JAVA

MULTITHREADING

Part -3

INTERVIEW Q&A

SOFTWARE ENGINEERS

PRACTICAL GUIDE TO
MASTERING JAVA
MULTITHREADING

@codeincubicles

QUESTIONS WITH ANSWERS

Q. List synchronization primitives

- Semaphores

- Binary semaphore

- Mutex (also known as a lock)

- Locks

- Events

- Signals

- Condition variables

-

Satyaranjan Swain

- Monitors - Class or segment of code that can only be executed by one thread at a time

Q. In what scenarios can we use asynchronous tasks?

Asynchronous programming provides a non-blocking, event-driven programming model.

To implement asynchrony in Java, you would need to use 'Future' or 'FutureTask', available in the java.util.concurrent package. Although the former is an interface, the latter is an implementation of the Future interface. In essence, in using 'Future' in your code, your asynchronous task will be executed immediately with the promise of the result being made available to the calling thread in the future.

The following code snippet shows an interface with two methods. One illustrates a synchronous method and the other an asynchronous method.

```
import java.util.concurrent.Future;

public interface IMyDataManager {

// synchronous method
```

```
public String getMyDataSynchronously();

// asynchronous method

public Future<String> getMyDataAsynchronously();

}
```

Q. How to asynchronously call a method in java?

a) Runnable interface

```
final String x = "abc";

new Thread(new Runnable() {

public void run() {

x.matches("something");
```

```
}

}).start();
```

b) Runnable interface as a lambda (Java 8)

You can use a lambda expression to shorten it to:

```
Thread a = new Thread(() -> {

//Do whatever

});
```

c) FutureTask

You may wish to also consider the class java.util.concurrent.FutureTask.

If you are using Java 5 or later, FutureTask is a turnkey implementation of "A cancellable asynchronous computation."

d) ScheduledExecutorService (java.util.concurrent)

e) Executors.newSingleThreadExecutor().submit(task);Task can be

Callable or Runnable.

f) @Async annotation

You can use @Async annotation from jcabi-aspects and AspectJ:

```
public class Foo {

@Async

public void save() {

// to be executed in the background

}
```

```
}
```

When you call save(), a new thread starts and executes its body. Your main thread continues without waiting for the result of save().

g) In Play framework you can use asynch Play API with java8CompletableFuture / CompletableFuture.supplyAsync

```
public CompletionStage<Result> index() {

return CompletableFuture.supplyAsync(() -> intensiveComputation())

.thenApply(i -> ok("Result: " + i));

}
```

And Play framework (MVC) works very well with Akka actor framework.

Q. What is the difference between parallelism and concurrency?

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they will ever both be running at the same instant. Eg. multitasking on a single-core machine.

Parallelism is when tasks literally run at the same time, eg. on a multicore processor.

Quoting Sun's Multithreaded Programming Guide:

- Concurrency: A condition that exists when at least two threads are makingprogress. A more generalized form of parallelism that can include timeslicing as a form of virtual parallelism.

- Parallelism: A condition that arises when at least two threads areexecuting simultaneously.

Q. What is the difference between Runnable and Callable interfaces?

1) The Runnable interface is older than Callable, there from JDK 1.0, whileCallable is added in Java 5.0.

2) Runnable interface has run() method to define task while Callableinterface uses call() method for task definition.

3) run() method does not return any value. It's return type is void while callmethod returns value. The Callable interface is a generic parameterized interface and Type of value is provided when an instance of Callable implementation is created.

4) run() method cannot throw checked exceptions while call() method canthrow checked exception in Java.

Callable interface uses Generic to define the return type of Object. Executors class provide useful methods to execute Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object. Callable tasks return java.util.concurrent.Future object. Using Future we can find out the status of the Callable task and get the returned Object. It provides get() method that can wait for the Callable to finish and then return the result.

```
@FunctionalInterface

public interface Callable<V> {

/**

* Computes a result, or throws an exception if unable to do so.
```

```
*

* @return computed result

* @throws Exception if unable to compute a result

*/

V call() throws Exception;

}

@FunctionalInterface

public interface Runnable {

/**

* When an object implementing interface 'Runnable' is used
```

```
* to create a thread, starting the thread causes the object's

* 'run' method to be called in that separately executing* thread.

*

* The general contract of the method 'run' is that it may* take any action

   whatsoever.

*

* @see java.lang.Thread#run()

*/

public abstract void run();

}
```

Q. What is the difference between Process and Thread?

A process is a self-contained execution environment and it can be seen as a program or application whereas Thread is a single task of execution within the process. Java runtime environment runs as a single process which contains different classes and programs as processes. Thread can be called lightweight process. Thread requires less resources to create and exists in the process. Thread shares the process resources.

Q. What is the difference between user Thread and daemon Thread?

When we create a Thread in java program, it's known as user thread. A daemon thread runs in background and doesn't prevent JVM from terminating. When there are no user threads running, JVM shuts down the program and quits. A child thread created from daemon thread is also a daemon thread.

Q. How can we create a Thread in Java?

There are two ways to create Thread in Java – first by implementing Runnable interface and then creating a Thread object from it and second is to extend the Thread Class.

a) implementing Runnable interface

```
final String x = "abc";

new Thread(new Runnable() {

public void run() {

x.matches("something");

}

}).start();
```

b) extend the Thread class

```
public class MyThread extends Thread {

public MyThread() {

super("MyThread");
```

```
}

public void run() {

//Code

}

}

//Started with a "new MyThread().start()" call
```

c) Implement Callable interface

d) Executors.newSingleThreadExecutor().submit(task);

Q. How to start a thread in java?

Call method start() on a thread.

Q. How to finish a thread in java?

a) Control Boolean flag (volatile / AtomicBoolean)'While

loop' condition should depend on a flag.

If flag == true/false => end tread.

b) Thread.stop (is deprecated)

c) Thread.interruptAd. a) Example: public class

MyThread implements Runnable{

private volatile boolean playing = false;

private volatile boolean abort = false;

@Override

Satyaranjan Swain

```
public void run(){

synchronized (this){

this.playing = true;

while (!abort){

//do something

while (!playing){

this.wait();

}

}
```

```
}

}

public void pauseThread(){

//paused

this.playing = false;

}

/**

* Resumes playback of thread (if possible).

*/

public void resumeThread(){
```

```
synchronized (this){

this.playing = true;

notify();

}

}

/**

* Sets the abort flag to true. Doing so makes the run() method finish*

   and thus stopping the thread properly.

*/

public void abortThread(){
```

```
this.abort = true;

}

}
```

Q. Why is Thread.stop() deprecated?

Because it is inherently unsafe. Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the ThreadDeath exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be damaged. When threads operate on damaged objects, arbitrary behaviour can result. This behaviour may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, ThreadDeath kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future.

Q. How do I stop a thread that waits for long periods (e.g., for input)?

That's what the Thread.interrupt() method is for. The state change can be followed by a call to Thread.interrupt, to interrupt the wait:

```
public void stop() {

Thread t = waiter;

waiter = null;

t.interrupt();

}
```

For this technique to work, it's critical that any method that catches an interrupt exception and is not prepared to deal with it immediately reasserts the exception. We say reasserts rather than rethrows, because it is not always possible to rethrow the exception. If the method that catches the InterruptedException is not declared to throw this (checked) exception, then it should "reinterrupt itself" in the following way:

```
Thread.currentThread().interrupt();
```

This ensures that the Thread will reraise the InterruptedException as soon as it is able.

Q. What if a thread doesn't respond to Thread.interrupt?

In some cases, you can use application specific tricks. For example, if a thread is waiting on a known socket, you can close the socket to cause the thread to return immediately. Unfortunately, there really isn't any technique that works in general. It should be noted that in all situations where a waiting thread doesn't respond to Thread.interrupt, it wouldn't respond to Thread.stop either. Such cases include deliberate denial-of-service attacks, and I/O operations for which thread.stop and thread.interrupt do not work properly.

Q. What are different states in lifecycle of Thread?

New, Runnable, Runing, Waiting, Blocked, Dead

When we create a Thread in java program, its state is New. Then we start the thread that change its state to Runnable. Thread Scheduler is responsible to allocate CPU to threads in Runnable thread pool and change their state to Running. Other Thread states are Waiting, Blocked and Dead. **Q. Can we call run() method of a Thread class?**

Yes, we can call run() method of a Thread class but then it will behave like a normal method. To actually execute it in a new thread, we need to start it using Thread.start() method.

Q. How can we pause the execution of a Thread for specific time?

We can use Thread class sleep() method to pause the execution of Thread for certain time. Note that this will not stop the processing of thread for specific time, once the thread awake from sleep, it's state gets changed to runnable and based on thread scheduling, it gets executed. **Q. What do you know about Thread Priority?**

Every thread has a priority. Usually higher priority thread gets precedence in execution but it depends on Thread Scheduler implementation that is OS dependent. We can specify the priority of thread but it doesn't guarantee that higher priority thread will get executed before lower priority thread. Thread priority is an int which value varies from 1 to 10 where 1 is the lowest priority thread and 10 is the highest priority thread.

Q. How can we make sure that main() is the last thread to finish in Java Program?

We can use Thread join() method to make sure all the threads created by the program are dead before finishing the main function.

public final void join(): This java thread join() method puts the current thread on wait until the thread on which it's called is dead. If the thread is interrupted, it throws InterruptedException.

```
public class ThreadJoinExample {

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable(), "t1");

Thread t2 = new Thread(new MyRunnable(), "t2");

Thread t3 = new Thread(new MyRunnable(), "t3");

//start 1st thread

t1.start();

//start second thread after waiting for 2 seconds or if it's dead

try {

t1.join(2000);
```

```
} catch (InterruptedException e) {

e.printStackTrace();

}

t2.start();

//start third thread only when first thread is dead

try {

t1.join();

} catch (InterruptedException e) {

e.printStackTrace();

}
```

```
t3.start();

//let all threads finish execution before finishing main thread

try {

t1.join();

t2.join();

t3.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

//All threads are dead, exiting main thread
```

```
}

}

class MyRunnable implements Runnable{

@Override

public void run() {

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

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

e.printStackTrace();
```

```
}

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

}

}
```

```
Thread started: t1

Thread started: t2

Thread ended: t1

Thread started: t3

Thread ended: t2

Thread ended: t3
```

All threads are dead, exiting main thread

Q. Why thread communication methods wait(), notify() and notifyAll() are in Object class?

In Java every Object has a monitor (LOCK) and wait(), notify() methods are used to wait for the Object monitor or to notify other threads that Object monitor is free now. There is no monitor on threads in java and synchronization can be used with any Object, that's why it's part of Object class so that every class in java has these essential methods for inter thread communication.

Besides that, these methods have to be called from synchronized block or method.

When a Thread calls wait() on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call notify() on this Object. Similarly when a thread calls notify() on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object. Since all these methods require Thread to have the Object monitor, that can be achieved only by synchronization, they need to be called from synchronized method or block.

class MyThread extends Thread {

```
Object lock;

MyThread (Object lock) {

this.lock = lock;

}

public void run() {

System.out.println(this.getName() + ": Started");

System.out.println(this.getName() + ": Starting for loop");

for (int i = 0; i < 1000000000; i++) {

}

System.out.println(this._getName() + ": Finished for loop.");
```

```
System.out.println(this.getName() + ": About to notify all"); synchronized
(lock) {

lock.notifyAll();

}

System.out.println(this.getName() + ": Finished");

}

}

public class WaitNotifyExample {

public static void main(String[] args) {

System.out.println("Main program: Started");

Object lock = new Object();
```

```
MyThread t1 = new MyThread(lock);

System.out.println("Main program: Spawing thread");

t1.start();

System.out.println("Main program: Waiting for lock");

synchronized (lock) {

try {

lock.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}
```

```
}

System.out.println("Main program: Resuming.");

System.out.println("Main program: Finished.");

}

}
```

Output:

```
# java WaitNotifyExample

Main program: Started

Main program: Spawing thread

Main program: Waiting for lock
```

Thread-0: Started Thread-

0: Starting for loop

Thread-0: Finished for loop.

Thread-0: About to notify all

Main program: Resuming.

Main program: Finished.

Thread-0: Finished

Q. Why Thread sleep() and yield() methods are static?

Thread sleep() and yield() methods work on the currently executing thread. So there is no point in invoking these methods on some other threads that are in wait state. That is why these methods are made static so that when this method is called statically, it works on the current executing thread and avoid confusion to the programmers who might think that they can invoke these methods on some non-running threads.

Thread.yield() - A hint to the scheduler that the current thread is willing to

yield its current use of a processor. The scheduler is free to ignore this hint.

Q. What is ThreadLocal?

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share it's variables, so if the variable is not thread safe, we can use synchronization but if we want to avoid synchronization, we can use ThreadLocal variables.

Every thread has its own ThreadLocal variable and they can use it's get() and set() methods to get the default value or change its value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread.

The ThreadLocal class in Java enables you to create variables that can only be read and written by the same thread.

Example: private ThreadLocal myThreadLocal = new

ThreadLocal<String>() {

```
@Override protected String initialValue() {

return "initial value";

}

};

public static class MyRunnable implements Runnable {

private ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>();

@Override

public void run() {

threadLocal.set( (int) (Math.random() * 100D) );

try {
```

```
Thread.sleep(2000);

} catch (InterruptedException e) {

}

System.out.println(threadLocal.get());

}

}
```

Q. What is Callable and Future?

Java 5 introduced java.util.concurrent.Callable interface in concurrency package that is similar to Runnable interface but it can return any Object and able to throw Exception.

Callable interface use Generic to define the return type of Object. Executors class provide useful methods to execute Callable in a thread pool. Since callable tasks run in parallel, we have to wait for the returned Object. Callable tasks return java.util.concurrent.Future object. Using Future we

can find out the status of the Callable task and get the returned Object. It provides get() method that can wait for the Callable to finish and then return the result.

And a few lines from javadoc. It is really worth to read it.

package java.util.concurrent;

/**

* A Future represents the result of an asynchronous computation. Methodsare provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method 'get' when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the 'cancel' method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled.

If you would like to use a 'Future' for the sake of cancellability but not provide a usable result, you can declare types of the form 'Future<?>' and return null as a result of the underlying task.

* Sample Usage (Note that the following classes are all made-up.)

*

* interface ArchiveSearcher { String search(String target); }

* class App {

* ExecutorService executor = ...

* ArchiveSearcher searcher = ...

* void showSearch(final String target) throws InterruptedException {

* Future<String> future

* = executor.submit(new Callable<String>() {

* public String call() {

* return searcher.search(target);

```
* }});

* displayOtherThings(); // do other things while searching

* try {

* displayText(future.get()); // use future

* } catch (ExecutionException ex) { cleanup(); return; }

* }

* }

*

* @see FutureTask

* @see Executor
```

```
* @since 1.5

* @param <V> The result type returned by this Future's get method

*/

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();

/**

* Waits if necessary for the computation to complete, and then* retrieves its

   result.
```

```
*

* @return the computed result

* @throws CancellationException if the computation was cancelled

* @throws ExecutionException if the computation threw an

* exception

* @throws InterruptedException if the current thread was interrupted

* while waiting

*/

V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit)
```

throws InterruptedException, ExecutionException, TimeoutException; }

Q. What is FutureTask Class?

FutureTask is the base implementation class of Future interface and we can use it with Executors for asynchronous processing. Most of the time we don’t need to use FutureTask class but it comes really handy if we want to override some of the methods of Future interface and want to keep most of the base implementation. We can just extend this class and override the methods according to our requirements.

* The 'FutureTask' class is an implementation of 'Future' that implements'Runnable', and so may be executed by an 'Executor'.

* For example, the above construction with 'submit' could be replaced by:

*

* FutureTask<String> future =

* new FutureTask<String>(new Callable<String>() {

* public String call() {

* return searcher.search(target);

* }});

* executor.execute(future);

Q. How can we achieve thread safety in Java?

There are several ways to achieve thread safety in java:

- synchronization,

- atomic concurrent classes,

- implementing concurrent Lock interface,

- using volatile keyword,

- using immutable classes,

- using Thread safe classes.

Q. What thread safe collections do you know of?

A. Synchronized Collections:

a) Vector

b) Hashtable

c) Static method: Collections.synchronizedSet() [conditionally thread-safe]

d) Static method: Collections.synchronizedMap() [conditionally thread-safe]

e) Static method: Collections.synchronizedList() [conditionally thread-

safe]B. Concurrent Collections:

a) Number of Queues from java.util.concurrent

- BlockingQueue (interface)

- LinkedBlockingQueue (FIFO)

- ArrayBlockingQueue (FIFO)

- PriorityBlockingQueue

- SynchronousQueue

b) ConcurrentHashMap

c) CopyOnWriteArrayList

d) CopyOnWriteArraySet

e) ConcurrentSkipListMap (since java 1.6)

f) ConcurrentSkipListSet (since java 1.6)

Q. What concurrent collections do you know?

a) Number of Queues from java.util.concurrent

- BlockingQueue (interface)

- LinkedBlockingQueue (FIFO)

- ArrayBlockingQueue (FIFO)

- PriorityBlockingQueue- SynchronousBlockingQueue

b) ConcurrentHashMap

c) CopyOnWriteArrayList

d) CopyOnWriteArraySet

e) ConcurrentSkipListMap (since java 1.6)

f) ConcurrentSkipListSet (since java 1.6)

Q. How to create synchronized collections in java?

A. Synchronized Collections:

a) Vector

b) Hashtable (this is a map)

c) Static method: Collections.synchronizedSet() [conditionally thread-

safe]Set s = Collections.synchronizedSet(new HashSet());

synchronized (s) {

Iterator i = s.iterator(); // Must be in the synchronized block

```
while (i.hasNext())

foo(i.next());

}
```

d) Static method: Collections.synchronizedMap() [conditionally thread-safe]

```
Map m = Collections.synchronizedMap(new HashMap());

Set s = m.keySet(); // Needn't be in synchronized block

synchronized (m) { // Synchronizing on m, not s!

Iterator i = s.iterator(); // Must be in synchronized block

while (i.hasNext()){

foo(i.next());
```

}

}

e) Static method: Collections.synchronizedList() [conditionally thread-safe]

Q. What different synchronized Map implementations are in the Java API?

- Hashtable

- Collections.synchronizedMap(Map)

- ConcurrentHashMap

Q. What is the difference between ConcurrentHashMap and map created by Collections.synchronizedMap(Map<K,V> m)?

ConcurrentHashMap allows concurrent modification of the Map from several threads without the need to block them.

Collections.synchronizedMap(map) creates a blocking Map which will degrade performance, albeit ensure consistency (if used properly).

Use the second option if you need to ensure data consistency, and each thread needs to have an up-to-date view of the map. Use the first if performance is critical, and each thread only inserts data to the map, with reads happening less frequently.

Q. What is atomic operation? What are atomic classes in Java Concurrency API?

Atomic operations are performed in a single unit of task without interference from other operations. Atomic operations are necessity in multi-threaded environment to avoid data inconsistency.

int++ is not an atomic operation. So by the time one threads read its value and increment it by one, other thread has read the older value leading to wrong result.

To solve this issue, we will have to make sure that increment operation on count is atomic, we can do that using Synchronization but Java 5 java.util.concurrent.atomic provides wrapper classes for int, long, boolean that can be used to achieve this atomically without usage of Synchronization.

Classes:

- AtomicInteger

- AtomicLong

- AtomicBoolean

- AtomicReference

Q. What AtomicInteger is for?

An int value that may be updated atomically.

Exemplary AtomicInteger operations:

- getAndIncrement()

- getAndDecrement()

- getAndAdd()

- incrementAndGet

- decrementAndGet

- getAndUpdate

- updateAndGet

Q. What AtomicBoolean is for?

A boolean value that may be updated atomically.

Operations:

- get

- compareAndSet

- weakCompareAndSet

- set

- lazySet(boolean val)

- getAndSet(boolean val)

Q. What is a critical section?

A critical section is a part of a multi-process program that may not be concurrently executed by more than one of the program's processes. In other words, it is a piece of a program that requires mutual exclusion of access. Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that does not allow multiple concurrent accesses.

Q. How to create a critical section in java?

a) synchronized keyword

public void foo() {

```
synchronized (this) {

// do something thread-safe

}

}
```

The same is:

```
public synchronized void foo() {

// do something thread-safe

}
```

b) with lock:

- java.util.concurrent.locks.ReadWriteLock. - ReentrantLock, - ReentrantReadWriteLock

c) semaphores

Q. How does thread communicate with each other?

When threads share resources, communication between Threads is important to coordinate their efforts. Object class wait(), notify() and notifyAll() methods allows threads to communicate about the lock status of a resource.

In order to invoke Object.wait(), this call must be placed in synchronized block, otherwise an IllegalMonitorStateException is thrown.

Q. How can you exchange data between different threads?

- Shared variables / volatile variables

- Concurrent collections (listed above)

Q. What volatile keyword is for?

When we use volatile keyword with a variable, all the threads read it's value directly from the memory and don't cache it. This makes sure that the value read is the same as in the memory.

Each thread has its own stack, and so its own copy of variables it can access. When the thread is created, it copies the value of all accessible variables to its own memory. The volatile keyword is used to say to the jvm "This variable may be modified in another Thread". Without this keyword the JVM is free to make some optimizations, like never refreshing those local copies in some threads. The volatile force the thread to update the original variable for each variable. The volatile keyword could be used on every kind of variable, either primitive or objects. **Q. How to synchronize threads?**

- Wait()/notify() on Object

- Synchronized keyword

- semaphores

- java.util.concurrent.locks.ReadWriteLock

Satyaranjan Swain

- ReentrantLock

Q. What is Lock contention and how to reduce it?

Essentially thread contention is a condition where one thread is waiting for a lock/object that is currently being held by another thread. Therefore, this waiting thread cannot use that object until the other thread has unlocked that particular object.

Q. How to reduce lock contention?

1. Protect data, not code. The whole method does not need to besynchronized.

2. Get rid of expensive calculations while in locks.

3. Use different locks for different data whenever possible.

4. Use atomic operations

5. Use synchronized data structures

Satyaranjan Swain

6. Use Reader-Writer Locks where applicable

7. Use Read-Only data whenever possible

8. Avoid Object Pooling

9. Use local variables or thread-local storage**Q. What is the purpose of thread pool?**

A thread pool is a group of threads initially created that waits for jobs and executes them. The idea is to have the threads always existing, so that we won't have to pay overhead time for creating them every time. They are appropriate when we know there's a stream of jobs to process, even though there could be some time when there are no jobs. **Q. How to make a pool of threads in java?**

- ThreadPoolExecutor

- java.util.concurrent.Executors#newCachedThreadPool

- java.util.concurrent.Executors#newFixedThreadPool-

java.util.concurrent.Executors#newScheduledThreadPool

Q. What is deadlock and how to prevent it?

Deadlock is a programming situation where two or more threads are blocked forever, this situation arises with at least two threads and two or more resources.

To analyse a deadlock, we need to look at the java thread dump of the application, we need to look out for the threads with state as BLOCKED and then the resources it's waiting to lock, every resource has a unique ID using which we can find which thread is already holding the lock on the object.

These are some of the guidelines using which we can avoid most of the deadlock situations:

- Avoid Nested Locks: This is the most common reason for deadlocks,avoid locking another resource if you already hold one. It's almost impossible to get deadlock situation if you are working with only one object lock. For example, here is the implementation of run() method without nested lock and program runs successfully without deadlock situation.

```
public void run() {

String name = Thread.currentThread().getName();

System.out.println(name + " acquiring lock on " + obj1);

synchronized (obj1) {

System.out.println(name + " acquired lock on " + obj1);

work();

}

System.out.println(name + " released lock on " + obj1);

System.out.println(name + " acquiring lock on " + obj2);

synchronized (obj2) {
```

```
System.out.println(name + " acquired lock on " + obj2);

work();

}

System.out.println(name + " released lock on " + obj2);

System.out.println(name + " finished execution.");

}
```

- Lock Only What is Required: You should acquire lock only on theresources you have to work on, for example in above program I am locking the complete Object resource but if we are only interested in one of its fields, then we should lock only that specific field not complete object.

- Avoid waiting indefinitely: You can get deadlock if two threads arewaiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it's always best to use join with maximum time you want to wait for thread to finish.

- Synchronization with wait / notify

- Semaphore

- ReentrantLock

Q. How to resolve readers and writers problem?

Blocking Queues are particular interesting data structures as they can be used both to store information and coordinate actions between threads. They fit very naturally in producer consumer situations.

Normally if you have one or more threads producing data and one or more threads consuming information you will need communications mechanisms such as semaphores to coordinate between the two sets of threads. The semaphores will track when the queue is empty or when it is full. A blocking thread can handle much of the communication overhead.

The standard put operation used to place items into the queue is modified to block if the queue is full. The take operation removes an item from the queue if one is available, otherwise it blocks until an item is available.

BlockingQueue itself is an interface. There are a number of classes which support the BlockingQueue behavior. LinkedBlockingQueue and

ArrayBlockingQueue provide standard FIFO behavior. There is also a PriorityBlockingQueue. The SynchronousBlockingQueue is particularly interesting in that it doesn't actually store data, it's designed to allow threads to communicate using put and take without actually involving any real data storage. The storage size is zero, threads calling put pass data directly to threads calling take. The data structure acts as more of a communications mechanism than a true structure storing data.

Example:

```
class Producer implements Runnable {

private final BlockingQueue queue;

Producer(BlockingQueue q) { queue = q; }

public void run() {

try {

while (true) { queue.put(produce()); }
```

```
} catch (InterruptedException ex) { //handle it}

}

Object produce() { ... }

}

class Consumer implements Runnable {

private final BlockingQueue queue;

Consumer(BlockingQueue q) { queue = q; }

public void run() {

try {

while (true) { consume(queue.take()); }
```

```
} catch (InterruptedException ex) { //handle it}

}

void consume(Object x) { ... }

}

class MyMain {

void main() {

BlockingQueue q = new SomeQueueImplementation();

Producer p = new Producer(q);

Consumer c1 = new Consumer(q);

Consumer c2 = new Consumer(q);
```

```
new Thread(p).start();

new Thread(c1).start();

new Thread(c2).start();

}

}}
```

Q. What memory barriers (fences) are about?

Memory barriers, or fences, are a set of processor instructions used to apply ordering limitations on memory operations.