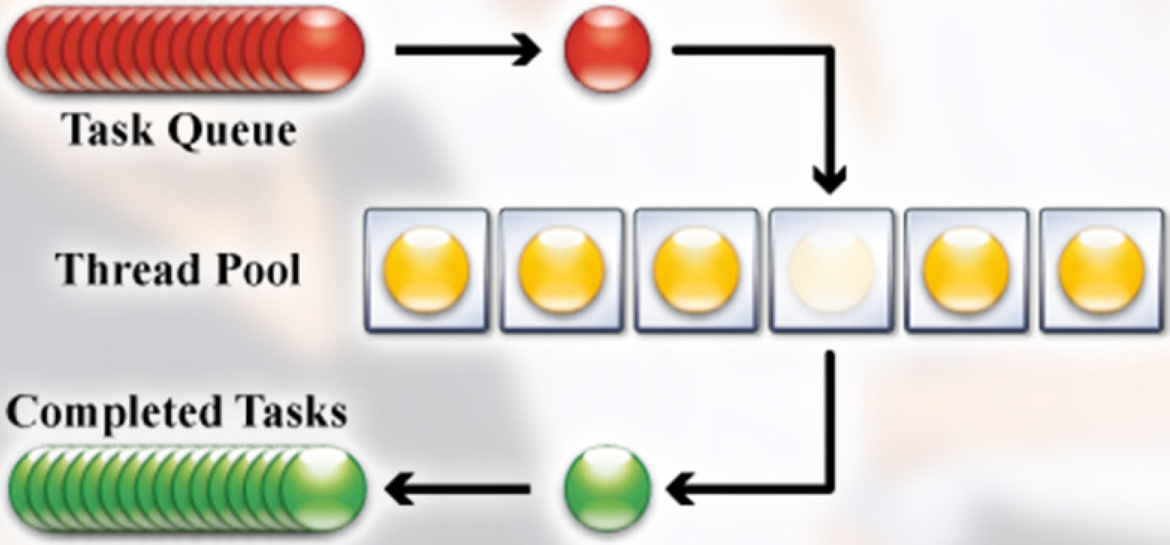
MULTITHREADING AND CONCURRENCY INTERVIEW QUESTIONS AND ANSWERS

BOOSTING YOUR JAVA CAREER



MARTIN MOIS

Java Code Geeks
Java 2 Java Developers Resource Center

Multithreading and Concurrency Questions

Contents

Preface

Concurrency is always a challenge for developers and writing concurrent programs can be extremely hard.

There is a number of things that could potentially blow up and the complexity of systems rises considerably when concurrency is introduced.

However, the ability to write robust concurrent programs is a great tool in a developer's belt and can help build sophisticated, enterprise level applications.

In this guide we will discuss different types of questions that can be used in a programming interview in order to assess a candidate's understanding of concurrency and multithreading.

The questions are not only Java specific, but revolve around general programming principles.

About the Author

Martin is a software engineer with more than 10 years of experience in software development. He has been involved in different positions in application development in a variety of software projects ranging from reusable software components, mobile applications over fat-client GUI projects up to larg-scale, clustered enterprise applications with real-time requirements.

After finishing his studies of computer science with a diploma, Martin worked as a Java developer and consultant for international operating insurance companies. Later on he designed and implemented web applications and fat-client applications for companies on the energy market. Currently Martin works for an international operating company in the Java EE domain and is concerned in his day-to-day work with larg-scale big data systems.

His current interests include Java EE, web applications with focus on HTML5 and performance optimizations. When time permits, he works on open source projects, some of them can be found at this github account. Martin is blogging at Martin's Developer World.

Chapter 1

MultiThreading Questions

1.1 What do we understand by the term concurrency?

Concurrency is the ability of a program to execute several computations simultaneously. This can be achieved by distributing the computations over the available CPU cores of a machine or even over different machines within the same network.

1.2 What is the difference between processes and threads?

A process is an execution environment provided by the operating system that has its own set of private resources (e.g. memory, open files, etc.). Threads, in contrast to processes, live within a process and share their resources (memory, open files, etc.) with the other threads of the process. The ability to share resources between different threads makes thread more suitable for tasks where performance is a significant requirement.

1.3 In Java, what is a process and a thread?

In Java, processes correspond to a running Java Virtual Machine (JVM) whereas threads live within the JVM and can be created and stopped by the Java application dynamically at runtime.

1.4 What is a scheduler?

A scheduler is the implementation of a scheduling algorithm that manages access of processes and threads to some limited resource like the processor or some I/O channel. The goal of most scheduling algorithms is to provide some kind of load balancing for the available processes/threads that guarantees that each process/thread gets an appropriate time frame to access the requested resource exclusively.

1.5 How many threads does a Java program have at least?

Each Java program is executed within the main thread; hence each Java application has at least one thread.

1.6 How can a Java application access the current thread?

The current thread can be accessed by calling the static method `currentThread()` of the JDK class `java.lang.Thread`:

```java
public class MainThread {

        public static void main(String[] args) {
                long id = Thread.currentThread().getId();
                String name = Thread.currentThread().getName();
                ...
        }
}
```

1.7 What properties does each Java thread have?

Each Java thread has the following properties:

- an identifier of type long that is unique within the JVM
- a name of type String
- a priority of type int
- a state of type `java.lang.Thread.State`
- a thread group the thread belongs to

1.8 What is the purpose of thread groups?

Each thread belongs to a group of threads. The JDK class `java.lang.ThreadGroup` provides some methods to handle a whole group of Threads. With these methods we can, for example, interrupt all threads of a group or set their maximum priority.

1.9 What states can a thread have and what is the meaning of each state?

- `NEW`: A thread that has not yet started is in this state.
- `RUNNABLE`: A thread executing in the Java virtual machine is in this state.
- `BLOCKED`: A thread that is blocked waiting for a monitor lock is in this state.
- `WAITING`: A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- `TIMED_WAITING`: A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- `TERMINATED`: A thread that has exited is in this state.

1.10 How do we set the priority of a thread?

The priority of a thread is set by using the method `setPriority(int)`. To set the priority to the maximum value, we use the constant `Thread.MAX_PRIORITY` and to set it to the minimum value we use the constant `Thread.MIN_PRIORITY` because these values can differ between different JVM implementations.

1.11 How is a thread created in Java?

Basically, there are two ways to create a thread in Java.

The first one is to write a class that extends the JDK class `java.lang.Thread` and call its method `start()`:

```
public class MyThread extends Thread {

        public MyThread(String name) {
                super(name);
        }

        @Override
        public void run() {
                System.out.println("Executing thread "+Thread.currentThread().getName());
        }

        public static void main(String[] args) throws InterruptedException {
                MyThread myThread = new MyThread("myThread");
                myThread.start();
        }
}
```

The second way is to implement the interface `java.lang.Runnable` and pass this implementation as a parameter to the constructor of `java.lang.Thread`:

```
public class MyRunnable implements Runnable {

        public void run() {
                System.out.println("Executing thread "+Thread.currentThread().getName());
        }

        public static void main(String[] args) throws InterruptedException {
                Thread myThread = new Thread(new MyRunnable(), "myRunnable");
                myThread.start();
        }
}
```

1.12 How do we stop a thread in Java?

To stop a thread one can use a volatile reference pointing to the current thread that can be set to null by other threads to indicate the current thread should stop its execution:

```
private static class MyStopThread extends Thread {
        private volatile Thread stopIndicator;

        public void start() {
                stopIndicator = new Thread(this);
                stopIndicator.start();
        }

        public void stopThread() {
                stopIndicator = null;
        }

        @Override
        public void run() {
                Thread thisThread = Thread.currentThread();
                while(thisThread == stopIndicator) {
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                        }
                }
        }
}
```

1.13 Why should a thread not be stopped by calling its method `stop()`?

A thread should not be stopped by using the deprecated methods `stop()` of `java.lang.Thread`, as a call of this method causes the thread to unlock all monitors it has acquired. If any object protected by one of the released locks was in an inconsistent state, this state gets visible to all other threads. This can cause arbitrary behavior when other threads work this this inconsistent object.

1.14 Is it possible to start a thread twice?

No, after having started a thread by invoking its `start()` method, a second invocation of `start()` will throw an `IllegalThreadStateException`.

1.15 What is the output of the following code?

```
public class MultiThreading {

        private static class MyThread extends Thread {

                public MyThread(String name) {
                        super(name);
                }

                @Override
                public void run() {
                        System.out.println(Thread.currentThread().getName());
                }
        }

        public static void main(String[] args) {
                MyThread myThread = new MyThread("myThread");
                myThread.run();
        }
}
```

The code above produces the output "main" and not "myThread". As can be seen in line two of the `main()` method, we invoke by mistake the method `run()` instead of `start()`. Hence, no new thread is started, but the method `run()` gets executed within the main thread.

1.16 What is a daemon thread?

A daemon thread is a thread whose execution state is not evaluated when the JVM decides if it should stop or not. The JVM stops when all user threads (in contrast to the daemon threads) are terminated. Hence daemon threads can be used to implement for example monitoring functionality as the thread is stopped by the JVM as soon as all user threads have stopped:

```
public class Example {

        private static class MyDaemonThread extends Thread {

                public MyDaemonThread() {
                        setDaemon(true);
                }

                @Override
                public void run() {
                        while (true) {
                                try {
                                        Thread.sleep(1);
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                }
        }

        public static void main(String[] args) throws InterruptedException {
                Thread thread = new MyDaemonThread();
                thread.start();
        }
}
```

The example application above terminates even though the daemon thread is still running in its endless while loop.

1.17 Is it possible to convert a normal user thread into a daemon thread after it has been started?

A user thread cannot be converted into a daemon thread once it has been started. Invoking the method `thread.setDaemon(true)` on an already running thread instance causes a `IllegalThreadStateException`.

1.18 What do we understand by busy waiting?

Busy waiting means implementations that wait for an event by performing some active computations that let the thread/process occupy the processor although it could be removed from it by the scheduler. An example for busy waiting would be to spend the waiting time within a loop that determines the current time again and again until a certain point in time is reached:

```
Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {
                long millisToStop = System.currentTimeMillis() + 5000;
                long currentTimeMillis = System.currentTimeMillis();
                while (millisToStop > currentTimeMillis) {
                        currentTimeMillis = System.currentTimeMillis();
                }
        }
});
```

1.19 How can we prevent busy waiting?

One way to prevent busy waiting is to put the current thread to sleep for a given amount of time. This can be done by calling the method `java.lang.Thread.sleep(long)` by passing the number of milliseconds to sleep as an argument.

1.20 Can we use Thread.sleep() for real-time processing?

The number of milliseconds passed to an invocation of Thread.sleep(long) is only an indication for the scheduler how long the current thread does not need to be executed. It may happen that the scheduler lets the thread execute again a few milliseconds earlier or later depending on the actual implementation. Hence an invocation of Thread.sleep() should not be used for real-time processing.

1.21 How can a thread be woken up that has been put to sleep before using Thread.sleep()?

The method interrupt() of java.lang.Thread interrupts a sleeping thread. The interrupted thread that has been put to sleep by calling Thread.sleep() is woken up by an InterruptedException:

```
public class InterruptExample implements Runnable {

        public void run() {
                try {
                        Thread.sleep(Long.MAX_VALUE);
                } catch (InterruptedException e) {
                        System.out.println("["+Thread.currentThread().getName()+"]  ↩
                            Interrupted by exception!");
                }
        }

        public static void main(String[] args) throws InterruptedException {
                Thread myThread = new Thread(new InterruptExample(), "myThread");
                myThread.start();

                System.out.println("["+Thread.currentThread().getName()+"] Sleeping in main ↩
                     thread for 5s...");
                Thread.sleep(5000);

                System.out.println("["+Thread.currentThread().getName()+"] Interrupting  ↩
                    myThread");
                myThread.interrupt();
        }
}
```

1.22 How can a thread query if it has been interrupted?

If the thread is not within a method like Thread.sleep() that would throw an InterruptedException, the thread can query if it has been interrupted by calling either the static method Thread.interrupted() or the method isInterrupted() that it has inherited from java.lang.Thread.

1.23 How should an InterruptedException be handled?

Methods like sleep() and join() throw an InterruptedException to tell the caller that another thread has interrupted this thread. In most cases this is done in order to tell the current thread to stop its current computations and to finish them unexpectedly. Hence ignoring the exception by catching it and only logging it to the console or some log file is often not the appropriate way to handle this kind of exception. The problem with this exception is, that the method run() of the Runnable interface does not allow that run() throws any exceptions. So just rethrowing it does not help. This means the implementation of run() has to handle this checked exception itself and this often leads to the fact that it its caught and ignored.

1.24 After having started a child thread, how do we wait in the parent thread for the termination of the child thread?

Waiting for a thread's termination is done by invoking the method `join()` on the thread's instance variable:

```
Thread thread = new Thread(new Runnable() {
        @Override
        public void run() {

        }
});
thread.start();
thread.join();
```

1.25 What is the output of the following program?

```
public class MyThreads {

        private static class MyDaemonThread extends Thread {

                public MyDaemonThread() {
                        setDaemon(true);
                }

                @Override
                public void run() {
                        try {
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                        }
                }
        }

        public static void main(String[] args) throws InterruptedException {
                Thread thread = new MyDaemonThread();
                thread.start();
                thread.join();
                System.out.println(thread.isAlive());
        }
}
```

The output of the above code is "false". Although the instance of MyDaemonThread is a daemon thread, the invocation of `join()` causes the main thread to wait until the execution of the daemon thread has finished. Hence calling `isAlive()` on the thread instance reveals that the daemon thread is no longer running.

1.26 What happens when an uncaught exception leaves the `run()` method?

I can happen that an unchecked exception escapes from the `run()` method. In this case the thread is stopped by the Java Virtual Machine. It is possible to catch this exception by registering an instance that implements the interface `UncaughtExceptionHandler` as an exception handler.

This is either done by invoking the static method `Thread.setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler)`, which tells the JVM to use the provided handler in case there was no specific handler registerd on the thread itself, or by invoking `setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler)` on the thread instance itself.

1.27 What is a shutdown hook?

A shutdown hook is a thread that gets executed when the JVM shuts down. It can be registered by invoking `addShutdownHook(Runnable)` on the Runtime instance:

```
Runtime.getRuntime().addShutdownHook(new Thread() {
                @Override
                public void run() {

                }
        });
```

1.28 For what purposes is the keyword synchronized used?

When you have to implement exclusive access to a resource, like some static value or some file reference, the code that works with the exclusive resource can be embraced with a synchronized block:

```
synchronized (SynchronizedCounter.class) {
        counter++;
}
```

1.29 What intrinsic lock does a synchronized method acquire?

A synchronized method acquires the intrinsic lock for that method's object and releases it when the method returns. Even if the method throws an exception, the intrinsic lock is released. Hence a synchronized method is equal to the following code:

```
public void method() {
        synchronized(this) {
                ...
        }
}
```

1.30 Can a constructor be synchronized?

No, a constructor cannot be synchronized. The reason why this leads to an syntax error is the fact that only the constructing thread should have access to the object being constructed.

1.31 Can primitive values be used for intrinsic locks?

No, primitive values cannot be used for intrinsic locks.

1.32 Are intrinsic locks reentrant?

Yes, intrinsic locks can be accessed by the same thread again and again. Otherwise code that acquires a lock would have to pay attention that it does not accidently tries to acquire a lock it has already acquired.

1.33 What do we understand by an atomic operation?

An atomic operation is an operation that is either executed completely or not at all.

1.34 Is the statement c++ atomic?

No, the incrementation of an integer variable consist of more than one operation. First we have to load the current value of c, increment it and then finally store the new value back. The current thread performing this incrementation may be interrupted in-between any of these three steps, hence this operation is not atomic.

1.35 What operations are atomic in Java?

The Java language provides some basic operations that are atomic and that therefore can be used to make sure that concurrent threads always see the same value:

- Read and write operations to reference variables and primitive variables (except long and double)
- Read and write operations for all variables declared as volatile

1.36 Is the following implementation thread-safe?

```
public class DoubleCheckedSingleton {
        private DoubleCheckedSingleton instance = null;

        public DoubleCheckedSingleton getInstance() {
                if(instance == null) {
                        synchronized (DoubleCheckedSingleton.class) {
                                if(instance == null) {
                                        instance = new DoubleCheckedSingleton();
                                }
                        }
                }
                return instance;
        }
}
```

The code above is not thread-safe. Although it checks the value of instance once again within the synchronized block (for performance reasons), the JIT compiler can rearrange the bytecode in a way that the reference to instance is set before the constructor has finished its execution. This means the method `getInstance()` returns an object that may not have been initialized completely. To make the code thread-safe, the keyword volatile can be used since Java 5 for the instance variable. Variables that are marked as volatile get only visible to other threads once the constructor of the object has finished its execution completely.

1.37 What do we understand by a deadlock?

A deadlock is a situation in which two (or more) threads are each waiting on the other thread to free a resource that it has locked, while the thread itself has locked a resource the other thread is waiting on: Thread 1: locks resource A, waits for resource B Thread 2: locks resource B, waits for resource A

1.38 What are the requirements for a deadlock situation?

In general the following requirements for a deadlock can be identified:

- Mutual exclusion: There is a resource which can be accessed only by one thread at any point in time.
- Resource holding: While having locked one resource, the thread tries to acquire another lock on some other exclusive resource.
- No preemption: There is no mechanism, which frees the resource if one thread holds the lock for a specific period of time.
- Circular wait: During runtime a constellation occurs in which two (or more) threads are each waiting on the other thread to free a resource that it has locked.

1.39 Is it possible to prevent deadlocks at all?

In order to prevent deadlocks one (or more) of the requirements for a deadlock has to be eliminated:

- Mutual exclusion: In some situation it is possible to prevent mutual exclusion by using optimistic locking.
- Resource holding: A thread may release all its exclusive locks, when it does not succeed in obtaining all exclusive locks.
- No preemption: Using a timeout for an exclusive lock frees the lock after a given amount of time.
- Circular wait: When all exclusive locks are obtained by all threads in the same sequence, no circular wait occurs.

1.40 Is it possible to implement a deadlock detection?

When all exclusive locks are monitored and modelled as a directed graph, a deadlock detection system can search for two threads that are each waiting on the other thread to free a resource that it has locked. The waiting threads can then be forced by some kind of exception to release the lock the other thread is waiting on.

1.41 What is a livelock?

A livelock is a situation in which two or more threads block each other by responding to an action that is caused by another thread. In contrast to a deadlock situation, where two or more threads wait in one specific state, the threads that participate in a livelock change their state in a way that prevents progress on their regular work. An example would be a situation in which two threads try to acquire two locks, but release a lock they have acquired when they cannot acquire the second lock. It may now happen that both threads concurrently try to acquire the first thread. As only one thread succeeds, the second thread may succeed in acquiring the second lock. Now both threads hold two different locks, but as both want to have both locks, they release their lock and try again from the beginning. This situation may now happen again and again.

1.42 What do we understand by thread starvation?

Threads with lower priority get less time for execution than threads with higher priority. When the threads with lower priority performs a long enduring computations, it may happen that these threads do not get enough time to finish their computations just in time. They seem to "starve" away as threads with higher priority steal them their computation time.

1.43 Can a synchronized block cause thread starvation?

The order in which threads can enter a synchronized block is not defined. So in theory it may happen that in case many threads are waiting for the entrance to a synchronized block, some threads have to wait longer than other threads. Hence they do not get enough computation time to finish their work in time.

1.44 What do we understand by the term race condition?

A race condition describes constellations in which the outcome of some multi-threaded implementation depends on the exact timing behavior of the participating threads. In most cases it is not desirable to have such a kind of behavior, hence the term race condition also means that a bug due to missing thread synchronization leads to the differing outcome. A simple example for a race condition is the incrementation of an integer variable by two concurrent threads. As the operation consists of more than one single and atomic operation, it may happen that both threads read and increment the same value. After this concurrent incrementation the amount of the integer variable is not increased by two but only by one.

1.45 What do we understand by fair locks?

A fair lock takes the waiting time of the threads into account when choosing the next thread that passes the barrier to some exclusive resource. An example implementation of a fair lock is provided by the Java SDK: `java.util.concurrent.locks.ReentrantLock`. If the constructor with the boolean flag set to true is used, the ReentrantLock grants access to the longest-waiting thread.

1.46 Which two methods that each object inherits from `java.lang.Object` can be used to implement a simple producer/consumer scenario?

When a worker thread has finished its current task and the queue for new tasks is empty, it can free the processor by acquiring an intrinsic lock on the queue object and by calling the method `wait()`. The thread will be woken up by some producer thread that has put a new task into the queue and that again acquires the same intrinsic lock on the queue object and calls `notify()` on it.

1.47 What is the difference between `notify()` and `notifyAll()`?

Both methods are used to wake up one or more threads that have put themselves to sleep by calling `wait()`. While `notify()` only wakes up one of the waiting threads, `notifyAll()` wakes up all waiting threads.

1.48 How it is determined which thread wakes up by calling `notify()`?

It is not specified which threads will be woken up by calling `notify()` if more than one thread is waiting. Hence code should not rely on any concrete JVM implementation.

1.49 Is the following code that retrieves an integer value from some queue implementation correct?

```java
public Integer getNextInt() {
        Integer retVal = null;
        synchronized (queue) {
                try {
                        while (queue.isEmpty()) {
                                queue.wait();
                        }
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
        synchronized (queue) {
                retVal = queue.poll();
                if (retVal == null) {
                        System.err.println("retVal is null");
                        throw new IllegalStateException();
                }
        }
        return retVal;
}
```

Although the code above uses the queue as object monitor, it does not behave correctly in a multi-threaded environment. The reason for this is that it has two separate synchronized blocks. When two threads are woken up in line 6 by another thread that calls `notifyAll()`, both threads enter one after the other the second synchronized block. It this second block the queue has now only one new value, hence the second thread will poll on an empty queue and get null as return value.

1.50 Is it possible to check whether a thread holds a monitor lock on some given object?

The class `java.lang.Thread` provides the static method `Thread.holdsLock(Object)` that returns true if and only if the current thread holds the lock on the object given as argument to the method invocation.

1.51 What does the method `Thread.yield()` do?

An invocation of the static method `Thread.yield()` gives the scheduler a hint that the current thread is willing to free the processor. The scheduler is free to ignore this hint. As it is not defined which thread will get the processor after the invocation of `Thread.yield()`, it may even happen that the current thread becomes the "next" thread to be executed.

1.52 What do you have to consider when passing object instances from one thread to another?

When passing objects between threads, you will have to pay attention that these objects are not manipulated by two threads at the same time. An example would be a `Map` implementation whose key/value pairs are modified by two concurrent threads. In order to avoid problems with concurrent modifications you can design an object to be immutable.

1.53 Which rules do you have to follow in order to implement an immutable class?

- All fields should be final and private.
- There should be not setter methods.
- The class itself should be declared final in order to prevent subclasses to violate the principle of immutability.
- If fields are not of a primitive type but a reference to another object:
  - There should not be a getter method that exposes the reference directly to the caller.
  - Don’t change the referenced objects (or at least changing these references is not visisble to clients of the object).

1.54 What is the purpose of the class `java.lang.ThreadLocal`?

As memory is shared between different threads, `ThreadLocal` provides a way to store and retrieve values for each thread separately. Implementations of `ThreadLocal` store and retrieve the values for each thread independently such that when thread A stores the value A1 and thread B stores the value B1 in the same instance of `ThreadLocal`, thread A later on retrieves value A1 from this `ThreadLocal` instance and thread B retrieves value B1.

1.55 What are possible use cases for `java.lang.ThreadLocal`?

Instances of `ThreadLocal` can be used to transport information throughout the application without the need to pass this from method to method. Examples would be the transportation of security/login information within an instance of `ThreadLocal` such that it is accessible by each method. Another use case would be to transport transaction information or in general objects that should be accessible in all methods without passing them from method to method.

1.56 Is it possible to improve the performance of an application by the usage of multi-threading? Name some examples.

If we have more than one CPU core available, the performance of an application can be improved by multi-threading if it is possible to parallelize the computations over the available CPU cores. An example would be an application that should scale all images that are stored within a local directory structure. Instead of iterating over all images one after the other, a producer/consumer implementation can use a single thread to scan the directory structure and a bunch of worker threads that perform the actual scaling operation. Another example would be an application that mirrors some web page. Instead of loading one HTML page after the other, a producer thread can parse the first HTML page and issue the links it found into a queue. The worker threads monitor the queue and load the web pages found by the parser. While the worker threads wait for the page to get loaded completely, other threads can use the CPU to parse the already loaded pages and issue new requests.

1.57 What do we understand by the term scalability?

Scalability means the ability of a program to improve the performance by adding further resources to it.

1.58 Is it possible to compute the theoretical maximum speed up for an application by using multiple processors?

Amdahl's law provides a formula to compute the theoretical maximum speed up by providing multiple processors to an application. The theoretical speedup is computed by `S(n) =1 /(B + (1-B)/n)` where `n` denotes the number of processors and `B` the fraction of the program that cannot be executed in parallel. When n converges against infinity, the term `(1-B)/n` converges against zero. Hence the formula can be reduced in this special case to `1/B`. As we can see, the theoretical maximum speedup behaves reciprocal to the fraction that has to be executed serially. This means the lower this fraction is, the more theoretical speedup can be achieved.

1.59 What do we understand by lock contention?

Lock contention occurs, when two or more threads are competing in the acquisition of a lock. The scheduler has to decide whether it lets the thread, which has to wait sleeping and performs a context switch to let another thread occupy the CPU, or if letting the waiting thread busy-waiting is more efficient. Both ways introduce idle time to the inferior thread.

1.60 Which techniques help to reduce lock contention?

In some cases lock contention can be reduced by applying one of the following techniques:

- The scope of the lock is reduced.
- The number of times a certain lock is acquired is reduced (lock splitting).
- Using hardware supported optimistic locking operations instead of synchronization.
- Avoid synchronization where possible.
- Avoid object pooling.

1.61 Which technique to reduce lock contention can be applied to the following code?

```
synchronized (map) {
        UUID randomUUID = UUID.randomUUID();
        Integer value = Integer.valueOf(42);
        String key = randomUUID.toString();
        map.put(key, value);
}
```

The code above performs the computation of the random UUID and the conversion of the literal 42 into an Integer object within the synchronized block, although these two lines of code are local to the current thread and do not affect other threads. Hence they can be moved out of the synchronized block:

```
UUID randomUUID = UUID.randomUUID();
Integer value = Integer.valueOf(42);
String key = randomUUID.toString();
synchronized (map) {
        map.put(key, value);
}
```

1.62 Explain by an example the technique lock splitting.

Lock splitting may be a way to reduce lock contention when one lock is used to synchronize access to different aspects of the same application. Suppose we have a class that implements the computation of some statistical data of our application. A first version of this class uses the keyword synchronized in each method signature in order to guard the internal state before corruption by multiple concurrent threads. This also means that each method invocation may cause lock contention as other threads may try to acquire the same lock simultaneously. But it may be possible to split the lock on the object instance into a few smaller locks for each type of statistical data within each method. Hence thread T1 that tries to increment the statistical data D1 does not have to wait for the lock while thread T2 simultaneously updates the data D2.

1.63 What kind of technique for reducing lock contention is used by the SDK class ReadWriteLock?

The SDK class `ReadWriteLock` uses the fact that concurrent threads do not have to acquire a lock when they want to read a value when no other thread tries to update the value. This is implemented by a pair of locks, one for read-only operations and one for writing operations. While the read-only lock may be obtained by more than one thread, the implementation guarantees that all read operation see an updated value once the write lock is released.

1.64 What do we understand by lock striping?

In contrast to lock splitting, where we introduce different locks for different aspects of the application, lock striping uses multiple locks to guard different parts of the same data structure. An example for this technique is the class `ConcurrentHashMap` from JDK's `java.util.concurrent` package. The `Map` implementation uses internally different buckets to store its values. The bucket is chosen by the value's key. `ConcurrentHashMap` now uses different locks to guard different hash buckets. Hence one thread that tries to access the first hash bucket can acquire the lock for this bucket, while another thread can simultaneously access a second bucket. In contrast to a synchronized version of `HashMap` this technique can increase the performance when different threads work on different buckets.

1.65 What do we understand by a CAS operation?

CAS stands for compare-and-swap and means that the processor provides a separate instruction that updates the value of a register only if the provided value is equal to the current value. CAS operations can be used to avoid synchronization as the thread can try to update a value by providing its current value and the new value to the CAS operation. If another thread has meanwhile updated the value, the thread's value is not equal to the current value and the update operation fails. The thread then reads the new value and tries again. That way the necessary synchronization is interchanged by an optimistic spin waiting.

1.66 Which Java classes use the CAS operation?

The SDK classes in the package `java.util.concurrent.atomic` like `AtomicInteger` or `AtomicBoolean` use internally the CAS operation to implement concurrent incrementation.

```java
public class CounterAtomic {
        private AtomicLong counter = new AtomicLong();

        public void increment() {
                counter.incrementAndGet();
        }

        public long get() {
                return counter.get();
        }
}
```

1.67 Provide an example why performance improvements for single-threaded applications can cause performance degradation for multi-threaded applications.

A prominent example for such optimizations is a `List` implementation that holds the number of elements as a separate variable. This improves the performance for single-threaded applications as the `size()` operation does not have to iterate over all elements but can return the current number of elements directly. Within a multi-threaded application the additional counter has to be guarded by a lock as multiple concurrent threads may insert elements into the list. This additional lock can cost performance when there are more updates to the list than invocations of the `size()` operation.

1.68 Is object pooling always a performance improvement for multi-threaded applications?

Object pools that try to avoid the construction of new objects by pooling them can improve the performance of single-threaded applications as the cost for object creation is interchanged by requesting a new object from the pool. In multi-threaded applications such an object pool has to have synchronized access to the pool and the additional costs of lock contention may outweigh the saved costs of the additional construction and garbage collection of the new objects. Hence object pooling may not always improve the overall performance of a multi-threaded application.

1.69 What is the relation between the two interfaces Executor and ExecutorService?

The interface `Executor` only defines one method: `execute(Runnable)`. Implementations of this interface will have to execute the given Runnable instance at some time in the future. The `ExecutorService` interface is an extension of the `Executor` interface and provides additional methods to shut down the underlying implementation, to await the termination of all submitted tasks and it allows submitting instances of `Callable`.

1.70 What happens when you `submit()` a new task to an ExecutorService instance whose queue is already full?

As the method signature of `submit()` indicates, the `ExecutorService` implementation is supposed to throw a `RejectedExecutionException`.

1.71 What is a ScheduledExecutorService?

The interface `ScheduledExecutorService` extends the interface `ExecutorService` and adds method that allow to submit new tasks to the underlying implementation that should be executed a given point in time. There are two methods to schedule one-shot tasks and two methods to create and execute periodic tasks.

1.72 Do you know an easy way to construct a thread pool with 5 threads that executes some tasks that return a value?

The SDK provides a factory and utility class `Executors` whose static method `newFixedThreadPool(int nThreads)` allows the creation of a thread pool with a fixed number of threads (the implementation of `MyCallable` is omitted):

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        Future<Integer>[] futures = new Future[5];
        for (int i = 0; i < futures.length; i++) {
                futures[i] = executorService.submit(new MyCallable());
        }
        for (int i = 0; i < futures.length; i++) {
                Integer retVal = futures[i].get();
                System.out.println(retVal);
        }
        executorService.shutdown();
}
```

1.73 What is the difference between the two interfaces `Runnable` and `Callable`?

The interface `Runnable` defines the method `run()` without any return value whereas the interface `Callable` allows the method `run()` to return a value and to throw an exception.

1.74 Which are use cases for the class `java.util.concurrent.Future`?

Instances of the class `java.util.concurrent.Future` are used to represent results of asynchronous computations whose result are not immediately available. Hence the class provides methods to check if the asynchronous computation has finished, canceling the task and to the retrieve the actual result. The latter can be done with the two `get()` methods provided. The first `get()` methods takes no parameter and blocks until the result is available whereas the second `get()` method takes a timeout parameter that lets the method invocation return if the result does not get available within the given timeframe.

1.75 What is the difference between `HashMap` and `Hashtable` particularly with regard to thread-safety?

The methods of `Hashtable` are all synchronized. This is not the case for the `HashMap` implementation. Hence `Hashtable` is thread-safe whereas `HashMap` is not thread-safe. For single-threaded applications it is therefore more efficient to use the "newer" `HashMap` implementation.

1.76 Is there a simple way to create a synchronized instance of an arbitrary implementation of `Collection`, `List` or `Map`?

The utility class Collections provides the methods `synchronizedCollection(Collection)`, `synchronizedList(List)` and `synchronizedMap(Map)` that return a thread-safe collection/list/map that is backed by the given instance.

1.77 What is a semaphore?

A semaphore is a data structure that maintains a set of permits that have to be acquired by competing threads. Semaphores can therefore be used to control how many threads access a critical section or resource simultaneously. Hence the constructor of `java.util.concurrent.Semaphore` takes as first parameter the number of permits the threads compete about. Each invocation of its `acquire()` methods tries to obtain one of the available permits. The method `acquire()` without any parameter blocks until the next permit gets available. Later on, when the thread has finished its work on the critical resource, it can release the permit by invoking the method `release()` on an instance of Semaphore.

1.78 What is a `CountDownLatch`?

The SDK class `CountDownLatch` provides a synchronization aid that can be used to implement scenarios in which threads have to wait until some other threads have reached the same state such that all thread can start. This is done by providing a synchronized counter that is decremented until it reaches the value zero. Having reached zero the `CountDownLatch` instance lets all threads proceed. This can be either used to let all threads start at a given point in time by using the value 1 for the counter or to wait until a number of threads has finished. In the latter case the counter is initialized with the number of threads and each thread that has finished its work counts the latch down by one.

1.79 What is the difference between a `CountDownLatch` and a `CyclicBarrier`?

Both SDK classes maintain internally a counter that is decremented by different threads. The threads wait until the internal counter reaches the value zero and proceed from there on. But in contrast to the `CountDownLatch` the class `CyclicBarrier` resets the internal value back to the initial value once the value reaches zero. As the name indicates instances of `CyclicBarrier` can therefore be used to implement use cases where threads have to wait on each other again and again.

1.80 What kind of tasks can be solved by using the Fork/Join framework?

The base class of the Fork/Join Framework `java.util.concurrent.ForkJoinPool` is basically a thread pool that executes instances of `java.util.concurrent.ForkJoinTask`. The class `ForkJoinTask` provides the two methods `fork()` and `join()`. While `fork()` is used to start the asynchronous execution of the task, the method `join()` is used to await the result of the computation. Hence the Fork/Join framework can be used to implement divide-and-conquer algorithms where a more complex problem is divided into a number of smaller and easier to solve problems.

1.81 Is it possible to find the smallest number within an array of numbers using the Fork/Join-Framework?

The problem of finding the smallest number within an array of numbers can be solved by using a divide-and-conquer algorithm. The smallest problem that can be solved very easily is an array of two numbers as we can determine the smaller of the two numbers directly by one comparison. Using a divide-and-conquer approach the initial array is divided into two parts of equal length and both parts are provided to two instances of `RecursiveTask` that extend the class `ForkJoinTask`. By forking the two tasks they get executed and either solve the problem directly, if their slice of the array has the length two, or they again recursively divide the array into two parts and fork two new RecursiveTasks. Finally each task instance returns its result (either by having it computed directly or by waiting for the two subtasks). The root tasks then returns the smallest number in the array.

1.82 What is the difference between the two classes `RecursiveTask` and `RecursiveAction`?

In contrast to `RecursiveTask` the method `compute()` of `RecursiveAction` does not have to return a value. Hence `RecursiveAction` can be used when the action works directly on some data structure without having to return the computed value.

1.83 Is it possible to perform stream operations in Java 8 with a thread pool?

Collections provide the method `parallelStream()` to create a stream that is processed by a thread pool. Alternatively you can call the intermediate method `parallel()` on a given stream to convert a sequential stream to a parallel counterpart.

1.84 How can we access the thread pool that is used by parallel stream operations?

The thread pool used for parallel stream operations can be accessed by `ForkJoinPool.commonPool()`. This way we can query its level of parallelism with `commonPool.getParallelism()`. The level cannot be changed at runtime but it can be configured by providing the following JVM parameter: `-Djava.util.concurrent.ForkJoinPool.common.parallelism=5`.