

Thread concurrency - Top 50 interview questions and answer in java for fresher and experienced (detailed explanation with programs)

You are here : [Home](#) / [Core Java Tutorials](#) / [Java Interview Questions and answers](#) / [Thread Concurrency Tutorial in java](#)



Interviewers always have great focus on testing interviewees command on thread concurrency, so it's time to discuss thread concurrency in detail. So, I have tried to cover maximum number of the possible question which could be framed on thread concurrency. For comprehensive understanding, i'll recommend you refer thread concurrency tutorial and [THREADS - Top 80 interview questions and answers](#). Every **experienced developer** must be well versed in thread concurrency.

Thread concurrency interview Question 1. What is executor framework in java?

Answer. This is very important question to start your interview with. [Executor and ExecutorService](#) are used for following purposes >

- creating thread in java,
- starting threads in java,
- managing whole [life cycle of Threads](#) in java.

Executor creates [pool of threads](#) and manages life cycle of all threads in it.

In Executor framework, **Executor** interface and **ExecutorService** class are most prominently used in java.

[Executor](#) interface defines very important execute() method which executes command in java.

[ExecutorService](#) interface extends **Executor** interface.

An Executor interface provides following type of methods >

- methods for managing termination and
- methods that can produce a Future for tracking progress of tasks in java.

An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

For more information read [Executor and ExecutorService framework in java](#).

Thread concurrency interview Question 2. What are differences between execute() and submit() method of executor framework in java?

Answer. This is basic thread concurrency interview question, beforehand you must know about [Executor Service Framework](#).

execute() method	submit() method
execute() method is defined in Executor interface in java.	submit() method is defined in ExecutorService interface in java.
It can be used for executing runnable task in java in java .	It can be used for executing runnable task or callable task , submitted callable returns future and Future's get method will return the task's result in java.
Signature of execute method is > void execute(Runnable task)	submit method has 3 forms > <T> Future<T> submit(Callable<T> task) Submits a callable task for execution.

	<p>Method returns a Future which represents pending results of the task.</p> <p>Once task is completed Future's get method will return the task's result.</p> <p><T> Future<T> submit(Runnable task, T result)</p> <p>Submits a Runnable task for execution.</p> <p>Method returns a Future which represents that task. Once task is completed Future's get method will return result.</p> <p>Future<?> submit(Runnable task)</p> <p>Submits a Runnable task for execution.</p> <p>Method returns a Future which represents that task. Once task is completed Future's get method will return null.</p>
--	---

Thread concurrency interview Question 3.What is Semaphore in java ??

Answer. This is very important thread concurrency interview question for freshers and experienced. A [semaphore](#) controls access to a shared resource by using permits in java.

- If permits are greater than zero, then semaphore **allow access to shared resource**.
- If permits are zero or less than zero, then semaphore **does not allow access to shared resource in java**.

These permits are sort of counters, which allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore in java.

Semaphore has 2 constructors >

- **Semaphore(int permits)**

permits is the initial number of permits available.

This value can be negative, in which case releases must occur before any acquires will be granted, **permits** is number of threads that can access shared resource at a time.

If **permits** is 1, then only one threads that can access shared resource at a time in java.

- **Semaphore(int permits, boolean fair)**

permits is the initial number of permits available.

This value can be negative, in which case releases must occur before any acquires will be granted.

By setting **fair** to **true**, we ensure that **waiting threads are granted a permit in the order in which they requested access**.

Semaphore's acquire() method has 2 forms :

- **void acquire()** throws **InterruptedException**

Acquires a permit if one is available and **reduces the number of available permits by 1**.

If no permit is available then the current thread becomes dormant until

- >some other thread calls release() method on this semaphore or,
- >some other thread interrupts the current thread.

- **void acquire(int permits)** throws **InterruptedException**

Acquires **permits** number of permits if available and **reduces the number of available permits by permits**.

If **permits** number of permits are not available then the current thread becomes dormant until one of the following things happens -

- >some other thread calls release() method on this semaphore and available permits become equal to **permits** or,
- >some other thread interrupts the current thread.

Semaphore's release() method has 2 forms in java :

- **void release()**

Releases a permit and **increases the number of available permits by 1.**

For releasing lock by calling release() method it's not mandatory that thread must have acquired permit by calling acquire() method in java.

- **void release(int permits)**

Releases **permits** number of permits and **increases the number of available permits by permits.**

For releasing lock by calling release(int *permits*) method it's not mandatory that thread must have acquired permit by calling acquire()/acquire(int permit) method in java.

Read more about [Semaphore in java](#).

Thread concurrency interview Question 4. How can you implement Producer Consumer pattern using Semaphore in java?

Answer. This is tricky thread concurrency interview question for even experienced guys. [Semaphore](#) on producer is created with permit =1. So, that producer can get the permit to produce. Semaphore on consumer is created with permit =0. So, that consumer could wait for permit to consume. [because initially producer hasn't produced any product]

Producer gets permit by calling semaphoreProducer.acquire() and starts producing, after producing it calls semaphoreConsumer.release(). So, that consumer could get the permit to consume.

```
semaphoreProducer.acquire();
System.out.println("Produced : "+i);
semaphoreConsumer.release();
```

Consumer gets permit by calling semaphoreConsumer.acquire() and starts consuming, after consuming it calls semaphoreProducer.release(). So, that producer could get the permit to produce.

```
semaphoreConsumer.acquire();
System.out.println("Consumed : "+i);
semaphoreProducer.release();
```

Read more on [Semaphore used for implementing Producer Consumer pattern](#).

Thread concurrency interview Question 5. How can you implement your own Semaphore?

Answer. Experienced developers must be able to answer this thread concurrency interview question. [Implementation of custom/own Semaphore in java](#).

Thread concurrency interview Question 6. What is significance of atomic classes in java 7?

Answer. Another important and basic thread concurrency interview question for freshers. Java provides some classes in [java.util.concurrent.atomic](#) which offers an alternative to the other [synchronization](#) in java.

Classes found in java.util.concurrent.atomic are >

- AtomicInteger,
- AtomicLong, and
- AtomicBoolean.

Methods provided by these classes >

- get(),
- set(),
- getAndSet(),
- compareAndSet(), and
- decrementAndGet().

In [multithreading](#) environment we can use these classes without any explicit synchronization, as all these classes are [thread safe](#) in java.

For more information on atomic read [Atomic operations in java](#).

Thread concurrency interview Question 7. What are Future and Callable? How are they related in java?

Answer. This is **very very important** thread concurrency interview question. They are widely used in thread concurrency.

[Future<V>](#) interface provides methods >

- for **returning result** of computation, wait until computation is not completed and
- for **cancelling** the computation in between.

Future Methods >

V get() method returns the result of computation, method waits for computation to complete.

cancel method cancels the task.

[Callable<V>](#) interface provides method for computing a result and returning that computed result or throws an exception if unable to do so

Any class implementing Callable interface must override *call()* method for computing a result.

Method returns computed result or throws an exception if unable to do so in java.

what type of results Callable's call() method can return in java?

The Callable<V> is a generic interface, so its call method can return generic result specified by *V*.

How Callable and Future are related?

If you submit a Callable object to an Executor returned object is of Future type.

```
Future<Double> futureDouble=executor.submit(new SquareDoubleCallable(2.2));
```

where, SquareDoubleCallable is a class which implements Callable.

This Future object can check the status of a Callable call's method and wait until Callable's call() method is not completed.

For more information read [Executor and ExecutorService framework in java](#).

Thread concurrency interview Question 8. Similarity and differences between java.util.concurrent.Callable and java.lang.Runnable in java?

Answer. This is basic thread concurrency interview question.

Similarity between java.util.concurrent.Callable and java.lang.Runnable in java?

Instances of class which implements callable are executed by another thread.

Difference between java.util.concurrent.Callable and java.lang.Runnable in java?

Class implementing Callable interface must override call() method. call() method returns computed result or throws an exception if unable to do so.

Class implementing Runnable interface must override run() method.

A Runnable does not return a result and can neither throw a checked exception in java.

Thread concurrency interview Question 9. What is CountdownLatch in java?

Answer. This is very important thread concurrency interview question. Fresher and experienced both be well versed with this. There might be situation where we might like our thread to wait until one or more threads completes certain operation in java.

A [CountDownLatch](#) is initialized with a given *count* .

count specifies the number of events that must occur before latch is released.

Every time an event happens *count* is reduced by 1. Once count reaches 0 latch is released.

CountDownLatch's constructor >

- **CountDownLatch(int count)**

CountDownLatch is initialized with given **count**.

count specifies the number of events that must occur before latch is released.

CountDownLatch's await() method has 2 forms :

- **void await()** throws **InterruptedException**

Causes the current thread to wait until one of the following things happens-

- latch **count** has down to reached 0, or
- unless the thread is interrupted.

- **boolean await(long timeout, TimeUnit unit)**

Causes the current thread to wait until one of the following things happens-

- latch **count** has down to reached 0,
- unless the thread is interrupted, or
- specified **timeout** elapses.

CountDownLatch's countDown() method in java :

- **void countDown()**

Reduces latch **count** by 1.

If **count** reaches 0, all waiting threads are released.

Read more about [CountDownLatch in java](#).

Thread concurrency interview Question 10. Where can you use CountDownLatch in real world?

Answer. Very interesting question. It will test your real time thread concurrency implementation skills. When you go in amusement park, you must have seen on certain rides there is mandate that at least 3 people (**3 is count**) should be there to take a ride. So, ride keeper (**ride keeper is main thread**) waits for 3 persons (**ride keeper has called await()**).

Every time a person comes count is reduced by 1 (**let's say every person is calling countDown() method**).

Ultimately when 3 persons reach count becomes 0 & wait for ride keeper comes to end.

Read more about [CountDownLatch in java](#).

Thread concurrency interview Question 11. How can you implement your own CountDownLatch in java?

Answer. This is important and complex interview question for developers. Please see [Implementation of custom/own CountDownLatch in java](#).

Thread concurrency interview Question 12. What is CyclicBarrier in java?

Answer. This is very important thread concurrency interview question. Fresher and experienced both will be versed with this. There might be situation where we might have to trigger event only when one or more threads complete certain operation in java.

2 or more threads wait for each other to reach a common barrier point. When all threads have reached common barrier point (i.e. when all threads have called await() method) >

- **All waiting threads are released**, and
- **Event can be triggered** as well.

[CyclicBarrier's constructor in java](#) >

- **CyclicBarrier(int parties)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released.

- **CyclicBarrier(int parties, Runnable barrierAction)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released and **barrierAction** (event) is triggered.

CyclicBarrier's await() method has 2 forms :

- `int await()` throws `InterruptedException`, `BrokenBarrierException`

If the current thread is not the last to arrive(i.e. call `await()` method) then it waits until one of the following things happens -

- The last thread to call arrive(i.e. call `await()` method), or
- Some other thread interrupts the current thread, or
- Some other thread interrupts one of the other waiting threads, or
- Some other thread times out while waiting for barrier, or
- Some other thread invokes `reset()` method on this `cyclicBarrier`.

- `int await(long timeout, TimeUnit unit)` throws `InterruptedException`, `BrokenBarrierException`, `TimeoutException`

If the current thread is not the last to arrive(i.e. call `await()` method) then it waits until one of the following things happens -

- The last thread to call arrive(i.e. call `await()` method), or
- The specified **timeout** elapses, or
- Some other thread interrupts the current thread, or
- Some other thread interrupts one of the other waiting threads, or
- Some other thread times out while waiting for barrier, or
- Some other thread invokes `reset()` method on this `cyclicBarrier`.

Read more about [CyclicBarrier in java](#).

Thread concurrency interview Question 13. Why is CyclicBarrier cyclic in java?

Answer. This is very interesting thread concurrency interview question for developers. The barrier is called *cyclic* because [CyclicBarrier](#) can be reused after -

- All the waiting threads are released in java and
- event has been triggered in java.

Thread concurrency interview Question 14. Where could we use CyclicBarrier in real world?

Answer. Another very interesting question. It will test your real time thread concurrency implementation skills. Let's say 10 friends (**friends are threads**) have planned for picnic on place A (Here **place A is common barrier** point). And they all decided to play certain game (**game is event**) only on everyones arrival at place A. So, all 10 friends must wait for each other to reach place A before launching event.

Now, when all **threads** have **reached** common **barrier point** (i.e. all friends have reached place A) >

- **All waiting threads are released** (All friends can play game), and
- **Event can be triggered** (they will start playing game).

Thread concurrency interview Question 15. How can you implement your own CyclicBarrier in java?

Answer. This is another important and complex interview for developers. Please see [Implementation of custom/own CyclicBarrier in java](#).

Thread concurrency interview Question 16. Similarity and Difference between CyclicBarrier and CountdownLatch in Java?

Answer. This is **very very important** thread concurrency interview question. Fresher and experienced both must be well versed with this.

1. [CyclicBarrier](#) and [CountDownLatch](#) are similar because they wait for specified number of thread to reach certain point and make count/parties equal to 0. But,
for completing wait in `CountDownLatch` specified number of threads must call `countDown()` method in java.
for completing wait in `CyclicBarrier` specified number of threads must call `await()` method.

2. Let's see there constructor's >

<code>CountDownLatch(int count)</code>	<code>CyclicBarrier(int parties)</code>
--	---

CountDownLatch is initialized with given count . count specifies the number of events that must occur before latch is released.	New CyclicBarrier is created where parties number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, parties number of waiting threads are released.
---	--

3. **CyclicBarrier** can be **awaited repeatedly**, but **CountDownLatch** can't be awaited repeatedly. i.e. once count has become 0 cyclicBarrier can be used again but CountDownLatch cannot be used again in java.
4. **CyclicBarrier** can be used to trigger event, but **CountDownLatch** can't be used to launch event. i.e. once count has become 0 cyclicBarrier can trigger event but CountDownLatch can't in java.

How can cyclicBarrier launch event?

CyclicBarrier provides constructor for triggering event.

CyclicBarrier(int parties, Runnable barrierAction)

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released and **barrierAction (event) is triggered**.

Thread concurrency interview Question 17. What is Phaser in java? Is Phaser similar to CyclicBarrier?

Answer. This is **another very important** thread concurrency interview question. [Phaser](#) is somewhat similar in functionality of [CyclicBarrier](#) and [CountDownLatch](#) but it provides more flexibility than both of them.

Phaser provides us flexibility of registering and deRegistering parties at any time.

For registering parties, we may use any of the following -

- [constructors](#), or
- [int register\(\)](#), or
- [bulkRegister\(\)](#).

For deRegistering parties, we may use any of the following -

- [arriveAndDeregister\(\)](#)

we have methods like **getPhase()** which returns the current phase number. And

isTerminated() method returns **true** if phaser has been **terminated**.

Read more about [Phaser in java](#)

Thread concurrency interview Question 18. Differences and similarity between Phaser and CyclicBarrier in java?

Answer. Another interesting thread concurrency interview question. Like a [CyclicBarrier](#), a [Phaser](#) can be **awaited repeatedly**.

But, in CyclicBarrier we used to register parties in constructor but Phaser provides us flexibility of registering and deRegistering parties at any time in java.

Thread concurrency interview Question 19. Difference between arrive() and arriveAndAwaitAdvance() method of Phaser in java?

Answer. This is thread concurrency interview question for experienced developers. **arrive()** method of [Phaser](#) **does not cause current thread to wait for other registered threads to complete current phase**.

That means current thread can immediately start next phase without waiting for any other registered thread to complete current phase.

But, **arriveAndAwaitAdvance()** method **causes current thread to wait for other registered threads to complete current phase**. That means current thread can proceed to next phase only when all other threads have completed current phase (i.e. by calling **arriveAndAwaitAdvance()** method).

Thread concurrency interview Question 20. When is phaser terminated in java?

Answer. This is another thread concurrency interview question for experienced developers. When calling `arriveAndDeregister()` method of [Phaser](#) has caused the number of registered parties to become 0. Termination can also be triggered when an `onAdvance()` method returns **true**.

Question 21. How can you control number of phase you want to execute in Phaser in java?

Answer. Yet another thread concurrency interview question for experienced developers. We can override the `onAdvance()` method of [Phaser](#) to control number of phases which we want to execute.

Signature of `onAdvance` method is [boolean onAdvance\(int phase, int registeredParties\)](#).

Where, **phase** is the current phase number when we enter `onAdvance()` method i.e. before advancing to next phase.

registeredParties is the current number of registered parties

Every Time before advancing to next phase overridden onAdvance() method is called and returns either true or false.

If method returns **true** then **phaser is terminated** ,or

If method returns **false** then **phaser continues** and can **advance to next phase**.

[Program to demonstrate usage of how we can override Phaser's onAdvance method to control number of phase we want to execute](#)

Thread concurrency interview Question 22. Where could we use Phaser in real world?

Answer. Another interesting thread concurrency interview question. Software process management is done in phases.

- First phase could be **requirement gathering**,
- second could be **software development** and
- third could be **testing**.

Second phase will not start until first is not completed, like wise third phase will not start until second is not completed.

Thread concurrency interview Question 23. What is maximum number of parties that could be registered with phaser at a time in java ?

Answer. Complex and challenging interview question even for experienced. Maximum number of parties that could be registered with phaser at a time is **65535**, if we try to register more parties **IllegalStateException** will be thrown in java.

Thread concurrency interview Question 24. What is exchanger in Java?

Answer. This is very important thread concurrency interview question for freshers and experienced. [Exchanger](#) enables two threads to exchange their data between each other. Exchanger can be handy in solving Producer Consumer pattern where Producer and consumer threads can exchange their data.

- [exchange\(V x\)](#)

`exchange()` method enables two threads to exchange their data between each other.

If current thread is first one to call exchange() method then it will wait until one of following things happen >

- Some other thread calls `exchange()` method, or
- Some other thread interrupts the current thread, or

If some other thread has already called exchanger() method then it resumes its execution and following things happen -

- waiting thread is resumed and receives data from current thread.
- current thread receives data from that waiting thread and it returns immediately.

- [V exchange\(V x, long timeout, TimeUnit unit\)](#)

`exchanger()` method enables two threads to exchange their data between each other.

If current thread is first one to call `exchange()` method then it will wait until one of following things happen >

- Some other thread calls `exchange()` method, or
- Some other thread interrupts the current thread, or
- The specified **timeout** elapses.

If some other thread has already called `exchanger()` method then it resumes its execution and following things happen -

- waiting thread is resumed and receives data from current thread.
- current thread receives data from that waiting thread and it returns immediately.

Read more about [Exchanger in java](#).

Thread concurrency interview Question 25. How can you implement Producer Consumer pattern using Exchanger in java?

Answer. Very interesting thread concurrency interview question for experienced developers. Exchanger is created, which will enable Producer and consumer threads to exchange their data.

Producer thread produces and calls `exchange()` method, now it will wait for consumer thread to call `exchange()` method.

Consumer thread calls `exchange()` method and following things will happen >

- current thread (consumerThread) will receive data from that waiting thread (producerThread) and it returns immediately.
- waiting thread (producerThread) will resume and receive data from current thread (consumerThread).

[Read program to implement Producer Consumer pattern using Exchanger](#)

Question 26. How can you solve consumer producer pattern by using BlockingQueue in java?

Answer. It is **very very important** thread concurrency interview question for all developers.. 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 uses consumer producer pattern in very optimized manner, BlockingQueue is designed in such a manner that it offers us the best performance.

[BlockingQueue is an 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 27. How can you implement your own LinkedBlockingQueue to solve consumer producer pattern in java?

Answer. Another challenging, logical and complex thread concurrency interview question, Please read

[Producer Consumer pattern using Custom implementation of BlockingQueue interface](#)

Thread concurrency interview Question 28. What is Lock in java?

Answer. Important thread concurrency interview question. The `java.util.concurrent.locks.Lock` is an interface and its implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.

A lock helps in controlling access to a shared resource by multiple threads. Only one thread at a time can acquire the lock and access the shared resource in java.

If a second thread attempts to acquire the lock on shared resource when it is acquired by another thread, the second thread will wait until the lock is released. In this way we can achieve [synchronization](#) and [race conditions](#) can be avoided in java.

Read lock of a ReadWriteLock may allow concurrent access to a shared resource in java.

Thread concurrency interview Question 29. Explain key methods of Lock interface in java?

Answer. This is very important thread concurrency interview question for freshers and experienced.

[Lock interface key methods in java >](#)

void lock()

Acquires the lock if it is not held by another thread. And sets **lock hold count** to 1.

If current thread already holds lock then **lock hold count** is increased by 1.

If the lock is held by another thread then the current thread waits for another thread to release lock.

void unlock()

If the current thread is the holding the lock then the **lock hold count** is decremented by 1. If the **lock hold count** has reached 0, then the lock is released.

If **lock hold count** is still greater than 0 then lock is not released.

If the current thread is not holding the lock then **IllegalMonitorStateException** is thrown.

boolean tryLock()

Acquires the lock if it is not held by another thread and returns true. And sets **lock hold count** to 1.

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If lock is held by another thread then method return false.

boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException

Acquires the lock if it is not held by another thread and returns true. And sets **lock hold count** to 1.

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If lock is held by another thread then current thread will wait until one of the following things happen -

- **Another thread releases lock and the lock is acquired by the current thread, or**
- **Some other thread interrupts the current thread, or**
- **The specified timeout elapses .**

If the **lock is acquired** then method **returns true**. And sets **lock hold count** to 1.

If **specified timeout elapses** then method return false.

Condition newCondition()

Method returns a Condition instance to be used with this Lock instance.

Condition instance are similar to using [Wait\(\), notify\(\) and notifyAll\(\)](#) methods.

- **IllegalMonitorStateException** is thrown if this lock is not held when any of the **Condition waiting or signalling methods** are called.
- **Lock is released** when the **condition waiting methods are called** and before they return, the lock is reacquired and the **lock hold count** restored to what it was when the method was called.
- If a **thread is interrupted while waiting** then **InterruptedException** will be thrown and following things will happen -
 - the **wait will be over**, and
 - **thread's interrupted status will be cleared**.
- Waiting threads are signalled in FIFO (first in first out order) order.
- When lock is **fair**, first lock is obtained by longest-waiting thread.

If lock is not **fair**, any waiting thread could get lock, at discretion of implementation.

can it be used to implement producer consumer pattern in java?

Answer. It is very complex thread concurrency interview question. Even most of the experienced developers are not aware of this question. Please read [ReentrantLock class provides implementation of Lock's newCondition\(\) method in java - description and solving producer consumer program using this method.](#)

Thread concurrency interview Question 31. Explain key methods of ReentrantLock class in java?

Answer. This is very important thread concurrency interview question for freshers and experienced. [ReentrantLock class](#) provides implementation of all Lock interface methods

```
void lock()
void unlock()
boolean tryLock()
boolean tryLock(long timeout, TimeUnit unit)
```

Additional methods provided by ReentrantLock class are >

void lockInterruptibly() throws InterruptedException

If current thread already holds lock then method returns true. And increments lock hold count by 1.

If the lock is held by another thread then the current thread waits until one of the following thing

happens -

- The lock is acquired by the current thread, or
- Some other thread **interrupts the current thread**.

As soon as current thread acquires the lock it sets lock hold count to 1.

int getWaitQueueLength(Condition condition)

Method returns number of threads that may be waiting to acquire this lock.

Method is used just for monitoring purposes and not for any kind of synchronization purposes.

boolean isHeldByCurrentThread()

Method returns true if lock is held by current thread. Its similar to **Thread.holdsLock()** method.

Read more about [ReEntrantLocks in java](#)

Thread concurrency interview Question 32. Write Program to demonstrate usage of ReentrantLock in java?

Answer. Interesting question for developers. [Read program to demonstrate usage of ReentrantLock.](#)

Thread concurrency interview Question 33. How can you implement your own ReentrantLock in java?

Answer. This is important and complex interview question for developers. [Implementation of custom/own Lock and ReEntrantLock in java](#)

Thread concurrency interview Question 34. What is Fork/Join Framework in java ?

Answer. This is **very very important** thread concurrency interview question. Fresher and experienced both must be well versed with this..

*Fork/Join Framework has been added in JDK 7 and is defined in the **java.util.concurrent** package in java.*

*Fork/Join framework enables **parallel programming**. Parallel programming means taking **advantage two or more processors (multicore) in computers**. Parallel programming improves program performance in java.*

The Fork/Join Framework also improves program performance in following ways >

- Fork/Join framework makes use of multiple processors available in computer. Hence enabling parallel processing, and
- It managing whole [life cycle of Threads](#).

Read more about [Fork/Join Framework - Parallel programming in java](#).

Thread concurrency interview Question 35. What is Divide-and-conquer in Fork/Join framework in java ?

Answer. Basic thread concurrency interview question, which tests the developers in depth knowledge about the fork join framework in java. The **divide-and-conquer** strategy recursively divides a task into smaller subtasks until subtask isn't small enough to be solved independently.

Thread concurrency interview Question 36. What approach does ForkJoinPool uses for managing tasks in java?

Answer. Another basic thread concurrency interview question, which tests the developers in depth knowledge about the fork join framework in java. [ForkJoinPool](#) uses **work-stealing approach for managing threads**. Each thread in ForkJoinPool maintains a queue of tasks. If one thread's queue is empty, it can take task from another thread. This overall improves the program/applications performance in java.

Thread concurrency interview Question 37. What are ForkJoinPool and ForkJoinTask in java?

Answer.

[ForkJoinPool in java](#)

ForkJoinPool implements ExecutorService framework. The execution of **ForkJoinTasks** takes place within a **ForkJoinPool**, which also manages the execution of the tasks.

ForkJoinPool constructors >

- **ForkJoinPool()**
 - Creates a pool in java.
 - **level of parallelism = number of processors available in the system**
- **ForkJoinPool(int parallelism)**
 - The **parallelism** is the **level of parallelism**. Its value must be greater than 0 and must not be more than number of processors in system.
 - **level of parallelism** determines the number of threads that can execute simultaneously. As a number of threads are determined it also determines number of tasks that could be executed **parallelly** in java.

ForkJoinPool important methods in java >

After you have created an instance of **ForkJoinPool**, you can start a task in a number of different ways. **The first task started is the main task. Main task begins subtasks that are also managed by the pool.** Different methods for starting tasks have been discussed below >

- **<T> T invoke(ForkJoinTask<T> task)**

This method starts the **task** and returns the result of the **task**. Calling code waits until method returns.

- **void execute(ForkJoinTask<?> task)**

The execute() method can be used to start a **task** without waiting for its completion.

This method starts the **task**. Calling code continues its execution asynchronously and does not wait for method completion like in `invoke` method.

- ***submit()* method comes in 4 different forms.**

`submit()` method can also be used for submitting task.

- ***int getParallelism()***

The method returns **level of parallelism** i.e. number of processors available in the system.

- ***void shutdown()***

Initiates shutdown, previously submitted tasks are executed, but no new tasks will be accepted.

- ***List<Runnable> shutdownNow()***

- attempts to stop all actively executing tasks,
- submitted tasks may or may not execute.
- awaiting tasks will never execute, and
- method cancels both existing and unexecuted tasks, so it returns empty list.

ForkJoinTask<V> in java

ForkJoinTask is abstract class for tasks that run within a **ForkJoinPool**.

ForkJoinTask<V> is an abstract class that defines a task that can be managed by a **ForkJoinPool**.

The **V** specifies the result type of the task in java.

Threads managed by ForkJoinPool executes ForkJoinTasks. Small number of threads are used to serve large number of tasks.

ForkJoinTask important methods >

ForkJoinTask core methods are **fork()** and **join()**

- ***ForkJoinTask<V> fork()***

The **fork()** method **submits the task** for asynchronous execution, means that the thread that calls **fork()** method to submit task continues to run. Task are executed in the **compute()** method, which is running within a **ForkJoinPool**.

- ***V join()***

The **join()** method waits for task completion on which it is called. The method returns result of the task.

- ***In short, about fork() and join()*** are used for starting one or more new tasks and then wait for them to complete.

- ***V invoke()***

The **invoke()** method combines the functionality of **fork()** and **join()** methods. **invoke()** submits the task and waits for completion of submitted task.

The method returns result of task.

- ***static void invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2)***

invokeAll() method submits **t1** and **t2** and waits for completion of **t1** and **t1**.

- ***static void invokeAll(ForkJoinTask<?> ... tasks)***

invokeAll() method submits list of tasks i.e. **tasks** and waits for completion of all tasks in list.

The **invokeAll()** method can only be called from within the overridden **compute()** method of another **ForkJoinTask**, which is running within a **ForkJoinPool**.

Some *other important methods* for checking status of submitted task

- *boolean isDone()* method returns true if a task completes.
- *boolean isCompletedNormally()* method returns true if a task completed normally without cancellation or without throwing any exception.
- *boolean isCompletedAbnormally()* returns true if a task completed abnormally either by cancellation or by throwing any exception.
- *boolean isCancelled()* returns true if the task was cancelled.

Question 38. Similarity and Difference between RecursiveAction and RecursiveTask in java?

Answer. Another important thread concurrency interview question.

Difference between RecursiveAction and RecursiveTask in java

<i>RecursiveAction</i>	<i>RecursiveTask<V></i>
This submits a task and does not return a result in java.	This submits a task and returns a result in java.
Definition of compute method <i>protected abstract void compute()</i>	<i>protected abstract V compute()</i> The <i>V</i> specifies the result type of the task.

Similarity between RecursiveAction and RecursiveTask

- > Both **extends ForkJoinPool**.
- > All **computations by tasks** are performed inside **compute()** method.

Question 39. How can we use Fork/Join framework in real world?

Answer. This thread concurrency interview question will test your real time thread concurrency implementation skills. We can use Fork/Join framework for calculating sum of array of 100000 or even may be more numbers. *Fork/Join framework uses divide-and-conquer strategy for enabling parallel programming.* Divide-and-conquer strategy recursively divides a array into smaller subarrays until subarray isn't small enough to be solved independently.

Also, **ForkJoinPool** uses **work-stealing approach** for managing threads. Each thread in **ForkJoinPool** maintains a queue of tasks. If one thread's queue is empty, it can take task from another thread. This overall improve the programs performance. Please see [program](#) to calculate sum of array of 100000 numbers.

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
```

Thread concurrency interview Question 40. Difference between synchronized and ReentrantLock in java?

Answer. This is another **very very important** thread concurrency interview question. Fresher and experienced both must be well versed with this.

<i>synchronized in java</i>	<i>ReentrantLock in java</i>
Does not provide any fair locks in java.	provides fair locks , when lock is fair - first lock is obtained by longest-waiting thread in java. Constructor to provide fairness - <i>ReentrantLock(boolean fair)</i> Creates an instance of ReentrantLock. When fair is set true, first lock is obtained by longest-waiting thread in java. If fair is set false, any waiting thread could get lock, at discretion of implementation in java.

<p>Does not provide tryLock() method or its functionality. Thread always waits for lock in java.</p>	<p>Provide tryLock() method. If lock is held by another thread then method return false in java.</p> <p><i>boolean tryLock()</i></p> <p>Acquires the lock if it is not held by another thread and returns true. And sets lock hold count to 1.</p> <p>If current thread already holds lock then method returns true. And increments lock hold count by 1.</p> <p>If lock is held by another thread then method return false in java.</p>
<p>There is no method for lock interruptibility, though current thread waits until one of the following thing happens -</p> <ul style="list-style-type: none"> • The lock is acquired by the current thread in java, or • Some other thread interrupts the current thread in java. 	<p><i>void lockInterruptibly()</i></p> <p>If current thread already holds lock then method returns true. And increments lock hold count by 1.</p> <p>If the lock is held by another thread then the current thread waits until one of the following thing happens -</p> <ul style="list-style-type: none"> • The lock is acquired by the current thread in java, or • Some other thread interrupts the current thread. <p>As soon as current thread acquires the lock it sets lock hold count to 1.</p>
<p>Does not provide any method to return number of threads that may be waiting to acquire this lock in java.</p>	<p>provide <i>int getQueueLength()</i> method to return number of threads that may be waiting to acquire this lock in java.</p>
<p>holdsLock() method is used to find out whether lock is held by current thread or not. If current thread holds lock method returns true in java.</p>	<p><i>isHeldByCurrentThread()</i> method is used to find out whether lock is held by current thread or not. If current thread holds lock method returns true in java.</p>
<p>Thread can hold lock on object monitor only once in java.</p>	<p>if current thread already holds lock then lock hold count is increased by 1 when lock() method is called.</p> <p>method to maintain lock hold count -</p> <p><i>void lock()</i></p> <p>Acquires the lock if it is not held by another thread. And sets lock hold count to 1.</p> <p>If current thread already holds lock then lock hold count is increased by 1.</p>
<p>Does not provide any new condition() method in java.</p>	<p>provides <i>newCondition()</i> method.</p> <p>Method returns a Condition instance to be used with this Lock instance.</p> <p>Condition instance are similar to using <u>Wait(),_notify().and notifyAll()</u> methods on object.</p> <ul style="list-style-type: none"> • IllegalMonitorStateException is thrown if this lock is not held when any of the Condition waiting or signalling methods are called. • Lock is released when the condition waiting methods are called and before they return, the lock is reacquired and the lock hold count restored to what it was when the method was called. • If a thread is interrupted while waiting then InterruptedException will be thrown and following things will happen - <ul style="list-style-type: none"> ◦ the wait will be over, and ◦ thread's interrupted status will be cleared. • Waiting threads are signalled in FIFO (first in first out order) order in java.

	<ul style="list-style-type: none"> When lock is <i>fair</i>, first lock is obtained by longest-waiting thread in java. <p>If lock is not <i>fair</i>, any waiting thread could get lock, at discretion of implementation in java.</p>
--	--

Question 41. Difference between traditional multithreading and parallel programming in java?

Answer. Basic interview question. Not a important one. MultiThreading primarily was designed to work with single CPU and utilize idle time of CPU. If two or more processors are there multithreading won't be able to utilize multi processors but parallel programming using Fork/Join framework can utilize multiple processors available in computer in java.

Question 42. Explain atomic operations in java?

Answer. Developers must have knowledge of atomic operations in thread concurrency java. Java provides some classes in `java.util.concurrent.atomic` which offers an alternative to the other [synchronization](#) in java.

Please see [Atomic operations in java](#).

Question 43. What is AtomicInteger in java? Explain key methods of AtomicInteger?

Answer. Another very important thread concurrency interview question for freshers and experienced. *AtomicInteger* provides you with **int value** that is updated atomically. i.e. we can use these classes without any explicit synchronization in [multithreading](#) environment, because any operation done on these classes is [thread safe](#) in java.

[AtomicInteger important Methods >](#)

- **int get()**
method returns the current value
- **void set(int newValue)**
Sets to **newValue**.
- **int getAndSet(int newValue)**

Sets to **newValue** and returns the old value.

- **boolean compareAndSet(int expect, int update)**

Compare with *expect*, if equal, set to *update* and return true.

[Addition methods >](#)

- **int addAndGet(int value)**

adds *value* to the current value. And **return updated value**.

- **int incrementAndGet()**

increments current value by 1. And **return updated value**.

- **int getAndAdd(int value)**

Method **return current value**. And adds *value* to the current value.

- **int getAndIncrement()**

Method **return current value**. And increments current value by 1.

[Subtraction methods >](#)

- `int decrementAndGet()`

decrements current value by 1. And **return updated value**.

- `int getAndDecrement()`

Method **return current value**. And decrements current value by 1.

Thread concurrency interview Question 44. How can you implement your own AtomicInteger in java?

Answer. Another important and complex interview question for developers. Please see [Implementation of custom/own AtomicInteger in java](#).

Question 45. What is AtomicLong? Explain key methods of AtomicLong in java?

Answer. This is another important thread concurrency interview question for freshers and experienced. AtomicLong provides you with **long value** that is updated atomically. i.e. we can use these classes without any explicit synchronization in [multithreading](#) environment, because any operation done on these classes is [thread safe](#).

[AtomicLong important Methods >](#)

- `long get()`
method returns the current value
- `void set(long newValue)`
Sets to **newValue**.
- `long getAndSet(long newValue)`

Sets to **newValue** and returns the old value.

- `boolean compareAndSet(long expect, long update)`

[Addition methods >](#)

- `long addAndGet(long value)`

adds *value* to the current value. And **return updated value**.

- `long incrementAndGet()`

increments current value by 1. And **return updated value**.

- `long getAndAdd(long value)`

Method **return current value**. And adds *value* to the current value.

- `long getAndIncrement()`

Method **return current value**. And increments current value by 1.

[Subtraction methods >](#)

- `long decrementAndGet()`

decrements current value by 1. And **return updated value**.

- `long getAndDecrement()`

Method **return current value**. And decrements current value by 1.

Thread concurrency interview Question 46. How can you implement your own AtomicLong in java?

Answer. Another important and complex interview question for developers. Please see [Implementation of custom/own AtomicLong in java](#).

Thread concurrency interview Question 47. What will be the output of below question in java?

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockTryLockTest {
    public static void main(String[] args) {

        Lock lock=new ReentrantLock();
        MyRunnable myRunnable=new MyRunnable(lock);
        new Thread(myRunnable,"Thread-1").start();
        new Thread(myRunnable,"Thread-2").start();

    }
}

class MyRunnable implements Runnable{

    Lock lock;
    public MyRunnable(Lock lock) {
        this.lock=lock;
    }

    public void run(){

        System.out.println(Thread.currentThread().getName()
            +" is Waiting to acquire lock");

        if(lock.tryLock()){
            System.out.println(Thread.currentThread().getName()
                +" has acquired lock.");

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

        }
        else{
            System.out.println(Thread.currentThread().getName()
                +" didn't got lock.");

        }

    }

}
```

Answer.

Thread-1 is Waiting to acquire lock

Thread-2 is Waiting to acquire lock

Thread-1 has acquired lock.

Thread-2 didn't got lock.

Thread concurrency interview Question 48. What will be the output of below question in java?

```
import java.util.concurrent.Phaser;

public class PhaserParentChildTest {
    public static void main(String[] args) {

        /*
        * Creates a new phaser with no registered unArrived parties.
        */
        Phaser parentPhaser = new Phaser();

        /*
        * Creates a new phaser with the given parent &
        * no registered unArrived parties.
        */
        Phaser childPhaser = new Phaser(parentPhaser,0);

        childPhaser.register();

        System.out.println("parentPhaser isTerminated : "+parentPhaser.isTerminated());
        System.out.println("childPhaser isTerminated : "+childPhaser.isTerminated());

        childPhaser.arriveAndDeregister();
        System.out.println("\n--childPhaser has called arriveAndDeregister()-- \n");

        System.out.println("parentPhaser isTerminated : "+parentPhaser.isTerminated());
        System.out.println("childPhaser isTerminated : "+childPhaser.isTerminated());

    }
}
```

Answer.

parentPhaser isTerminated : false

```
childPhaser isTerminated : false
--childPhaser has called arriveAndDeregister()--
parentPhaser isTerminated : true
childPhaser isTerminated : true
```

Thread concurrency interview Question 49. Which atomic classes are available and which are not available in java 7? And why jdk developers didn't created those classes?

Answer. This is confusing thread concurrency interview question for freshers.

Java provides following classes in [java.util.concurrent.atomic](#) >

Classes found in [java.util.concurrent.atomic](#) are >

- **AtomicInteger**,
- **AtomicLong**, and
- **AtomicBoolean**.

Classes **NOT** found in [java.util.concurrent.atomic](#) are >

- **AtomicByte**,
- **AtomicShort**,
- **AtomicFloat**,
- **AtomicDouble** and
- **AtomicCharacter**

Read why jdk developers didn't created those classes. [Click here](#).

Having any doubt? or you liked the tutorial! Please comment in below section.

Please express your love by liking [JavaMadeSoEasy.com](#) ([JMSE](#)) on [facebook](#), following on [google+](#) or [Twitter](#).

