# How ConcurrentHashMap is implemented in Java

ConcurrentHashMap is introduced as an alternative of Hashtable and provided all functions supported by Hashtable with additional feature called "concurrency level", which allows ConcurrentHashMap to partition Map. ConcurrentHashMap allows multiple readers to read concurrently without any blocking. This is achieved by partitioning Map into different parts based on concurrency level and locking only a portion of Map during updates. Default concurrency level is 16, and accordingly Map is divided into 16 part and each part is governed with different lock. This means, 16 thread can operate on Map simultaneously, until they are operating on different part of Map. This makes ConcurrentHashMap high performance despite keeping thread-safety intact. Though, it comes with caveat. Since update operations like put(), remove(), putAll() or clear() is not synchronized, concurrent retrieval may not reflect most recent change on Map.

In case of putAll() or clear(), which operates on whole Map, concurrent read may reflect insertion and removal of only some entries. Another important point to remember is iteration over CHM, Iterator returned by keySet of ConcurrentHashMap are weekly consistent and they only reflect state of ConcurrentHashMap and certain point and may not reflect any recent change. Iterator of ConcurrentHashMap's keySet area also fail-safe and doesn’t throw ConcurrentModificationExceptoin..

Default concurrency level is 16 and can be changed, by providing a number which make sense and work for you while creating ConcurrentHashMap. Since concurrency level is used for internal sizing and indicate number of concurrent update without contention, so, if you just have few writers or thread to update Map keeping it low is much better. ConcurrentHashMap also uses ReentrantLock to internally lock its segments.

# Summary

1. **ConcurrentHashMap allows concurrent read and thread-safe update operation**.

2. **During update operation, ConcurrentHashMap only lock a portion of Map instead of whole Map**.

3. Concurrent update is achieved by internally dividing Map into small portion which is defined by concurrency level.

4. Choose concurrency level carefully as a significant higher number can be waste of time and space and lower number may introduce thread contention in case writers over number concurrency level.

5. All operations of ConcurrentHashMap are thread-safe.

6. Since ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.

7. Iterator returned by ConcurrentHashMap is weekly consistent, fail safe and never throw ConcurrentModificationException. Iterator does not support remove() method ConcurrentHashMap.

8. ConcurrentHashMap doesn't allow null as key or value.

9. You can use ConcurrentHashMap in place of Hashtable but with caution as CHM doesn't lock whole Map.

10. During putAll() and clear() operations, concurrent read may only reflect insertion or deletion of some entries.

**Blocking Queue Implementation using Condition**

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();

}

}

}

Of course if you actually need a blocking queue, then you should use an implementation of the BlockingQueue interface. The wait() method causes the current thread to wait (possibly forever) until another thread notifies it of a condition change. You use wait() in conjunction with notify() to coordinate the activities of multiple threads using the same resources.

**ThreadPoolExecutor**

<http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/ThreadPoolExecutor.html>

An ExecutorService that executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods.

Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks.

To be useful across a wide range of contexts, this class provides many adjustable parameters and extensibility hooks. However, programmers are urged to use the more convenient Executors factory methods Executors.newCachedThreadPool() (unbounded thread pool, with automatic thread reclamation), Executors.newFixedThreadPool(int) (fixed size thread pool) and Executors.newSingleThreadExecutor() (single background thread), that preconfigure settings for the most common usage scenarios. Otherwise, use the following guide when manually configuring and tuning this class:

Core and maximum pool sizes

A ThreadPoolExecutor will automatically adjust the pool size (see getPoolSize()) according to the bounds set by corePoolSize (see getCorePoolSize()) and maximumPoolSize (see getMaximumPoolSize()). When a new task is submitted in method execute(java.lang.Runnable), and fewer than corePoolSize threads are running, a new thread is created to handle the request, even if other worker threads are idle. If there are more than corePoolSize but less than maximumPoolSize threads running, a new thread will be created only if the queue is full. By setting corePoolSize and maximumPoolSize the same, you create a fixed-size thread pool. By setting maximumPoolSize to an essentially unbounded value such as Integer.MAX\_VALUE, you allow the pool to accommodate an arbitrary number of concurrent tasks. Most typically, core and maximum pool sizes are set only upon construction, but they may also be changed dynamically using setCorePoolSize(int) and setMaximumPoolSize(int).

On-demand construction

By default, even core threads are initially created and started only when new tasks arrive, but this can be overridden dynamically using method prestartCoreThread() or prestartAllCoreThreads(). You probably want to prestart threads if you construct the pool with a non-empty queue.

Creating new threads

New threads are created using a ThreadFactory. If not otherwise specified, a Executors.defaultThreadFactory() is used, that creates threads to all be in the same ThreadGroup and with the same NORM\_PRIORITY priority and non-daemon status. By supplying a different ThreadFactory, you can alter the thread's name, thread group, priority, daemon status, etc. If a ThreadFactory fails to create a thread when asked by returning null from newThread, the executor will continue, but might not be able to execute any tasks.

Keep-alive times

If the pool currently has more than corePoolSize threads, excess threads will be terminated if they have been idle for more than the keepAliveTime (see getKeepAliveTime(java.util.concurrent.TimeUnit)). This provides a means of reducing resource consumption when the pool is not being actively used. If the pool becomes more active later, new threads will be constructed. This parameter can also be changed dynamically using method setKeepAliveTime(long, java.util.concurrent.TimeUnit). Using a value of Long.MAX\_VALUE TimeUnit.NANOSECONDS effectively disables idle threads from ever terminating prior to shut down. By default, the keep-alive policy applies only when there are more than corePoolSizeThreads. But method allowCoreThreadTimeOut(boolean) can be used to apply this time-out policy to core threads as well, so long as the keepAliveTime value is non-zero.

Queuing

Any BlockingQueue may be used to transfer and hold submitted tasks. The use of this queue interacts with pool sizing:

If fewer than corePoolSize threads are running, the Executor always prefers adding a new thread rather than queuing.

If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.

If a request cannot be queued, a new thread is created unless this would exceed maximumPoolSize, in which case, the task will be rejected.

There are three general strategies for queuing:

Direct handoffs. A good default choice for a work queue is a SynchronousQueue that hands off tasks to threads without otherwise holding them. Here, an attempt to queue a task will fail if no threads are immediately available to run it, so a new thread will be constructed. This policy avoids lockups when handling sets of requests that might have internal dependencies. Direct handoffs generally require unbounded maximumPoolSizes to avoid rejection of new submitted tasks. This in turn admits the possibility of unbounded thread growth when commands continue to arrive on average faster than they can be processed.

Unbounded queues. Using an unbounded queue (for example a LinkedBlockingQueue without a predefined capacity) will cause new tasks to wait in the queue when all corePoolSize threads are busy. Thus, no more than corePoolSize threads will ever be created. (And the value of the maximumPoolSize therefore doesn't have any effect.) This may be appropriate when each task is completely independent of others, so tasks cannot affect each others execution; for example, in a web page server. While this style of queuing can be useful in smoothing out transient bursts of requests, it admits the possibility of unbounded work queue growth when commands continue to arrive on average faster than they can be processed.

Bounded queues. A bounded queue (for example, an ArrayBlockingQueue) helps prevent resource exhaustion when used with finite maximumPoolSizes, but can be more difficult to tune and control. Queue sizes and maximum pool sizes may be traded off for each other: Using large queues and small pools minimizes CPU usage, OS resources, and context-switching overhead, but can lead to artificially low throughput. If tasks frequently block (for example if they are I/O bound), a system may be able to schedule time for more threads than you otherwise allow. Use of small queues generally requires larger pool sizes, which keeps CPUs busier but may encounter unacceptable scheduling overhead, which also decreases throughput.

Rejected tasks

New tasks submitted in method execute(java.lang.Runnable) will be rejected when the Executor has been shut down, and also when the Executor uses finite bounds for both maximum threads and work queue capacity, and is saturated. In either case, the execute method invokes the RejectedExecutionHandler.rejectedExecution(java.lang.Runnable, java.util.concurrent.ThreadPoolExecutor) method of its RejectedExecutionHandler. Four predefined handler policies are provided:

In the default ThreadPoolExecutor.AbortPolicy, the handler throws a runtime RejectedExecutionException upon rejection.

In ThreadPoolExecutor.CallerRunsPolicy, the thread that invokes execute itself runs the task. This provides a simple feedback control mechanism that will slow down the rate that new tasks are submitted.

In ThreadPoolExecutor.DiscardPolicy, a task that cannot be executed is simply dropped.

In ThreadPoolExecutor.DiscardOldestPolicy, if the executor is not shut down, the task at the head of the work queue is dropped, and then execution is retried (which can fail again, causing this to be repeated.)

It is possible to define and use other kinds of RejectedExecutionHandler classes. Doing so requires some care especially when policies are designed to work only under particular capacity or queuing policies.

Hook methods

This class provides protected overridable beforeExecute(java.lang.Thread, java.lang.Runnable) and afterExecute(java.lang.Runnable, java.lang.Throwable) methods that are called before and after execution of each task. These can be used to manipulate the execution environment; for example, reinitializing ThreadLocals, gathering statistics, or adding log entries. Additionally, method terminated() can be overridden to perform any special processing that needs to be done once the Executor has fully terminated.

If hook or callback methods throw exceptions, internal worker threads may in turn fail and abruptly terminate.

Queue maintenance

Method getQueue() allows access to the work queue for purposes of monitoring and debugging. Use of this method for any other purpose is strongly discouraged. Two supplied methods, remove(java.lang.Runnable) and purge() are available to assist in storage reclamation when large numbers of queued tasks become cancelled.

Finalization

A pool that is no longer referenced in a program AND has no remaining threads will be shutdown automatically. If you would like to ensure that unreferenced pools are reclaimed even if users forget to call shutdown(), then you must arrange that unused threads eventually die, by setting appropriate keep-alive times, using a lower bound of zero core threads and/or setting allowCoreThreadTimeOut(boolean).

Extension example. Most extensions of this class override one or more of the protected hook methods. For example, here is a subclass that adds a simple pause/resume feature:

class PausableThreadPoolExecutor extends ThreadPoolExecutor {

private boolean isPaused;

private ReentrantLock pauseLock = new ReentrantLock();

private Condition unpaused = pauseLock.newCondition();

public PausableThreadPoolExecutor(...) { super(...); }

protected void beforeExecute(Thread t, Runnable r) {

super.beforeExecute(t, r);

pauseLock.lock();

try {

while (isPaused) unpaused.await();

} catch (InterruptedException ie) {

t.interrupt();

} finally {

pauseLock.unlock();

}

}

public void pause() {

pauseLock.lock();

try {

isPaused = true;

} finally {

pauseLock.unlock();

}

}

public void resume() {

pauseLock.lock();

try {

isPaused = false;

unpaused.signalAll();

} finally {

pauseLock.unlock();

}

}

}

<http://www.javacodegeeks.com/2011/12/using-threadpoolexecutor-to-parallelize.html>

Using a ThreadPoolExecutor to Parallelize Independent Single-Threaded Tasks

The task execution framework, introduced in Java SE 5.0, is a giant leap forward to simplify the design and the development of multi threaded applications. The framework provides facilities to manage the concept of task, to manage thread life cycles and their execution policy.

In this blog post we'll describe the power, the flexibility and the simplicity of this framework showing off a simple use case.

The Basics

The executor framework introduces an interface to manage task execution: Executor. Executor is the interface you use to submit tasks, represented as Runnable instances. This interface also isolates a task submission from a task execution: executors with different execution policies all publish the same submission interface: should you change your execution policy, your submission logic wouldn't be affected by the change.

If you want to submit a Runnable instance for execution, it's as simple as:

Thread Pools

As outlined in the previous section, how the executor is going to execute your runnable isn't specified by the Executor contract: it depends on the specific type of executor you're using. The framework provides some different types of executors, each one with a specific execution policy tailored for different use cases.

The most common type of executors you'll be dealing with are thread pool executors., which are instances of the ThreadPoolExecutor class (and its subclasses). Thread pool executors manage a thread pool, that is the pool of worker threads that's going to execute the tasks, and a work queue.

You surely have seen the concept of pool in other technologies. The primary advantage of using a pool is reducing the overhead of resources creation, reusing structures (in this case, threads) that have been released after use. Another implicit advantage of using a pool is the capability of sizing your resource usage: you can tune the thread pool sizes to achieve the load you desire, without jeopardizing system resources.

The framework provides a factory class for thread pools called Executors. Using this factory you'll be able to create thread pools of different characteristics. Often, the underlying implementation is often the same (ThreadPoolExecutor) but the factory class helps you quickly configure a thread pool without using its more complex constructor. The factory methods are:

newFixedThreadPool: this method returns a thread pool whose maximum size is fixed. It will create new threads as needed up to the maximum configured size. When the number of threads hits the maximum, the thread pool will maintain the size constant.

newCachedThreadPool: this method returns an unbounded thread pool, that is a thread pool without a maximum size. However, this kind of thread pool will tear down unused thread when the load reduces.

newSingleThreadedExecutor: this method returns an executor that guarantees that tasks will be executed in a single thread.

newScheduledThreadPool: this method returns a fixed size thread pool that supports delayed and timed task execution.

This is just the beginning. Executors also provide other facilities that are out of scope in this tutorial and that I strongly encourage you to study about:

Life cycle management methods, declared by the ExecutorService interface (such as shutdown() and awaitTermination()).

Completion services to poll for a task status and retrieve its return value, if applicable.

The ExecutorService interface is particularly important since it provides a way to shutdown a thread pool, which is something you almost surely want to be able to do cleanly. Fortunately, the ExecutorService interface is pretty simple and self-explanatory and I recommend you study its JavaDoc thoroughly.

Basically, you send a shutdown() message to an ExecutorService, after which it won't accept new submitted tasks, but will continue processing the already enqueued jobs. You can pool for an executor service's termination status with isTerminated(), or wait until termination using the awaitTermination(…) method. The awaitTermination method won't wait forever, though: you'll have to pass the maximum wait timeout as a parameter.

Warning: a source of errors and confusion is a understanding why a JVM process never exits. If you don't shutdown your executor services, thus tearing down the underlying threads, the JVM will never exit: a JVM exits when its last non-daemon thread exits.

Configuring a ThreadPoolExecutor

If you decide to create a ThreadPoolExecutor manually instead of using the Executors factory class, you will need to create and configure one using one of its constructors. The most extensive constructor of this class is:

public ThreadPoolExecutor(

int corePoolSize,

int maxPoolSize,

long keepAlive,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

RejectedExecutionHandler handler);

As you can see, you can configure:

The core pool size (the size the thread pool will try to stick with).

The maximum pool size.

The keep alive time, which is a time after which an idle thread is eligible for being torn down.

The work queue to hold tasks awaiting execution.

The policy to apply when a task submission is rejected.

Limiting the Number of Queued Tasks

Limiting the number of concurrent tasks being executing, sizing your thread pool, represents a huge benefit for your application and its execution environment in terms of predictability and stability: an unbounded thread creation will eventually exhaust the runtime resources and your application might experience as a consequence, serious performance problems that may lead even to application instability.

That's a solution to just one part of the problem: you're capping the number of tasks being executed but aren't capping the number of jobs that can be submitted and enqueued for later execution. The application will experience resource shortage later, but it will eventually experience it if the submission rate consistently outgrows the execution rate.

The solution to this problem is:

Providing a blocking queue to the executor to hold the awaiting tasks. In the case the queue fills up, the submitted task will be "rejected".

The RejectedExecutionHandler is invoked when a task submission is rejected, and that's why the verb rejected was quoted in the previous item. You can implement you're own rejection policy or use one of the built-in policies provided by the framework.

The default rejection policies has the executor throw a RejectedExecutionException. However, other built-in policies let you:

Discard a job silently.

Discard the oldest job and try to resubmit the last one.

Execute the rejected task on the caller's thread.

When and why would one use such a thread pool configuration? Let's see an example.

An Example: Parallelizing Independent Single-Threaded Tasks

Recently, I was called to solve a problem with an old job my client was running since a long time ago. Basically, the job is made up of a component that awaits for file system events on a set of directory hierarchies. Whenever an event is fired, a file must be processed. The file processing is performed by a proprietary single threaded process. Truth be said, by its own nature, even if I could, I don't if I could parallelize it. The arrival rate of events is very high throughout part of the day and there's no need to process file in real time, they just to get processed before the next day.

The current implementation was a mix and match of technologies, including a UNIX shell script that was responsible for scanning huge directory hierarchies to detect where changes were applied. When that implementation was put in place, the number of cores in the execution environment were two, as much. Also, the rate of events was pretty lower: nowadays they're in the order of the millions, for a total of between 1 and 2 terabytes of raw data to be processed.

The servers the client is running these processes nowadays are twelve core machines: a huge opportunity to parallelize those old single-threaded tasks. We've got basically all of the ingredients for the recipe, we just need to decide how to build and tune it. Some thoughts before writing any code were necessary to understand the nature of the load and these are the constraints I detected:

A really huge number of files is to be scanned periodically: each directory contains between one and two millions of files.

The scanning algorithm is very quick and can be parallelized.

Processing a file will take at least 1 second, with spikes of even 2 or 3 seconds.

When processing a file, there is no other bottleneck than CPU.

CPU usage must be tunable, in order to use a different load profile depending on the time of the day.

I'll thus need a thread pool whose size is determined by the load profile active at the moment of invoking the process. I'm inclined to create, then, a fixed size thread pool executor configured according to the load policy. Since a processing thread is only CPU-bound, its core usage is 100% and waits on no other resources, the load policy is very easy to calculate: just take the number of core available in the processing environment and scale it down using the load factor that's active at that moment (and check that at least one core is used in the moment of peak):

int cpus = Runtime.getRuntime().availableProcessors();

int maxThreads = cpus \* scaleFactor;

maxThreads = (maxThreads > 0 ? maxThreads : 1);

Then, I need to create a ThreadPoolExecutor using a blocking queue to bound the number of submitted tasks. Why? Well: the directory scanning algorithms are very quick and will generate a huge number of files to process very quickly. How huge? It's hard to predict and its variability is pretty high. I'm not going to let the internal queue of my executor fill up indiscriminately with the objects representing my tasks (which include a pretty huge file descriptor). I'll prefer let the executor reject the files when the queue fills up.

Also, I'll use the ThreadPoolExecutor.CallerRunsPolicy as rejection policy. Why? Well, because when the queue is filled up and while the threads in the pools are busy processing the file, I'll have the thread that is submitting the task executing it. This way, the scanning stops to process a file and will resume scanning as soon as it finishes executing the current task.

Here's the code that creates the executor:

**ExecutorService executorService =**

**new ThreadPoolExecutor(**

**maxThreads, // core thread pool size**

**maxThreads, // maximum thread pool size**

**1, // time to wait before resizing pool**

**TimeUnit.MINUTES,**

**new ArrayBlockingQueue<Runnable>(maxThreads, true),**

**new ThreadPoolExecutor.CallerRunsPolicy());**

**ExecutorService executorService =**

**new ThreadPoolExecutor(**

**maxThreads, // core thread pool size**

**maxThreads, // maximum thread pool size**

**1, // time to wait before resizing pool**

**TimeUnit.MINUTES,**

**new ArrayBlockingQueue<Runnable>(maxThreads, true),**

**new ThreadPoolExecutor.CallerRunsPolicy());**

The skeleton of the code is the following (greatly simplified):

// scanning loop: fake scanning

while (!dirsToProcess.isEmpty()) {

File currentDir = dirsToProcess.pop();

// listing children

File[] children = currentDir.listFiles();

// processing children

for (final File currentFile : children) {

// if it's a directory, defer processing

if (currentFile.isDirectory()) {

dirsToProcess.add(currentFile);

continue;

}

executorService.submit(new Runnable() {

@Override

public void run() {

try {

// if it's a file, process it

new ConvertTask(currentFile).perform();

} catch (Exception ex) {

// error management logic

}

}

});

}

// ...

// wait for all of the executor threads to finish

executorService.shutdown();

try {

if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {

// pool didn't terminate after the first try

executorService.shutdownNow();

}

if (!executorService.awaitTermination(60, TimeUnit.SECONDS)) {

// pool didn't terminate after the second try

}

} catch (InterruptedException ex) {

executorService.shutdownNow();

Thread.currentThread().interrupt();

}

Conclusion

As you can see, the Java concurrency API is very easy to use, very flexible and extremely powerful. Some years ago, I would have taken much more effort to write such a simple program. This way, I could quickly solve a scalability problem caused by a legacy single threaded component in a matter of hours.

<http://programmingexamples.wikidot.com/threadpoolexecutor>

public class ThreadPoolExecutor extends AbstractExecutorService

An ExecutorService that executes each submitted task using one of possibly several pooled threads, normally configured using Executors factory methods.

Thread pools address two different problems: they usually provide improved performance when executing large numbers of asynchronous tasks, due to reduced per-task invocation overhead, and they provide a means of bounding and managing the resources, including threads, consumed when executing a collection of tasks. Each ThreadPoolExecutor also maintains some basic statistics, such as the number of completed tasks.[1]

Programmers can configure the creation for a thread pool using ThreadPoolExecutor constructor. The following parameters can be configure:

corePoolSize: This value (core pool size) tells how many threads will be created before implementation (execution policy) starts looking for existing free thread.

maximumPoolSize: The maximum pool size is the upper bound on how many pool threads can be active at once.

keepAliveTime: If the pool has more than corePoolSize threads and there are more tasks to execute then implementation (execution policy) will terminate all excess threads (maximumPoolSize – coreThreadSize) which are idle for more than keepAliveTime.

workQueue: This is a BlockingQueue used for holding tasks awaiting execution. There is direct relation between work queue and pool sizing.

import java.util.concurrent.\*;

import java.util.\*;

class MyThreadPoolExecutor {

int poolSize = 2;

int maxPoolSize = 2;

long keepAliveTime = 10;

ThreadPoolExecutor threadPool = null;

final ArrayBlockingQueue<Runnable> queue = new ArrayBlockingQueue<Runnable>(

5);

public MyThreadPoolExecutor() {

threadPool = new ThreadPoolExecutor(poolSize, maxPoolSize,

keepAliveTime, TimeUnit.SECONDS, queue);

}

public void runTask(Runnable task) {

// System.out.println("Task count.."+threadPool.getTaskCount() );

// System.out.println("Queue Size before assigning the

// task.."+queue.size() );

threadPool.execute(task);

// System.out.println("Queue Size after assigning the

// task.."+queue.size() );

// System.out.println("Pool Size after assigning the

// task.."+threadPool.getActiveCount() );

// System.out.println("Task count.."+threadPool.getTaskCount() );

System.out.println("Task count.." + queue.size());

}

public void shutDown() {

threadPool.shutdown();

}

public static void main(String args[]) {

MyThreadPoolExecutor mtpe = new MyThreadPoolExecutor();

// start first one

mtpe.runTask(new Runnable()

{

public void run()

{

for (int i = 0; i < 10; i++)

{

try

{

System.out.println("First Task");

Thread.sleep(1000);

} catch (InterruptedException ie)

{

}

}

}

});

// start second one

/\*

\* try{ Thread.sleep(500); }catch(InterruptedException

\* ie){}

\*/

mtpe.runTask(new Runnable() {

public void run() {

for (int i = 0; i < 10; i++) {

try {

System.out.println("Second Task");

Thread.sleep(1000);

} catch (InterruptedException ie)

{

}

}

}

});

// start third one

/\*

\* try{ Thread.sleep(500); }catch(InterruptedException

\* ie){}

\*/

mtpe.runTask(new Runnable() {

public void run() {

for (int i = 0; i < 10; i++)

{

try

{

System.out.println("Third Task");

Thread.sleep(1000);

} catch (InterruptedException ie)

{

}

}

}

});

// start fourth one

/\*

\* try{ Thread.sleep(500); }catch(InterruptedException

\* ie){}

\*/

mtpe.runTask(new Runnable()

{

public void run()

{

for (int i = 0; i < 10; i++)

{

try

{

System.out.println("Fourth Task");

Thread.sleep(1000);

} catch (InterruptedException ie)

{

}

}

}

});

// start fifth one

/\*

\* try{ Thread.sleep(500); }catch(InterruptedException

\* ie){}

\*/

mtpe.runTask(new Runnable()

{

public void run()

{

for (int i = 0; i < 10; i++)

{

try

{

System.out.println("Fifth Task");

Thread.sleep(1000);

} catch (InterruptedException ie)

{

}

}

}

});

// start Sixth one

/\*

\* try{ Thread.sleep(500); }catch(InterruptedException

\* ie){}

\*/

mtpe.runTask(new Runnable()

{

public void run()

{

for (int i = 0; i < 10; i++)

{

try

{

System.out.println("Sixth Task");

Thread.sleep(1000);

} catch (InterruptedException ie)

{

}

}

}

});

mtpe.shutDown();

}

}

**How AtomicInteger or AtomicLong works in Java**

First consider this code:

public class MyApp {

private volatile int count = 0;

public void upateVisitors() {

++count; //increment the visitors count

}

}

Actually the problem is simply **marking count as volatile does not guarantee atomicity and ++count is not an atomic operations**. Can we solve this problem if we mark the method itself synchronized as shown below:

public class MyApp {

private int count = 0;

public **synchronized** void upateVisitors() {

++count; //increment the visitors count

}

}

Although the above problem solves the atomicity and visibility but it makes use of locking and that introduces lot of delay and overhead. Check this [article](http://flex4java.blogspot.in/2015/03/is-multi-threading-really-worth-it.html). This is very expensive way of making things work. To overcome these problems atomic constructs were introduced. If we make use of an AtomicInteger to track the count it will work.

public class MyApp {

private **AtomicInteger count = new AtomicInteger(0);**

public void upateVisitors() {

count.incrementAndGet(); //increment the visitors count

}

}

The classes that support atomic operations e.g. AtomicInteger, AtomicLong etc. makes use of CAS. CAS does not make use of locking rather it is very optimistic in nature. It follows these steps:

* Compare the value of the primitive to the value we have got in hand.
* If the values do not match it means some thread in between has changed the value. Else it will go ahead and swap the value with new value.

Check the following code in AtomicLong class:

public final long incrementAndGet() {

for (;;) {

long current = get();

long next = current + 1;

if (compareAndSet(current, next))

return next;

}

}

In JDK 8 the above code has been changed to a single intrinsic:

**public final long incrementAndGet() {**

**return unsafe.getAndAddLong(this, valueOffset, 1L) + 1L;**

**}**

**public** **final** **boolean** [http://grepcode.com/static/app/images/1x1.gif](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java)compareAndSet(**long** expect, **long** update) { //As for JDK 7

**return** [**unsafe**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java#AtomicLong.0unsafe)**.**[**compareAndSwapLong**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/sun/misc/Unsafe.java#Unsafe.compareAndSwapLong%28java.lang.Object%2Clong%2Clong%2Clong%29)**(this,**[**valueOffset**](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/atomic/AtomicLong.java#AtomicLong.0valueOffset)**, expect, update);**

}

[Asynchronous VS Multithreading](https://stackoverflow.com/questions/34680985/what-is-the-difference-between-asynchronous-programming-and-multithreading)

<https://stackoverflow.com/questions/34680985/what-is-the-difference-between-asynchronous-programming-and-multithreading>

Many people are taught that multithreading and asynchrony are the same thing, but they are not.

An analogy usually helps. You are cooking in a restaurant. An order comes in for eggs and toast.

* Synchronous: you cook the eggs, then you cook the toast.
* Asynchronous, single threaded: you start the eggs cooking and set a timer. You start the toast cooking, and set a timer. While they are both cooking, you clean the kitchen. When the timers go off you take the eggs off the heat and the toast out of the toaster and serve them.
* Asynchronous, multithreaded: you hire two more cooks, one to cook eggs and one to cook toast. Now you have the problem of coordinating the cooks so that they do not conflict with each other in the kitchen when sharing resources. And you have to pay them.

Now does it make sense that multithreading is only one kind of asynchrony? **Threading is about workers; asynchrony is about tasks**. In multithreaded workflows you assign tasks to workers. In asynchronous single-threaded workflows you have a graph of tasks where some tasks depend on the results of others; as each task completes it invokes the code that schedules the next task that can run, given the results of the just-completed task. But you (hopefully) only need one worker to perform all the tasks, not one worker per task.

It will help to realize that many tasks are not processor-bound. For processor-bound tasks it makes sense to hire as many workers (threads) as there are processors, assign one task to each worker, assign one processor to each worker, and have each processor do the job of nothing else but computing the result as quickly as possible. But for tasks that are not waiting on a processor, you don't need to assign a worker at all. You just wait for the message to arrive that the result is available and *do something else while you're waiting*. When that message arrives then you can schedule the continuation of the completed task as the next thing on your to-do list to check off.

In-browser Javascript is a great example of an asynchronous program that has no threads.

This is an essential truth of async in its purest form: **There is no thread.**

<https://medium.com/@punitkmr/does-async-programming-mean-multi-threading-fb8d1add56dc>

In a **multi-threaded environment**, many individual threads of programming are running at the same time. (Depending upon the number of CPUs and the support of the operating system, this may be literally true, or it may be an illusion created by sophisticated scheduling algorithms). For this reason, multi-threaded environments are difficult and involve issues of threads locking each other’s memory to prevent them from overrunning one another.

In an **asychronous environment**, a single process thread runs all the time, but it may, for event-driven reasons (and that is the key), switch from one function to another. When an event happens, *and when the currently running process hits a point at which it must wait for another event*, the javascript core then scans its list of events and delivers the next one, in a (formally) indeterminate (but probably deterministic) order, to the event manager.

Asynchronous programming means that the engine runs in an event loop.

<https://blog.slaks.net/2014-12-23/parallelism-async-threading-explained/>

An asynchronous operation is an operation which continues in the background after being initiated, without forcing the caller to wait for it to finish before running other code.

Multi-threading means running more than one thread of execution at a time. In this model, all operations are still synchronous, but the CPU will execute multiple threads of synchronous operations at the same time.

Multi-threading makes most sense when calling multiple (and independent) CPU-bound operations, on a multi-core processor. For example, a program that independently analyzes every pixel in an image could divide the image into one strip for each CPU core, then analyze each strip in its own thread at the same time.

Note that there is no advantage gained in running more threads than there are CPU cores. Threads do not magically let CPUs do more work for free; once you run out of dedicated cores to run your threads, you’ll end up with single cores running bits of each thread in turn, adding additional context-switching overhead as the core switches to each thread, and not getting any performance benefit.

If you have strictly CPU-bound work, multi-threading is your only option. The whole point of asynchrony is to leave the CPU (and your thread(s)) free, so that you can run other code while waiting for the asynchronous operations to finish. If your work is CPU-bound, running it asynchronously does not make sense.

If you have non-CPU-bound work (disk IO, network requests, etc), both options can work. However, running the operations asynchronously will offer better performance. Parallelizing non-CPU-bound work using multiple threads means making each request on its own dedicated thread, blocking that thread until the response arrives. However, threads are not cheap; creating each thread occupies a megabyte or more of memory for the stack and limited kernel resources, and also adds more context-switching overhead. This places artificial limits on the total number of in-flight requests.

In contrast, asynchrony offers much better parallelism with non-blocking operations. You can kick off thousands of non-blocking requests from a single thread with very little per-request overhead, then schedule callbacks to run when each request finishes. Your original thread is then free to do whatever else it likes (such as actual CPU-bound work, kicking off more async requests, or accepting incoming connections).

Switching non-CPU-bound operations to non-blocking implementations can offer tremendous performance and scalability benefits, simply because threads are not cheap. At a previous job, I rewrote a long-running and massively parallel network server to use non-blocking IO, and the throughput improved from 900 connections per server to over 9,000 connections per server.

<https://stackoverflow.com/questions/4844637/what-is-the-difference-between-concurrency-parallelism-and-asynchronous-methods>

*Concurrency means multiple tasks which start, run, and complete in overlapping time periods, in no specific order. Parallelism is when multiple tasks OR several part of a unique task literally run at the same time, e.g. on a multi-core processor.*

Remember that Concurrency and parallelism are NOT the same thing.

**Differences between concurrency vs. parallelism**

Now let’s list down remarkable differences between concurrency and parallelism.

Concurrency is when two tasks can start, run, and complete in overlapping time periods. Parallelism is when tasks literally run at the same time, eg. on a multi-core processor.

Concurrency is the composition of independently executing processes, while parallelism is the simultaneous execution of (possibly related) computations.

Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.

An application can be concurrent – but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at same time instant.

An application can be parallel – but not concurrent, which means that it processes multiple sub-tasks of a task in multi-core CPU at same time.

An application can be neither parallel – nor concurrent, which means that it processes all tasks one at a time, sequentially.

An application can be both parallel – and concurrent, which means that it processes multiple tasks concurrently in multi-core CPU at same time.

**Concurrency**

Concurrency is essentially applicable when we talk about minimum two tasks or more. When an application is capable of executing two tasks virtually at same time, we call it concurrent application. Though here tasks run looks like simultaneously, but essentially they MAY not. They take advantage of CPU time-slicing feature of operating system where each task run part of its task and then go to waiting state. When first task is in waiting state, CPU is assigned to second task to complete it’s part of task.

Operating system based on priority of tasks, thus, assigns CPU and other computing resources e.g. memory; turn by turn to all tasks and give them chance to complete. To end user, it seems that all tasks are running in parallel. This is called concurrency.

**Parallelism**

Parallelism does not require two tasks to exist. It literally physically run parts of tasks OR multiple tasks, at the same time using multi-core infrastructure of CPU, by assigning one core to each task or sub-task.

Parallelism requires hardware with multiple processing units, essentially. In single core CPU, you may get concurrency but NOT parallelism.

**Asynchronous methods**

This is not related to Concurrency and parallelism, asynchrony is used to present the impression of concurrent or parallel tasking but effectively an asynchronous method call is normally used for a process that needs to do work away from the current application and we don't want to wait and block our application awaiting the response.

<http://tutorials.jenkov.com/java-concurrency/concurrency-vs-parallelism.html>

**Parallelism**

Parallelism means that an application splits its tasks up into smaller subtasks which can be processed in parallel, for instance on multiple CPUs at the exact same time. To achieve true parallelism your application must have more than one thread running, or at least be able to schedule tasks for execution in other threads, processes, CPUs, graphics cards etc.

**Concurrency**

Concurrency is related to how an application handles multiple tasks it works on. An application may process one task at at time (sequentially) or work on multiple tasks at the same time (concurrently).

Parallelism on the other hand, is related to how an application handles each individual task. An application may process the task serially from start to end, or split the task up into subtasks which can be completed in parallel.

An application can also be parallel but not concurrent. This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel.

<https://medium.com/swift-india/concurrency-parallelism-threads-processes-async-and-sync-related-39fd951bc61d>

Recently, a friend of mine asked me his queries on Concurrency and Parallelism. When I started explaining him his queries, we started discussing other related concepts and nomenclatures such as Threads -> Multi-threaded and Single, Asynchronous and Synchronous. At a point, we were confused with queries like:

*How is concurrency related to parallelism?*

*What is synchronous and asynchronous execution?*

*What is the importance of synchronous and asynchronous programming in concurrency and parallelism?*

*How threads fit along with all these concepts?*

Many of us sometimes get confused with such queries.

So let us check whether you have grasped it right. We would analyse and understand what actually they are and their relationship between one another. We will keep our discussion easy and concise.

**Concurrency & Parallelism**

**Concurrency**

Consider you are given a task of singing and eating at the same time. At a given instance of time either you would sing or you would eat as in both cases your mouth is involved. So in order to do this, you would eat for some time and then sing and repeat this until your food is finished or song is over. So you performed your tasks concurrently.

Concurrency means executing multiple tasks at the same time but not necessarily simultaneously. In a concurrent application, two tasks can start, run, and complete in overlapping time periods i.e Task-2 can start even before Task-1 gets completed.

In the computer science world, the way how concurrency is achieved in various processors is different. In a single core environment (i.e your processor is having a single core), concurrency is achieved via a process called [context-switching](http://www.linfo.org/context_switch.html). If its a multi-core environment, concurrency can be achieved through parallelism.

**Parallelism**

Consider you are given two tasks of cooking and speaking to your friend over the phone. You could do these two things simultaneously. You could cook as well as speak over the phone. Now you are doing your tasks parallelly.

Parallelism means performing two or more tasks simultaneously. Parallel computing in computer science refers to the process of performing multiple calculations simultaneously.

**How is concurrency related to parallelism?**

* Concurrency and Parallelism refer to computer architectures which focus on how our tasks or computations are performed.
* ***In a single core environment, concurrency happens*** with tasks executing over same time period via context switching i.e at a particular time period, only a single task gets executed.
* ***In a multi-core environment, concurrency can be achieved via parallelism***in which multiple tasks are executed simultaneously.

**Threads & Processes**

**Threads**

Threads are a sequence of execution of code which can be executed independently of one another. ***It is the smallest unit of tasks that can be executed by an OS. A program can be single threaded or multi-threaded.***

**Process**

A process is an instance of a running program. ***A program can have multiple processes. A process usually starts with a single thread i.e a primary thread but later down the line of execution it can create multiple threads.***

**Synchronous and Asynchronous**

**Synchronous**

Imagine you were given to write two letters one to your mom and another to your best friend. You can not at the same time write two letters unless you are a pro ambidextrous.

In a synchronous programming model, tasks are executed one after another. Each task waits for any previous task to complete and then gets executed.

**Asynchronous**

Imagine you were given to make a sandwich and wash your clothes in a washing machine. You could put your clothes in the washing machine and without waiting for it to be done, you could go and make the sandwich. Here you performed these two tasks asynchronously.

In an asynchronous programming model, when one task gets executed, you could switch to a different task without waiting for the previous to get completed.

Is Volatile keyword required in case of Reentrant Lock?

<https://stackoverflow.com/questions/1570589/is-the-volatile-keyword-required-for-fields-accessed-via-a-reentrantlock>

Answer by **Jon Skeet: It's safe without volatility**. ReentrantLock implements Lock.

**All Lock implementations must enforce the same memory synchronization semantics as provided by the built-in monitor lock, as described in The Java Language Specification, Third Edition (17.4 Memory Model):**

* **A successful lock operation has the same memory synchronization effects as a successful Lock action.**
* **A successful unlock operation has the same memory synchronization effects as a successful Unlock action.**

Volatile variables are not cached in registers or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread. ... The visibility effects of volatile variables extend beyond the value of the volatile variable itself. When thread A writes to a volatile variable and subsequently thread B reads that same variable, the values of all variables that were visible to A prior to writing to the variable become visible to B after reading the volatile variable.

**Usage of Volatile Variable**

1) **Use volatile fields when writes do not depend on its current value.** An example is a flag to stop a worker thread from another thread:

public class WorkerThread extends Thread {

    private volatile boolean isRunning = true;

    @Override

    public void run() {

        while(isRunning) {

            // execute a task

        }

    }

    public void stopWorker() {

        isRunning = false;

    }

}

The WorkerThread executes his tasks in a while loop, line 5. It checks the volatile field isRunning in each iteration and stops processing if the field is false. This allows other threads to stop the WorkerThread by calling the method stopWorker which sets the value of the field to false. Since a thread can call the method stopWorker even if the WorkerThread is already stopped, the write to the field can be executed independently of its current value. By declaring the field volatile we make sure that the WorkerThread sees the update done in another Thread and does not run forever.

2) **Use volatile fields for reading and locks for writing.**

The java.util.concurrent.CopyOnWriteArrayList get and set methods are an example of this tip:

public class CopyOnWriteArrayList<E>

    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

**private transient volatile Object[] array;**

  final Object[] getArray() {

        return array;

   }

**final void setArray(Object[] a) {**

**array = a;**

**}**

   private E get(Object[] a, int index) {

        return (E) a[index];

   }

   public E get(int index) {

        return get(getArray(), index);

   }

   public E set(int index, E element) {

**final ReentrantLock lock = this.lock;**

**lock.lock();**

        try {

            Object[] elements = getArray();

            E oldValue = get(elements, index);

            if (oldValue != element) {

                int len = elements.length;

**Object[] newElements = Arrays.copyOf(elements, len);**

                newElements[index] = element;

**setArray(newElements);**

            } else {

                // Not quite a no-op; ensures volatile write semantics

**setArray(elements);**

            }

            return oldValue;

        } finally {

**lock.unlock();**

        }

    }

    // Other fields and methods omitted

}

Is it necessary to use volatile with AtomicInteger?

<https://stackoverflow.com/questions/14338533/atomicinteger-and-volatile>

**Answer by Jon Skeet**

**Atomic\* actually gives *both* atomicity and volatility.** **So when you call (say) AtomicInteger.get(), you're guaranteed to get the *latest* value. This is documented in the java.util.concurrent.atomic**[**package documentation**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html)**:**

**get has the memory effects of reading a volatile variable.**

**set has the memory effects of writing (assigning) a volatile variable.**

**lazySet** has the memory effects of writing (assigning) a volatile variable except that it permits reorderings with subsequent (but not previous) memory actions that do not themselves impose reordering constraints with ordinary non-volatile writes. Among other usage contexts, > - lazySet may apply when nulling out, for the sake of garbage collection, a reference that is never accessed again.

**weakCompareAndSet** atomically reads and conditionally writes a variable but does not create any happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the weakCompareAndSet.

**compareAndSet** and all other read-and-update operations such as getAndIncrement have the memory effects of both reading and writing volatile variables.

Now if you have volatile AtomicInteger count;

the volatile part means that each thread will use the latest AtomicInteger reference, and the fact that it's an AtomicInteger means that you'll *also* see the latest value for that object.

It's not common (IME) to need this - because normally you wouldn't reassign count to refer to a different object. Instead, you'd have: **private final AtomicInteger count = new AtomicInteger();**

At that point, the fact that it's a final variable means that all threads will be dealing with the same object - and the fact that it's an Atomic\* object means they'll see the latest value within that object.

**Difference between Atomic, Volatile and Synchronized**

<https://stackoverflow.com/questions/9749746/what-is-the-difference-between-atomic-volatile-synchronized>

**No synchronization**

private int counter;

public int getNextUniqueIndex() {

return counter++;

}

It basically reads value from memory, increments it and puts back to memory. This works in single thread but nowadays, in the era of multi-core, multi-CPU, multi-level caches it won't work correctly. **First of all it introduces race condition (several threads can read the value at the same time), but also visibility problems.** **The value might only be stored in "*local*" CPU memory (some cache) and not be visible for other CPUs/cores (and thus - threads).** This is why many refer to *local copy* of a variable in a thread. It is very unsafe. Consider this popular but broken thread-stopping code:

private boolean stopped;

public void run() {

while(!stopped) {

//do some work

}

}

public void pleaseStop() {

stopped = true;

}

Add volatile to stopped variable and it works fine - if any other thread modifies stopped variable via pleaseStop() method, you are guaranteed to see that change immediately in working thread's while(!stopped) loop. BTW this is not a good way to interrupt a thread either, see: [How to stop a thread that is running forever without any use](https://stackoverflow.com/questions/6410721) and [Stopping a specific java thread](https://stackoverflow.com/questions/7786305).

**AtomicInteger**

private AtomicInteger counter = new AtomicInteger();

public int getNextUniqueIndex() {

return counter.getAndIncrement();

}

**The AtomicInteger class uses CAS (**[**compare-and-swap**](http://en.wikipedia.org/wiki/Compare-and-swap)**) low-level CPU operations (no synchronization needed!)** They allow you to modify a particular variable only if the present value is equal to something else (and is returned successfully). So when you execute **getAndIncrement()** it actually runs in a loop (simplified real implementation):

**int current;**

**do {**

**current = get();**

**} while(!compareAndSet(current, current + 1));**

So basically: read; try to store incremented value; if not successful (the value is no longer equal to current), read and try again. **The compareAndSet() is implemented in native code (assembly).**

**Volatile without synchronization**

private volatile int counter;

public int getNextUniqueIndex() {

return counter++;

}

**This code is not correct. It fixes the visibility issue (volatile makes sure other threads can see change made to counter) but still has a race condition.** This has been [explained](https://stackoverflow.com/questions/25168062/why-is-i-not-atomic) multiple times: pre/post-incrementation is not atomic. **The only side effect of volatile is "*flushing*" caches so that all other parties see the freshest version of the data.** This is too strict in most situations; that is why volatile is not default.

**Volatile without synchronization (2)**

**volatile int i = 0;**

**void incIBy5() {**

**i += 5;**

**}**

The same problem as above, but even worse because i is not private. The race condition is still present. Why is it a problem? **If, say, two threads run this code simultaneously, the output might be + 5 or + 10**. **However, you are guaranteed to see the change.**

**Multiple independent synchronized**

void incIBy5() {

int temp;

synchronized(i) { temp = i }

synchronized(i) { i = temp + 5 }

}

**Surprise, this code is incorrect as well. In fact, it is completely wrong. First of all you are synchronizing on i, which is about to be changed (moreover, i is a primitive, so I guess you are synchronizing on a temporary Integer created via autoboxing...) Completely flawed**. You could also write:

synchronized(new Object()) {

//thread-safe, SRSLy?

}

**No two threads can enter the same synchronized block** **with the same lock**. In this case (and similarly in your code) the lock object changes upon every execution, so synchronized effectively has no effect.

Even if you have used a final variable (or this) for synchronization, the code is still incorrect. Two threads can first read i to temp synchronously (having the same value locally in temp), then the first assigns a new value to i (say, from 1 to 6) and the other one does the same thing (from 1 to 6).

The synchronization must span from reading to assigning a value. Your first synchronization has no effect (reading an int is atomic) and the second as well. In my opinion, these are the correct forms:

**void synchronized incIBy5() {**

**i += 5**

**}**

**void incIBy5() {**

**synchronized(this) {**

**i += 5**

**}**

**}**

**void incIBy5() {**

**synchronized(this) {**

**int temp = i;**

**i = temp + 5;**

**}**

**}**

Declaring a variable as **volatile** means that modifying its value immediately affects the actual memory storage for the variable. The compiler cannot optimize away any references made to the variable. This guarantees that when one thread modifies the variable, all other threads see the new value immediately. (This is not guaranteed for non-volatile variables.)

Declaring an **atomic** variable guarantees that operations made on the variable occur in an atomic fashion, i.e., that all of the substeps of the operation are completed within the thread they are executed and are not interrupted by other threads. For example, an increment-and-test operation requires the variable to be incremented and then compared to another value; an atomic operation guarantees that both of these steps will be completed as if they were a single indivisible/uninterruptible operation.

**Synchronizing** **all accesses to a variable allows only a single thread at a time to access the variable, and forces all other threads to wait for that accessing thread to release its access to the variable.**

**Synchronized access is similar to atomic access, but the atomic operations are generally implemented at a lower level of programming.** Also, it is entirely possible to synchronize only some accesses to a variable and allow other accesses to be unsynchronized (e.g., synchronize all writes to a variable but none of the reads from it).

**Atomicity, synchronization, and volatility are independent attributes, but are typically used in combination to enforce proper thread cooperation for accessing variables.**

**Addendum** *(April 2016)*

**Synchronized access to a variable is usually implemented using a *monitor* or *semaphore*.** These are low-level *mutex* (mutual exclusion) mechanisms that allow a thread to acquire control of a variable or block of code exclusively, forcing all other threads to wait if they also attempt to acquire the same mutex. Once the owning thread releases the mutex, another thread can acquire the mutex in turn.

**Addendum** *(July 2016)*

**Synchronization occurs on an *object*.** This means that calling a synchronized method of a class will lock the this object of the call. **Static synchronized methods will lock the Class object itself**.

Likewise, entering a synchronized block requires locking the this object of the method.

**This means that a synchronized method (or block) can be executing in multiple threads at the same time if they are locking on *different* objects, but only one thread can execute a synchronized method (or block) at a time for any given *single* object.**

**StampedLock**

*StampedLock* is introduced in Java 8. It also supports both read and write locks. However, lock acquisition methods returns a stamp that is used to release a lock or to check if the lock is still valid:

public class StampedLockDemo {

    Map<String,String> map = new HashMap<>();

**private StampedLock lock = new StampedLock();**

    public void put(String key, String value){

**long stamp = lock.writeLock();**

        try {

            map.put(key, value);

        } finally {

            lock.unlockWrite(stamp);

        }

    }

    public String get(String key) throws InterruptedException {

**long stamp = lock.readLock();**

        try {

            return map.get(key);

        } finally {

**lock.unlockRead(stamp);**

        }

    }

}

Another feature provided by *StampedLock* is optimistic locking. Most of the time read operations doesn’t need to wait for write operation completion and as a result of this, the full-fledged read lock isn’t required. Instead, we can upgrade to read lock:

public String readWithOptimisticLock(String key) {

**long stamp = lock.tryOptimisticRead();**

    String value = map.get(key);

**if(!lock.validate(stamp)) {**

**stamp = lock.readLock();**

**try {**

**return map.get(key);**

**} finally {**

**lock.unlock(stamp);**

**}**

    }

    return value;

}

**Volatile Variable with ReentrantLock**

**Is it necessary to use volatile keyword in case of ReentrantLock >**

<https://stackoverflow.com/questions/1570589/is-the-volatile-keyword-required-for-fields-accessed-via-a-reentrantlock>

Answer by Jon Skeet

For next example using a ReentrantLock however, is the volatile keyword on the field necessary?

class B

{

private final ReentrantLock lock = new ReentrantLock();

**private volatile double sharedData;**

public void method()

{

lock.lock();

try

{

double temp = sharedData;

temp \*= 2.5;

sharedData = temp + 1;

}

finally

{

lock.unlock();

}

}

}

**Answer by Jon Skeet**

It's safe without volatility. ReentrantLock implements Lock, and the [docs for Lock](http://java.sun.com/javase/6/docs/api/java/util/concurrent/locks/Lock.html) include this:

All Lock implementations must enforce the same memory synchronization semantics as provided by the built-in monitor lock, as described in The Java Language Specification, Third Edition (17.4 Memory Model):

* A successful lock operation has the same memory synchronization effects as a successful Lock action.
* A successful unlock operation has the same memory synchronization effects as a successful Unlock action.

<https://vmlens.com/articles/3_tips_volatile_fields/>

Volatile variables are not cached in registers or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread. ... The visibility effects of volatile variables extend beyond the value of the volatile variable itself. When thread A writes to a volatile variable and subsequently thread B reads that same variable, the values of all variables that were visible to A prior to writing to the variable become visible to B after reading the volatile variable.  
— [Java Concurrency in Practice - Brian Goetz, et al.](http://jcip.net/)

1) Use volatile fields when writes do not depend on its current value.

An example is a flag to stop a worker thread from another thread:

public class WorkerThread extends Thread {

    private volatile boolean isRunning = true;

    @Override

    public void run() {

        while(isRunning)

        {

            // execute a task

        }

    }

    public void stopWorker()

    {

        isRunning = false;

    }

}

The WorkerThread executes his tasks in a while loop, line 5. It checks the volatile field isRunning in each iteration and stops processing if the field is false. This allows other threads to stop the WorkerThread by calling the method stopWorker which sets the value of the field to false. Since a thread can call the method stopWorker even if the WorkerThread is already stopped, the write to the field can be executed independently of its current value.

By declaring the field volatile we make sure that the WorkerThread sees the update done in another Thread and does not run forever.

2) Use volatile fields for reading and locks for writing

The java.util.concurrent.CopyOnWriteArrayList get and set methods are an example of this tip:

public class CopyOnWriteArrayList<E>

    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

  private transient volatile Object[] array;

  final Object[] getArray() {

        return array;

   }

   final void setArray(Object[] a) {

        array = a;

   }

   private E get(Object[] a, int index) {

        return (E) a[index];

   }

   public E get(int index) {

        return get(getArray(), index);

   }

   public E set(int index, E element) {

        final ReentrantLock lock = this.lock;

        lock.lock();

        try {

            Object[] elements = getArray();

            E oldValue = get(elements, index);

            if (oldValue != element) {

                int len = elements.length;

                Object[] newElements = Arrays.copyOf(elements, len);

                newElements[index] = element;

                setArray(newElements);

            } else {

                // Not quite a no-op; ensures volatile write semantics

                setArray(elements);

            }

            return oldValue;

        } finally {

            lock.unlock();

        }

    }

    // Other fields and methods omitted

}

**Is it necessary to use volatile with AtomicInteger**

<https://stackoverflow.com/questions/14338533/atomicinteger-and-volatile>

I know volatile allows for visibility, AtomicInteger allows for atomicity. So if I use a volatile AtomicInteger, does it mean I don't have to use any more synchronization mechanisms?

Eg.

class A {

private volatile AtomicInteger count;

void someMethod(){

// do something

if(count.get() < 10) {

count.incrementAndGet();

}

}

Is this threadsafe?

**Answer by Jon Skeet**

I believe that Atomic\* actually gives *both* atomicity and volatility. So when you call (say) AtomicInteger.get(), you're guaranteed to get the *latest* value. This is documented in the java.util.concurrent.atomic [package documentation](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/package-summary.html):

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in section 17.4 of The Java™ Language Specification.

* get has the memory effects of reading a volatile variable.
* set has the memory effects of writing (assigning) a volatile variable.
* lazySet has the memory effects of writing (assigning) a volatile variable except that it permits reorderings with subsequent (but not previous) memory actions that do not themselves impose reordering constraints with ordinary non-volatile writes. Among other usage contexts, > - lazySet may apply when nulling out, for the sake of garbage collection, a reference that is never accessed again.
* weakCompareAndSet atomically reads and conditionally writes a variable but does not create any happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the weakCompareAndSet.
* compareAndSet and all other read-and-update operations such as getAndIncrement have the memory effects of both reading and writing volatile variables.

Now if you have

volatile AtomicInteger count;

the volatile part means that each thread will use the latest AtomicInteger reference, and the fact that it's an AtomicInteger means that you'll *also* see the latest value for that object.

It's not common (IME) to need this - because normally you wouldn't reassign count to refer to a different object. Instead, you'd have:

private final AtomicInteger count = new AtomicInteger();

At that point, the fact that it's a final variable means that all threads will be dealing with the same object - and the fact that it's an Atomic\* object means they'll see the latest value within that object.

<https://stackoverflow.com/questions/9749746/what-is-the-difference-between-atomic-volatile-synchronized>

You are specifically asking about how they *internally work*, so here you are:

**No synchronization**

private int counter;

public int getNextUniqueIndex() {

return counter++;

}

It basically reads value from memory, increments it and puts back to memory. This works in single thread but nowadays, in the era of multi-core, multi-CPU, multi-level caches it won't work correctly. First of all it introduces race condition (several threads can read the value at the same time), but also visibility problems. The value might only be stored in "*local*" CPU memory (some cache) and not be visible for other CPUs/cores (and thus - threads). This is why many refer to *local copy* of a variable in a thread. It is very unsafe. Consider this popular but broken thread-stopping code:

private boolean stopped;

public void run() {

while(!stopped) {

//do some work

}

}

public void pleaseStop() {

stopped = true;

}

Add volatile to stopped variable and it works fine - if any other thread modifies stopped variable via pleaseStop() method, you are guaranteed to see that change immediately in working thread's while(!stopped) loop. BTW this is not a good way to interrupt a thread either, see: [How to stop a thread that is running forever without any use](https://stackoverflow.com/questions/6410721) and [Stopping a specific java thread](https://stackoverflow.com/questions/7786305).

**AtomicInteger**

private AtomicInteger counter = new AtomicInteger();

public int getNextUniqueIndex() {

return counter.getAndIncrement();

}

The AtomicInteger class uses CAS ([compare-and-swap](http://en.wikipedia.org/wiki/Compare-and-swap)) low-level CPU operations (no synchronization needed!) They allow you to modify a particular variable only if the present value is equal to something else (and is returned successfully). So when you execute getAndIncrement() it actually runs in a loop (simplified real implementation):

int current;

do {

current = get();

} while(!compareAndSet(current, current + 1));

So basically: read; try to store incremented value; if not successful (the value is no longer equal to current), read and try again. The compareAndSet() is implemented in native code (assembly).

**volatile without synchronization**

private volatile int counter;

public int getNextUniqueIndex() {

return counter++;

}

This code is not correct. It fixes the visibility issue (volatile makes sure other threads can see change made to counter) but still has a race condition. This has been [explained](https://stackoverflow.com/questions/25168062/why-is-i-not-atomic) multiple times: pre/post-incrementation is not atomic.

The only side effect of volatile is "*flushing*" caches so that all other parties see the freshest version of the data. This is too strict in most situations; that is why volatile is not default.

**volatile without synchronization (2)**

volatile int i = 0;

void incIBy5() {

i += 5;

}

The same problem as above, but even worse because i is not private. The race condition is still present. Why is it a problem? If, say, two threads run this code simultaneously, the output might be + 5 or + 10. However, you are guaranteed to see the change.

**Multiple independent synchronized**

void incIBy5() {

int temp;

synchronized(i) { temp = i }

synchronized(i) { i = temp + 5 }

}

Surprise, this code is incorrect as well. In fact, it is completely wrong. First of all you are synchronizing on i, which is about to be changed (moreover, i is a primitive, so I guess you are synchronizing on a temporary Integer created via autoboxing...) Completely flawed. You could also write:

synchronized(new Object()) {

//thread-safe, SRSLy?

}

No two threads can enter the same synchronized block **with the same lock**. In this case (and similarly in your code) the lock object changes upon every execution, so synchronized effectively has no effect.

Even if you have used a final variable (or this) for synchronization, the code is still incorrect. Two threads can first read i to temp synchronously (having the same value locally in temp), then the first assigns a new value to i (say, from 1 to 6) and the other one does the same thing (from 1 to 6).

The synchronization must span from reading to assigning a value. Your first synchronization has no effect (reading an int is atomic) and the second as well. In my opinion, these are the correct forms:

void synchronized incIBy5() {

i += 5

}

void incIBy5() {

synchronized(this) {

i += 5

}

}

void incIBy5() {

synchronized(this) {

int temp = i;

i = temp + 5;

}

}

Declaring a variable as **volatile** means that modifying its value immediately affects the actual memory storage for the variable. The compiler cannot optimize away any references made to the variable. This guarantees that when one thread modifies the variable, all other threads see the new value immediately. (This is not guaranteed for non-volatile variables.)

Declaring an **atomic** variable guarantees that operations made on the variable occur in an atomic fashion, i.e., that all of the substeps of the operation are completed within the thread they are executed and are not interrupted by other threads. For example, an increment-and-test operation requires the variable to be incremented and then compared to another value; an atomic operation guarantees that both of these steps will be completed as if they were a single indivisible/uninterruptible operation.

**Synchronizing** all accesses to a variable allows only a single thread at a time to access the variable, and forces all other threads to wait for that accessing thread to release its access to the variable.

Synchronized access is similar to atomic access, but the atomic operations are generally implemented at a lower level of programming. Also, it is entirely possible to synchronize only some accesses to a variable and allow other accesses to be unsynchronized (e.g., synchronize all writes to a variable but none of the reads from it).

Atomicity, synchronization, and volatility are independent attributes, but are typically used in combination to enforce proper thread cooperation for accessing variables.

**Addendum** *(April 2016)*

Synchronized access to a variable is usually implemented using a *monitor* or *semaphore*. These are low-level *mutex* (mutual exclusion) mechanisms that allow a thread to acquire control of a variable or block of code exclusively, forcing all other threads to wait if they also attempt to acquire the same mutex. Once the owning thread releases the mutex, another thread can acquire the mutex in turn.

**Addendum** *(July 2016)*

Synchronization occurs on an *object*. This means that calling a synchronized method of a class will lock the this object of the call. Static synchronized methods will lock the Class object itself.

Likewise, entering a synchronized block requires locking the this object of the method.

This means that a synchronized method (or block) can be executing in multiple threads at the same time if they are locking on *different* objects, but only one thread can execute a synchronized method (or block) at a time for any given *single* object.

[**Volatile Vs Atomic**](http://stackoverflow.com/questions/19744508/volatile-vs-atomic)

private int counter;

public int getNextUniqueIndex() {

return counter++;

}

It basically reads value from memory, increments it and puts back to memory. This works in single thread but nowadays, in the era of multi-core, multi-CPU, multi-level caches it won't work correctly. First of all it introduces race condition (several threads can read the value at the same time), but also visibility problems. The value might only be stored in "*local*" CPU memory (some cache) and not be visible for other CPUs/cores (and thus - threads).

private AtomicInteger counter = new AtomicInteger();

public int getNextUniqueIndex() {

return counter.getAndIncrement();

}

The AtomicInteger class uses CAS ([compare-and-swap](http://en.wikipedia.org/wiki/Compare-and-swap)) low-level CPU operations (no synchronization needed!) They allow you to modify particular variable only if the present value is equal to something else (and return it it succeed). So when you execute getAndIncrement() it actually runs in a loop (simplified real implementation):

int current;

do {

current = get();

} while(!compareAndSet(current, current + 1)

**No two threads can enter the same synchronized block with the same lock. This means that two threads can enter the same block on different objects.**

There are two important concepts in multithreading environment.

1. atomicity
2. visibility

Volatile eradicates visibility problem but it does not deal with atomicity. Volatile will prevent compiler to reorder the instruction which involves write and subsequent read of a volatile variable. e.g.k++ Here k++ is not a single machine instruction rather it is three machine instructions.

* copy the value to register
* increment it
* place it back

So even though you declare variable to volatile it will not make this operation atomic that means another thread can see a intermediate result which is a stale or unwanted value for the other thread.

**Atomicity, Visibility and Ordering**

* ***Atomicity*** deals with which actions and sets of actions have indivisible effects.
* ***Visibility*** determines when the effects of one thread can be seen by another.
* ***Ordering*** determines when actions in one thread can be seen to occur out of order with respect to another. Let's talk about them.

**Exchanger – By Debadatta Mishra**

<http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Exchanger.html>

A synchronization point at which threads can pair and swap elements within pairs. Each thread presents some object on entry to the exchange method, matches with a partner thread, and receives its partner's object on return. An Exchanger may be viewed as a bidirectional form of a SynchronousQueue. Exchangers may be useful in applications such as genetic algorithms and pipeline designs.

http://robaustin.wikidot.com/an-example-of-using-exchanger

An example is given below.

package test;

import java.util.concurrent.Exchanger;

import java.util.concurrent.atomic.AtomicReference;

public class Main {

public static void main(String[] args) {

final Exchanger<Integer> e = new Exchanger<Integer>();

new Thread(new Runnable() {

private final AtomicReference<Integer> last = new AtomicReference<Integer>(1);

@Override

public void run() {

try {

while (true) {

last.set(e.exchange(last.get()));

System.out.println("Thread A has value: " + last.get());

Thread.sleep(2000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}).start();

new Thread(new Runnable() {

private final AtomicReference<Integer> last = new AtomicReference<Integer>(2);

@Override

public void run() {

try {

while (true) {

last.set(e.exchange(last.get()));

System.out.println("Thread B has value: " + last.get());

Thread.sleep(2000);

}

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}).start();

}

}

http://tutorials.jenkov.com/java-util-concurrent/exchanger.html

The java.util.concurrent.Exchanger class represents a kind of rendezvous point where two threads can exchange objects. Here is an illustration of this mechanism: Exchanging objects is done via one of the two exchange() methods. Here is an example:

Exchanger exchanger = new Exchanger();

ExchangerRunnable exchangerRunnable1 = new ExchangerRunnable(exchanger, "A");

ExchangerRunnable exchangerRunnable2 = new ExchangerRunnable(exchanger, "B");

new Thread(exchangerRunnable1).start();

new Thread(exchangerRunnable2).start();

Here is the ExchangerRunnable code:

public class ExchangerRunnable implements Runnable{

Exchanger exchanger = null;

Object object = null;

public ExchangerRunnable(Exchanger exchanger, Object object) {

this.exchanger = exchanger;

this.object = object;

}

public void run() {

try {

Object previous = this.object;

this.object = this.exchanger.exchange(this.object);

System.out.println(

Thread.currentThread().getName() +

" exchanged " + previous + " for " + this.object

);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

This example prints out this:

Thread-0 exchanged A for B

Thread-1 exchanged B for A

http://vanillajava.blogspot.in/2011/09/exchange-and-gc-less-java.html

**The Exchanger and GC-less Java Overview**

The Exchanger class is very efficient at passing work between thread and recycling the objects used. AFAIK, It is also one of the least used Concurrency classes.

As @Marksim Sipos points out, if you don't need GC less logging using an ArrayBlockingQueue is much simpler.

Exchanger class

The Exchanger class is useful for passing data back and forth between two threads. e.g. Producer/Consumer. It has the property of naturally recycling the data structures used to pass the work and supports GC-less sharing of work in an efficient manner. Here is an example, passing logs to a background logger. Work (a log entry) is batched into LogEntries and passed to a background thread which later passes it back to the thread so it can add more work. Provided the background thread is always finished before the batch is full, it is almost transparent. Increasing the size of the batch reduces how often the batch is full but increase the number of unprocessed entries waiting at any one time. Calling flush() can push out the data. The key line is the following which exchanges the batch in the current thread with the batch in the other thread. The producer fills up the batch while the consumer is emptying it. The exchange when it occurs typically takes 1-4 micro-seconds. In this case, once every 64 lines.

entries = logEntriesExchanger.exchange(entries);

How does this compare to the LMAX disruptor pattern

This approach has similar principles to the Disruptor. No GC using recycled, pre-allocated buffers and lock free operations (The Exchanger not completely lock free and doesn't busy wait, but it could) Two keys difference are:

there is only one producer/consumer in this case, the disruptor supports multiple consumers.

this approach re-uses a much smaller buffer efficiently. If you are using ByteBuffer (as I have in the past) an optimal size might be 32 KB. The disruptor library was designed to exploit large amounts of memory on the assumption it is relative cheap and can use medium sized (MBs) to very large buffers (GBs). e.g. it was design for servers with 144 GB. I am sure it works well on much smaller servers. ;)

Thank you @Doug, for reminding me to mention the Disruptor pattern.

If you have dozen logs files (for different purposes) and you want to minimise memory foot print and you prefer the consuming thread to be blocking rather than busy waiting which consumes 100% of a thread (which adds a small latency of up to 10 us) then the Exchanger is better suited.

Exchanger example

import java.util.concurrent.Exchanger;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class BackgroundLogger implements Runnable {

static final int ENTRIES = 64;

static class LogEntry {

long time;

int level;

final StringBuilder text = new StringBuilder();

}

static class LogEntries {

final LogEntry[] lines = new LogEntry[ENTRIES];

int used = 0;

}

private final ExecutorService executor = Executors.newSingleThreadExecutor();

final Exchanger<LogEntries> logEntriesExchanger = new Exchanger<LogEntries>();

LogEntries entries = new LogEntries();

BackgroundLogger() {

executor.submit(this);

}

public StringBuilder log(int level) {

try {

if (entries.used == ENTRIES)

entries = logEntriesExchanger.exchange(entries);

LogEntry le = entries.lines[entries.used++];

le.time = System.currentTimeMillis();

le.level = level;

return le.text;

} catch (InterruptedException e) {

throw new RuntimeException(e);

}

}

public void flush() throws InterruptedException {

if(entries.used > 0)

entries = logEntriesExchanger.exchange(entries);

}

public void stop() {

try {

flush();

} catch (InterruptedException e) {

e.printStackTrace(); // use standard logging.

}

executor.shutdownNow();

}

@Override

public void run() {

LogEntries entries = new LogEntries();

try {

while (!Thread.interrupted()) {

entries = logEntriesExchanger.exchange(entries);

for (int i = 0; i < entries.used; i++) {

bgLog(entries.lines[i]);

entries.lines[i].text.delete(0, entries.lines[i].text.length());

}

entries.used = 0;

}

} catch (InterruptedException ignored) {

} finally {

System.out.println("Warn: logger stopping."); // use standard logging.

}

}

private void bgLog(LogEntry line) {

// log the entry to a file.

}

}

SYNCHRONOUS QUEUE BROKER

Having explored the rather boring wait-notify option we can now move onto more interesting implementations. The next thing that comes to mind is a SynchronousQueue. Why? Because the problem here is to facilitate single message exchange and for this SQ are perfect. Synchronous queues are effectively zero capacity queues and only pass messages across threads when consuming threads are ready to take handover.

package name.dhruba.kb.concurrency.pc.broker;

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.SynchronousQueue;

public class SyncQueueBroker<T> implements Broker<T> {

private final BlockingQueue<T> queue = new SynchronousQueue<T>();

@Override

public T take() throws InterruptedException {

return queue.take();

}

@Override

public void put(T message) throws InterruptedException {

queue.put(message);

}

}

In this example we could have used any blocking queue implementation but I used SQ for lightweight and immediate thread handovers without buffering.

EXCHANGER BROKER

So far we’ve tackled wait-notify and queues but how else can we achieve the same effect. The next solution uses an Exchanger. The Exchanger is a little known but tremendously interesting and powerful utility to swap messages between threads in a really simple way. It’s usage is as follows.

package name.dhruba.kb.concurrency.pc.broker;

import java.util.concurrent.Exchanger;

public class ExchangerBroker<T> implements Broker<T> {

private T message;

private final Exchanger<T> exchanger = new Exchanger<T>();

public void put(T message) throws InterruptedException {

message = exchanger.exchange(message);

}

public T take() throws InterruptedException {

return exchanger.exchange(message);

}

}

I have to admit that I’ve never actually needed or found a legitimate use for an exchanger in production code and I’m curious to hear if you have.

CONDITION BROKER

And, finally, we come to our last implementation using Condition which is again, like the Exchanger, little known and rarely seen out there (at least in my experience). I’ve only seen it in production code once and that too was for a very unusual use case. However conditions have a key advantage over their predecessor – wait/notify. To quote the javadoc on this one: “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”.

package name.dhruba.kb.concurrency.pc.broker;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.ReentrantLock;

public class ConditionBroker<T> implements Broker<T> {

private T message;

private boolean empty = true;

private final ReentrantLock lock = new ReentrantLock();

private final Condition fullState = lock.newCondition();

private final Condition emptyState = lock.newCondition();

@Override

public T take() throws InterruptedException {

lock.lock();

try {

while (empty) {

fullState.await();

}

empty = true;

emptyState.signal();

return message;

} finally {

lock.unlock();

}

}

@Override

public void put(T message) throws InterruptedException {

lock.lock();

try {

while (!empty) {

emptyState.await();

}

this.message = message;

empty = false;

fullState.signal();

} finally {

lock.unlock();

}

}

}

As you can see this is a much more effective semantic representation of the solution than the equivalent wait-notify.

Conclusion

The producer/consumer pattern is a useful scenario within which to explore solutions to bounded buffers, blocking queues and in-process eventing. Though it is usually of little use as higher level synchronisation primitives in Java 5 negate the need for us to implement any such thing in our processes. Employers love to ask this question in interviews. And for that bear in mind that it’s useful to know not only the higher level utilities but also the lower level wait notify behaviour even though it’s virtually obsolete at this point. If you can think of any other ways of implementing a single message exchange producer/consumer pattern please comment on here.

Basic syntax of using SynchronousQueue -blocking take and put methods

take method blocking if no one is put anything into the queue

package egtry.thread.queue;

import java.util.concurrent.SynchronousQueue;

public class SynchronousQueue1Test {

public static void main(String[] args) throws InterruptedException {

SynchronousQueue<String> queue=new SynchronousQueue();

System.out.println("taking a work item from queue");

String item=queue.take();//block

System.out.println("item take from queue: "+item);

}

}

Output

taking a work item from queue

put method blocking if no one is taking item from the queue

package egtry.thread.queue;

import java.util.concurrent.SynchronousQueue;

public class SynchronousQueue2Test {

public static void main(String[] args) throws InterruptedException {

SynchronousQueue<String> queue=new SynchronousQueue();

System.out.println("putting a item into the queu");

queue.put("first"); //block

System.out.println("Done");

}

}

For reference

--------------

http://www.dcs.warwick.ac.uk/~sgm/cs237/java/

http://www.egtry.com/java/thread/semaphoretest