We can prevent a thread from execution by using any of the 3 methods of Thread class:

1. yield()
2. join()
3. sleep()

1. yield() method pauses the currently executing thread temporarily for giving a chance to the remaining waiting threads of the same priority to execute. If there is no waiting thread or all the waiting threads have a lower priority then the same thread will continue its execution. The yielded thread when it will get the chance for execution is decided by the thread scheduler whose behavior is vendor dependent.
2. join() If any executing thread t1 calls join() on t2 i.e; t2.join() immediately t1 will enter into waiting state until t2 completes its execution.
3. sleep() Based on our requirement we can make a thread to be in sleeping state for a specified period of time (hope not much explanation required for our favorite method).

What is thread pool? Why should we use thread pools?

A thread pool is a collection of threads on which task can be scheduled. Instead of creating a new thread for each task, you can have one of the threads from the thread pool pulled out of the pool and assigned to the task. When the thread is finished with the task, it adds itself back to the pool and waits for another assignment. One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Below are key reasons to use a Thread Pool

- Using thread pools minimizes the JVM overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and de-allocating many thread objects creates a significant memory management overhead.
- You have control over the maximum number of tasks that are being processed in parallel (= number of threads in the pool).

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

From the point of view of an interviewer, there are basically just 3 main things that I want to hear, besides obvious things like a process can have multiple threads:

1. Threads share same memory space, which means a thread can access memory from other's thread memory. Processes normally can not.
2. Resources. Resources (memory, handles, sockets, etc) are release at process termination, not thread termination.
3. Security. A process has a fixed security token. A thread, on the other hand, can impersonate different users/tokens.

When InvalidMonitorStateException is thrown? Why?

This exception is thrown when you try to call wait()/notify()/notifyAll() any of these methods for an Object from a point in your program where u are NOT having a lock on that object.(i.e. u r not executing any synchronized block/method of that object and still trying to call wait()/notify()/notifyAll()) wait(), notify() and notifyAll() all throw IllegalMonitorStateException. since This exception is a subclass of RuntimeException so we r not bound to catch it (although u may if u want to). and being a RuntimeException this exception is not mentioned in the signature of wait(), notify(), notifyAll() methods.

What happens when I make a static method as synchronized?

Synchronized static methods have a lock on the class "Class", so when a thread enters a synchronized static method, the class itself gets locked by the thread monitor and no other thread can enter any static synchronized methods on that class. This is unlike instance methods, as multiple threads can access "same synchronized instance methods" at same time for different instances.

Can a thread call a non-synchronized instance method of an Object when a synchronized method is being executed ?

Yes, a Non synchronized method can always be called without any problem. In fact Java does not do any check for a non-synchronized method. The Lock object check is performed only for synchronized methods/blocks. In case the method is not declared synchronized Jave will call even if you are playing with shared data. So you have to be careful while doing such thing. The decision of declaring a method as synchronized has to be based on critical section access. If your method does not access a critical section (shared resource or data structure) it need not be declared synchronized.

Can two threads call two different synchronized instance methods of an Object?

No. If a object has synchronized instance methods then the Object itself is used a lock object for controlling the synchronization. Therefore all other instance methods need to wait until previous method call is completed. See the below sample code which demonstrate it very clearly. The Class Common has 2 methods called synchronizedMethod1() and synchronizedMethod2() MyThread class is calling both the methods

```
public class Common {
public synchronized void synchronizedMethod1() {
System.out.println("synchronizedMethod1 called");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("synchronizedMethod1 done");
}
public synchronized void synchronizedMethod2() {
System.out.println("synchronizedMethod2 called");
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}
System.out.println("synchronizedMethod2 done");
}
}
public class MyThread extends Thread {
private int id = 0;
private Common common;

public MyThread(String name, int no, Common object) {
super(name);
common = object;
id = no;
}
public void run() {
System.out.println("Running Thread" + this.getName());
try {
if (id == 0) {
common.synchronizedMethod1();
} else {
common.synchronizedMethod2();
}
} catch (Exception e) {
e.printStackTrace();
}
}
public static void main(String[] args) {
Common c = new Common();
MyThread t1 = new MyThread("MyThread-1", 0, c);
MyThread t2 = new MyThread("MyThread-2", 1, c);
t1.start();
```

```
t2.start();
}
}
```

What is a deadlock?

Deadlock is a situation where two or more threads are blocked forever, waiting for each other. This may occur when two threads, each having a lock on one resource, attempt to acquire a lock on the other's resource. Each thread would wait indefinitely for the other to release the lock, unless one of the user processes is terminated. In terms of Java API, thread deadlock can occur in following conditions:

- When two threads call Thread.join() on each other.
- When two threads use nested synchronized blocks to lock two objects and the blocks lock the same objects in different order.

What is Starvation? And what is a Livelock?

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

LiveLock

Livelock occurs when all threads are blocked, or are otherwise unable to proceed due to unavailability of required resources, and the non-existence of any unblocked thread to make those resources available. In terms of Java API, thread livelock can occur in following conditions:

- When all the threads in a program execute Object.wait(0) on an object with zero parameter. The program is live-locked and cannot proceed until one or more threads call Object.notify() or Object.notifyAll() on the relevant objects. Because all the threads are blocked, neither call can be made.
- When all the threads in a program are stuck in infinite loops.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked. Starvation occurs when one thread cannot access the CPU because one or more other threads are monopolizing the CPU. In Java, thread starvation can be caused by setting thread priorities inappropriately. A lower-priority thread can be starved by higher-priority threads if the higher-priority threads do not yield control of the CPU from time to time.

What is thread pool? Why should we use thread pools?

A thread pool is a collection of threads on which task can be scheduled. Instead of creating a new thread for each task, you can have one of the threads from the thread pool pulled out of the pool and assigned to the task. When the thread is finished with the task, it adds itself back to the pool and waits for another assignment. One common type of thread pool is the fixed thread pool. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Below are key reasons to use a Thread Pool

- Using thread pools minimizes the JVM overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and de-allocating many thread objects creates a significant memory management overhead.
- You have control over the maximum number of tasks that are being processed in parallel (= number of threads in the pool).

Most of the executor implementations in java.util.concurrent use thread pools, which consist of worker threads. This kind of thread exists separately from the Runnable and Callable tasks it executes and is often used to execute multiple tasks.

Can we synchronize the run method? If yes then what will be the behavior?

Yes, the run method of a runnable class can be synchronized. If you make run method synchronized then the lock on runnable object will be occupied before executing the run method. In case we start multiple threads using the same runnable object in the constructor of the Thread then it would work. But until the 1st thread ends the 2nd thread cannot start and until the 2nd thread ends the next cannot start as all the threads depend on lock on same object