Thread Questions:

**1) What is multithreaded programming? Does Java supports multithreaded programming? Explain with an example?**

In a program or in an application, when two or more threads execute their task simultaneously then it is called multi threaded programming.

Yes, Java supports multithreaded programming.

For example, in the below code, main thread which is responsible for executing the main() method, creates two threads – t1 and t2. t1 prints numbers from 1 to 1000 and t2 prints numbers from 1001 to 2000. These two threads execute their task simultaneously not one after the other. This is called multi threaded programming.

//Thread1 : The task of this thread is to print numbers from 1 to 1000

**class** Thread1 **extends** Thread

{

    @Override

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

    {

**for** (**int** i = 1; i <= 1000; i++)

        {

            System.out.println(i);

        }

    }

}

//Thread2 : The task of this thread is to print numbers from 1001 to 2000

**class** Thread2 **extends** Thread

{

    @Override

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

    {

**for** (**int** i = 1001; i <= 2000; i++)

        {

            System.out.println(i);

        }

    }

}

**public** **class** JavaThreadsInterviewQuestions

{

    //Main Thread : The task of this thread is to execute main() method

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

    {

        //Creating and starting first thread

        Thread1 t1 = **new** Thread1();

        t1.start();

        //Creating and starting second thread

        Thread2 t2 = **new** Thread2();

        t2.start();

        //Both these two threads will be executed simultaneously

    }

}

**2) In how many ways, you can create threads in Java? What are those? Explain with examples?**

There are two ways two create threads in Java.

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

1) Creating thread by extending java.lang.Thread class :

Your thread must extend Thread class and override run() method. Whatever the task which you want to be performed by this thread, keep that task in the overridden run() method.

|  |
| --- |
| **class** MyThread **extends** Thread  {      @Override  **public** **void** run()      {          //Keep the task to be performed here      }  } |

Where ever you want this task to be performed, create an object to your thread class and call start() method.

MyThread myThread = **new** MyThread();

myThread.start();

2) By implementing java.lang.Runnable interface

Runnable interface has only one methode i.e run() method. Your thread class must implement Runnable interface and override run() method and keep the task to be performed in this run() method.

**class** MyRunnable **implements** Runnable

{

    @Override

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

    {

        //Keep the task to be performed here

    }

}

Whenever you want this task to be performed, create an object to java.lang.Thread class by passing an object of your thread class which implements Runnable interface and call start() method.

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

t.start();

**3) How many types of threads are there in Java? Explain?**

There are two types of threads in Java. They are,

1. User Threads
2. Daemon Threads

User threads are threads which are created by the application or user. They are high priority threads. JVM will not exit until all user threads finish their execution. JVM wait for user threads to finish their task. These threads are foreground threads.

Daemon threads are threads which are mostly created by the JVM. These threads always run in background. These threads are used to perform some background tasks like garbage collection. These threads are less priority threads. JVM will not wait for these threads to finish their execution. JVM will exit as soon as all user threads finish their execution.

**4.) What is the default daemon status of a thread? How do you check it?**

Default daemon status of a thread is inherited from it’s parent thread i.e a thread created by user thread will be a user thread and a thread created by a daemon thread will be a daemon thread.

isDaemon() method is used to check whether a thread is daemon thread or not.

**5) Can you convert user tread into daemon thread and vice-versa? Explain with example?**

Yes, you can convert user thread into daemon thread and vice-versa using setDaemon() method. But, it has to be done before starting the thread. If you call this method after starting the thread, you will get java.lang.IllegalThreadStateException.

**class** UserThread **extends** Thread

{

    @Override

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

    {

        System.out.println("Keep user thread task here...");

    }

}

**public** **class** JavaThreadsInterviewQuestions

{

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

    {

        UserThread userThread = **new** UserThread();

        userThread.setDaemon(**true**);

        userThread.start();

    }

}

**6) Is it possible to give a name to a thread? If yes, how do you do that? What will be the default name of a thread if you don’t name a thread?**

Yes, it is possible to give a name to a thread. It can be done via setName() method or else you can pass the name while creating the thread itself.

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

{

**public** MyThread(String name)

    {

**super**(name);

    }

    @Override

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

    {

        System.out.println("Keep the task to be performed here...");

    }

}

**public** **class** JavaThreadsInterviewQuestions

{

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

    {

        MyThread myThread = **new** MyThread("My\_Thread");

        myThread.start();

        System.out.println(myThread.getName());   //Output : My\_Thread

        myThread.setName("My\_Thread\_2.0");

        System.out.println(myThread.getName());   //Output : My\_Thread\_2.0

    }

}

If you don’t name a thread, thread will get default name. Default name of the thread will consist of a word “Thread”, followed by hyphen (-) and followed by an integer number starting from 0 like Thread-0, Thread-1, Thread-2.

**7) Can we change the name of the main thread? If yes, How?**

Yes, we can change the name of the main thread. Below code shows how to do it.

**public** **class** JavaThreadsInterviewQuestions

{

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

    {

        Thread t = Thread.currentThread();

        System.out.println(t.getName());       //Output : main

        t.setName("My\_Main\_Thread");

        System.out.println(t.getName());      //Output : My\_Main\_Thread

    }

}

**8) Do two threads can have same name? If yes then how do you identify the threads having the same name?**

Yes, two threads can have same name. In such scenarios, Thread ID can be used to identify the threads. Thread ID is a unique long number which remains unchanged through out the life of a thread. Thread ID can be retrieved using getID() method.

**9) What are MIN\_PRIORITY, NORM\_PRIORITY and MAX\_PRIORITY?**

MIN\_PRIORITY, NORM\_PRIORITY and MAX\_PRIORITY are three constant fields in java.lang.Thread class which define lowest, normal and highest priority of a thread respectively.

MIN\_PRIORITY : It defines the lowest priority that a thread can have and it’s value is 1.

NORM\_PRIORITY : It defines the normal priority that a thread can have and it’s value is 5.

MAX\_PRIORITY : It defines the highest priority that a thread can have and it’s value is 10.

**10) What is the default priority of a thread? Can we change it? If yes, how?**

The default priority of a thread is same as that of it’s parent. We can change the priority of a thread at any time using setPriority() method.

**11) What is the priority of main thread? Can we change it?**

The priority of a main thread, if explicitly not set, is always NORM\_PRIORITY i.e 5.

Yes, we can change the priority of a main thread using setPriority() method.

**public** **class** JavaThreadsInterviewQuestions

{

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

    {

        Thread t = Thread.currentThread();

        System.out.println(t.getPriority());       //Output : 5

        t.setPriority(8);

        System.out.println(t.getPriority());      //Output : 8

    }

}

**12) What is the purpose of Thread.sleep() method?**

Thread.sleep() is used to pause the execution of current thread for a specified period of time.

**13) Can you tell which thread is going to sleep after calling myThread.sleep(5000) in the below program? is it main thread or myThread?**

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

{

    @Override

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

    {

**for** (**int** i = 0; i <= 10000; i++)

        {

            System.out.println(i);

        }

    }

}

**public** **class** JavaThreadsInterviewQuestions

{

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

    {

        MyThread myThread = **new** MyThread();

        myThread.start();

**try**

        {

            myThread.sleep(5000);

        }

**catch** (InterruptedException e)

        {

            e.printStackTrace();

        }

    }

}

t is the main thread which is going to sleep not myThread. Because, when you call sleep() method, it is currently executing thread which is going to sleep, not on which you have called it.

To sleep myThread in the above program, call Thread.sleep() inside the run() method of MyThread class.

**14) Does the thread releases the lock it holds when it is going for sleep?**

No. When the thread is going for sleep, it does not release the synchronized locks it holds.

**15) What is the purpose of join() method? Explain with an example?**

join() method can be used to apply the order of execution on threads. Using join() method, you can make the currently executing thread to wait for the some other threads to finish their task. For example, let’s us assume that there are two threads – thread1 and thread2. You can make thread1 to hold it’s execution for some time so that thread2 can finish it’s task. After thread2 finishes it’s task, thread1 resumes it’s execution. For this to happen, you should call join() method on thread2 within thread1.

**16) What do you mean by synchronization? Explain with an example?**

Through synchronization, we can make the threads to execute particular method or block in sync not simultaneously. When a method or block is declared as synchronized, only one thread can enter into that method or block. When one thread is executing synchronized method or block, the other threads which wants to execute that method or block have to wait until first thread executes that method or block. Thus avoiding the thread interference and achieving the thread safeness.

**class** Shared

{

**int** i;

**synchronized** **void** SharedMethod()

    {

        Thread t = Thread.currentThread();

**for**(i = 0; i <= 1000; i++)

        {

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

        }

    }

}

**public** **class** ThreadsInJava

{

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

    {

**final** Shared s1 = **new** Shared();

        Thread t1 = **new** Thread("Thread - 1")

        {

            @Override

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

            {

                s1.SharedMethod();

            }

        };

        Thread t2 = **new** Thread("Thread - 2")

        {

            @Override

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

            {

                s1.SharedMethod();

            }

        };

        t1.start();

        t2.start();

    }

}

In the above example, both threads t1 and t2 wants to execute sharedMethod() of s1 object. But, sharedMethod() is declared as synchronized. So, whichever thread enters first into sharedMethod(), it continues to execute that method. The other thread waits for first thread to finish it’s execution of sharedMethod(). It never enters into sharedMethod() until first thread is done with that method. That means, both threads are executing sharedMethod() one by one not simultaneously.

**17) What is object lock or monitor?**

The synchronization in Java is built around an entity called object lock or monitor. Below is the brief description about lock or monitor.

* Whenever an object is created to any class, an object lock is created and is stored inside the object.
* One object will have only one object lock associated with it.
* Any thread wants to enter into synchronized methods or blocks of any object, they must acquire object lock associated with that object and release the lock after they are done with the execution.
* The other threads which wants to enter into synchronized methods of that object have to wait until the currently executing thread releases the object lock.
* To enter into static synchronized methods or blocks, threads have to acquire class lock associated with that class as static members are stored inside the class memory.

**18) I want only some part of the method to be synchronized, not the whole method? How do you achieve that?**

This can be done using synchronized blocks.

**19) What is the use of synchronized blocks?**

Synchronization slows down the application. Because, at any given time, only one thread can enter into synchronized method. Other threads have to wait until first thread finishes it’s execution of that method. This slows down the execution of whole application.

Instead of synchronizing the whole method, synchronizing the only that part which is to be monitored for thread safe saves the time. This can be done using synchronized blocks.

**20) What is mutex?**

synchronized block takes one argument and it is called mutex. If synchronized block is defined inside non-static definition blocks like non-static methods, instance initializer or constructors, then this mutex must be an instance of that class. If synchronized block is defined inside static definition blocks like static methods or static initializer, then this mutex must be like ClassName.class.

**21) Is it possible to make constructors synchronized?**

Not possible. Synchronized keyword can not be used with constructors. But, constructors can have synchronized blocks.

**22) Can we use synchronized keyword with variables?**

No, you can’t use synchronized keyword with variables. You can use synchronized keyword only with methods but not with variables, constructors, static initializers and instance initializers.

**23) As you know that synchronized static methods need class level lock and synchronized non-static methods need object level lock. Is it possible to run these two methods simultaneously?**

Yes. It is possible.

**24) If a particular thread caught with exceptions while executing a synchronized method, does executing thread releases lock or not?**

Thread must release the lock whether the execution is completed normally or caught with exceptions.

**25) Synchronized methods or synchronized blocks – which one do you prefer?**

Synchronized blocks are better than synchronized methods. Because, synchronizing some part of a method improves the performance than synchronizing the whole method.

**26) What is deadlock in Java?**

Deadlock in Java is a condition which occurs when two or more threads get blocked waiting for each other for an infinite period of time to release the resources(Locks) they hold.

**27) How do you programatically detect the deadlocked threads in Java?**

**import** java.lang.management.ManagementFactory;

**import** java.lang.management.ThreadInfo;

**import** java.lang.management.ThreadMXBean;

**public** **class** JavaThreadsInterviewQuestions

{

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

    {

        ThreadMXBean bean = ManagementFactory.getThreadMXBean();

**long** ids[] = bean.findMonitorDeadlockedThreads();

**if**(ids != **null**)

        {

            ThreadInfo threadInfo[] = bean.getThreadInfo(ids);

**for** (ThreadInfo threadInfo1 : threadInfo)

            {

                System.out.println(threadInfo1.getThreadId());    //Prints the ID of deadlocked thread

                System.out.println(threadInfo1.getThreadName());  //Prints the name of deadlocked thread

                System.out.println(threadInfo1.getLockName());    //Prints the string representation of an object for which thread has entered into deadlock.

                System.out.println(threadInfo1.getLockOwnerId());  //Prints the ID of thread which currently owns the object lock

                System.out.println(threadInfo1.getLockOwnerName());  //Prints name of the thread which currently owns the object lock.

            }

        }

    }

}

**28) What do you know about lock ordering and lock timeout?**

Lock ordering and lock timeout are two methods which are used to avoid the deadlock in Java.

Lock Ordering : In this method of avoiding the deadlock, some predefined order is applied for threads to acquire the locks they need. For example, If there are three threads t1, t2 and t3 running concurrently and they needed locks A, B and C. t1 needs A and B locks, t2 needs A and C locks and t3 needs A, B and C locks. If you define an order to acquire the locks like, Lock A must be acquired before Lock B and Lock B must be acquired before Lock c, then deadlock never occurs.

Lock Timeout : It is another deadlock preventive method in which we specify the time for a thread to acquire the lock. If it fails to acquire the specified lock in the given time, then it should give up trying for a lock and retry after some time.

**29. How threads communicate with each other in Java?**

Threads in Java communicate with each other using wait(), notify() and notifyAll() methods.

wait() : This method tells the currently executing thread to release the lock of this object and wait until some other thread acquires the same lock and notify it using either notify() or notifyAll() methods.

notify() : This method wakes up one thread randomly that called wait() method on this object.

notifyAll() : This method wakes up all the threads that called wait() method on this object. But, only one thread will acquire lock of this object depending upon the priority.

**30) What is the difference between wait() and sleep() methods in Java?**

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| The thread which calls wait() method releases the lock it holds. | The thread which calls sleep() method doesn’t release the lock it holds. |
| The thread regains the lock after other threads call either notify() or notifyAll() methods on the same lock. | No question of regaining the lock as thread doesn’t release the lock. |
| wait() method must be called within the synchronized block. | sleep() method can be called within or outside the synchronized block. |
| wait() method is a member of java.lang.Object class. | sleep() method is a member of java.lang.Thread class. |
| wait() method is always called on objects. | sleep() method is always called on threads. |
| wait() is a non-static method of Object class. | sleep() is a static method of Thread class. |
| Waiting threads can be woken up by other threads by calling notify() or notifyAll() methods. | Sleeping threads can not be woken up by other threads. If done so, thread will throw InterruptedException. |
| To call wait() method, thread must have object lock. | To call sleep() method, thread need not to have object lock. |

31. **What is the difference between notify() and notifyAll() in Java?**

notify() : When a thread calls notify() method on a particular object, only one thread will be notified which is waiting for the lock or monitor of that object. The thread chosen to notify is random i.e randomly one thread will be selected for notification. Notified thread doesn’t get the lock of the object immediately. It gets once the calling thread releases the lock of that object.

notifyAll() : When a thread calls notifyAll() method on a particular object, all threads which are waiting for the lock of that object are notified. All notified threads will move from WAITING state to BLOCKED state. All these threads will get the lock of the object on a priority basis. The thread which gets the lock of the object moves to RUNNING state. The remaining threads will remain in BLOCKED state until they get the object lock.

32. **What do you know about interrupt() method? Why it is used?**

interrupt() method is used to interrupt sleeping or waiting thread. The whole thread interruption mechanism depends on an internal flag called interrupt status. The initial value of this flag for any thread is false. When you call interrupt() method on a thread, interrupt status of that thread will be set to true. When a thread throws InterruptedException, this status will be set to false again.

33. **How do you check whether a thread is interrupted or not?**

isInterrupted() or interrupted() method is used to check whether a particular thread is interrupted or not.

**34)What is the difference between isInterrupted() and interrupted() methods?**

Both, isInterrupted() and interrupted() methods are used to check whether a particular thread is interrupted or not. Both these methods return current interrupt status of a thread. isInterrupted() is a non-static method where as interrupted() is a static method of java.lang.Thread class. The main difference between these two methods is that isInterrupted() doesn’t clear the interrupt status where as interrupted() clears the interrupt status of a thread.

**35) Can a thread interrupt itself? Is it allowed in Java?**

Yes, a thread can interrupt itself. It is very much legal in Java.

36) **Can a thread interrupt itself? Is it allowed in Java?**

Yes, a thread can interrupt itself. It is very much legal in Java.

37) **In what state deadlocked threads will be?**

Deadlocked threads will be in BLOCKED state.

38) **What is the difference between BLOCKED and WAITING states?**

a thread will be in WAITING state if it is waiting for notification from other threads. A thread will be in BLOCKED state if it is waiting for other thread to release the lock it wants.

A thread enters into WAITING state when it calls wait() or join() method on an object. Before entering into WAITING state, thread releases the lock of the object it holds. It will remain in WAITING state until any other thread calls either notify() or notifyAll() on the same object.

Once the other thread calls notify() or notifyAll() on the same object, one or all the threads which are WAITING for lock of that object will be notified. All the notified threads will not get the object lock immediately. They will get the object lock on a priority basis once the current thread releases the lock. Until that they will be in BLOCKED state.

39) **Can we call start() method twice?**

No, start() method must be called only once. If you call start() method second time, it will throw IllegalThreadStateException as thread is already started.

40) **What is the difference between calling start() method and calling run() method directly as anyhow start() method internally calls run() method?**

When you call start() method, a new thread is created and that newly created thread executes the task kept in the run() method. If you call run() method directly, no new thread is created. Any task kept in run() method is executed by the calling thread itself.

If you are calling run() method directly, then you are not making use of the multi-threaded programming concept. Because, when you call run() method directly, no new thread is created. run() method is executed by the calling thread itself. It just acts as normal method invocation. You are not using the concept of multi-threading.

41) **How do you stop a thread?**

As stop() method has been deprecated, there are two ways through which you can stop a thread in Java. One is using boolean variable and second one is using interrupt() method.

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

{

    //Initially setting the flag as true

**private** **volatile** **boolean** flag = **true**;

    //This method will set flag as false

**public** **void** stopRunning()

    {

        flag = **false**;

    }

    @Override

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

    {

        //Keep the task in while loop

        //This will make thread continue to run until flag becomes false

**while** (flag)

        {

            System.out.println("I am running....");

        }

        System.out.println("Stopped Running....");

    }

}

**public** **class** MainClass

{

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

    {

        MyThread thread = **new** MyThread();

        thread.start();

**try**

        {

            Thread.sleep(100);

        }

**catch** (InterruptedException e)

        {

            e.printStackTrace();

        }

        //call stopRunning() method whenever you want to stop a thread

        thread.stopRunning();

    }

}

### 42) How To Stop A Thread In Java Using interrupt() Method?

In this method, we use *interrupt()* method to stop a thread. Whenever you call *interrupt()* method on a thread, it sets the *interrupted status* of a thread. This status can be obtained by *interrupted()* method. This status is used in a *while* loop to stop a thread.

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

{

    @Override

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

    {

**while** (!Thread.interrupted())

        {

            System.out.println("I am running....");

        }

        System.out.println("Stopped Running.....");

    }

}

**public** **class** MainClass

{

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

    {

        MyThread thread = **new** MyThread();

        thread.start();

**try**

        {

            Thread.sleep(100);

        }

**catch** (InterruptedException e)

        {

            e.printStackTrace();

        }

        //interrupting the thread

        thread.interrupt();

    }

}