100

200

300

400

500

600

700

800

900

1000

1000

2000

3000

4000

5000

6000

7000

8000

9000

10000

Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference .class name .class. A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

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

Deadlock in Java

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Example of Deadlock in Java

**TestDeadlockExample1.java**

1. **public** **class** TestDeadlockExample1 {
2. **public** **static** **void** main(String[] args) {
3. **final** String resource1 = "ratan jaiswal";
4. **final** String resource2 = "vimal jaiswal";
5. // t1 tries to lock resource1 then resource2
6. Thread t1 = **new** Thread() {
7. **public** **void** run() {
8. **synchronized** (resource1) {
9. System.out.println("Thread 1: locked resource 1");
11. **try** { Thread.sleep(100);} **catch** (Exception e) {}
13. **synchronized** (resource2) {
14. System.out.println("Thread 1: locked resource 2");
15. }
16. }
17. }
18. };
20. // t2 tries to lock resource2 then resource1
21. Thread t2 = **new** Thread() {
22. **public** **void** run() {
23. **synchronized** (resource2) {
24. System.out.println("Thread 2: locked resource 2");
26. **try** { Thread.sleep(100);} **catch** (Exception e) {}
28. **synchronized** (resource1) {
29. System.out.println("Thread 2: locked resource 1");
30. }
31. }
32. }
33. };

36. t1.start();
37. t2.start();
38. }
39. }

**Output:**

Thread 1: locked resource 1

Thread 2: locked resource 2

More Complicated Deadlocks

A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock. Here is an example in which four threads have deadlocked:

Thread 1 locks A, waits for B

Thread 2 locks B, waits for C

Thread 3 locks C, waits for D

Thread 4 locks D, waits for A

Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

How to avoid deadlock?

A solution for a problem is found at its roots. In deadlock it is the pattern of accessing the resources A and B, is the main issue. To solve the issue we will have to simply re-order the statements where the code is accessing shared resources.

**DeadlockSolved.java**

1. **public** **class** DeadlockSolved {
3. **public** **static** **void** main(String ar[]) {
4. DeadlockSolved test = **new** DeadlockSolved();
6. **final** resource1 a = test.**new** resource1();
7. **final** resource2 b = test.**new** resource2();
9. // Thread-1
10. Runnable b1 = **new** Runnable() {
11. **public** **void** run() {
12. **synchronized** (b) {
13. **try** {
14. /\* Adding delay so that both threads can start trying to lock resources \*/
15. Thread.sleep(100);
16. } **catch** (InterruptedException e) {
17. e.printStackTrace();
18. }
19. // Thread-1 have resource1 but need resource2 also
20. **synchronized** (a) {
21. System.out.println("In block 1");
22. }
23. }
24. }
25. };
27. // Thread-2
28. Runnable b2 = **new** Runnable() {
29. **public** **void** run() {
30. **synchronized** (b) {
31. // Thread-2 have resource2 but need resource1 also
32. **synchronized** (a) {
33. System.out.println("In block 2");
34. }
35. }
36. }
37. };

40. **new** Thread(b1).start();
41. **new** Thread(b2).start();
42. }
44. // resource1
45. **private** **class** resource1 {
46. **private** **int** i = 10;
48. **public** **int** getI() {
49. **return** i;
50. }
52. **public** **void** setI(**int** i) {
53. **this**.i = i;
54. }
55. }
57. // resource2
58. **private** **class** resource2 {
59. **private** **int** i = 20;
61. **public** **int** getI() {
62. **return** i;
63. }
65. **public** **void** setI(**int** i) {
66. **this**.i = i;
67. }
68. }
69. }

**Output:**

In block 1

In block 2

In the above code, class DeadlockSolved solves the deadlock kind of situation. It will help in avoiding deadlocks, and if encountered, in resolving them.

How to Avoid Deadlock in Jaxva?

Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

1. **Avoid Nested Locks**: We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
2. **Avoid Unnecessary Locks**: The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
3. **Using Thread Join**: A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use **join** with a maximum time that a thread will take.

# Inter-thread Communication in Java

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) 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()
* notify()
* notifyAll()

### 1) wait() method

The wait() method causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final void wait()throws InterruptedException | It waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException | It waits for the specified amount of time. |

### 2) notify() method

The notify() method wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

**Syntax:**

1. **public** **final** **void** notify()

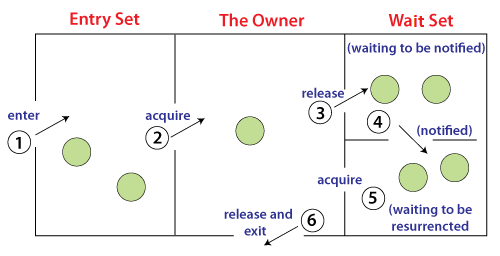
### 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.

**Syntax:**

1. **public** **final** **void** notifyAll()

## **Understanding the process of inter-thread communication**



The point to point explanation of the above diagram is as follows:

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.

### Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

### Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| The wait() method releases the lock. | The sleep() method doesn't release the lock. |
| It is a method of Object class | It is a method of Thread class |
| It is the non-static method | It is the static method |
| It should be notified by notify() or notifyAll() methods | After the specified amount of time, sleep is completed. |

### Example of Inter Thread Communication in Java

Let's see the simple example of inter thread communication.

**Test.java**

1. **class** Customer{
2. **int** amount=10000;
4. **synchronized** **void** withdraw(**int** amount){
5. System.out.println("going to withdraw...");
7. **if**(**this**.amount<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. **try**{wait();}**catch**(Exception e){}
10. }
11. **this**.amount-=amount;
12. System.out.println("withdraw completed...");
13. }
15. **synchronized** **void** deposit(**int** amount){
16. System.out.println("going to deposit...");
17. **this**.amount+=amount;
18. System.out.println("deposit completed... ");
19. notify();
20. }
21. }
23. **class** Test{
24. **public** **static** **void** main(String args[]){
25. **final** Customer c=**new** Customer();
26. **new** Thread(){
27. **public** **void** run(){c.withdraw(15000);}
28. }.start();
29. **new** Thread(){
30. **public** **void** run(){c.deposit(10000);}
31. }.start();
33. }}

**Output:**

going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed