**Java thread model, thread priorities, synchronization, messaging, thread class and runnable interface, creating thread, creating multiple threads, thread priorities, synchronizing threads, interthread communication, thread life cycle.**

**Real-life Example of Java Multithreading**

Suppose you are using two tasks at a time on the computer, be it using Microsoft Word and listening to music. These two tasks are called processes . So you start typing in Word and at the same time start music app, this is called multitasking . Now you committed a mistake in a Word and spell check shows exception, this means Word is a process that is broken down into sub-processes. Now if a machine is dual-core then one process or task is been handled by one core and music is been handled by another core.

1. **Multiprocessing**: Process-based multitasking is a heavyweight process and occupies different address spaces in memory. Hence, while switching from one process to another, it will require some time be it very small, causing a lag because of switching. This happens as registers will be loaded in memory maps and the list will be updated.
2. **Multithreading**: Thread-based multitasking is a lightweight process and occupies the same address space. Hence, while switching cost of communication will be very less.

## **What is Thread in java**

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

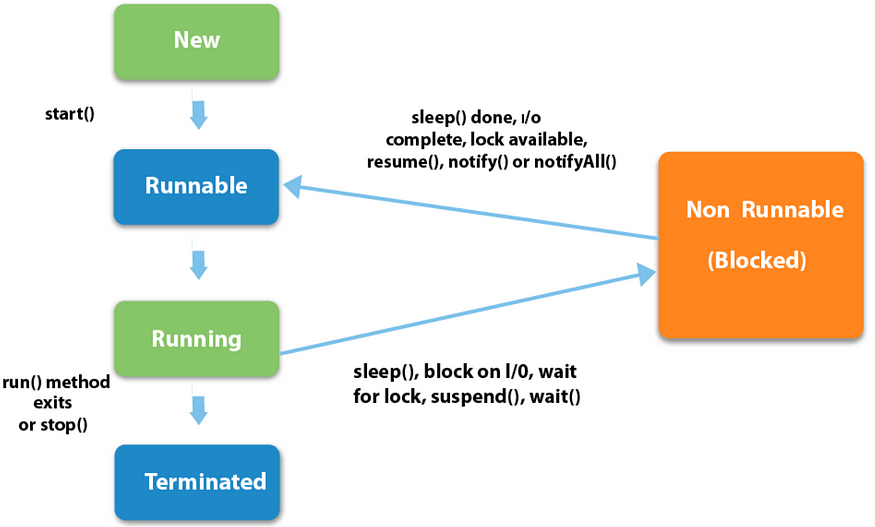
Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.



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.

**Life Cycle of a Thread in Java Multithreading**

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.

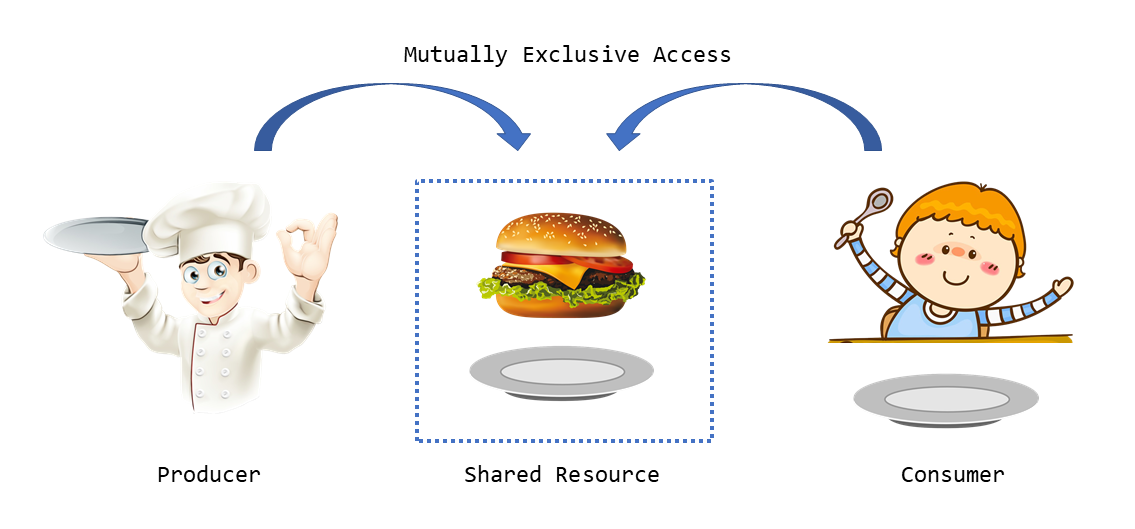
****

According to Sun microsystems, there are 4 states in the java thread life cycle. They are:

* New — A thread is in the “New” state, when an object of the thread class is instantiated but the “start” method is not invoked.
* Runnable — When the “start” method has been invoked on the thread object. In this state, the thread is either waiting for the scheduler to pick it up for execution or it’s already running. Let us call the state when the thread is already picked for execution, the “running” state.
* Non-Runnable(Blocked , Timed-Waiting)— When the thread is alive, i.e., the thread class object exists, but it cannot be picked by the scheduler for execution. It is temporarily inactive.
* Terminated — When the thread completes execution of its “run” method, it goes into the “terminated” state. At this stage, the task of the thread is completed.

## **Synchronization**

Java supports an asynchronous multithreading, any number of thread can run simultaneously without disturbing other to access individual resources at different instant of time or shareable resources. But some time it may be possible that shareable resources are used by at least two threads or more than two threads, one has to write at the same time, or one has to write and other thread is in the middle of reading it. For such type of situations and circumstances Java implements synchronization model called *monitor.* The monitor was first defined by C.A.R. Hoare. You can consider the monitor as a box, in which only one thread can reside. As a thread enter in monitor, all other threads have to wait until that thread exits from the monitor. In such a way, a monitor protects the shareable resources used by it being manipulated by other waiting threads at the same instant of time. Java provides a simple methodology to implement synchronization.



## **Messaging**

A program is a collection of more than one thread. Threads can communicate with each other. Java supports messaging between the threads with lost-cost. It provides methods to all objects for inter-thread communication. As a thread exits from synchronization state, it notifies all the waiting threads.

**creating 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 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.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.

**Example**

class ThreadExample1 **extends** Thread

{

**public** **void** run()

{

System.out.println("thread is running...");

}

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

{

ThreadExample1 t1=**new** ThreadExample1();

t1.start();

 }

}

**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.

**Example:**

class ThreadExample2 **implements** Runnable

{

**public** **void** run()

{

System.out.println("thread is running...");

}

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

{

ThreadExample2 m1=**new** ThreadExample2();

Thread t1 =**new** Thread(m1); // Using the constructor Thread(Runnable r)t1.start();

 }

}

**Key Points**

* **extending**Thread: Use this method if you don’t need to extend another class.
* **implementing**Runnable: Use this method for more flexibility, especially if your class needs to extend another class.

**Creating multiple threads:**

class MyThread extends Thread

{

public void run()

{

for (int i = 1; i <= 5; i++)

{

System.out.println(Thread.currentThread().getName() + ": " + i);

try

{

// Pause the thread for 500 milliseconds.

Thread.sleep(500); // Sleep for 500 milliseconds

}

catch(InterruptedException e)

{

System.out.println(e); // Handle any interruptions.

}

}

}

}

public class MultithreadingTest

{

public static void main(String[] args)

{

// Creating two objects of MyThread.

MyThread t1 = new MyThread();

MyThread t2 = new MyThread();

// Start the first thread.

t1.start();

// Start the second thread.

t2.start();

}

}

## **Thread Priorities**

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

**Daemon threads in java:**

In Java, a **daemon thread** is a low-priority thread that runs in the background to perform tasks such as garbage collection and other housekeeping activities. Here are some key points about daemon threads:

1. **Purpose**: Daemon threads provide services to user threads. [They are typically used for background tasks that do not need to be completed before the program exits](https://www.baeldung.com/java-daemon-thread).
2. **Lifecycle**: The JVM terminates daemon threads automatically when all user threads have finished execution. [This means that daemon threads do not prevent the JVM from shutting down](https://www.baeldung.com/java-daemon-thread).
3. **Creation:** To create a daemon thread, you can use the setDaemon(true) method before starting the thread. For example:

Thread daemonThread = new Thread(()

{

// Daemon thread task

});

daemonThread.setDaemon(true);

daemonThread.start();

**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

**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

**Understanding the problem without Synchronization**

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

class Table

{

void printTable(int n)

{//method not synchronized

for(int i=1;i<=5;i++)

{

System.out.println(n\*i);

try

{

Thread.sleep(400);

}

catch(Exception e)

{

System.out.println(e);

}

}

}

}

class MyThread2 extends Thread

{

Table t;

MyThread2(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(5);

}

}

class MyThread3 extends Thread

{

Table t;

MyThread3(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(100);

}

}

class TestSynchronization1

{

public static void main(String args[])

{

Table obj = new Table();//only one object

MyThread2 t1=new MyThread2(obj);

MyThread3 t2=new MyThread3(obj);

t1.start();

t2.start();

}

}

**1.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.

**Example:**

synchronized void printTable(int n)

{//synchronized method

for(int i=1;i<=5;i++)

{

System.out.println(n\*i);

try

{

Thread.sleep(400);

}

catch(Exception e)

{

System.out.println(e);

}

}

}

**2.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.

**Syntax**

**synchronized** (object reference expression){

  //code block

}

**Example:**

void printTable(int n)

{

synchronized(this)

{//synchronized block

for(int i=1;i<=5;i++)

{

System.out.println(n\*i);

try

{

Thread.sleep(400);

}

catch(Exception e)

{

System.out.println(e);

}

}

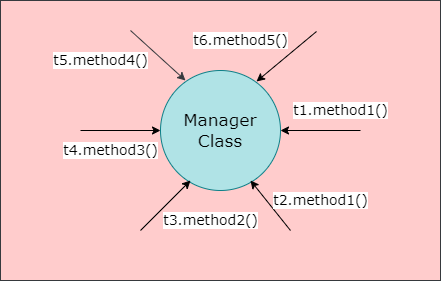
}

}//end of the method

**3.Static Synchronization**

n Java, a static synchronised method is also a technique of synchronising a method so that no two threads can act on the synchronised method simultaneously. The only change is that Static Synchronised is used. We're achieving a **class-level lock**, which means that only one thread can use the method. The Thread class will obtain a class level lock of a java class, allowing only one thread to interact with the static synchronised method.

Let us assume that there are six threads. The following will be the order of execution:



**Threads**

Here, we take six threads, namely, t1, t2, t3, t4, t5 and t6.

**Methods**

The complete declarations of the various methods are:

method1: public static synchronized void method1()

method2: public static synchronized void method2()

method3: public static void method3()

method4: public synchronized int method4()

method5: public String method5()

The execution flow of the method will be like this:

* When t1.method1() obtains class level lock of the Manager class, it begins execution.
* Because it is a static synchronised method, t2.method2() must wait for its time to begin execution. Because t1 has already obtained the class level lock, t2 must wait till t1 executes.
* t3.method2() must wait until t1 releases the lock because it requires a class level lock.
* t4.method3() begins execution immediately because it is a static method that does not require a lock.
* Because t5.method4() is an instance (or normal) level synchronised method that requires object level lock, it obtains object level lock.
* t6.method5() executes as an instance method or as a regular method.

**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:**

**public** **final** **void** notify()

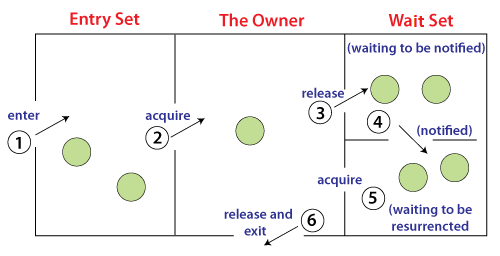
**3) notifyAll() method**

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

**Syntax:**

**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.

**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.

