Multithreading in Java

**Multithreading in java** is a process of executing multiple threads simultaneously.

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

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

### **Advantages of Java Multithreading**

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at 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 exception occur in a single thread.

### **Multitasking**

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

* Process-based Multitasking(Multiprocessing)
* Thread-based Multitasking(Multithreading)

### **1) Process-based Multitasking (Multiprocessing)**

* Each process have its own address in memory i.e. each process allocates separate memory area.
* Process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

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

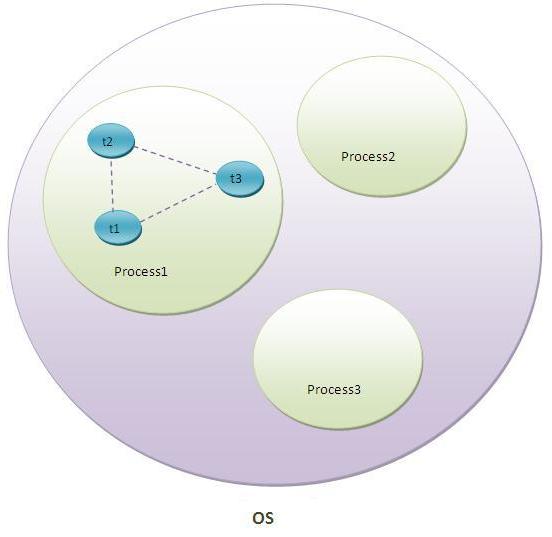
* Threads share the same address space.
* Thread is lightweight.
* Cost of communication between the thread is low.

#### Note: At least one process is required for each thread.

## **What is Thread in java**

A thread is a lightweight sub-process 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 shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

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

# 

Once we created a Thread object then the Thread is said to be in new state or

born state.

Once we call start() method then the Thread will be entered into Ready or

Runnable state.

If Thread Scheduler allocates CPU then the Thread will be entered into running

state.

Blocked/Waiting

A thread can be waiting for other thread to finish using [thread join](https://www.journaldev.com/1024/java-thread-join-example) or it can be waiting for some resources to available.

# once run() method completes then the Thread will entered into dead state

# How to create thread

There are two ways to create a thread:

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

**Best approach to define a Thread:**

 Among the 2 ways of defining a Thread, implements Runnable approach is

always recommended.

 In the 1st approach our class should always extends Thread class there is no

chance of extending any other class hence we are missing the benefits of

inheritance.

 But in the 2nd approach while implementing Runnable interface we can extend

some other class also. Hence implements Runnable mechanism is recommended

to define a Thread.

### **Thread class:**

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

### **Commonly used Constructors of Thread class:**

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

### **Commonly used methods of Thread class:**

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.StategetState():**returns the state of the thread. 13. **public booleanisAlive():**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 booleanisDaemon():**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 booleanisInterrupted():**tests if the thread has been interrupted. 22. **public static booleaninterrupted():**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. |

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### **Java 8 Runnable Lambda Example with Argument**

Java 8 supports lambda expression. In java 8 Runnable interface has been annotated with @FunctionalInterface. Now we can create Runnable instance using lambda expression.

**package**com.mgs.scala;

**import**java.util.Arrays;

**import**java.util.List;

**class** Book {

**privateint**id;

**private** String name;

**public**Book(**int**id, String name) {

**super**();

**this**.id = id;

**this**.name = name;

}

**publicint**getId() {

**return**id;

}

**publicvoid**setId(**int**id) {

**this**.id = id;

}

**public** String getName() {

**return**name;

}

**publicvoid**setName(String name) {

**this**.name = name;

}

}

**publicclass**JavaDemo {

**publicstaticvoid**main(String[] args) {

List<Book>list= Arrays.*asList*(**new** Book(101,"scala"),

**new** Book(102,"java"),**new** Book(103,"angular"));

Runnable r1 = () ->list.forEach((e) ->

System.***out***.println(e.getId() +" : "+e.getName()));

Thread t = **new** Thread(r1);

t.start();

}

}

### **Java 8 Callable Lambda Example with Argument**

Callable<V> interface has been introduced in Java 5 where **V** is a return type. In Java 8, Callable interface has been annotated with @FunctionalInterface. Now in java 8, we can create the object of Callable using lambda expression as follows.

Callable<Integer>callableObj=()->{return2\*3;};

The above code is equivalent to below code snippet.

Callable<Integer>callableObj=newCallable<Integer>(){

@Override

publicIntegercall()throwsException{

return2\*3;

}

};

To pass the argument to our call() method we should use **final** modifier.

finalintval=10;

Callable<Integer>callableObj=()->{return2\*val;};

**publicclass**JavaDemo {

**publicstaticvoid**main(String[] args) {

**final** List<Integer>list= Arrays.*asList*(1,2,3,4);

Callable<Integer>cb = () -> {

**int**result = list.stream().mapToInt(i ->i.intValue()).sum();

**return**result;

};

ExecutorServiceexecutorService = Executors.*newSingleThreadExecutor*();

Future<Integer>future = executorService.submit(cb);

Integer result = 0;

**try** {

result = future.get();

System.***out***.println(result);

} **catch** (InterruptedException | ExecutionExceptione) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

list1.removeAll(list2);//will delete all the objects of list1 that matches list2.  
list1.retainAll(list2);//will delete all the objects of list1 that not matches list2.

***yield():***

1. yield() method causes "to pause current executing Thread for giving the chance

of remaining waiting Threads of same priority".

2. If all waiting Threads have the low priority or if there is no waiting Threads then

the same Thread will be continued its execution.

3. If several waiting Threads with same priority available then we can't expect

exact which Thread will get chance for execution.

4. The Thread which is yielded when it get chance once again for execution is

depends on mercy of the Thread scheduler.

5. public static native void yield();

**Sleep() method:**

If a Thread don't want to perform any operation for a particular amount of time then

we should go for sleep() method.

1. public static native void sleep(long ms) throws InterruptedException

2. public static void sleep(long ms,int ns)throws InterruptedException

**Join:**

If a Thread wants to wait until completing some other Thread then we should go for

join() method.

Example: If a Thread t1 executes t2.join() then t1 should go for waiting state until

completing t2.

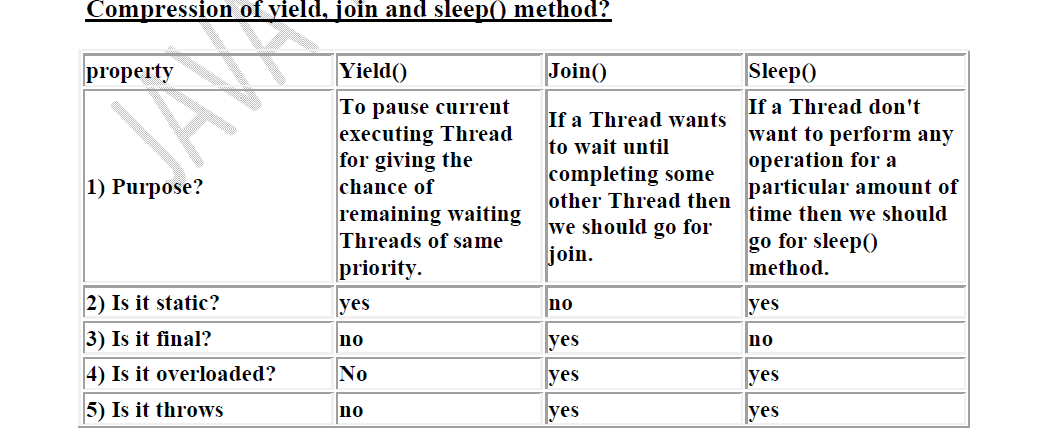
**Interrupting a Thread:**

How a Thread can interrupt another thread ?

If a Thread can interrupt a sleeping or waiting Thread by using interrupt()(break off)

method of Thread class.

public void interrupt();



**Synchronization**

1. Synchronized is the keyword applicable for methods and blocks but not for

classes and variables.

2. If a method or block declared as the synchronized then at a time only one

Thread is allow to execute that method or block on the given object.

3. The main advantage of synchronized keyword is we can resolve date

inconsistency problems.

4. But the main disadvantage of synchronized keyword is it increases waiting time

of the Thread and effects performance of the system.

5. Hence if there is no specific requirement then never recommended to use

synchronized keyword.

6. Internally synchronization concept is implemented by using lock concept.

7. Every object in java has a unique lock. Whenever we are using synchronized

keyword then only lock concept will come into the picture.

8. If a Thread wants to execute any synchronized method on the given object 1st it

has to get the lock of that object. Once a Thread got the lock of that object then

it's allow to execute any synchronized method on that object. If the synchronized

method execution completes then automatically Thread releases lock.

9. While a Thread executing any synchronized method the remaining Threads are

not allowed execute any synchronized method on that object simultaneously. But

remaining Threads are allowed to execute any non-synchronized method

simultaneously. [lock concept is implemented based on object but not based on

method].

***Class level lock:***

1. Every class in java has a unique lock. If a Thread wants to execute a static

synchronized method then it required class level lock.

2. Once a Thread got class level lock then it is allow to execute any static

synchronized method of that class.

3. While a Thread executing any static synchronized method the remaining

Threads are not allow to execute any static synchronized method of that class

simultaneously.

4. But remaining Threads are allowed to execute normal synchronized methods,

normal static methods, and normal instance methods simultaneously.

5. Class level lock and object lock both are different and there is no relationship

between these two.

***Synchronized block:***

1. If very few lines of the code required synchronization then it's never

recommended to declare entire method as synchronized we have to enclose those

few lines of the code with in synchronized block.

2. The main advantage of synchronized block over synchronized method is it

reduces waiting time of Thread and improves performance of the system.

***Example 1****:* To get lock of current object we can declare synchronized block as follows.

If Thread got lock of current object then only it is allowed to execute this block.

Synchronized(this){}

***Example 2****:* To get the lock of a particular object 'b' we have to declare a synchronized

block as follows.

If thread got lock of 'b' object then only it is allowed to execute this block.

Synchronized(b){}

***Example 3****:* To get class level lock we have to declare synchronized block as follows.

Synchronized(Display.class){}

If thread got class level lock of Display then only it allowed to execute this block.

Note: As the argument to the synchronized block we can pass either object reference or

".class file" and we can't pass primitive values as argument [because lock concept is

dependent only for objects and classes but not for primitives].

**What is synchronized statement**

**Ans**: The statements which present inside synchronized method and

synchronized block are called synchronized statements. [Interview people

created terminology]

**Inter Thread communication (wait(),notify(), notifyAll**

Two Threads can communicate with each other by using wait(), notify() and

notifyAll() methods**.**

**wait**

The java.lang.Object.wait() causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

**notify vs notifyAll():**

We can use notify() method to give notification for only one Thread. If multiple

Threads are waiting then only one Thread will get the chance and remaining

Threads has to wait for further notification. But which Thread will be

notify(inform) we can't expect exactly it depends on JVM.

We can use notifyAll() method to give the notification for all waiting Threads. All

waiting Threads will be notified and will be executed one by one, because they

are required lock

Note: On which object we are calling wait(), notify() and notifyAll() methods that

corresponding object lock we have to get but not other object locks

**DeadLock**

If 2 Threads are waiting for each other forever(without end) such type of

situation(infinite waiting) is called dead lock

**Daemon Threads**

The Threads which are executing in the background are called daemon Threads.

The main objective of daemon Threads is to provide support for non-daemon Threads

like main Thread.

Example:

*Garbage collector*

***ThreadGroup:***

Based on functionality we can group threads as a single unit which is nothing but

ThreadGroup.

i.e. thread group represent set of threads.

ThreadGroup provides a convenient way to perform common operations for all threads

belongs to a perticular group.

