1. 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.

2. What are the benefits of multi-threaded programming?

In Multi-Threaded programming, multiple threads are executing concurrently that improves the performance because CPU is not idle in case some thread is waiting to get some resources. Multiple threads share the heap memory, so it's good to create multiple threads to execute some task rather than creating multiple processes. For example, Servlets are better in performance than CGI because Servlet support multi-threading but CGI doesn't.

3. What is 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 shutdown the program and quits. A child thread created from daemon thread is also a daemon thread.

4. 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. Read this post to learn more about creating threads in java.

5. Thread Class vs Runnable Interface

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

6. What are different states in lifecycle of Thread?

When we create a Thread in java program, its state is New. Then we start the thread that change it's 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. Read this post to learn more about life cycle of thread.

7. 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 Thread, we need to start it using **Thread.start()** method.

8. 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.

9. What do you understand 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* whose value varies from 1 to 10 where 1 is the lowest priority thread and 10 is the highest priority thread.

10. What is Thread Scheduler and Time Slicing?

Thread Scheduler is the Operating System service that allocates the CPU time to the available runnable threads. Once we create and start a thread, it's execution depends on the implementation of Thread Scheduler. Time Slicing is the process to divide the available CPU time to the available runnable threads. Allocation of CPU time to threads can be based on thread priority or the thread waiting for longer time will get more priority in getting CPU time. Thread scheduling can't be controlled by java, so it's always better to control it from application itself.

11. What is context-switching in multi-threading?

Context Switching is the process of storing and restoring of CPU state so that Thread execution can be resumed from the same point at a later point of time. Context Switching is the essential feature for multitasking operating system and support for multi-threaded environment.

12. How can we make sure 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 is dead before finishing the main function. Here is an article about Thread join method.

13. 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. Check this post to learn more about thread wait, notify and notifyAll.

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

In Java every Object has a monitor 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.

15. Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?

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.

16. Difference between notify vs notifyAll in Java

Here is couple of main differences between notify and notifyAll method in Java :

1. First and main difference between notify() and notifyAll() method is that, if multiple threads is waiting on any lock in Java, notify method send notification to only one of waiting thread while notifyAll informs all threads waiting on that lock.

2. If you use notify method , It's not guaranteed that, which thread will be informed, but if you use notifyAll since all thread will be notified, they will compete for lock and the lucky thread which gets lock will continue. In a way, notifyAll method is safer because it sends notification to all threads, so if any thread misses the notification, there are other threads to do the job, while in the case of notify() method if the notified thread misses the notification then it could create subtle, hard to debug issues.

3. Some people argue that using notifyAll can drain more CPU cycles than notify itself but if you really want to sure that your notification doesn't get wasted by any reason, use notifyAll. Since wait method is called from the loop and they check condition even after waking up, calling notifyAll won't lead any side effect, instead it ensures that notification is not dropped.

17. 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’s 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.

18. What is the difference between wait and sleep in Java? (method)

One more classic Java multithreading question from the telephonic round of interviews. The key point to mention while answering this question is to mention that wait will release the lock and must be called from the synchronized context, while sleep will only pause the thread for some time and keep the lock.

By the way, both methods throw IntrupptedException

19. How do you solve producer consumer problem in Java?

One of my favorite questions during any Java multithreading interview, Almost half of the concurrency problems can be categorized in the producer-consumer pattern. There are basically two ways to solve this problem in Java, One by using wait and notify method and other by using BlockingQueue in Java. later is easy to implement and a good choice if you are coding in Java 5.

20. 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 and Thread safe classes. Learn more at thread safety tutorial.

21. What is volatile keyword in Java

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.

22. Which is more preferred – Synchronized method or Synchronized block?

Synchronized block is more preferred way because it doesn't lock the Object, synchronized methods lock the Object and if there are multiple synchronization blocks in the class, even though they are not related, it will stop them from execution and put them in wait state to get the lock on Object.

23. How to create daemon thread in Java?

Thread class setDaemon(true) can be used to create daemon thread in java. We need to call this method before calling start() method else it will throw IllegalThreadStateException.

24. 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 it's own ThreadLocal variable and they can use it's get() and set() methods to get the default value or change it's value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread. Check this post for small example program showing ThreadLocal Example.

25. What is Thread Group? Why it's advised not to use it?

ThreadGroup is a class which was intended to provide information about a thread group. ThreadGroup API is weak and it doesn't have any functionality that is not provided by Thread. Two of the major feature it had are to get the list of active threads in a thread group and to set the uncaught exception handler for the thread. But Java 1.5 has added *setUncaughtExceptionHandler(UncaughtExceptionHandler eh)* method using which we can add uncaught exception handler to the thread. So ThreadGroup is obsolete and hence not advised to use anymore.

```
t1.setUncaughtExceptionHandler(new UncaughtExceptionHandler(){

@Override
public void uncaughtException(Thread t, Throwable e) {
    System.out.println("exception occured:"+e.getMessage());
}

});
```

26. What is Java Thread Dump, How can we get Java Thread dump of a Program?

Thread dump is list of all the threads active in the JVM, thread dumps are very helpful in analyzing bottlenecks in the application and analyzing deadlock situations. There are many ways using which we can generate Thread dump – Using Profiler, Kill -3 command, jstack tool etc. I prefer jstack tool to generate thread dump of a program because it's easy to use and comes with JDK installation. Since it's a terminal based tool, we can create script to generate thread dump at regular intervals to analyze it later on. Read this post to know more about generating thread dump in java.

27. What is Deadlock? How to analyze and avoid deadlock situation?

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 analyze 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.

Avoid Nested Locks, Lock Only What is Required and Avoid waiting indefinitely are common ways to avoid deadlock situation, read this post to learn how to analyze deadlock in java with sample program.

28. What is Java Timer Class? How to schedule a task to run after specific interval?

java.util.Timer is a utility class that can be used to schedule a thread to be executed at certain time in future. Java Timer class can be used to schedule a task to be run one-time or to be run at regular intervals.

java.util.TimerTask is an **abstract class** that implements Runnable interface and we need to extend this class to create our own TimerTask that can be scheduled using java Timer class.

Check this post for java Timer example.

29. What is Thread Pool? How can we create Thread Pool in Java?

A thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed.

A thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue.

java.util.concurrent.Executors provide implementation of java.util.concurrent.Executor interface to create the thread pool in java. Thread Pool Example program shows how to create and use Thread Pool in java. Or read ScheduledThreadPoolExecutor Example to know how to schedule tasks after certain delay.

30. What will happen if we don't override Thread class run() method?

Thread class run() method code is as shown below.

```
public void run() {
    if (target != null) {
        target.run();
    }
}
```

Above target set in the init() method of Thread class and if we create an instance of Thread class asnew TestThread(), it's set to null. So nothing will happen if we don't override the run() method. Below is a simple example demonstrating this.

```
public class TestThread extends Thread {

  //not overriding Thread.run() method

  //main method, can be in other class too
  public static void main(String args[]){
         Thread t = new TestThread();
         System.out.println("Before starting thread");
         t.start();
         System.out.println("After starting thread");
  }
}
```

It will print only below output and terminate.

```
Before starting thread

After starting thread
```

31. Thread Contention

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

Java Concurrency Interview Questions and Answers

1. **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.

2. **What is AtomicInteger class and how it works internally**

Java 5 introduced java.util.concurrent.atomic package with a motive to provide:

A small toolkit of classes that support lock-free thread-safe programming on single variables.

AtomicInteger uses combination of volatile & CAS (compare and swap) to achieve thread-safety for Integer Counter. It is non-blocking in nature and thus highly usable in writing high throughput concurrent data structures that can be used under low to moderate thread contention.

Compare-And-Swap

In computer science, compare-and-swap (CAS) is an atomic instruction used in multi-threading to achieve synchronization. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value has been updated by another thread in the meantime, the write would fail.

https://en.wikipedia.org/wiki/Compare-and-swap

Since the compare-and-swap occurs (or appears to occur) instantaneously, if another process updates the location while we are in-progress, the compare-and-swap is guaranteed to fail.

3. How is volatile different from AtomicInteger

Read & write to volatile variables have same memory semantics as that of acquiring and releasing a monitor using synchronized code block. So the visibility of volatile field is guaranteed by the JMM (Java Memory Model).

AtomicInteger class stores its value field in a volatile variable, thus it is a decorator over the traditional volatile variable, but it provides unique non-blocking mechanism for updating the value after requiring the hardware level support for CAS (compare and set/swap).

Here is the implementation for getAndIncrement() method of AtomicInteger Class (as of Java 7).

Basic implementation of increment() operation using CAS

```
class AtomicInteger {

    public final int getAndIncrement() {
        for (;;) {
            int current = get();
            int next = current + 1;
            if (compareAndSet(current, next))
                return current;
        }
    }
    //Rest of the implementation
}
```

You can see that no lock is acquired to increment the value, rather CAS is used inside infinite loop to update the new value, that’s why it can be used to write scalable application where thread contention is low to medium.

General Advice

Use synchronized mechanism wherever thread contention is high i.e. multiple threads writes to the shared variable at the same point in time. Use AtomicInteger where thread contention is low to moderate to build highly scalable applications.

AtomicInteger class is best candidate for implementing thread-safe counters in your application.

4. **What is Lock interface in Java Concurrency API? What are it's benefits over synchronization?**

   Lock interface provide more extensive locking operations than can be obtained using synchronized methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated Condition objects. `ReentrantLock`
   The advantages of a lock are

   - it's possible to make them fair
   - it's possible to make a thread responsive to interruption while waiting on a Lock object.
   - it's possible to try to acquire the lock, but return immediately or after a timeout if the lock can't be acquired
   - it's possible to acquire and release locks in different scopes, and in different orders

   If you don't call the `unlock()` method at the end of the critical section, the other threads that are waiting for that block will be waiting forever, causing a deadlock situation.

   Read more at **Java Lock Example**.

5. **ThreadFactory - java.util.concurrent.ThreadFactory**

   The factory design pattern is one of the most used design patterns in the java. It is one of creational patterns and can be used to develop an object in demand of one or several classes. With this factory, we centralize the creation of objects.
   The centralization of creation logic brings us some advantages e.g.

   - It's easy to change the class of the objects created or the way we create these objects.
   - It's easy to limit the creation of objects for limited resources. For example, we can only have N objects of a type.
   - It's easy to generate statistical data about the creation of the objects.

   In java, we usually create threads using two ways i.e. extending thread class and implementing runnable interface. Java also provides an interface, the ThreadFactory interface, to create your own Thread object factory.

Here, the ThreadFactory interface has only one method called newThread(). It receives a Runnable object as a parameter and returns a Thread object. When you implement a ThreadFactory interface, you have to implement that interface and override this method.

6. What is Executors Framework?

In Java 5, Executor framework was introduced with the java.util.concurrent.Executor interface.

The Executor framework is a framework for standardizing invocation, scheduling, execution, and control of asynchronous tasks according to a set of execution policies.

Creating a lot many threads with no bounds to the maximum threshold can cause application to run out of heap memory. So, creating a ThreadPool is a better solution as a finite number of threads can be pooled and reused. Executors framework facilitate process of creating Thread pools in java. Check out this post to learn with example code to create thread pool using Executors framework.

```
//Executes only one thread
ExecutorService es = Executors.newSingleThreadExecutor();

//Internally manages thread pool of 2 threads
ExecutorService es = Executors.newFixedThreadPool(2);

//Internally manages thread pool of 10 threads to run scheduled tasks
ExecutorService es = Executors.newScheduledThreadPool(10);
```

We can create following 5 types of thread pool executors with pre-built methods in `java.util.concurrent.Executors` interface.

1. **Fixed thread pool executor** – Creates a thread pool that reuses a fixed number of threads to execute any number of tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. It is the best fit for most off the real-life use-cases.

   ```
   ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10);
   ```

2. **Cached thread pool executor** – Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. DO NOT use this thread pool if tasks are long-running. It can bring down the system if the number of threads goes beyond what the system can handle.

   ```
   ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();
   ```

3. **Scheduled thread pool executor** – Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newScheduledThreadPool(10);
```

4. **Single thread pool executor** – Creates single thread to execute all tasks. Use it when you have only one task to execute.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newSingleThreadExecutor();
```

5. **Work stealing thread pool executor** – Creates a thread pool that maintains enough threads to support the given parallelism level. Here parallelism level means the maximum number of threads which will be used to execute a given task, at a single point of time, in multi-processor machines.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newWorkStealingPool(4);
```

Java Thread Pool - Executor Framework in Java 5

```
public class ThreadPoolExample {

    public static void main(String args[]) {
       ExecutorService service = Executors.newFixedThreadPool(10);
       for (int i =0; i<100; i++){
           service.submit(new Task(i));
       }
    }

}

final class Task implements Runnable{
    private int taskId;

    public Task(int id){
        this.taskId = id;
    }

    @Override
    public void run() {
        System.out.println("Task ID : " + this.taskId +" performed by "
                           + Thread.currentThread().getName());
    }

}

Output:
Task ID : 0 performed by pool-1-thread-1
Task ID : 3 performed by pool-1-thread-4
Task ID : 2 performed by pool-1-thread-3
Task ID : 1 performed by pool-1-thread-2
Task ID : 5 performed by pool-1-thread-6
Task ID : 4 performed by pool-1-thread-5
```

7. What is the difference between submit() and execute() method of Executor and ExecutorService in Java? (answer)

The main difference between submit and execute method from ExecutorService interface is that former return a result in the form of a Future object, while later doesn't return a result. By the way, both are used to submit a task to thread pool in Java but one is defined in Executor interface,while other is added into ExecutorService interface.

8. **What is BlockingQueue? How can we implement Producer-Consumer problem using Blocking Queue?**

   java.util.concurrent.BlockingQueue is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.

   BlockingQueue doesn't accept null values and throw NullPointerException if you try to store null value in the queue.

   BlockingQueue implementations are thread-safe. All queuing methods are atomic in nature and use internal locks or other forms of concurrency control.

   BlockingQueue interface is part of java collections framework and it's primarily used for implementing producer consumer problem.
   Check this post for producer-consumer problem implementation using BlockingQueue.

9. **Java DelayQueue**

   DelayQueue class is an unbounded blocking queue of delayed elements, in which an element can only be taken when its delay has expired. `DelayQueue` class is part of `java.util.concurrent` package.

10. **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.
Check this post for Callable Future Example.

11. 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 real 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. Check out **Java FutureTask Example** post to learn how to use it and what are different methods it has.

12. What are Concurrent Collection Classes?

Java Collection classes are fail-fast which means that if the Collection will be changed while some thread is traversing over it using iterator, the iterator.next() will throw ConcurrentModificationException.

Concurrent Collection classes support full concurrency of retrievals and adjustable expected concurrency for updates.
Major classes are ConcurrentHashMap, CopyOnWriteArrayList and CopyOnWriteArraySet, check this post to learn how to avoid ConcurrentModificationException when using iterator.

13. What is Executors Class?

Executors class provide utility methods for Executor, ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes.

Executors class can be used to easily create Thread Pool in java, also this is the only class supporting execution of Callable implementations.

14. What are some of the improvements in Concurrency API in Java 8?

Some important concurrent API enhancements are:

- ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
- CompletableFuture that may be explicitly completed (setting its value and status).

- Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

15. What is the difference between CountDownLatch and CyclicBarrier in Java?

The CountDownLatch and CyclicBarrier in Java are two important concurrency utility which is added on Java 5 Concurrency API. Both are used to implement scenario, where one thread has to wait for other thread before starting processing but there is a difference between them.

The key point to mention, while answering this question is that CountDownLatch is not reusable once the count reaches to zero, while CyclicBarrier can be reused even after the barrier is broken.

16. What is Busy Spinning? Why will you use Busy Spinning as wait strategy? (answer)

The busy waiting is a wait strategy, where one thread wait for a condition to become true, but instead of calling wait or sleep method and releasing CPU, it just spins. This is particularly useful if the condition is going to be true quite quickly i.e. in a millisecond or microsecond.

The advantage of not releasing CPU is that all cached data and instruction remain unaffected, which may be lost, had this thread is suspended on one core and brought back to another thread. If you can answer this question, that rest assure of a good impression.

17. Why is ConcurrentHashMap faster than Hashtable in Java? (answer)

ConcurrentHashMap is introduced as an alternative of Hashtable in Java 5, it is faster because of its design. ConcurrentHashMap divides the whole map into different segments and only lock a particular segment during the update operation, instead of Hashtable, which locks whole Map.

The ConcurrentHashMap also provides lock-free read, which is not possible in Hashtable, because of this and lock striping, ConcurrentHashMap is faster than Hashtable, especially when a number of the reader is more than the number of writers.

18. How do you share data between two threads in Java?

You can share data between thread by using shared object or shared data structures like Queue. Depending upon, what you are using, you need to provide the thread-safety guarantee, and one way of providing thread-safety is using synchronized keyword.

If you use concurrent collection classes from Java 5 e.g. BlockingQueue, you can easily share data without being bothered about thread safety and inter-thread communication. I like this thread question, because of it's simplicity and effectiveness. This also leads further follow-up questions on issues which arise due to sharing data between threads e.g. race conditions.

19. What is ReentrantLock in Java? Have you used it before?

ReentrantLock is an alternative of synchronized keyword in Java, it is introduced to handle some of the limitations of synchronized keywords. Many concurrency utility classes and concurrent collection classes from Java 5, including ConcurrentHashMap uses ReentrantLock, to leverage optimization.

20. What is ReadWriteLock in Java? What is the benefit of using ReadWriteLock in Java?

The ReadWriteLock is again based upon the concept of lock striping, one of the advance thread-safety mechanism which advocates separating locks for reading and writing operations

21. Difference between ReentrantLock and synchronized keyword in Java

Though ReentrantLock provides same visibility and orderings guaranteed as implicit lock, acquired by synchronized keyword in Java, it provides more functionality and differ in certain aspect. As stated earlier, main difference between synchronized and ReentrantLock is ability to trying for lock interruptibly, and with timeout. Thread doesn’t need to block infinitely, which was the case with synchronized. Let’s see few more differences between synchronized and Lock in Java.

1) Another significant difference between ReentrantLock and synchronized keyword is fairness. synchronized keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property, while creating instance of ReentrantLock. Fairness property provides lock to longest waiting thread, in case of contention.

2) Second difference between synchronized and Reentrant lock is tryLock() method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or

not held by any other thread. This reduce blocking of thread waiting for lock in Java application.

3) One more worth noting difference between ReentrantLock and synchronized keyword in Java is, ability to interrupt Thread while waiting for Lock. In case of synchronized keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. ReentrantLock provides a method called lockInterruptibly(), which can be used to interrupt thread when it is waiting for lock. Similarly tryLock() with timeout can be used to timeout if lock is not available in certain time period.

4) ReentrantLock also provides convenient method to get List of all threads waiting for lock.

22.Semaphore

Use Semaphore to control thread access to resource.

A Semaphore is a thread synchronization construct that can be used either to send signals between threads to avoid **missed signals**, or to guard a **critical section** like you would with a **lock**. Java 5 comes with semaphore implementations in the java.util.concurrent package so you don't have to implement your own semaphores. Still, it can be useful to know the theory behind their implementation and use.

Java 5 comes with a built-in Semaphore so you don't have to implement your own. You can read more about it in the **java.util.concurrent.Semaphore** text, in my java.util.concurrent tutorial.

Simple Semaphore

```
public class Semaphore {
  private boolean signal = false;

  public synchronized void take() {
    this.signal = true;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(!this.signal) wait();
    this.signal = false;
  }

}
```

The take() method sends a signal which is stored internally in the Semaphore.
The release() method waits for a signal. When received the signal flag is cleared again, and the release() method exited.

Using a semaphore like this you can avoid missed signals. You will call take() instead of notify() and release() instead of wait(). If the call to take() happens before the call to release() the thread calling release() will still know that take() was called, because the signal is stored internally in the signal variable. This is not the case with wait() and notify().

The names take() and release() may seem a bit odd when using a semaphore for signaling. The names origin from the use of semaphores as locks, as explained later in this text. In that case the names make more sense.

Using Semaphores for Signaling

```
Semaphore semaphore = new Semaphore();

SendingThread sender = new SendingThread(semaphore);

ReceivingThread receiver = new ReceivingThread(semaphore);

receiver.start();
sender.start();
public class SendingThread {
  Semaphore semaphore = null;

  public SendingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
    while(true){
      //do something, then signal
      this.semaphore.take();

    }
  }
}
public class RecevingThread {
  Semaphore semaphore = null;

  public ReceivingThread(Semaphore semaphore){
    this.semaphore = semaphore;
  }

  public void run(){
    while(true){
      this.semaphore.release();
      //receive signal, then do something...
    }
  }
}
```

Counting Semaphore

The Semaphore implementation in the previous section does not count the number of signals sent to it by take() method calls. We can change the Semaphore to do so. This is called a counting semaphore. Here is a simple implementation of a counting semaphore:

```
public class CountingSemaphore {
  private int signals = 0;

  public synchronized void take() {
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
  }

}
```

Bounded Semaphore

The CoutingSemaphore has no upper bound on how many signals it can store. We can change the semaphore implementation to have an upper bound, like this:

```
public class BoundedSemaphore {
  private int signals = 0;
  private int bound   = 0;

  public BoundedSemaphore(int upperBound){
    this.bound = upperBound;
  }

  public synchronized void take() throws InterruptedException{
    while(this.signals == bound) wait();
    this.signals++;
    this.notify();
  }

  public synchronized void release() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
    this.notify();
  }
}
```

Notice how the take() method now blocks if the number of signals is equal to the upper bound. Not until a thread has called release() will the thread calling take() be allowed to deliver its signal, if the BoundedSemaphore has reached its upper signal limit.

Using Semaphores as Locks

It is possible to use a bounded semaphore as a lock. To do so, set the upper bound to 1, and have the call to take() and release() guard the critical section. Here is an example:

```
BoundedSemaphore semaphore = new BoundedSemaphore(1);

...

semaphore.take();
```

```
try{
  //critical section
} finally {
  semaphore.release();
}
```

In contrast to the signaling use case the methods take() and release() are now called by the same thread. Since only one thread is allowed to take the semaphore, all other threads calling take() will be blocked until release() is called. The call to release() will never block since there has always been a call to take() first.

You can also use a bounded semaphore to limit the number of threads allowed into a section of code. For instance, in the example above, what would happen if you set the limit of the BoundedSemaphore to 5? 5 threads would be allowed to enter the critical section at a time. You would have to make sure though, that the thread operations do not conflict for these 5 threads, or you application will fail.

The relase() method is called from inside a finally-block to make sure it is called even if an exception is thrown from the critical section.

23. Mutex

Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section. This thread only releases the Mutex when it exits the critical section.

24. Mutex vs Semaphore

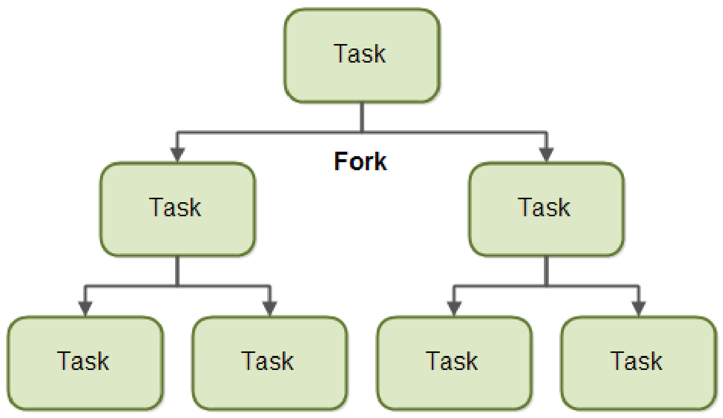
As per operating system terminology, mutex and semaphore are kernel resources that provide synchronization services (also called as synchronization primitives).

25. FORK JOIN

The ForkJoinPool was added to Java in Java 7. The ForkJoinPool is similar to the **Java ExecutorService** but with one difference. The ForkJoinPool makes it easy for tasks to split their work up into smaller tasks which are then submitted to the ForkJoinPool too. Tasks can keep splitting their work into smaller subtasks for as long as it makes to split up the task.

Fork

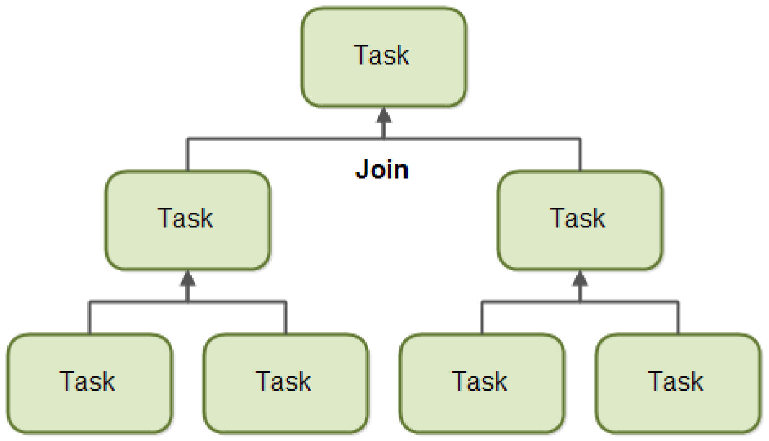
A task that uses the fork and join principle can *fork* (split) itself into smaller subtasks which can be executed concurrently. This is illustrated in the diagram below:



By splitting itself up into subtasks, each subtask can be executed in parallel by different CPUs, or different threads on the same CPU.
A task only splits itself up into subtasks if the work the task was given is large enough for this to make sense. There is an overhead to splitting up a task into subtasks, so for small amounts of work this overhead may be greater than the speedup achieved by executing subtasks concurrently.
The limit for when it makes sense to fork a task into subtasks is also called a threshold. It is up to each task to decide on a sensible threshold. It depends very much on the kind of work being done.

Join

When a task has split itself up into subtasks, the task waits until the subtasks have finished executing.
Once the subtasks have finished executing, the task may *join* (merge) all the results into one result. This is illustrated in the diagram below:



Of course, not all types of tasks may return a result. If the tasks do not return a result then a task just waits for its subtasks to complete. No result merging takes place then.

The ForkJoinPool

The ForkJoinPool is a special thread pool which is designed to work well with fork-and-join task splitting. The ForkJoinPool located in the java.util.concurrent package, so the full class name is java.util.concurrent.ForkJoinPool.

Creating a ForkJoinPool

You create a ForkJoinPool using its constructor. As a parameter to the ForkJoinPool constructor you pass the indicated level of parallelism you desire. The parallelism level indicates how many threads or CPUs you want to work concurrently on on tasks passed to the ForkJoinPool. Here is a ForkJoinPool creation example:

```
ForkJoinPool forkJoinPool = new ForkJoinPool(4);
```

This example creates a ForkJoinPool with a parallelism level of 4.

Submitting Tasks to the ForkJoinPool

You submit tasks to a ForkJoinPool similarly to how you submit tasks to an ExecutorService. You can submit two types of tasks. A task that does not return any result (an "action"), and a task which does return a result (a "task"). These two types of tasks are represented by the RecursiveAction and RecursiveTask classes. How to use both of these tasks and how to submit them will be covered in the following sections.

RecursiveAction

A RecursiveAction is a task which does not return any value. It just does some work, e.g. writing data to disk, and then exits.
A RecursiveAction may still need to break up its work into smaller chunks which can be executed by independent threads or CPUs.
You implement a RecursiveAction by subclassing it. Here is a RecursiveAction example:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveAction;

public class MyRecursiveAction extends RecursiveAction {

    private long workLoad = 0;

    public MyRecursiveAction(long workLoad) {
        this.workLoad = workLoad;
    }
    @Override
    protected void compute() {

        //if work is above threshold, break tasks up into smaller tasks
        if(this.workLoad > 16) {
            System.out.println("Splitting workLoad : " + this.workLoad);

            List<MyRecursiveAction> subtasks =
                new ArrayList<MyRecursiveAction>();

            subtasks.addAll(createSubtasks());

            for(RecursiveAction subtask : subtasks){
                subtask.fork();
            }

        } else {
            System.out.println("Doing workLoad myself: " + this.workLoad);
        }
    }

    private List<MyRecursiveAction> createSubtasks() {
        List<MyRecursiveAction> subtasks =
            new ArrayList<MyRecursiveAction>();

        MyRecursiveAction subtask1 = new MyRecursiveAction(this.workLoad / 2);
        MyRecursiveAction subtask2 = new MyRecursiveAction(this.workLoad / 2);

        subtasks.add(subtask1);
        subtasks.add(subtask2);

        return subtasks;
    }

}
```

This example is very simplified. The MyRecursiveAction simply takes a fictive workLoad as parameter to its constructor. If the workLoad is above a certain threshold, the work is split into subtasks which are also scheduled for execution (via the .fork() method of the subtasks. If the workLoad is below a certain threshold then the work is carried out by the MyRecursiveAction itself.
You can schedule a MyRecursiveAction for execution like this:

```
MyRecursiveAction myRecursiveAction = new MyRecursiveAction(24);

forkJoinPool.invoke(myRecursiveAction);
```

RecursiveTask

A RecursiveTask is a task that returns a result. It may split its work up into smaller tasks, and merge the result of these smaller tasks into a collective result. The splitting and merging may take place on several levels. Here is a RecursiveTask example:

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;


public class MyRecursiveTask extends RecursiveTask<Long> {

    private long workLoad = 0;

    public MyRecursiveTask(long workLoad) {
        this.workLoad = workLoad;
    }

    protected Long compute() {

        //if work is above threshold, break tasks up into smaller tasks
        if(this.workLoad > 16) {
            System.out.println("Splitting workLoad : " + this.workLoad);

            List<MyRecursiveTask> subtasks =
                new ArrayList<MyRecursiveTask>();
            subtasks.addAll(createSubtasks());

            for(MyRecursiveTask subtask : subtasks){
                subtask.fork();
            }

            long result = 0;
            for(MyRecursiveTask subtask : subtasks) {
                result += subtask.join();
            }
            return result;

        } else {
            System.out.println("Doing workLoad myself: " + this.workLoad);
            return workLoad * 3;
        }
    }

    private List<MyRecursiveTask> createSubtasks() {
        List<MyRecursiveTask> subtasks =
        new ArrayList<MyRecursiveTask>();

        MyRecursiveTask subtask1 = new MyRecursiveTask(this.workLoad / 2);
        MyRecursiveTask subtask2 = new MyRecursiveTask(this.workLoad / 2);

        subtasks.add(subtask1);
        subtasks.add(subtask2);

        return subtasks;
    }
}
```

```java
MyRecursiveTask myRecursiveTask = new MyRecursiveTask(128);

long mergedResult = forkJoinPool.invoke(myRecursiveTask);

System.out.println("mergedResult = " + mergedResult);
```

Notice how you get the final result out from the ForkJoinPool.invoke() method call.

Difference between Executor Framework and ForkJoinPool in Java?

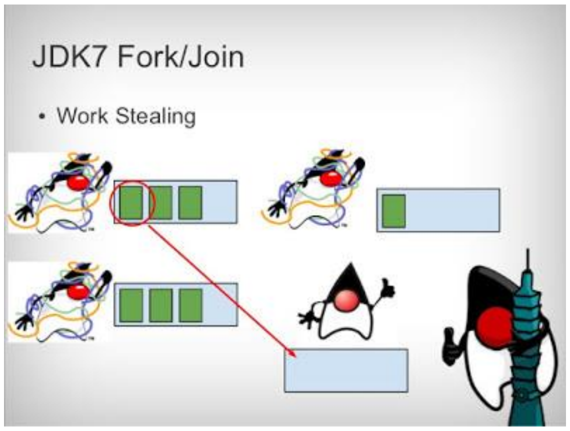
The Job of that thread pool is to accept the task and execute if there is free worker thread available, but ForJoinPool is a special kind of thread pool. They use **work stealing pattern**. All threads in a fork join pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks.

This enables efficient processing when most tasks spawn other subtasks (as do most ForkJoinTasks e.g. recursive actions), as well as when many small tasks are submitted to the pool from external clients.

In short, the *main difference between Executor framework and ForkJoinPoo*l is that former provides a **general purpose thread pool**, while later provides a **special implementation** which uses work stealing pattern for efficient processing of ForkJoinTask. Let's see a couple of more differences to answer this question better.
ForkJoinPool vs Executor Framework

1) The main difference between ForkJoinPool and ThreadPoolExecutor is that ForkJoinPool is designed to accept and execute ForkJoinTask, which is a lightweight version of FutureTask, while ThreadPoolExecutor is designed to provide a normal thread pool which executes each submitted task using one of possibly several pooled threads.

2) Another key difference between ThreadPoolExecutor and ForkJoinPool class is that ForkJoinPool uses work stealing pattern, which means one thread can also execute a pending task from another thread. This improves efficient in case of ForkJoinTask as most of ForkJoinTask algorithm spawn new tasks. You can further read Java Concurrency in Practice by Brian Goetz to learn more about work stealing pattern and other parallel computing patterns.



Callable VS Runnable Interface?

Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.

Callable.java

```
public interface Callable {

   V call() throws Exception;
}
```

In order to convert Runnable to Callable use the following utility method provided by Executors class

Convert Runnable to Callable

```
Callable callable = Executors.callable(Runnable task);
```

Callable, however must be executed using a ExecutorService instead of Thread as shown below.

Convert Runnable to Callable

```
Callable<String> aCallable = () -> "dummy";
ExecutorService executorService = Executors.newSingleThreadExecutor();
final Future<String> future = executorService.submit(aCallable);
final String result = future.get();
```

Submitting a callable to ExecutorService returns Future Object which represents the lifecycle of a task and provides methods to check if the task has been completed or cancelled, retrieve the results and cancel the task.
Here is the source for Future Interface

java/util/concurrent/Future.java

```
public interface Future {
    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled();

    boolean isDone();

    V get() throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit) throws InterruptedException,
ExecutionException, TimeoutException;
}
```

What is need for both the Interfaces (callable & Runnable)?

In my opinion - In theory, the Java team could have changed the signature of the Runnable.run() method, but this would have broken binary compatibility with pre 1.5 Java code, requiring re-coding when migrating old Java code to newer JVMs. That is a BIG NO-NO. Java strives to be backwards compatible … and that's been one of Java's biggest selling points for business computing.

Summary of differences between Runnable and Callable

1. A Runnable does not return a result
2. A Runnable can't throw checked Exception, while callable can.
3. Runnable cannot be parametrized while Callable is a parametrized type whose type parameter indicates the return type of its run method
4. Runnable has run() method while Callable has call() method
5. Runnable introduced in Java 1.0 while callable was added in Java 5

ThreadLocal in Java, where will you use this class

ThreadLocal provides a mechanism of maintaining thread confinement for a given object. which allows you to associate a per-thread value with a value-holding object.

ThreadLocal Java Docs

ThreadLocal class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable.

https://docs.oracle.com/javase/8/docs/api/java/lang/ThreadLocal.html#withInitial-java.util.function.Supplier

Pattern to use ThreadLocal

ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

For example, the class below generates counters local to each thread. A new copy ThreadLocalCounter is assigned the first time it invokes ThreadLocalCounter.get() and remains unchanged on subsequent calls.

ThreadLocalCounter class

```
public class ThreadLocalCounter {
    private int count;
    public ThreadLocalCounter(int count) {
        this.count = count;
    }

    public int increment() {
        return ++count;
    }

    private static final ThreadLocal<ThreadLocalCounter> threadLocal =
ThreadLocal.withInitial(() -> new ThreadLocalCounter(0));

    public static ThreadLocalCounter get() {
        return threadLocal.get();
    }
}
```

This ThreadLocalCounter can be used by multiple threads to keep their own counter copy which is not visible to other threads.

ThreadLocalCounter

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        ThreadLocalCounter threadLocalCounter = ThreadLocalCounter.get();
        while(true) {
            final int counter = threadLocalCounter.increment();
            doSomeWork(counter);
        }
    }

    private void doSomeWork(int counter) {
        //Dummy Work
    }
});
thread.start();
```

Garbage Collection Behaviour

Each thread holds an implicit reference to its copy of a thread-local variable as long as the thread is alive and the ThreadLocal instance is accessible; after a thread goes away, all of its copies of thread-local instances are subject to garbage collection (unless other references to these copies exist).

Benefits of using ThreadLocal

1. It makes multi-threading easy by not sharing the state of an object across threads. Since state is not shared so you don't need to synchronize it with other requests. Please be noted that ThreadLocal is not a substitute of synchronization, instead it isolates an object from being exposed to multiple threads. Something like this:

   Servlet filter with ThreadLocal

```
 doGet(HttpServletRequest req, HttpServletResponse resp) {
    User user = getLoggedInUser(req);
    StaticClass.getThreadLocal().set(user)
    try {
      doSomething()
      doSomethingElse()
      renderResponse(resp)
    }
    finally {
      StaticClass.getThreadLocal().remove()
    }
  }
```

Now any code that requires access to current user object, can obtain the reference using below code:

User user = StaticClass.getThreadLocal().get()
where Static class is defined as:

```
class StaticClass {
  static private ThreadLocal threadLocal = new ThreadLocal<User>();

  static ThreadLocal<User> getThreadLocal() {
    return threadLocal;
  }
}
```

Usecase

1. ThreadLocal is ideal for storing objects that are not thread-safe and object sharing across threads is not required. A good example is Hibernate Session which is not threadsafe and must not be shared across threads, so we can put Session into ThreadLocal and execute the transaction. In a servlet environment, this can happen in a filter which creates a new session for each request and commit the session after request is complete. Similar approach can be taken for JDBC connection.
2. SimpleDataFormat is not a thread-safe class, so you can use ThreadLocal to keep a copy of it per thread, thus avoiding the need for synchronization. The other option could be to create a new object on each invocation which requires more resources compared to ThreadLocal approach.

   ThreadSafeDateFormat

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class ThreadSafeDateFormat {
    //SimpleDateFormat is not thread-safe, so give one instance to each thread
    private static final ThreadLocal<SimpleDateFormat> formatter =
ThreadLocal.withInitial(() -> new SimpleDateFormat("yyyy.MM.dd HHmm a"));

    public static String formatIt(Date date) {
        return formatter.get().format(date);
    }

    public static void main(String[] args) {
        final String formatIt = ThreadSafeDateFormat.formatIt(new Date());
        System.out.println("formatIt = " + formatIt);
    }
}
```

3. ThreadLocal is very useful in web applications, a typical pattern is to store the state of web request in ThreadLocal (usually in a servlet filter or spring interceptor) at the very start of the processing and then access this state from any component involved in request processing. Normally all the processing of a web request happens in a single thread. In-fact ThreadLocal is widely used in implementing application frameworks. For example, J2EE containers associate a transaction context with an executing thread for the duration of an EJB call. This is implemented using a static Thread-Local holding the transaction context.
4. In creating a application that requires thread level stats collections for e.g. stress testing apps, performance monitoring app.

Issues with ThreadLocal

1. ThreadLocal introduces hidden coupling among classes, which makes them hard to test and debug. So it should be used with care.
2. It is easy to abuse ThreadLocal by treating its thread confinement property as a license to use global variables or as a means of creating “hidden” method arguments.

Threading Jargon in Java

Race Condition

A race condition occurs when the correctness of a computation depends on the relative timing of multiple threads by the runtime. In this scenario Getting the right result relies on the lucky timings.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don’t know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act".

For example, in the below code, if another thread changed the x in between point 1 and point 2, then y will not be equal to 10.

```
if (x == 5) // The "Check"
{
   y = x * 2; // The "Act"

   // If another thread changed x in between "if (x == 5)" and "y = x * 2"
above,
   // y will not be equal to 10.
}
```

Race condition can be avoided by proper understanding of Java Memory Model. Properly synchronizing the shared resource will not allow more than one thread to alter the state of shared resource in indeterministic manner.

Dead Lock

Dead lock occurs when two or more threads are blocked forever, waiting for each other to release up the shared resource. For two threads, it happens when two threads have a circular dependency on a pair of synchronized shared resources.

What is deadlock in java and how to troubleshoot it

Starvation

Describes a situation where a thread is unable to gain regular access to shared resource and is unable to make any 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.

Mutex

Mutex stands for mutually exclusive, only one kind of operation (READ or WRITE) is allowed at a given time frame.

Live Lock

A thread often acts in response to the action of other threads, if the other thread's action is also in response to another thread, then live lock may result. No progress but threads are not blocked.

Synchronizer

A synchronizer is any object that coordinates the control of flow of threads based on its state. For example, semaphore, CountDownLatch, FutureTask, Exchanger, CyclicBarrier, etc.

Latch

A synchronizer that can delay the progress of threads until it reaches the terminal state.

Semaphore

Counting semaphore are used to control the number of threads that can access a certain shared resource or perform a given action at the same time. Semaphores are normally used to implement resource pool or to impose a bound on a collection.

Example Usecases -

1. Considering iText PDF conversion to be a cpu intensive task, do not allow more than 10 itext pdf conversions to run at any given time.
2. There is a arbitrary algorithm in your program that takes lot of memory, do not allow more than 2 instances of this algorithm to run in parallel otherwise JVM will start throwing Out Of Memory Errors.

Exchanger

A two party barrier in which the parties exchange data at the barrier point.

How will you handle ConcurrentModificationException

ConcurrentModificationException is raised by fail-fast iterators when the underlying collection is modified structurally during iteration.

In the face of concurrent modification, the fail-fast iterator fails quickly and cleanly by throwing ConcurrentModificationException, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

There are two main scenarios when this exception can occur:

1. Collection is modified structurally while fail-fast iterator is iterating through it in the same thread.
2. One thread modifies the structure of collection, while other thread is iterating through it using fail-fast iterator.

There are two ways to handle this scenario:

1. Do not allow modification in underlying collection during the iteration.
   - Use synchronization mechanism to prevent any other thread from accessing the collection concurrently.
   - Do not use `Collection.remove()` method instead always use `iterator.remove()` method for any structural modification.
2. Do not use *fail-fast* iterators, instead use fail-safe iterators using concurrent collections. For example, ConcurrentHashMap instead of HashMap, CopyOnWriteArrayList instead of ArrayList, etc. Fail-safe iterators do not throw this exception

Double Checked Locking Problem in Multi-Threading?

Double-Checked Locking Problem

In earlier times (prior to JDK 1.6) a simple un-contended synchronization block was expensive and that lead many people to write double-checked locking to write lazy initialization code. The double-checked locking idiom tries to improve performance by avoiding synchronization over the common code path after the helper is allocated. But the DCL never worked because of the limitations of previous JMM.

Not Thread safe: JMM will not guarantee the expected execution of this static singleton.

```
public class Singleton {
    private Singleton() {}
    private static Singleton instance_ = null;

    public static Singleton instance() {
        if (instance_ == null) {
            synchronized(Singleton.class) {
                if (instance_ == null)
                    instance_ = new Singleton();
            }
        }
        return instance_;
    }
}
```

A global static variable that will hold the state

unsynchronized access to this fields may see partially constructed objects because of instruction reordering by the compiler or the cache

This is now fixed by new JMM (JDK 1.5 onwards) using volatile keyword.

Why above code idiom is broken in current JMM ?

DCL relies on the un synchronized use of _instance field. This appears harmless, but it is not. Suppose Thread A is inside synchronized block and it is creating new Singleton instance and assigning to _instance variable, while thread B is just entering the getInstance() method. Consider the effect on memory of this initialization. Memory for the new Singleton object will be allocated; the constructor for Singleton will be called, initializing the member fields of the new object; and the field resource of SomeClass will be assigned a reference to the newly created object. There could be two scenarios now

1. Suppose Thread A has completed initialization of _instance and exits synchronized block as thread B enters getInstance(). By this time, the _instance is fully initialized and Thread A has flushed its local memory to main memory (write barriers). Singleton's member fields may refer other objects stored

in memory which will also be flushed out.. While Thread B may see a valid reference to the newly created _instance, but because it didn't perform a read barrier, it could still see stale values of _instance's member fields.

2. Since thread B is not executing inside a synchronized block, it may see these memory operations in a different order than the one thread A executes. It could be the case that B sees these events in the following order (and the compiler is also free to reorder the instructions like this): allocate memory, assign reference to resource, call constructor. Suppose thread B comes along after the memory has been allocated and the resource field is set, but before the constructor is called. It sees that resource is not null, skips the synchronized block, and returns a reference to a partially constructed Resource! Needless to say, the result is neither expected nor desired.

Fixed double-checked Locking using volatile in new JMM (multi-threaded singleton pattern JDK 1.5)

The following code makes the helper volatile so as to stop the instruction reordering. This code will work with JDK 1.5 onwards only.

Thread-safe version using volatile

```
class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {

            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }

}
```

If Helper is an immutable object, such that all of the fields of Helper are final, then double-checked locking will work without having to use volatile fields. The idea is that a reference to an immutable object (such as a String or an Integer) should behave in much the same way as an int or float; reading and writing references to immutable objects are atomic.

Better Alternatives to DCL

Now a days JVM is much smarter and the relative expense of synchronized block over volatile is very less, so it does not really make sense to use DCL for performance reasons. The easiest way to avoid DCL is to avoid it. We can make the whole method synchronized instead of making the code block synchronized.

Another option is to use eager initialization instead of lazy initialization by assigning at the creation time Here is the example demonstrating eager initialization

1. Eager instantiation using static keyword

Eagerly instantiated singleton

```
class MySingleton {
    public static Resource resource = new Resource();
}
```

2. Using Initialization On Demand Holder idiom

Inner classes are not loaded until they are referenced. This fact can be used to utilize inner classes for lazy initialization as shown below

On demand holder idiom for thread-safe Singelton

```
public class Something {
    private Something() {}
    private static class LazyHolder {
        private static final Something INSTANCE = new Something();
    }
    public static Something getInstance() {
        return LazyHolder.INSTANCE;
    }
}
```

3. Enum for Thread-Safe

And finally we can use Enum developing for Thread-Safe Singleton class.

Thread-safe singleton using enum

```
public enum Singleton{
    INSTANCE;
}
```

Producer Consumer Problem using Blocking Queue

Producer Consumer Problem

Producer-consumer problem is a classic example of multi-threading synchronization problem. It is a must to know problem if you want to delve into Java concurrency & mutli-threading concepts.

1. Problem Description

The problem describes two entities, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time and processing it. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer, and at the same time ensure the thread-safety.

2. Steps to Solve problem

1. The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again.
2. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
3. while doing all of this, ensure the thread-safety.

Did you know?

JMS (Java Messaging Service) is a implementation of Producer Consumer problem. Multiple Producers & multiple Consumers can connect to JMS and distribute the work.

3. Solving Producer Consumer Problem in Core Java

There are multiple ways to solve this problem -

1. Semaphores can be used to coordinate b/w producer and consumers.
2. Using synchronization to solve the problem.
3. Using non-blocking algorithms to solve this problem.
4. **Using BlockingQueue to solve the problem.**

In this article we will focus only on blocking key approach.

Interviewers are mostly interested in solving producer-consumer problem from scratch to evaluate your multi-threading skills, so we will implement a simple version of blocking queue from scratch.

To solve this we need three different components:

1. A blocking queue
2. Producer thread(s)
3. Consumer thread(s)

We can implement our own simple thread-safe version of BlockingQueue using synchronization, as shown in below code:

BlockingQueue (Using Intrinsic Locking)

```
package com.shunya;

class BlockingQueue {
    final Object[] items = new Object[100];
    int putptr, takeptr, count;
    private boolean closed = false;

    public synchronized void put(Object x) throws InterruptedException {
        while (count == items.length)
            wait();
        items[putptr] = x;
        if (++putptr == items.length) putptr = 0;
        ++count;
        notifyAll();
    }

    public synchronized Object take() throws InterruptedException {
        while (count == 0)
            wait();
        Object x = items[takeptr];
        if (++takeptr == items.length) takeptr = 0;
        --count;
        notifyAll();
        return x;
    }

    public synchronized boolean isClosed() {
        return closed;
    }

    public synchronized void setClosed(boolean closed) {
        this.closed = closed;
    }
}
```

A producer is nothing but a thread puts task into BlockingQueue till the queue is full.

Producer Thread

```
import java.util.concurrent.ThreadLocalRandom;

public class Producer implements Runnable {
    private final BlockingQueue<SquareTask> queue;

    Producer(BlockingQueue<SquareTask> q) {
        queue = q;
    }

    public void run() {
        try {
            while (!queue.isClosed()) {
                queue.put(produce());
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    private SquareTask produce() {
        return new SquareTask(ThreadLocalRandom.current().nextInt(1, 200));
    }
}
```

A consumer listens on BlockingQueue and keeps consuming the tasks waiting if queue is empty.

Consumer Thread

```
public class Consumer implements Runnable {
    private final BlockingQueue<SquareTask> queue;

    Consumer(BlockingQueue<SquareTask> q) {
        queue = q;
    }

    public void run() {
        try {
            while (!queue.isClosed()) {
                consume(queue.take());
            }
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    private void consume(SquareTask x) {
        System.out.println(x.execute());
    }
}
```

Main Program

```
public class ProducerConsumer {

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<SquareTask> boundedBuffer = new BlockingQueue<>();
        Producer p = new Producer(boundedBuffer);
        Consumer c1 = new Consumer(boundedBuffer);
        Consumer c2 = new Consumer(boundedBuffer);
        new Thread(p).start();
        new Thread(c1).start();
        new Thread(c2).start();
        Thread.sleep(100);
        boundedBuffer.setClosed(true);
    }
}
```

4. Why producer-consumer problem is important?

1. It can be used to distribute the work among various workers that can be scaled up or down as per the load requirements.

2. It can be used to abstract Producer and Consumer connected through a shared pipeline. Producer does not need to know about the Consumer, thus there is abstraction of producers and consumers of work items i.e. separation of concerns. This leads to a better OOP Design.

3. Producer and Consumer does not need to be available at the same time. Consumer can pick up tasks produced by producer at a different time.

What do you understand by Java Memory Model?

Java Memory Model defines the legal interaction of threads with the memory in a real computer system. In a way, it describes what behaviors are legal in multi-threaded code. It determines when a Thread can reliably see writes to variables made by other threads. It defines semantics for volatile, final & synchronized, that makes guarantee of visibility of memory operations across the Threads.

Let's first discuss about Memory Barrier which are the base for our further discussions. There are two type of memory barrier instructions in JMM - read barriers & write barrier.

A read barrier invalidates the local memory (cache, registers, etc) and then reads the contents from the main memory, so that changes made by other threads becomes visible to the current Thread. A write barrier flushes out the contents of the processor's local memory to the main memory, so that changes made by the current Thread becomes visible to the other threads.

JMM semantics for synchronized

When a thread acquires monitor of an object, by entering into a synchronized block of code, it performs a read barrier (invalidates the local memory and reads from the heap instead). Similarly exiting from a synchronized block as part of releasing the associated monitor, it performs a write barrier (flushes changes to the main memory) Thus modifications to a shared state using synchronized block by one Thread, is guaranteed to be visible to subsequent synchronized reads by other threads. This guarantee is provided by JMM in presence of synchronized code block.

JMM semantics for Volatile fields

Read & write to volatile variables have same memory semantics as that of acquiring and releasing a monitor using synchronized code block. So the visibility of volatile field is guaranteed by the JMM. Moreover afterwards Java 1.5, volatile reads and writes are not reorderable with any other memory operations (volatile and non-volatile both). Thus when Thread A writes to a volatile variable V, and afterwards Thread B reads from variable V, any variable values that were visible to A at the time V was written are guaranteed now to be visible to B.

Let's try to understand the same using the following code.

```
Data data = null;
volatile boolean flag = false;

Thread A
-------------
```

data = new Data();
flag = true; <-- writing to volatile will flush data as well as flag to main memory

Thread B
----
if(flag==true) { <-- reading from volatile will perform read barrier for flag as well data.
use data; <--- data is guaranteed to visible even though it is not declared volatile because of the JMM semantics of volatile flag.
}

What is Slipped Condition in Multi-threading?

Slipped Condition is a special type of race condition that can occur in a multithreaded application. In this, a thread is suspended **after reading a condition** and **before performing the activities** related to it. It rarely occurs, however, one must look for it if the outcome is not as expected.

What is Deadlock in Java? How to troubleshoot and how to avoid deadlock

Deadlock

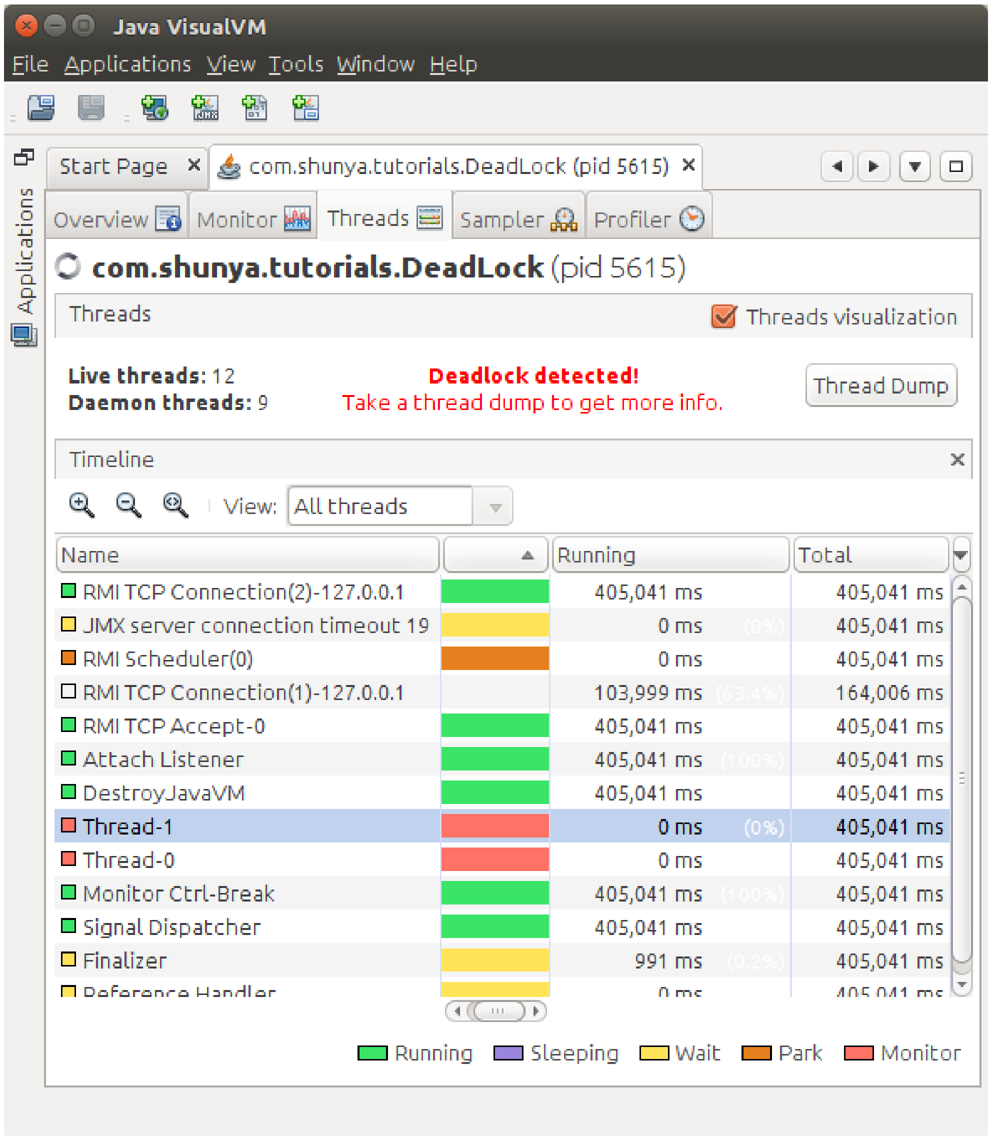
Deadlock describes a situation where two or more threads waiting on two or more shared resource in a particular order are blocked forever, waiting for each other to complete. Deadlocks can occur in Java when the synchronized keyword causes the executing thread to block while waiting to get the lock, associated with the specified object. Since the thread might already hold locks associated with other objects, two threads could each be waiting for the other to release a lock. In such case, they will end up waiting forever.

Java Source for producing a Deadlock Condition

```
package com.shunya.tutorials;

public class DeadLock {

    String resource1 = "Resource1";
    String resource2 = "Resource2";

    public void thread1Work() {
        Thread t1 = new Thread(() -> {
            while (true) {
                synchronized (resource1) {
                    synchronized (resource2) {
                        System.out.println(resource1 + resource2);
                    }
                }
            }
        });
        t1.start();
    }

    public void thread2Work() {
        Thread t2 = new Thread(() -> {
            while (true) {
                synchronized (resource2) {
                    synchronized (resource1) {
                        System.out.println(resource1 + resource2);
                    }
                }
            }
        });
        t2.start();
    }

    public static void main(String[] args) {
        DeadLock deadLock = new DeadLock();
        deadLock.thread1Work();
        deadLock.thread2Work();
    }
}
```

Using VisualVM to troubleshoot Deadlock Condition

Java Visual VM is a tool bundled with standard Java Development Kit


Java VisualVM
File Applications View Tools Window Help
Applications
Start Page
com.shunya.tutorials.DeadLock (pid 5615)
Overview
Monitor
Threads
Sampler
Profiler

com.shunya.tutorials.DeadLock (pid 5615)

Threads ✓ Threads visualization

Live threads: 12
Daemon threads: 9

Deadlock detected!
Take a thread dump to get more info.

Thread Dump

Timeline

View: All threads

Name	▲	Running	Total
🟩 RMI TCP Connection(2)-127.0.0.1	🟩	405,041 ms	405,041 ms
🟨 JMX server connection timeout 19	🟨	0 ms (0%)	405,041 ms
🟧 RMI Scheduler(0)	🟧	0 ms	405,041 ms
☐ RMI TCP Connection(1)-127.0.0.1		103,999 ms (63.4%)	164,006 ms
🟩 RMI TCP Accept-0	🟩	405,041 ms	405,041 ms
🟩 Attach Listener	🟩	405,041 ms (100%)	405,041 ms
🟩 DestroyJavaVM	🟩	405,041 ms	405,041 ms
🟥 Thread-1	🟥	0 ms (0%)	405,041 ms
🟥 Thread-0	🟥	0 ms	405,041 ms
🟩 Monitor Ctrl-Break	🟩	405,041 ms (100%)	405,041 ms
🟩 Signal Dispatcher	🟩	405,041 ms	405,041 ms
🟨 Finalizer	🟨	991 ms (0.2%)	405,041 ms
🟨 Reference Handler	🟨	0 ms	405,041 ms

🟩 Running 🟪 Sleeping 🟨 Wait 🟧 Park 🟥 Monitor

```
Stacktrace shown in JVisualVM for Deadlock Condition
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x00007f6228003828 (object 0x0000000771000720, a
java.lang.String),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x00007f6228004e28 (object 0x0000000771000738, a
java.lang.String),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at com.shunya.tutorials.DeadLock.lambda$thread2Work$1(DeadLock.java:26)
        - waiting to lock <0x0000000771000720> (a java.lang.String)
        - locked <0x0000000771000738> (a java.lang.String)
        at com.shunya.tutorials.DeadLock$$Lambda$2/1831932724.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:748)
"Thread-0":
        at com.shunya.tutorials.DeadLock.lambda$thread1Work$0(DeadLock.java:13)
        - waiting to lock <0x0000000771000738> (a java.lang.String)
        - locked <0x0000000771000720> (a java.lang.String)
        at com.shunya.tutorials.DeadLock$$Lambda$1/1096979270.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:748)

Found 1 deadlock.
```

How to Avoid Deadlock?

You can take few steps to ensure that deadlock does not occur in your application. Two main options for avoiding deadlock are-

1. All threads must acquire lock on shared objects in same order.
2. Use Explicit Locking in Java with a timeout specified on the lock so that if the requested resource is not available within time limit, code can proceed further.
3. We should shrink scope of synchronized blocks to reduce possibility deadlock. This will not avoid deadlock but prevent it to some extent.

We will see example code for both the options-

5. Acquire Lock in Same Order to avoid deadlock

If we acquire the lock on shared resource in same order from all threads, then deadlock will never occur. But in a real life this is not achievable due to large codebase. In the DeadLock example shown above, if you acquire lock on resource1 and resource2 in same order in both threads, deadlock will never occur.

6. Use Explicit Locking to avoid deadlock (tryLock() and wait(timeout))

`ReentrantLock` class provides a method `tryLock()` that Acquires the lock only if it is free at the time of invocation. This can be used to avoid un-necessary deadlock between two threads. For example,

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

public class DeadlockSolution {

    final Lock lock1 = new ReentrantLock();
    final Lock lock2 = new ReentrantLock();

    public void deadLockDemo() {
        Thread t1 = new Thread(new RunnableA());
        t1.setName("Thread A");
        t1.start();


        Thread t2 = new Thread(new RunnableB());
        t2.setName("Thread B");
        t2.start();
    }

    public static void main(String[] args) {
        new DeadlockSolution().deadLockDemo();
    }

    class RunnableA implements Runnable {

        public void run() {
            boolean done = false;
            while (!done) {
                if (lock1.tryLock()) {
                    try {
                        System.out.println(Thread.currentThread().getName() + ":
Got lockObject1. Trying for lockObject2");
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }

                        if (lock2.tryLock()) {
                            try {
                                System.out.println(Thread.currentThread().getName()
+ ": Got lockObject2.");
                                done = true;
                            } finally {
                                lock2.unlock();
                            }
                        }

                    } finally {
                        lock1.unlock();
                        try {
                            Thread.sleep(500);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }

    class RunnableB implements Runnable {
        public void run() {
            boolean done = false;
            while (!done) {
                if (lock2.tryLock()) {
                    try {
                        System.out.println(Thread.currentThread().getName() + ":
Got lockObject2. Trying for lockObject1");
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }

                        if (lock1.tryLock()) {
                            try {
                                System.out.println(Thread.currentThread().getName()
+ ": Got lockObject1.");
                                done = true;
                            } finally {
                                lock1.unlock();
                            }
                        }

                    } finally {
                        lock2.unlock();
                        try {
                            Thread.sleep(750);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        }
    }
}
```

We can even specify the timeout value for Lock Acquisition, like below

```
Lock lock = ...;
if (lock.tryLock(15L, TimeUnit.SECONDS)) {
    try {
       ........
    } finally {
        lock.unlock();
    }
} else {
      // do sumthing
}
```

This approach should be preferred because it does not require dependency on order of acquiring lock on shared resource, and quite practical solution for a real life project.

What is does Collections.unmodifiableCollection do

What does Collections.unmodifiableCollection() do? Is it safe to use the collection returned by this method in a multi-threading environment?

Collections.unmodifiableCollection() returns a unmodifiable dynamic view of underlying data structure. Any attempt direct or via iterator to modify this view throws UnsupportedOperationException, but any changes made in the underlying data structure will be reflected in the view.

> This method is no substitute for the other thread safety techniques because iterating over a collection using this view may throw ConcurrentModificationException if original collection is structurally modified during the iteration.

For example, the following code will throw ConcurrentModificationException in the for loop:

```
public class UnModifiableCollection {
    private List<String> names = new ArrayList<>();
    public void testConcurrency() {
        names.add("1");
        names.add("2");
        names.add("3");
        names.add("4");
        Collection < String > dynamicView =
Collections.unmodifiableCollection(names);
        for (String s: dynamicView) {
            System.out.println("s = " + s);
            names.remove(0);
        }
    }
    public static void main(String[] args) {
        UnModifiableCollection test = new UnModifiableCollection();
        test.testConcurrency();
    }
}
```

Hence, external synchronization is must if we are going to modify the underlying collection, even if you are using `Collections.unmodifiableCollection()`.

Design an Immutable class that has an java.util.Date member

As we know that java.util.Date is not immutable, we need to make a defensive copy of java.util.Date field while returning a reference to this instance variable.

Let’s create a hypothetical person class that has name and dob as the only two members.

Immutbale Person Class

```
import java.util.Date;

class Person {
    private String name;
    private Date dob;

    public Person(String name, Date dob) {
        this.name = name;
        this.dob = new Date(dob.getTime());
    }

    public String getName() {
        return name;
    }

    public Date getDob() {
        return new Date(dob.getTime());
    }
}
```

We are creating a new copy of Date field otherwise reference to dob field may leak

We are returning defensive copy of Date field instead of directly returning the reference of instance variable.

what are Key classes in java.util.concurrent package

`java.util.concurrent` package holds utility classes that are commonly useful in concurrent programming.

At a high level this package contains concurrent utilities like Executors, Queues, Timing, Synchronizers and Concurrent Collections.

Executors

Executor is a simple interface for defining custom thread-like subsystems, including thread pools, asynchronous I/O, and light weight task frameworks. ExecutorService provides a more complete asynchronous task execution framework.

Implementation classes include:

Executors

`Executors` class provides factory methods for most common kind of configurations of Executors.

ThreadPoolExecutor

Provides tunable and flexible thread pools.

ForkJoinPool

uses work-stealing scheduler for high throughput computation intensive parallel processing

ExecutorCompletionService

that assists in coordinating the processing of groups of asynchronous tasks.

Queues

ConcurrentLinkedQueue

The ConcurrentLinkedQueue class supplies an efficient scalable thread-safe non-blocking FIFO queue. The ConcurrentLinkedDeque class is similar, but additionally supports the java.util.Deque interface.

LinkedTransferQueue

LinkedTransferQueue introduce a synchronous transfer method (along with related features) in which a producer may optionally block awaiting its consumer.

There are five other implementations - LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue and DelayQueue.

Synchronizers

Five special purpose synchronizers are present in this package.

Semaphore

It is a classic concurrency tool which is often used to restric the number of threads that can access a shared resource (logical or physical).

CountDownLatch

It is a very simple yet very common utility for blocking until a given number of signals, events, or conditions hold.

CyclicBarrier

It is a resettable multiway synchronization point useful in some styles of parallel programming.

Exchanger

It allows two threads to exchange objects at a rendezvous point, and is useful in several pipeline designs.

Phaser

It provides a more flexible form of barrier that may be used to control phased computation among multiple threads.

Concurrent Collections

This package supplies Collection implementations designed for use in multithreaded contexts:

- ConcurrentHashMap
- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet

When many threads are expected to access a given collection, a ConcurrentHashMap is normally preferable to a synchronized HashMap, and a ConcurrentSkipListMap is normally preferable to a synchronized TreeMap.

A CopyOnWriteArrayList is preferable to a synchronized ArrayList when the expected number of reads and traversals greatly outnumber the number of updates to a list.

A concurrent collection is thread-safe, but not governed by a single exclusion lock. In the particular case of ConcurrentHashMap, it safely permits any number of concurrent reads as well as a tunable number of concurrent writes. "Synchronized" classes can be useful when you need to prevent all access to a collection via a single lock, at the expense of poorer scalability. In other cases in which multiple threads are expected to access a common collection, "concurrent" versions are normally preferable. And unsynchronized collections are preferable when either collections are unshared, or are accessible only when holding other locks.

Iterators and Spliterators returned by most concurrent Collection implementations provide weakly consistent rather than fail-fast traversal. That means:

- they may proceed concurrently with other operations
- they will never throw ConcurrentModificationException
- they are guaranteed to traverse elements as they existed upon construction exactly once, and may (but are not guaranteed to) reflect any modifications subsequent to construction.

How will you implement your custom threadsafe Semaphore in Java

We can implement a custom semaphore using `ReentrantLock` and `Condition` classes provided by Java. However this implementation is just for illustration purpose and not for production use.

Java Docs: Condition and Lock Interface

Condition factors out the Object monitor methods (wait, notify and notifyAll) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. Where a Lock replaces the use of synchronized methods and statements, a Condition replaces the use of the Object monitor methods.
Not for production use, instead use java.util.concurrent.Semaphore

```
import javax.annotation.concurrent.GuardedBy;
import javax.annotation.concurrent.ThreadSafe;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

@ThreadSafe
public class CustomSemaphore {
    private final Lock lock = new ReentrantLock();
    // CONDITION PREDICATE: permitsAvailable (permits > 0)
    private final Condition permitsAvailable = lock.newCondition();

    @GuardedBy("lock")
    private int permits;

    CustomSemaphore(int initialPermits) {
        lock.lock();
        try {
            permits = initialPermits;
        } finally {
            lock.unlock();
        }
    }

    /**
     * Blocks until permitsAvailable (permit > 0)
     * @throws InterruptedException
     */
    public void acquire() throws InterruptedException {
        lock.lock();
        try {
            while (permits <= 0)
                permitsAvailable.await();
            --permits;
        } finally {
            lock.unlock();
        }
    }

    /**
     * Release a single permit and notifies threads waiting on permitsAvailable Condition
     */
    public void release() {
        lock.lock();
        try {
            ++permits;
            permitsAvailable.signal();
        } finally {
            lock.unlock();
        }
    }

}
```

Derived from *Java Concurrency in Practice: Chapter14. BuildingCustomSynchronizers*

Code walk-through

1. A semaphore contains a number of permits and provides two methods - `acquire()` and `release()`

2. `acquire()` method is a blocking call which will decrease number of available permits by one, else wait for a permit to be available. Signalling is done through `Condition` interface (which is `permitsAvailable` in our case)
3. `release()` method will increment number of permits by one and notify all threads waiting on condition (`permitsAvailable`), so that one of waiting thread (if any) can acquire the next lock.
4. As you can see, both methods (acquire and release) call `lock()` on `lock` object, this is necessary for memory visibility and atomicity of shared mutable state (`permits` in this case)

How will you implement a Blocking Queue in Java

A blocking queue allows multiple threads to communicate with each other and pass data around. For example, a producer can put items to the queue while consumer can take items out from the queue.

> BlockingQueue implementations are designed to be used primarily for producer-consumer queues.

A blocking queue has below characteristics:

1. It is always thread-safe
2. It can hold arbitrary data
3. Producer has to wait if the queue is already full, similarly consumer has to be wait if no item is present in the queue.

Java implementation

A trivial implementation of BlockingQueue using *intrinsic locking* using synchronized keyword will look like the following:

BlockingQueue Implementation using synchronization

```
class SimpleBlockingQueue {
  final Object[] items = new Object[100];
  int putptr, takeptr, count;

  public synchronized void put(Object x) throws InterruptedException {
    while (count == items.length)
      wait();
    items[putptr] = x;
    if (++putptr == items.length) putptr = 0;
    ++count;
    notifyAll();
  }

  public synchronized Object take() throws InterruptedException {
    while (count == 0)
      wait();
    Object x = items[takeptr];
    if (++takeptr == items.length) takeptr = 0;
    --count;
```

```
        notifyAll();
        return x;
    }
}
```

Max capacity of blocking queue is 100

We are waiting inside a while loop while queue capacity is full. While loop is required to avoid spurious wakeups.

All other waiting threads are notified as soon as a new item is added to the queue.

Consumer thread waits inside a while loop for arrival of new item, if queue is empty. while loop prevents spurious wakeup problem.

Consumer thread notifies all waiting producer threads as soon as an item is removed from the queue.

A realistic implementation would have much more methods (peek(), remove(), offer(), isFull(), etc.) to the queue, but for brevity we have implemented only two methods.

A slightly improved version

Improved version of Queue will use explicit locking, to improve the multi-threading performance. *Lock* and *Condition* interface provides much better flexibility compared to intrinsic locking mechanism, but this flexibility brings more responsibility as we have to take care of calling lock and unlock ourselves. Since one lock can be associated with multiple conditions (notFull & notEmpty in this case), this results in better throughput due to lesser thread contention.

> Lock is analogical to synchronized keyword and Condition is similar to wait/notify.

SimpleBlockingQueue using Lock and Condition

```
class SimpleBlockingQueue {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    final Object[] items = new Object[100];
    int putptr, takeptr, count;

    public void put(Object x) throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0;
            ++count;
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public Object take() throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
        } finally {
            lock.unlock();
        }
    }
}
```

Here we are using ReentrantLock along with Condition to explicitly define the lock. This reduces thread contention under heavy load circumstances.

We can do few further improvements to the above version, for example we can use LinkedList instead of array and optionally make it generic to support any item type.

Generic BlockingQueue using LinkedList under the hood

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class BlockingQueue<T> {
    private Queue<T> queue = new LinkedList<T>();
    private int capacity;
    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public void put(T element) throws InterruptedException {
        lock.lock();
        try {
            while (queue.size() == capacity) {
                notFull.await();
            }
            queue.add(element);
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }

    public T take() throws InterruptedException {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                notEmpty.await();
            }
            T item = queue.remove();
            notFull.signal();
            return item;
        } finally {
            lock.unlock();
        }
    }
}
```

Please be noted that all above implementations are for learning purpose only, not for production use. For production, you shall prefer to use *LinkedBlockingQueue* or any other equivalent implementation from `java.util.concurrent` package.

Java 8 Parallel Stream with ThreadPool

Java does not provide any direct mechanism to control the number of threads and ThreadPool used by `parallel()` method in stream API, but there are two indirect way to configure the same.

Configure default Common Pool

Its documented that parallel() method utilizes the common pool available per classloader per jvm, and we have a mechanism to control the configuration of that default common pool using below 3 System properties

java.util.concurrent.ForkJoinPool.common.parallelism

The parallelism level, a non-negative integer

java.util.concurrent.ForkJoinPool.common.threadFactory

The class name of a ForkJoinPool.ForkJoinWorkerThreadFactory

java.util.concurrent.ForkJoinPool.common.exceptionHandler

The class name of a Thread.UncaughtExceptionHandler

For example, set the System property before calling the parallel stream

Changing the default common pool size JVM wide

```
public class CustomCommonPoolSize {
    public void testParallelOperation() {
        long start = System.currentTimeMillis();
        IntStream s = IntStream.range(0, 20);
        System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "20");
        s.parallel().forEach(i -> {
            try {
                Thread.sleep(100);
            } catch (Exception ignore) {}
            System.out.print((System.currentTimeMillis() - start) + " ms");
        });
        System.out.println("\nOverall time consumed: "+ (System.currentTimeMillis() - start)+" ms");
    }
}
```

Program output

```
192 192 192 192 192 192 192 192 192 192 192 192 192 192 192 192 192 192 192 192
Overall time consumed: 193 ms
```

Se see that all 20 tasks run in parallel and this the overall time is just 193 ms, even if individual task was taking 192ms each.

Run the parallel() operation inside a custom ForkJoinPool

There actually is a trick how to execute a parallel operation in a specific fork-join pool. If you execute it as a task in a fork-join pool, it stays there and does not use the common one. The trick is based on ForkJoinTask.

Fork documentation which specifies:

Arranges to asynchronously execute this task in the pool the current task is running in, if applicable, or using the ForkJoinPool.commonPool() if not in ForkJoinPool()

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ForkJoinPool;
import java.util.stream.IntStream;

import static java.lang.Math.sqrt;
import static java.util.stream.Collectors.toList;

class StreamExampleJava8 {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        ForkJoinPool forkJoinPool = new ForkJoinPool(4); // Configure the number of threads
        forkJoinPool.submit(() -> IntStream.range(1, 1_000_000)
                .parallel()
                .filter(StreamExampleJava8::isPrime).boxed()
                .collect(toList()))
                .get();
        forkJoinPool.shutdown();
    }

    private static boolean isPrime(long n) {
        return n > 1 && IntStream.rangeClosed(2, (int) sqrt(n)).noneMatch(divisor -> n % divisor == 0);
    }
}
```

In above code block, fork join pool will create 4 threads and run the parallel operations inside this custom fork join pool.

What are the key principles for designing a scalable software?

Carvia Tech | May 19, 2019 | 2 min read | 119 views

- Picking up right software architecture can lay a scalable foundation for your application. For example, Microservices Architecture of software development can help in building a highly scalable & distributed system. Essentially, microservice architecture is a method of developing software applications as a suite of independently deployable, small, modular services in which each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.
- Stateless design using REST can help achieve scalability wherever possible. In such application, minimal session elements need to be replicated while distributing the application over multiple hosts. Users can save their favorite URLs and thus there should be no need for the page flow, if we use REST.
- Removing centralized bottlenecks from your software can help increasing the scalability capabilities. For example, instead of using DB backed authentication mechanism, you can use JWT (JSON Web Tokens) for authentication which does not require token to be checked at database level (centralized), instead each microservice can check token using public key (cryptography).
- Logging can be done asynchronously to save precious time of a method call.
- More processes vs more threads can be configured based on the demand of the target application. Generally it is advised to have a JVM with up to 2 GB memory because increasing memory beyond 2 GB incurs heavy GC pauses, and if we require more processing then we prefer to have a separate process for the JVM altogether. Multiple independent tasks should be run in parallel. Tasks can be partitioned to improve the performance.
- If we improve upon the concurrency of the software piece, then we can increase its scalability. This can be achieved by reducing the dependency on the shared resources. We should try utilizing the latest hardware optimization through JAVA as much as possible. For example we can use Atomic utilities provided in java.util.concurrent.atomic package, or Fork & Join to achieve higher throughput in concurrent applications. We should try holding the shared locks for as little time as possible.
- Resource pooling and caching can be used to improve the processing time. Executing jobs in batches can further improve the performance.
- Picking up appropriate algorithm and data structure for a given scenario can help optimize the processing.
- If we are using SQL in our application then we should tune the SQL, use batching whereever possible and create indexes on the essentials table columns for faster retrievals.
- We should tune our JVM for optimum memory settings (Heap, PermGen, etc) and Garbage collection settings. For example if we do lot of text processing in our application with big temporary objects being created, then we should have larger Young Generation defined so that frequent gc run does not happen.
- Keep up to date with new technologies for performance benefits.