THREADS - Top 80 interview questions and answers in java for freshers and experienced(detailed explanation with programs) Set-1 > Q1- Q60

Interviewers always have great focus on testing interviewees multi threading skills, so it's time to discuss multithreading in detail. So, I have tried to cover maximum number of the possible question which could be framed from multithreading. To answers these questions in comprehensive manner I have given programs and detailed explanation for for each and every question. These questions will be very handy for **fresh learners** to **experienced java developers**.

Question 1. What is Thread in java?

Answer.

- Threads **consumes CPU in best possible manner**, hence enables multi processing. Multi threading **reduces idle time of CPU** which improves performance of application.
- Thread are **light weight process**.
- A thread class belongs to **java.lang package**.
- We can create multiple threads in java, **even if we don't create any Thread, one Thread at least do exist** i.e. **main thread**.
- **Multiple threads run parallely in java.**
- Threads have their **own stack**.
- **Advantage** of Thread : Suppose one thread needs 10 minutes to get certain task, 10 threads used at a time could complete that task in 1 minute, because threads can run parallely.

Also Read : Top and most important Interview Questions and answers in Java

Question 2. What is difference between Process and Thread in java?

Answer. One process can have multiple Threads,

Thread are **subdivision** of Process. One or more Threads runs in the context of process. Threads can execute any part of process. And same part of process can be executed by multiple Threads.

Processes have their own **copy of the data segment of the parent process** while Threads have **direct access to the data segment of its process**.

Processes have their **own address** while Threads share the **address space of the process that created it**.

Process creation needs whole lot of stuff to be done, we **might need to copy whole parent process**, but Thread can be **easily created**.

Processes can **easily communicate with child processes** but **interprocess communication is difficult**. While, Threads **can easily communicate with other threads of the same process using wait() and notify() methods**.

In process **all threads share system resource** like heap Memory etc. while Thread has its **own stack.**

Any change made to process **does not affect child processes**, but any change made to thread **can affect the behavior of the other threads of the process**.

Example to see where threads on are created on different processes and same process.

Question 3. How to implement Threads in java?

Answer. This is very basic threading question. Threads can be created in two ways i.e. by implementing **java.lang.Runnable** interface or extending **java.lang.Thread** class and then extending run method.

Thread has its own variables and methods, it lives and dies on the heap. But a thread of execution is an individual process that has its own call stack. Thread are lightweight process in java.

1. Thread creation by implementing **java.lang.Runnable** interface.

We will create object of class which implements Runnable interface :

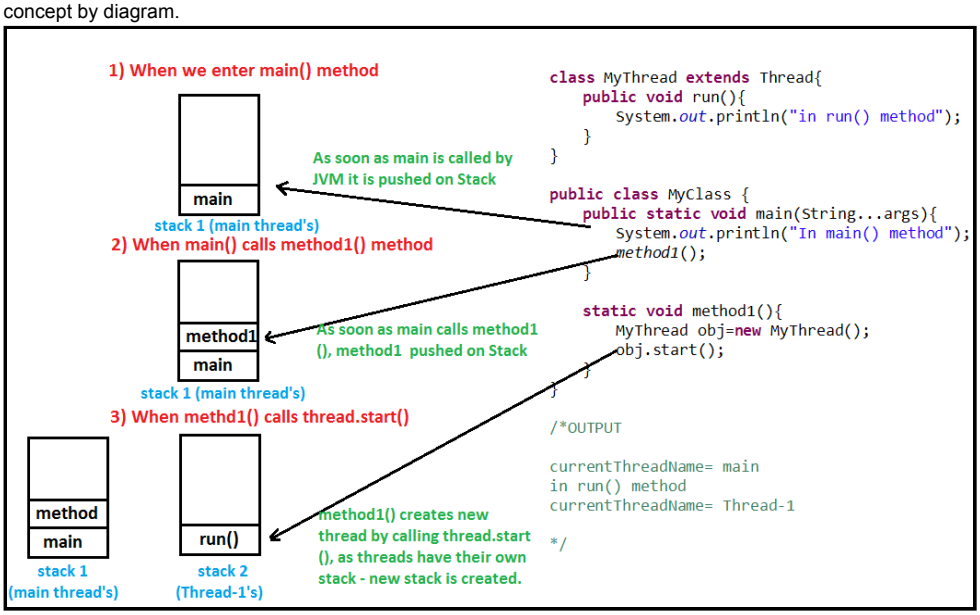
```
MyRunnable runnable=new MyRunnable();
Thread thread=new Thread(runnable);
```

2) And then create Thread object by calling constructor and passing reference of Runnable interface i.e. `runnable` object :

```
Thread thread=new Thread(runnable);
```

Question 4 . Does Thread implements their own Stack, if yes how? (Important)

Answer. **Yes**, Threads have their own stack. This is very interesting question, where interviewer tends to check your basic knowledge about how threads internally maintains their own stacks. I'll be explaining you the concept by diagram.



Question 5. We should implement Runnable interface or extend Thread class. What are differences between implementing Runnable and extending Thread?

Answer. Well the answer is you must **extend Thread** only when you are looking to **modify run() and other methods as well**. If you are simply looking to **modify only the run() method implementing Runnable is the best option (**Runnable interface has only one abstract method i.e. run() **)**.

Differences between implementing Runnable interface and extending Thread class **-**

1. ***Multiple inheritance in not allowed in java :*** When we implement Runnable interface **we can extend another class as well**, but if we extend Thread class **we cannot extend any other class** because java does not allow multiple inheritance. So, same work is done by implementing Runnable and extending Thread but in case of implementing Runnable we are still left with option of extending some other class. **So, it's better to implement Runnable.**

2. ***Thread safety :*** When we implement Runnable interface, **same object is shared amongst multiple threads**, but when we extend Thread class **each and every thread gets associated with new object**.

3. ***Inheritance (Implementing Runnable is lightweight operation) :*** When we extend Thread **unnecessary all Thread class features are inherited**, but when we implement Runnable interface no **extra feature are inherited**, as Runnable only consists only of one abstract method i.e. run() method. **So, implementing Runnable is lightweight operation.**

4. ***Coding to interface :*** Even **java recommends coding to interface**. So, we must implement Runnable rather than extending thread. Also, Thread class implements Runnable interface.

5. ***Don't extend unless you wanna modify fundamental behaviour of class, Runnable interface has only one abstract method i.e. run() :*** We must **extend Thread** only when you are looking to **modify run() and other methods as well**. If you are simply looking to **modify only the run() method implementing Runnable is the best option (**Runnable interface has only one abstract method

i.e. run() **). We must not extend Thread class unless we're looking to modify fundamental behaviour of Thread class.**

6. ***Flexibility in code when we implement Runnable :*** When we extend Thread first a fall all thread features are inherited and **our class becomes direct subclass of Thread , so whatever action we are doing is in Thread class**. But, when we implement Runnable **we create a new thread and pass runnable object as parameter, we could pass runnable object to executorService & much more**. So, we have more options when we implement Runnable and **our code becomes more flexible.**

7. ***ExecutorService :*** If we implement Runnable, **we can start multiple thread created on runnable object with ExecutorService** (because we can start Runnable object with new threads), **but not in the case when we extend** Thread (because thread can be started only once).

Question 6. How can you say Thread behaviour is unpredictable? (Important)

Answer. The solution to question is quite simple, Thread behaviour is unpredictable because execution of Threads depends on Thread scheduler, thread scheduler may have different implementation on different platforms like windows, unix etc. Same threading program may produce different output in subsequent executions even on same platform.

To achieve we are going to create 2 threads on same Runnable Object, create for loop in run() method and start both threads. There is no surety that which threads will complete first, both threads will enter anonymously in for loop.

Question 7 . When threads are not lightweight process in java?

Answer. Threads are **lightweight process only if threads of same process are executing concurrently**. But **if threads of different processes are executing concurrently then threads are heavy weight process**.

Question 8. How can you ensure all threads that started from main must end in order in which they started and also main should end in last? (Important)

Answer. Interviewers tend to know interviewees knowledge about Thread methods. So this is time to prove your point by answering correctly. We can use **join() method** to ensure all threads that started from main must end in order in which they started and also main should end in last.In other words **waits for this thread to die**. **Calling join() method internally calls join(0);**

DETAILED DESCRIPTION : Join() method - ensure all threads that started from main must end in order in which they started and also main should end in last. Types of join() method with programs- 10 salient features of join.

Question 9.What is difference between starting thread with run() and start() method? (Important)

Answer. This is quite interesting question, it might confuse you a bit and at time may make you think is there really any difference between starting thread with run() and start() method.
When you **call start()** method, **main thread internally calls run() method** to start newly created Thread, so **run() method is ultimately called by newly created thread**.
When you **call run()** method **main thread** rather than starting run() method with newly thread it start **run() method by itself**.

Question 10. What is significance of using Volatile keyword? (Important)

Answer. Java allows threads to **access shared variables**. As a rule, to ensure that **shared variables are consistently updated**, a thread should ensure that it has **exclusive use of such variables by obtaining a lock** that enforces mutual exclusion for those shared variables.

If a field is declared volatile, in that case the Java memory model ensures that all threads see a consistent value for the variable.

Few small questions>

Q. Can we have volatile methods in java?

A. **No**, volatile is only a keyword, can be used only with variables.

Q. Can we have synchronized variable in java?

A. **No**, synchronized can be used only with methods, i.e. in method declaration.

DETAILED DESCRIPTION : Volatile keyword in java- difference between synchronized and volatile with programs, 10 key points about volatile keyword, why volatile variables are not cached in memory

Question 11. Differences between synchronized and volatile keyword in Java? (Important)

Answer. Its very important question from interview perspective.

1. Volatile can be used as a **keyword** against the variable, we **cannot** use volatile against method declaration.

```
volatile void method1(){} //it's illegal, compilation error.
volatile int i; //legal
```

While synchronization can be used in method declaration or we can create synchronization blocks (In both cases thread acquires lock on object's monitor). Variables cannot be synchronized.

```
Synchronized method:
synchronized void method2(){} //legal

Synchronized block:
    void method2(){
       synchronized (this) {
           //code inside synchronized block.
           }
 }

Synchronized variable (illegal):
synchronized int i; //it's illegal, compilatiomn error.
```

2. Volatile does not acquire any lock on variable or object, but Synchronization acquires lock on method or block in which it is used.

3. Volatile variables are not cached, but variables used inside synchronized method or block are cached.

4. When volatile is used will never create deadlock in program, as volatile never obtains any kind of lock . But in case if synchronization is not done properly, we might end up creating dedlock in program.

5. Synchronization may cost us performance issues, as one thread might be waiting for another thread to release lock on object. But volatile is never expensive in terms of performance.

DETAILED DESCRIPTION : **Differences between synchronized and volatile keyword in detail with programs.**

Question 12. Can you again start Thread?

Answer. No, we cannot start Thread again, doing so will throw runtimeException java.lang.IllegalThreadStateException. The reason is once run() method is executed by Thread, it goes into **dead state**.
Let's take an example-
Thinking of starting thread again and calling start() method on it (which internally is going to call run() method) for us is some what like asking dead man to wake up and run. As, after completing his life person goes to **dead state**.

Question 13. What is race condition in multithreading and how can we solve it? (Important)

Answer. This is very important question, this forms the core of multi threading, you should be able to explain about race condition in detail. When more than one thread try to access same resource without synchronization causes race condition.
So we can solve race condition by using either synchronized block or synchronized method. When no two threads can access same resource at a time phenomenon is also called as **mutual exclusion**.

Few sub questions>

What if two threads try to **read** same resource without synchronization?
When two threads try to read on same resource without synchronization, **it's never going to create any problem**.

What if two threads try to **write** to same resource without synchronization?
When two threads try to **write** to same resource without synchronization, **it's going to create synchronization problems**.

Question 14. How threads communicate between each other?

Answer. This is very must know question for all the interviewees, you will most probably face this question in almost every time you go for interview.
Threads can communicate with each other by using **wait(), notify() and notifyAll()** methods.

Question 15. Why wait(), notify() and notifyAll() are in Object class and not in Thread class? (Important)

Answer.

1. **Every Object has a monitor**, acquiring that monitors allow thread to hold lock on object. But **Thread class does not have any monitors**.

2. wait(), notify() and notifyAll() are called on objects only > **When wait() method** is called on object by thread **it waits for another thread** on that object to **release object monitor by calling notify() or notifyAll()** method on that object.

   When notify() method is called on object by thread **it notifies all the threads** which are **waiting for that object monitor** that object monitor is available now.

So, this shows that wait(), notify() and notifyAll() are called on objects only.

Now, Straight forward question that comes to mind is how thread acquires object lock by acquiring object monitor? Let's try to understand this basic concept in detail?

3. Wait(), notify() and notifyAll() method being in Object class allows all the **threads created on that object** to **communicate** with other. [As multiple threads may exist on same object].

4. As **multiple threads exists on same object**. Only one thread can hold object monitor at a time. As a result thread can notify other threads of same object that lock is available now. But, thread having these methods does not make any sense because multiple threads exists on object its not other way around (i.e. multiple objects exists on thread).

5. Now let's discuss one **hypothetical** scenario, **what will happen if Thread class contains wait(), notify() and notifyAll() methods**?

   Having wait(), notify() and notifyAll() methods **means Thread class also must have their monitor**. Every thread having their monitor will create few problems -

   >**Thread communication problem.**

   >**Synchronization on object won't be possible**- Because object has monitor, one object can have multiple threads and thread hold lock on object by holding object monitor. But if each thread will have monitor, we won't have any way of achieving synchronization.

   >**Inconsistency in state of object** (because synchronization won't be possible).

Question 16. Is it important to acquire object lock before calling wait(), notify() and notifyAll()?

Answer. Yes, it's mandatory to acquire object lock before calling these methods on object. As discussed above **wait(), notify() and notifyAll() methods are always called from Synchronized block only**, and **as soon as thread enters synchronized block it acquires object lock** (by holding object monitor). If we call these methods without acquiring object lock i.e. from outside synchronize block then java.lang. IllegalMonitorStateException is thrown at runtime.
Wait() method needs to enclosed in try-catch block, because it throws compile time exception i.e. InterruptedException.

Question 17. How can you solve consumer producer problem by using wait() and notify() method? (Important)

Answer. Here come the time to answer **very very important question from interview perspective**. Interviewers tends to check how sound you are in threads inter communication. Because for solving this problem we got to **use synchronization blocks, wait() and notify() method very cautiously**. **If you misplace synchronization block or any of the method**, that **may cause your program to go horribly wrong**. So, before going into this question first i'll recommend you to understand how to use synchronized blocks, wait() and notify() methods.

Key points we need to ensure before programming :

>Producer will produce total of 10 products and cannot produce more than 2 products at a time until products are being consumed by consumer.

Example> when `sharedQueue's size` is 2, wait for consumer to consume (consumer will consume by calling remove(0) method on `sharedQueue` and reduce `sharedQueue's size).` As soon as size is less than 2, producer will start producing.

>Consumer can consume only when there are some products to consume.

Example> when `sharedQueue's size` is 0, wait for producer to produce (producer will produce by calling add() method on `sharedQueue` and increase `sharedQueue's size).` As soon as size is greater than 0, consumer will start consuming.

Explanation of **Logic** >

We will create sharedQueue that will be shared amongst Producer and Consumer. We will now start consumer and producer thread.
Note: it does not matter order in which threads are started (because rest of code has taken care of synchronization and key points mentioned above)

First we will start consumerThread >

```
consumerThread.start();
```

consumerThread will enter run method and call consume() method. There it will check for sharedQueue's size.
-if size is equal to 0 that means producer hasn't produced any product, wait for producer to produce by using below piece of code-

```
synchronized (sharedQueue) {
    while (sharedQueue.size() == 0) {
          sharedQueue.wait();
  }
}
```

-if size is greater than 0, consumer will start consuming by using below piece of code.

```
        synchronized (sharedQueue) {
           Thread.sleep((long)(Math.random() * 2000));
         System.out.println("consumed : "+ sharedQueue.remove(0));
         sharedQueue.notify();
        }
```

Than we will start producerThread >

```
producerThread.start();
```

producerThread will enter run method and call produce() method. There it will check for sharedQueue's size.
-if size is equal to 2 (i.e. maximum number of products which sharedQueue can hold at a time), wait for consumer to consume by using below piece of code-

```
synchronized (sharedQueue) {
    while (sharedQueue.size() == maxSize) { //maxsize is 2
      sharedQueue.wait();
    }
}
```

-if size is less than 2, producer will start producing by using below piece of code.

```
synchronized (sharedQueue) {
     System.out.println("Produced : " + i);
     sharedQueue.add(i);
     Thread.sleep((long)(Math.random() * 1000));
     sharedQueue.notify();
}
```

DETAILED DESCRIPTION **with program : Solve Consumer Producer problem by using wait() and notify() methods in multithreading.**

Another illustration with program : How to solve **Consumer Producer** problem by using **wait() and notify()** methods, where **consumer can consume only when production is over.**

Question 18. How to solve Consumer Producer problem without using wait() and notify() methods, where consumer can consume only when production is over.?

Answer. In this problem, producer will allow consumer to consume only when 10 products have been produced (i.e. when production is over).

We will approach by keeping one boolean variable **productionInProcess** and initially setting it to **true**, and later when production will be over we will set it to **false**.

DETAILED DESCRIPTION : How to solve **Consumer Producer** problem **without using wait() and notify()** methods, where **consumer can consume only when production is over.**

Question 19. How can you solve consumer producer pattern by using BlockingQueue? (Important)

Answer. Now it's time to gear up to face question which is most probably going to be followed up by previous question i.e. after how to solve consumer producer problem using wait() and notify() method. Generally you might wonder why interviewer's are so much interested in asking about solving consumer producer problem using BlockingQueue, answer is they want to know how strong knowledge you have about java concurrent Api's, this Api use consumer producer pattern in very optimized manner, BlockingQueue is designed is such a manner that it offer us the best performance.
BlockingQueue is a interface and we will use its **implementation class LinkedBlockingQueue.**

Key methods for solving consumer producer pattern are >

```
put(i);      //used by producer to put/produce in sharedQueue.
take(); //used by consumer to take/consume from sharedQueue.
```

Question 20. What is deadlock in multithreading? Write a program to form DeadLock in multi threading and also how to solve DeadLock situation. What measures you should take to avoid deadlock? (Important)

Answer. This is very important question from interview perspective. But, what makes this question important is it checks interviewees capability of **creating and detecting deadlock**. If you can write a code to

form deadlock, than I am sure you must be well capable in solving that deadlock as well. If not, later on this post we will learn how to solve deadlock as well.
First question comes to mind is, **what is deadlock in multi threading program?**
Deadlock is a situation where two threads are waiting for each other to release lock holded by them on resources.

But how deadlock could be formed :

Thread-1 acquires lock on String.class and then calls sleep() method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and **Thread-2 acquires lock on Object.class** then calls sleep() method and **now it waits for Thread-1 to release lock on String.class**.
Conclusion:
Now, **Thread-1 is waiting for Thread-2 to release lock on Object.class** and **Thread-2 is waiting for Thread-1 to release lock on String.class** and deadlock is formed.

Code called by Thread-1

```
public void run() {
          synchronized (String.class) {
                   Thread.sleep(100);
                 synchronized (Object.class) {
                 }
          }
}
```

Code called by Thread-2

```
   public void run() {
          synchronized (Object.class) {
                   Thread.sleep(100);
                 synchronized (String.class) {
                 }
          }
}
```

Here comes the **important** part, how above formed **deadlock** could be **solved** :

Thread-1 acquires lock on String.class and then calls sleep() method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and **Thread-2 tries to acquire lock on String.class** but lock is holded by Thread-1. Meanwhile, Thread-1 completes successfully. As Thread-1 has completed successfully it releases lock on String.class, Thread-2 can now acquire lock on String.class and complete successfully without any deadlock formation.
Conclusion: No deadlock is formed.

Code called by Thread-1

```
public void run() {
          synchronized (String.class) {
                   Thread.sleep(100);
                 synchronized (Object.class) {
                 }
          }
}
```

Code called by Thread-2

```
   public void run() {
          synchronized (String.class) {
                   Thread.sleep(100);
                 synchronized (Object.class) {
                 }
          }
}
```

Few important measures to avoid Deadlock >

1. **Lock specific member variables of class rather than locking whole class**: We must try to lock specific member variables of class rather than locking whole class.

2. **Use join() method:** If possible try to use join() method, although it may refrain us from taking full advantage of multithreading environment because threads will start and end sequentially, but it can be handy in avoiding deadlocks.

3. **If possible try avoid using nested synchronization blocks.**

Question 21. Have you ever generated thread dumps or analyzed Thread Dumps? (Important)

Answer. Answering this questions will show your in depth knowledge of Threads. Every experienced must know how to generate Thread Dumps.

VisualVM is most popular way to generate Thread Dump and is most widely used by developers. It's important to understand usage of VisualVM for in depth knowledge of VisualVM. I'll recommend every developer must understand this topic to become master in multi threading.
It helps us in analyzing threads performance, thread states, CPU consumed by threads, garbage collection and much more. For detailed information see **Generating and analyzing Thread Dumps using VisualVM - step by step detail to setup VisualVM with screenshots**

jstack is very easy way to generate Thread dump and is widely used by developers. I'll recommend every developer must understand this topic to become master in multi threading. For creating Thread dumps we **need not to download any jar or any extra software**. For detailed information see **Generating and analyzing Thread Dumps using JSATCK - step by step detail to setup JSTACK with screenshots**.

Question 22. What is life cycle of Thread, explain thread states? (Important)

Answer. **Thread states/ Thread life cycle is very basic question, before going deep into concepts we must understand Thread life cycle.**

Thread have following states >

- **New**
- **Runnable**
- **Running**
- **Waiting/blocked/sleeping**
- **Terminated (Dead)**

Thread states/ Thread life cycle in diagram >

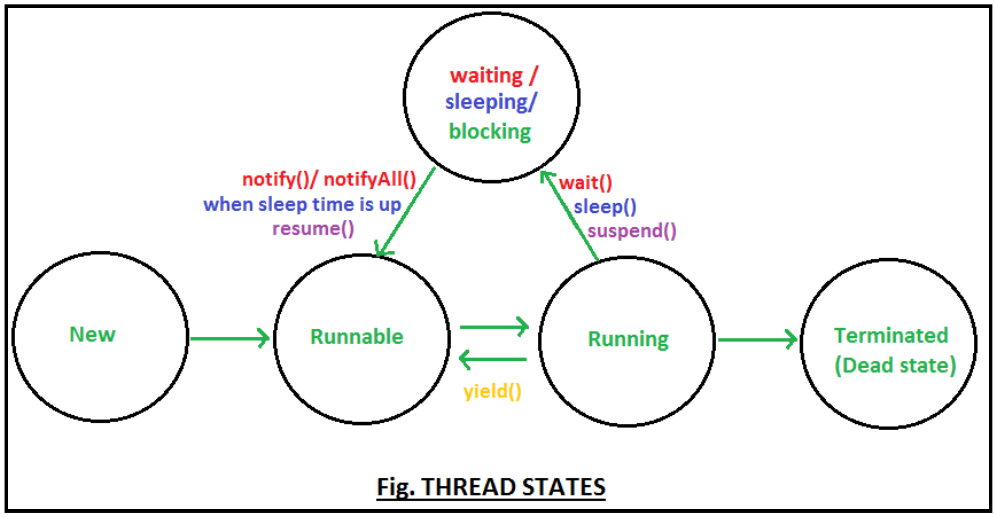


Fig. THREAD STATES

Thread states in detail >

New : **When instance of thread is created using new operator it is in new state**, but the start() method has not been invoked on the thread yet, thread is not eligible to run yet.

Runnable : **When start() method is called on thread it enters runnable state**.

Running : Thread scheduler selects thread to go from runnable to running state. In running state Thread starts executing by entering run() method.

Waiting/blocked/sleeping : In this state a thread is not eligible to run.
>Thread is still alive, but currently it's not eligible to run. In other words.

> How can Thread go from running to waiting state?
By calling **wait()** method thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.

> How can Thread go from running to sleeping state?
By calling **sleep()** method thread go from running to sleeping state. In sleeping state it will wait for sleep time to get over.

Terminated (Dead) : A thread is considered dead **when its run() method completes**.

You may like to have in depth knowledge of **Thread states/ Thread life cycle in java & explanation of thread methods which method puts thread from which state to which state.**

Question 23. Are you aware of preemptive scheduling and time slicing?

Answer. In **preemptive scheduling**, the **highest priority thread executes until** it enters into the **waiting or dead state**.
In time slicing, a **thread executes for a certain predefined time** and **then enters runnable pool**. Than thread can enter running state when selected by thread scheduler.

Question 24. What are daemon threads?

Answer. Daemon threads are low priority threads which **runs intermittently in background** for doing **garbage collection**.

12 Few salient features of daemon() threads>

- **Thread scheduler schedules these threads** only **when CPU is idle**.
- Daemon threads are **service oriented threads**, they **serves all other threads**.
- These threads are **created before user threads are created** and **die after all other user threads dies**.
- **Priority of daemon threads is always 1** (i.e. MIN_PRIORITY).
- **User created threads are non daemon threads**.
- **JVM can exit** when only daemon threads exist in system.
- we can use **isDaemon()** method to check whether thread is daemon thread or not.
- we can use **setDaemon(boolean on)** method to make any user method a daemon thread.
- If **setDaemon(boolean on)** is called on thread after calling start() method than IllegalThreadStateException is thrown.
- You may like to see how daemon threads work, for that you can use VisualVM or jStack. I have provided Thread dumps over there which shows daemon threads which were intermittently running in background.

Some of the daemon threads which intermittently run in background are >

```
"RMI TCP Connection(3)-10.175.2.71" daemon
"RMI TCP Connection(idle)" daemon
"RMI Scheduler(0)" daemon
"C2 CompilerThread1" daemon
"GC task thread#0 (ParallelGC)"
```

Question 25. Why suspend() and resume() methods are deprecated?

Answer. Suspend() method is **deadlock prone**. If the target thread holds a lock on object when it is suspended, no thread can lock this object until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, it results in **deadlock formation**.
These **deadlocks** are generally called **Frozen processes**.

Suspend() method puts thread from running to waiting state. And thread can go **from waiting to runnable** state **only when resume() method is called** on thread. It is deprecated method.

Resume() method is **only used with suspend()** method that's why it's also deprecated method.

Question 26. Why destroy() methods is deprecated?

Answer. This question is again going to check your in depth knowledge of thread methods i.e. destroy() method is **deadlock prone**. If the target thread holds a lock on object when it is destroyed, no thread can lock this object (Deadlock formed are similar to deadlock formed when suspend() and resume() methods are used improperly). It results in **deadlock formation**. These **deadlocks** are generally called **Frozen processes**.
Additionally you must know calling destroy() method on Threads throw runtimeException i.e. NoSuchMethodError. **Destroy() method puts thread from running to dead state.**

Question 27. As stop() method is deprecated, How can we terminate or stop infinitely running thread in java? (Important)

Answer. This is very interesting question where interviewees thread basics basic will be tested. Interviewers tend to know user's knowledge about main thread's and thread invoked by main thread.
We will try to address the problem by creating new thread which will run infinitely until certain condition is satisfied and will be called by main Thread.

1. Infinitely running thread can be stopped **using boolean variable.**
2. Infinitely running thread can be stopped **using interrupt() method.**

Let's understand Why stop() method is deprecated :

Stopping a thread with Thread.stop() causes it to release all of the monitors that it has locked. If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, which might lead to unpredictable behavior.

Question 28. what is significance of yield() method, what state does it put thread in?

yield() is a **native method** it's implementation in java 6 has been changed as compared to its implementation java 5. As method is native it's implementation is provided by JVM.
In java 5, yield() method **internally used to call sleep() method** giving all the other threads of same or higher priority to execute before yielded thread by leaving allocated CPU for time gap of 15 millisec.

But java 6, calling **yield() method gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor**. The **thread scheduler is free to ignore this hint**. So, sometimes even after using yield() method, you may not notice any difference in output.

salient features of yield() method >

- **Definition** : yield() **method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint**.
- **Thread state : when** yield() **method is called on thread it goes from running to runnable state**, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.
- **Waiting time :** yield() **method stops thread for unpredictable time.**
- **Static method :** yield() is a **static method**, hence calling Thread.yield() causes currently executing thread to yield.
- **Native method :** implementation of yield() method is provided by **JVM**.
  Let's see definition of yield() method as given in java.lang.Thread -

```
public static native void yield();
```

- **synchronized block : thread need not to to acquire object lock** before calling yield() method i.e. yield() method **can be called from outside synchronized block**.

Question 29.What is significance of sleep() method in detail, what state does it put thread in ?

sleep() is a **native method**, its implementation is provided by JVM.

10 salient features of sleep() method >

- **Definition** : sleep() **methods causes current thread to sleep for specified number of milliseconds** (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.
- **Thread state : when** sleep() **is called on thread it goes from running to waiting state** and can return to runnable state when sleep time is up.
- **Exception :** sleep() **method must catch or throw compile time exception** i.e. `InterruptedException`.
- **Waiting time :** sleep() **method have got few options.**
  1. **sleep(long millis) -** Causes the currently executing thread to sleep for the specified number of milliseconds

```
public static native void sleep(long millis) throws
InterruptedException;
```

  2. **sleep(long millis, int nanos) -** Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

```
public static native void sleep(long millis,int nanos) throws
InterruptedException;
```

- **static method :** sleep() is a static method, causes the currently executing thread to sleep for the specified number of milliseconds.

- **Native method :** implementation of sleep() method is provided by **JVM**.

  Let's see definition of yield() method as given in java.lang.Thread -

```
public static native void sleep(long millis) throws InterruptedException;
```

- **Belongs to which class :** sleep() **method belongs to java.lang.Thread** class.

- **synchronized block :** thread **need not to to acquire object lock** before calling sleep() method i.e. sleep() method **can be called from outside synchronized block**.

Question 30. Difference between wait() and sleep() ? (Important)

Answer.

- **Should be called from synchronized block :** wait() **method is always called from synchronized block** i.e. wait() method needs to lock object monitor before object on which it is called. But sleep() **method can be called from outside synchronized block** i.e. sleep() method doesn't need any object monitor.

- **IllegalMonitorStateException : if** wait() **method is called without acquiring object lock** than IllegalMonitorStateException is thrown at runtime, but sleep() method **never throws such exception**.

- **Belongs to which class : wait() method belongs to java.lang.Object** class but **sleep() method belongs to java.lang.Thread** class.

- **Called on object or thread :** wait() **method is called on objects** but sleep() **method is called on Threads** not objects**.**

- **Thread state : when** wait() **method is called on object, thread that holded object's monitor goes from running to waiting state** and can **return to runnable state only when notify() or notifyAll() method is called on that object**. And later thread scheduler schedules that thread to go from from runnable to running state.

  when sleep() **is called on thread it goes from running to waiting state** and can **return to runnable state when sleep time is up.**

- **When called from synchronized block :** when wait() method is called **thread leaves the object lock**. But sleep() method **when called from synchronized block or method thread doesn't leaves object lock.**

Question 31. Differences and similarities between yield() and sleep() ?

Answer.

Differences yield() and sleep() :

- **Definition** : yield() **method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint**. sleep() **methods causes current thread to sleep for specified number of milliseconds** (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.

- **Thread state : when** sleep() **is called on thread it goes from running to waiting state** and can return to runnable state when sleep time is up. **when** yield() **method is called on thread it goes from running to runnable state**, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.

- **Exception :** yield() **method need not to catch or throw any exception.** But sleep() **method must catch or throw compile time exception** i.e. `InterruptedException`**.**

- **Waiting time :** yield() **method stops thread for unpredictable time, that depends on thread scheduler.** But sleep() **method have got few options.**
  1. **sleep(long millis) -** Causes the currently executing thread to sleep for the specified number of milliseconds
  2. **sleep(long millis, int nanos) -** Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

similarity between yield() and sleep():

> yield() and sleep() method **belongs to java.lang.Thread** class.

> yield() and sleep() method can be **called from outside synchronized block**.

> yield() and sleep() method are **called on Threads not objects**.

Question 32. Mention some guidelines to write thread safe code, most important point we must take care of in multithreading programs?

Answer. In multithreading environment it's important very important to write thread safe code, thread unsafe code can cause a major threat to your application. I have posted many articles regarding thread safety. So overall this will be revision of what we have learned so far i.e. writing thread safe healthy code and avoiding any kind of deadlocks.

1. If method is exposed in multithreading environment and it's not synchronized (thread unsafe) than it might lead us to race condition, we must try to use synchronized block and **synchronized methods**. Multiple threads may exist on same object but only one thread of that object can enter **synchronized method** at a time, though threads on different object can enter same method at same time.

2. Even static variables are not thread safe, they are used in static methods and if static methods are not synchronized then thread on same or different object can enter method concurrently. Multiple threads may exist on same or different objects of class but only one thread can enter **static synchronized method** at a time, we must consider making static methods as synchronized.

3. If possible, try to use **volatile** variables. If a field is declared volatile all threads see a consistent value for the variable. Volatile variables at times can be used as alternate to synchronized methods as well.

4. **Final variables** are thread safe because once assigned some reference of object they cannot point to reference of other object.

   s is pointing to String object.

```
public class MyClass {
final String s=new String("a");
   void method(){
          s="b"; //compilation error, s cannot point to new reference.
   }
}
```

   If final is holding some primitive value it cannot point to other value.

```
public class MyClass {
final int i=0;
   void method(){
          i=0;  //compilation error, i cannot point to new value.
   }
}
```

5. Usage of **local variables** : If possible try to use local variables, local variables are thread safe, because every thread has its own **stack**, i.e. every thread has its own local variables and its pushes all the local variables on stack.

```
public class MyClass {
  void method(){
   int i=0; //Local variable, is thread safe.
  }
}
```

6. We must avoid using **deadlock prone** deprecated thread methods such as destroy(), stop(), suspend() and resume().

7. Using thread safe **collections** : Rather than using ArrayList we must Vector and in place of using HashMap we must use ConcurrentHashMap or HashTable.

8. We must use VisualVM or jstack to detect problems such as deadlocks and time taken by threads to complete in multi threading programs.

9. Using *ThreadLocal* : ThreadLocal is a class which provides thread-local variables. Every thread has its own ThreadLocal value that makes ThreadLocal value threadsafe as well.

10. Rather than StringBuffer try using **immutable classes** such as String. Any change to String produces new String.

Question 33. How thread can enter waiting, sleeping and blocked state and how can they go to runnable state ?

Answer. This is very prominently asked question in interview which will test your knowledge about thread states. And it's very important for developers to have in depth knowledge of this thread state transition. I will try to explain this thread state transition by framing few sub questions. I hope reading sub questions will be quite interesting.

> How can Thread go from running to waiting state ?

By calling **wait()** method thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.

> How can Thread return from waiting to runnable state ?

Once **notify() or notifyAll()** method is called object monitor/lock becomes available and thread can again return to runnable state.

> How can Thread go from running to sleeping state ?

By calling **sleep()** method thread go from running to sleeping state. In sleeping state it will wait for sleep time to get over.

> How can Thread return from sleeping to runnable state ?

Once specified **sleep time is up** thread can again return to runnable state.

Suspend() method can be used to put thread in waiting state and **resume()** method is the only way which could put thread in runnable state.

Thread also may go from running to waiting state if it is waiting for some I/O operation to take place. Once input is available thread may return to running state.

>When threads are in running state, **yield()** method can make thread to go in Runnable state.

Question 34. Difference between notify() and notifyAll() methods, can you write a code to prove your point?

Answer. Goodness. Theoretically you must have heard or you must be aware of differences between notify() and notifyAll().But have you created program to achieve it? If not let's do it.

First, I will like give you a brief description of what notify() and notifyAll() methods do.

notify() - 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 random and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the wait methods.
The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.

```
public final native void notify();
```

notifyAll() - Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.
The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object.

```
public final native void notifyAll();
```

Now it's time to write down a program to prove the point.

Question 35. Does thread leaves object lock when sleep() method is called?

Answer. When sleep() method is called Thread does not leaves object lock and goes from running to waiting state. Thread waits for sleep time to over and once sleep time is up it goes from waiting to runnable state.

Question 36. Does thread leaves object lock when wait() method is called?

Answer. When wait() method is called Thread leaves the object lock and goes from running to waiting state. Thread waits for other threads on same object to call notify() or notifyAll() and once any of notify() or notifyAll() is called it goes from waiting to runnable state and again acquires object lock.

Question 37. What will happen if we don't override run method?

Answer. This question will test your basic knowledge how start and run methods work internally in Thread Api.

When we call start() method on thread, **it internally calls run() method** with newly created thread. **So, if we don't override run() method newly created thread won't be called and nothing will happen**.

```
class MyThread extends Thread {
    //don't override run() method
}

public class DontOverrideRun {
    public static void main(String[] args) {
           System.out.println("main has started.");
           MyThread thread1=new MyThread();
           thread1.start();
           System.out.println("main has ended.");
    }

}

/*OUTPUT

main has started.
main has ended.
```

```
*/
```

As we saw in output, we didn't override run() method that's why on calling start() method nothing happened.

Question 38. What will happen if we override start method?

Answer. This question will again test your basic core java knowledge how overriding works at runtime, what what will be called at runtime and how start and run methods work internally in Thread Api.

When we call start() method on thread, **it internally calls run()** method with newly created thread. **So, if we override start() method, run() method will not be called** until we write code for calling run() method.

```
class MyThread extends Thread {

    @Override
    public void run() {
           System.out.println("in run() method");
    }

    @Override
    public void start(){
           System.out.println("In start() method");
    }

}

public class OverrideStartMethod {
    public static void main(String[] args) {
           System.out.println("main has started.");

           MyThread thread1=new MyThread();
           thread1.start();

           System.out.println("main has ended.");
    }

}

/*OUTPUT

main has started.
In start() method
main has ended.

*/
```

If we note output. we have overridden start method and didn't called run() method from it, so, run() method wasn't call.

Question 39. Can we acquire lock on class? What are ways in which you can acquire lock on class?

Answer. **Yes**, we can acquire lock on class's class object in 2 ways to acquire lock on class.

Thread can acquire lock on class's class object by-

1. Entering **synchronized block or**

Let's say there is one class `MyClass`. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized `MyClass`

```
synchronized (MyClass.class) {
   //thread has acquired lock on MyClass's class object.
}
```

2. by entering **static synchronized methods.**

```
public static synchronized void method1() {
   //thread has acquired lock on MyRunnable's class object.
}
```

As soon as thread entered Synchronization method, thread acquired lock on class's class object.
Thread will leave lock when it exits static synchronized method.

Question 40. Difference between object lock and class lock?

Answer. It is very important question from multithreading point of view. We must understand difference between object lock and class lock to answer interview, ocjp answers correctly.

Object lock	**Class lock**
Thread can acquire object lock by- 1. Entering **synchronized block or**	Thread can acquire lock on class's class object by-

2. by entering **synchronized methods.**	1. Entering **synchronized block or** 2. by entering **static synchronized methods.**
Multiple threads may exist on same object but only one thread of that object can enter **synchronized method** at a time. Threads on different object can enter same method at same time.	Multiple threads may exist on same or different objects of class but only one thread can enter **static synchronized method** at a time.
Multiple objects of class may exist and every object has it's own lock.	**Multiple objects of class may exist but there is always one class's class object lock available**.

First let's acquire object lock by entering **synchronized block**.
 	
Example- Let's say there is one class `MyClass` and we have created it's object and reference to that object is `myClass`. Now we can create synchronization block, and parameter passed with synchronization tells which object has to be synchronized. In below code, we have synchronized object reference by `myClass`.
`MyClass myClass=new Myclass();`
`synchronized (myClass) {`
`}`
As soon thread entered Synchronization block, thread acquired object lock on object referenced by `myClass` (by acquiring object's monitor.)
Thread will leave lock when it exits synchronized block.
	First let's acquire lock on class's class object by entering **synchronized block.**
	
	Example- Let's say there is one class `MyClass`. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized `MyClass`
	`synchronized (MyClass.class) {`
	`}`
	
	As soon as thread entered Synchronization block, thread acquired `MyClass's class object`. Thread will leave lock when it exits synchronized block.

`public synchronized void method1() {`
`}`
 	
As soon as thread entered **Synchronization method**, thread acquired object lock.
Thread will leave lock when it exits synchronized method.
	`public static synchronized void method1() {}`
	As soon as thread entered **static Synchronization method**, thread acquired lock on class's class object.
	Thread will leave lock when it exits synchronized method.

Let's me give you some tricky situation based question,

Question 41. Suppose you have **2 threads (Thread-1 and Thread-2) on same object**. **Thread-1** is in **synchronized method1(),** can **Thread-2** enter **synchronized method2()** at same time?

Answer. No, here when Thread-1 is in **synchronized method1()** it must be **holding lock on object's monitor** and will release lock on object's monitor only when it exits **synchronized method1()**. So, Thread-2 will have to wait for Thread-1 to release lock on object's monitor so that it could enter **synchronized method2()**.

Likewise, Thread-2 even cannot enter **synchronized method1()** which is being executed by Thread-1. Thread-2 will have to wait for Thread-1 to release lock on object's monitor so that it could enter **synchronized method1()**. Now, let's see a program to prove our point.

Question 42. Suppose you have **2 threads (Thread-1 and Thread-2) on same object**. **Thread-1** is in **static synchronized method1(),** can **Thread-2** enter **static synchronized method2()** at same time?

Answer. No, here when Thread-1 is in **static synchronized method1()** it must be **holding lock on class's class object** and will release lock on class's class object only when it exits **static synchronized method1()**. So, Thread-2 will have to wait for Thread-1 to release lock on class's class object so that it could enter **static synchronized method2()**.

Likewise, Thread-2 even cannot enter **static synchronized method1()** which is being executed by Thread-1. Thread-2 will have to wait for Thread-1 to release lock on class's class object so that it could enter **static synchronized method1()**. Now, let's see a program to prove our point.

Question 43. Suppose you have **2 threads (Thread-1 and Thread-2) on same object**. **Thread-1** is in **synchronized method1(),** can **Thread-2** enter **static synchronized method2()** at same time?

Answer. Yes, here when Thread-1 is in **synchronized method1()** it must be **holding lock on object's monitor** and Thread-2 can enter **static synchronized method2()** by acquiring lock on class's class object. Now, let's see a program to prove our point.

Question 44. Suppose you have thread and it is in **synchronized method** and now can thread **enter other synchronized method** from that method?

Answer. Yes, here when thread is in **synchronized method** it must be **holding lock on object's monitor** and **using that lock** thread can **enter other synchronized method**. Now, let's see a program to prove our point.

Question 45. Suppose you have thread and it is in **static synchronized method** and now can thread **enter other static synchronized method** from that method?

Answer. Yes, here when thread is in **static synchronized method** it must be **holding lock on class's class object** and **using that lock** thread can **enter other static synchronized method**. Now, let's see a program to prove our point.

Question 46. Suppose you have thread and it is in **static synchronized method** and now can thread **enter other non static synchronized method** from that method?

Answer. Yes, here when thread is in **static synchronized method** it must be **holding lock on class's class object** and when it **enters synchronized method** it will **hold lock on object's monitor as well**.
So, now thread holds 2 locks (it's also called nested synchronization)-
>first one on **class's class object.**
>second one on **object's monitor** (This lock will be released when thread exits non static method)**.** Now, let's see a program to prove our point.

Question 47. Suppose you have thread and it is in **synchronized method** and now can thread **enter other static synchronized method** from that method?

Answer. Yes, here when thread is in synchronized method it must be holding **lock on object's monitor** and when it enters static synchronized method it will hold lock on class's class object as well.
So, now thread holds 2 locks (it's also called nested synchronization)-
>first one on **object's monitor.**
>second one on **class's class object.**(This lock will be released when thread exits static method)**.** Now, let's see a program to prove our point.

Question 48. Suppose you have **2 threads (Thread-1 on object1 and Thread-2 on object2)**. **Thread-1** is in **synchronized method1(),** can **Thread-2** enter **synchronized method2()** at same time?

Answer. Yes, here when Thread-1 is in **synchronized method1()** it must be **holding lock on object1's monitor**. Thread-2 will acquire lock on **object2's monitor** and enter **synchronized method2()**.

Likewise, Thread-2 even enter **synchronized method1()** as well which is being executed by Thread-1 (because threads are created on different objects). Now, let's see a program to prove our point.

Question 49. Suppose you have **2 threads (Thread-1 on object1 and Thread-2 on object2)**. **Thread-1** is in **static synchronized method1(),** can **Thread-2** enter **static synchronized method2()** at same time?

Answer. No, it might confuse you a bit that threads are created on different objects. But, not to forgot that **multiple objects may exist but there is always one class's class object lock available**.
Here, when Thread-1 is in **static synchronized method1()** it must be **holding lock on class class's object** and will release lock on class's class object only when it exits **static synchronized method1()**. So, Thread-2 will have to wait for Thread-1 to release lock on class's class object so that it could enter **static synchronized method2()**.

Likewise, Thread-2 even cannot enter **static synchronized method1()** which is being executed by Thread-1. Thread-2 will have to wait for Thread-1 to release lock on class's class object so that it could enter **static synchronized method1()**. Now, let's see a program to prove our point.

Question 50. Difference between wait() and wait(long timeout), What are thread states when these method are called?

Answer.

wait()	**wait(long timeout)**
When wait() method is called on object, it causes causes the current	**wait(long timeout) -** Causes the current thread to wait until either another thread invokes the notify() or

thread to wait until another thread invokes the notify() or notifyAll() method for this object.	notifyAll() methods for this object, or a specified timeout time has elapsed.
When wait() is called on object - Thread enters from **running to waiting state**. **It waits for some other thread to call notify so that it could enter runnable state**.	**When wait(1000) is called** on object - Thread enters from **running to waiting state**. Than **even if notify() or notifyAll() is not called after timeout time has elapsed thread will go from waiting to runnable state.**

Question 51. How can you implement your own Thread Pool in java?

Answer.

What is ThreadPool?

ThreadPool is a pool of threads which **reuses a fixed number of threads** to execute tasks.
At any point, **at most nThreads threads will be active processing tasks**. **If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available**.
ThreadPool implementation internally uses LinkedBlockingQueue for adding and removing tasks.
In this post i will be using LinkedBlockingQueue provide by java Api, you can refer this post for implementing ThreadPool using custom LinkedBlockingQueue.

Need/Advantage of ThreadPool?

Instead of creating new thread every time for executing tasks, we can create **ThreadPool** which **reuses a fixed number of threads for executing tasks**.
As threads are reused, performance of our application improves drastically.

How ThreadPool works?

We will instantiate ThreadPool, in ThreadPool's **constructor** nThreads number of threads are created and started.

```
ThreadPool threadPool=new ThreadPool(2);
```

Here 2 threads will be created and started in ThreadPool.

Then, threads will enter **run()** method of **ThreadPoolsThread** class and will call take() method on `taskQueue`.

- If tasks are available thread will execute task by entering run() method of task (As tasks executed always implements Runnable).

```
public void run() {
 . . .
    while (true) {
        . . .
        Runnable runnable = taskQueue.take();
        runnable.run();
        . . .
    }
 . . .
}
```

- Else waits for tasks to become available.

When tasks are added?

When execute() method of **ThreadPool** is called, it internally calls put() method on `taskQueue` to add tasks.

```
taskQueue.put(task);
```

Once tasks are available all waiting threads are notified that task is available.

More detail on how to Implement Thread pool in java.

Question 52. What is significance of using ThreadLocal?

Answer. This question will test your command in multi threading, can you really create some perfect multithreading application or not. ThreadLocal is a class which provides thread-local variables.

What is ThreadLocal ?

ThreadLocal is a class which provides thread-local variables. Every thread has its own ThreadLocal value that makes ThreadLocal value threadsafe as well.

For how long Thread holds ThreadLocal value?

Thread holds ThreadLocal value till it hasn't entered dead state.

Can one thread see other thread's ThreadLocal value?

No, thread can see only it's ThreadLocal value.

Are ThreadLocal variables thread safe. Why?

Yes, ThreadLocal variables are thread safe. As every thread has its own ThreadLocal value and one thread can't see other threads ThreadLocal value.

Application of ThreadLocal?

1. ThreadLocal are **used by many web frameworks** for maintaining some context (may be session or request) related value.
    - In any **single threaded application**, same thread is assigned for every request made to same action, so ThreadLocal values will be available in next request as well.
    - In **multi threaded application**, different thread is assigned for every request made to same action, so ThreadLocal values will be different for every request.

2. When threads have started at different time they might like to store time at which they have started. **So, thread's start time can be stored in ThreadLocal.**

Creating ThreadLocal >

```
private ThreadLocal<String> threadLocal =   new ThreadLocal<String>();
```

We will create instance of ThreadLocal. ThreadLocal is a generic class, i will be using String to demonstrate threadLocal.
All threads will see same instance of ThreadLocal, **but a thread will be able to see value which was set by it only**.

How thread set value of ThreadLocal >

```
threadLocal.set( new Date().toString());
```

Thread set value of ThreadLocal by calling set("") method on threadLocal.

How thread get value of ThreadLocal >

```
threadLocal.get()
```

Thread get value of ThreadLocal by calling get() method on threadLocal.

See here for detailed explanation of threadLocal.

Question 53. What is busy spin?

Answer.

What is busy spin?

When one thread loops continuously waiting for another thread to signal.

Performance point of view - Busy spin is **very bad** from performance point of view, because one thread keeps on looping continuously (and consumes CPU) waiting for another thread to signal.

Solution to busy spin -

We must use sleep() or wait() and notify() method. Using wait() is better option.

Why using wait() and notify() is much better option to solve busy spin?

Because in case when we use sleep() method, thread will wake up again and again after specified sleep time until boolean variable is true. But, in case of wait() thread will wake up only when when notified by calling notify() or notifyAll(), hence end up consuming CPU in best possible manner.

Program - Consumer Producer problem with busy spin >

Consumer thread continuously execute (**busy spin**) in while loop till `productionInProcess` is true. Once producer thread has ended it will make boolean variable `productionInProcess` false and **busy spin** will be over.

```
while(productionInProcess){
   System.out.println("BUSY SPIN - Consumer waiting for production to get
over");
}
```

See here for Busy spin in detail.

Question 54. Can a constructor be synchronized?

Answer. No, constructor cannot be synchronized. Because constructor is used for instantiating object, when we are in constructor object is under creation. So, until object is not instantiated it does not need any synchronization.

Enclosing constructor in synchronized block will generate compilation error.

Using synchronized in **constructor definition** will also show compilation error.

COMPILATION ERROR = Illegal modifier for the constructor in type ConstructorSynchronizeTest; only public, protected & private are permitted

Though we can use synchronized block inside constructor.

Read More about : Constructor in java cannot be synchronized

Question 55. Can you find whether thread holds lock on object or not?

Answer. holdsLock(object) method can be used to find out whether current thread holds the lock on monitor of specified object.
holdsLock(object) method returns true if the current thread holds the lock on monitor of specified object.

Question 56. What do you mean by thread starvation?

Answer. When thread does not enough CPU for its execution **Thread starvation happens.**

Thread starvation may happen in following scenarios >

- Low priority threads gets less CPU (time for execution) as compared to high priority threads. **Lower priority thread** may **starve** away waiting to get enough CPU to perform calculations.
- In deadlock two threads waits for each other to release lock holded by them on resources. There both **Threads starves away to get CPU.**
- Thread might be waiting indefinitely for lock on object's monitor (by calling wait() method), because no other thread is calling notify()/notifAll() method on object. In that case, **Thread starves** away to get CPU.
- Thread might be waiting indefinitely for lock on object's monitor (by calling wait() method), but notify() may be repeatedly awakening some other threads. In that case also **Thread starves** away to get CPU.

Question 57. What is addShutdownHook method in java?

Answer. addShutdownHook method in java >

- addShutdownHook method **registers a new virtual-machine shutdown hook**.
- A shutdown hook is a **initialized but unstarted thread**.
- When **JVM starts its shutdown** it will **start all registered shutdown hooks** in some unspecified order and let them run concurrently.

When JVM (Java virtual machine) shuts down >

- When the last non-daemon thread finishes, or
- when the System.exit is called.

Once JVM's shutdown has begun **new shutdown hook cannot be registered** neither **previously-registered hook can be de-registered**. Any attempt made to do any of these operations causes an IllegalStateException.

For more detail with program read : Threads addShutdownHook method in java

Question 58. How you can handle uncaught runtime exception generated in run method?

Answer. We can use setDefaultUncaughtExceptionHandler method which can handle uncaught unchecked(runtime) exception generated in run() method.

What is setDefaultUncaughtExceptionHandler method?

setDefaultUncaughtExceptionHandler method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.

setDefaultUncaughtExceptionHandler method features >

- **setDefaultUncaughtExceptionHandler** method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.
- **setDefaultUncaughtExceptionHandler** is a static method method, so we can directly call `Thread.setDefaultUncaughtExceptionHandler` to set the default handler to handle uncaught unchecked(runtime) exception.
- It avoids abrupt termination of thread caused by uncaught runtime exceptions.

Defining setDefaultUncaughtExceptionHandler method >

```
Thread.setDefaultUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler(){
 public void uncaughtException(Thread thread, Throwable throwable) {
  System.out.println(thread.getName() + " has thrown " + throwable);
 }
});
```

*For more detail read : Program to demonstrate **setDefaultUncaughtExceptionHandler** method.*

Question 59. What is ThreadGroup in java, What is default priority of newly created threadGroup, mention some important ThreadGroup methods ?

Answer. When program starts **JVM creates a ThreadGroup** named ***main***. Unless specified, all newly created threads become members of the ***main*** thread group.

ThreadGroup is initialized with default priority of 10.

*ThreadGroup **important methods** >*

- `getName()`
  - name of ThreadGroup.
- `activeGroupCount()`
  - count of active groups in ThreadGroup.
- `activeCount()`
  - count of active threads in ThreadGroup.
- `list()`
  - list() method has prints ThreadGroups information
- `getMaxPriority()`
  - Method returns the maximum priority of ThreadGroup.
- `setMaxPriority(int pri)`
  - Sets the maximum priority of ThreadGroup.

Read more about ThreadGroup in java.

Question 60. What are thread priorities?

Answer.

Thread Priority range is from 1 to 10.

Where **1 is minimum priority** and **10 is maximum priority.**

Thread class provides variables of **final static int** type for setting thread priority.

```
/* The minimum priority that a thread can have. */
public final static int MIN_PRIORITY = 1;

/* The default priority that is assigned to a thread. */
public final static int NORM_PRIORITY = 5;

/* The maximum priority that a thread can have. */
public final static int MAX_PRIORITY = 10;
```

Thread with **MAX_PRIORITY is likely to get more CPU** as compared to low priority threads. But **occasionally low priority thread might get more CPU**. Because thread scheduler schedules thread on discretion of implementation and thread behaviour is totally unpredictable.

Thread with **MIN_PRIORITY is likely to get less CPU** as compared to high priority threads. But **occasionally high priority thread might less CPU**. Because thread scheduler schedules thread on discretion of implementation and thread behaviour is totally unpredictable.

setPriority() **method is used for Changing the priority of thread.**

getPriority() **method returns the thread's priority.**