1. [Java\_Concurrency](#Java_Concurrency)
2. [Concurrent\_Collections](#Concurrent_Collections)
3. [Concurrency\_utilities](#Concurrency_utilities)
4. [deadlock\_Racecondition](#deadlock_Racecondition)
5. [Executor](#Executor)
6. [What\_Executors](#What_Executors)
7. [Process\_threads](#Process_threads)
8. [RetrantLock](#RetrantLock)
9. [Locks](#Locks)
10. [Object level vs Class Level locking](#object_lelevel_locs_classlevel)
11. [LockExample](#LockExample)
12. [Countdownlatch](#Countdownlatch)
13. [Schedule\_executors](#Schedule_executors)
14. [Synchronizers\_count\_cyclic\_exc](#Synchronizers_count_cyclic_exc)
15. [ReadWriteLock](#ReadWriteLock)
16. [BlockingQueue](#BlockingQueue)(PRODUCER AND CONSUMER PROBLEM), Uses FIFO order. **thread-safe, no nulls**
17. [Difference\_CyclicBarrier\_CountDownLatch](#Difference_CyclicBarrier_CountDownLatch)
18. [Busy\_spinnning](#Busy_spinnning) **(wait strategy)**
19. [Future Object.](#Future)
20. [Cyclic Barrier](#CycliciBarrier)
21. [How ConcurrentHashMap internally works](#concurrent_hash_map)
22. **Difference between ExecutorService.submit and ExecutorService.execute()**

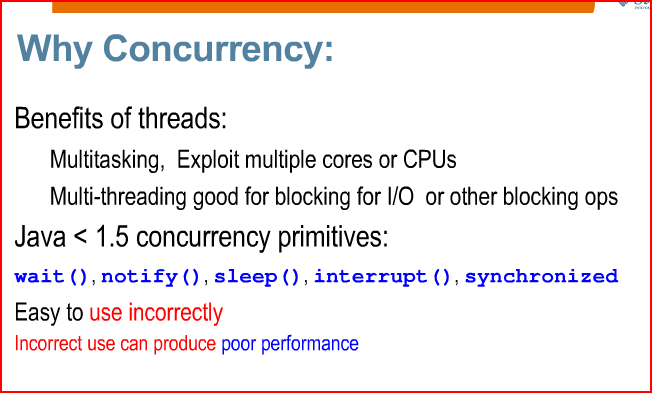
<http://javadecodedquestions.blogspot.com/2012/01/java-concurrency.html>

The words **synchronization** and **concurrency** are overlapping and sometimes synonymous terms. The word synchronization generally means sharing data between multiple processors or threads, while concurrency refers to a measure of– or the art of improving– how effectively an application allows multiple jobs required by that application (e.g. serving web page requests from a web server) to run simultaneously.

### Java Concurrency

**Java Concurrency interview question**- In year 2004 when technology gurus said innovation in Java is gone down and Sun Microsystems [Now Oracle] came with the Tiger release with very important changes with in java 1.5 and most important feature was Concurrency and programming features. This is hot topic for java interviews from past few years. Interviewers are mainly focused on java 1.5 concurrent package and they can ask how to use these changes and what are the benefits**. They will focus on how concurrency is better than synchronisation and how executors** are better than old java thread implementation? How you can avoid locks ? and What is java memory model changes for volatile etc. In investment banks we need to work on multithreaded applications due to high volume so clear understanding of this topic is very important. This topic is key for clearing any core java interview.

<http://www.slideshare.net/caroljmcdonald/java-concurrency-memory-model-and-trends-4961797>





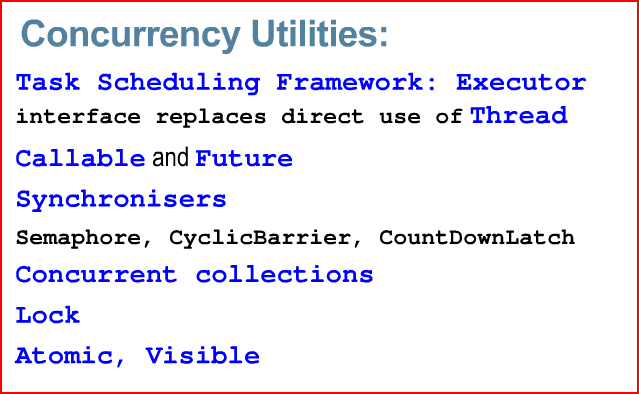
Differences and the common points (if any) between a **race** and a **dead lock** ? An detailed answer would be appreciated ;).

**Race Conditions**

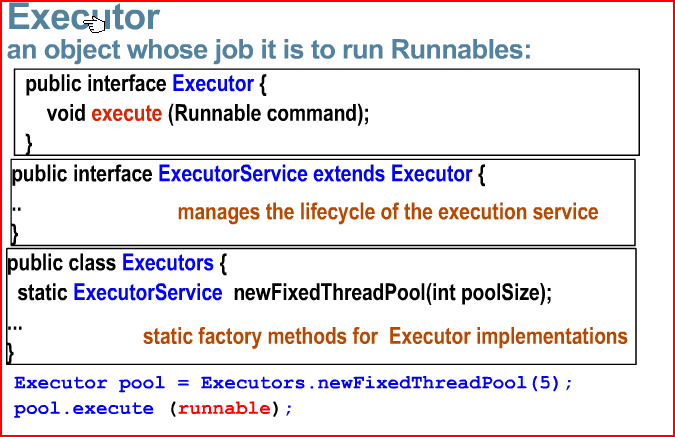
A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

**Deadlocks**

A deadlock occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked. As a result, each thread stops executing and waits for the other thread to release the variable. Because each thread is holding the variable that the other thread wants, nothing occurs, and the threads remain deadlocked.



[Future object?](#Future)



A **ThreadPoolExecutor** uses a pool of threads to execute the tasks. The trick is in configuring it. I wanted:

<http://thesoftwarelife.blogspot.com/2011/03/threadpoolexecutor.html>

**public** **static** ExecutorService getExecutor() {

**if**(*execService*==**null**){

*execService* = **new** ThreadPoolExecutor(9, 27, 60L, TimeUnit.*SECONDS*, **new** SynchronousQueue<Runnable>());

}

**return** *execService*;

}

**Executors.newCachedThreadPool** use SynchronousQueue when the others (**Executors.newSingleThreadExecutor** and **Executors.newFixedThreadPool**) use LinkedBlockingQueue?

SynchronousQueue is a very special kind of queue - it implements a rendezvous approach (producer waits until consumer is ready, consumer waits until producer is ready) behind the interface of Queue.

futureObjList = execService.invokeAll(callableSet);

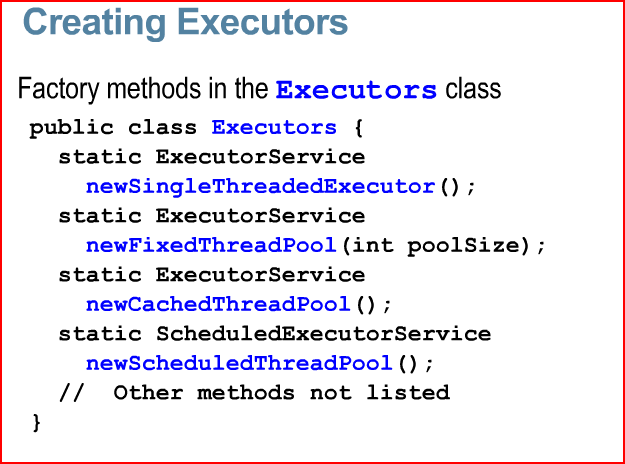
**for** (Future<Object> futureObj : futureObjList) {

Workbook resultWrkbook = (Workbook)futureObj.get();

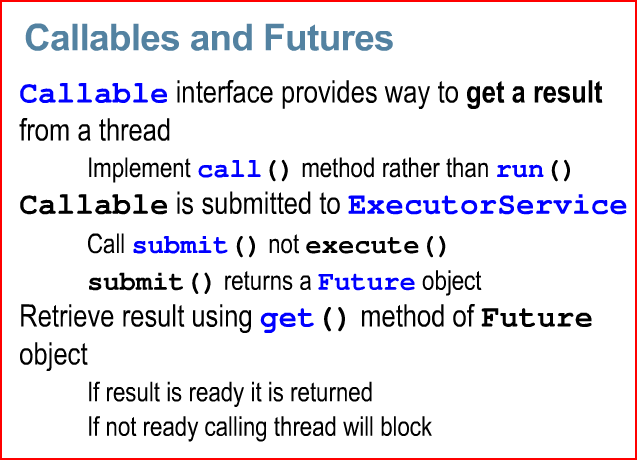
*log*.debug("future.get done ");

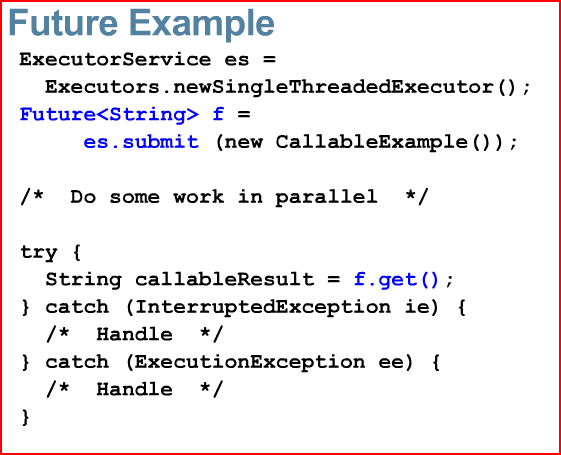
copySheet(workbook,resultWrkbook);

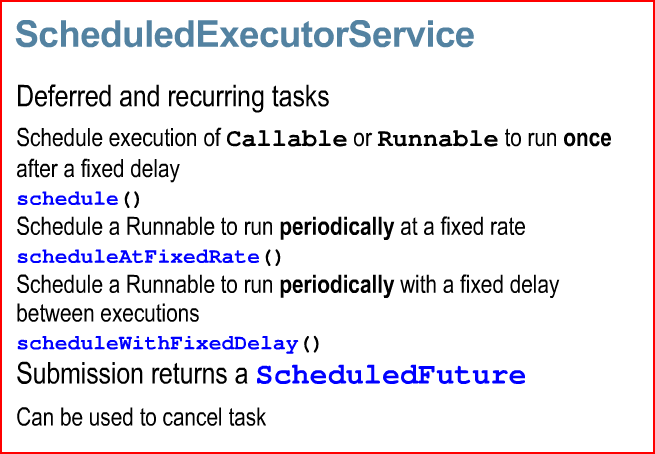
}

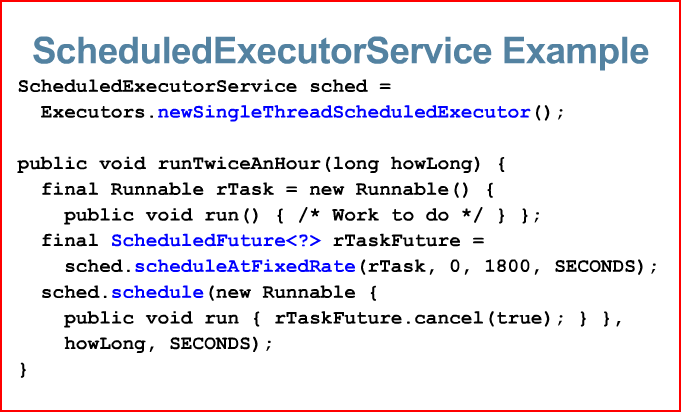


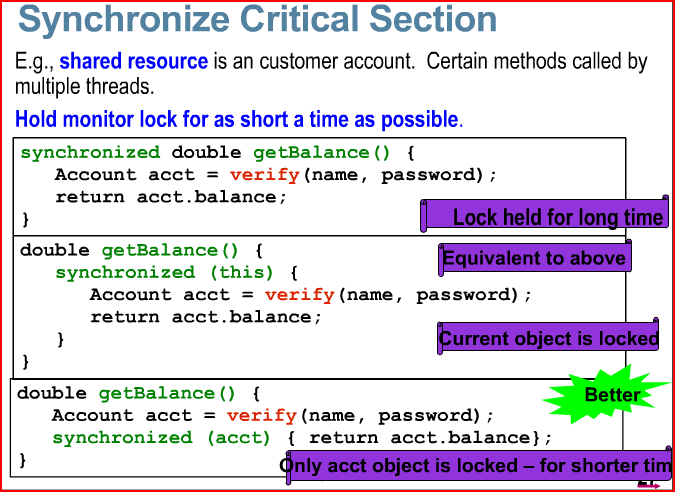
ExecutorService sExecutor = Executors.newSingleThreadExecutor().

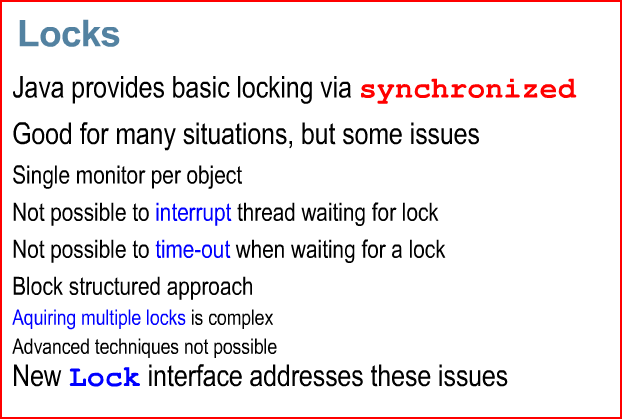


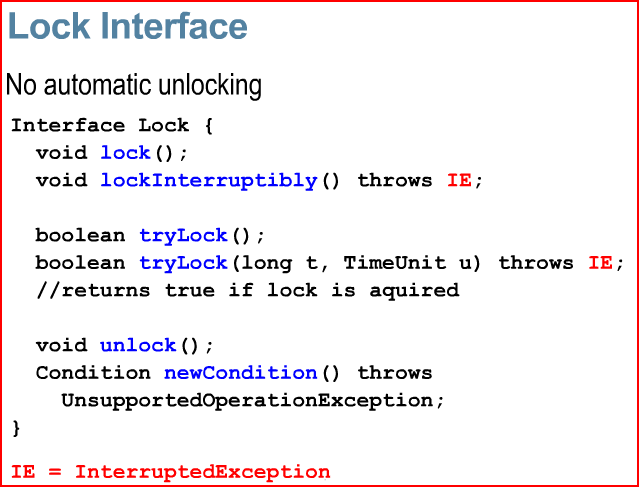


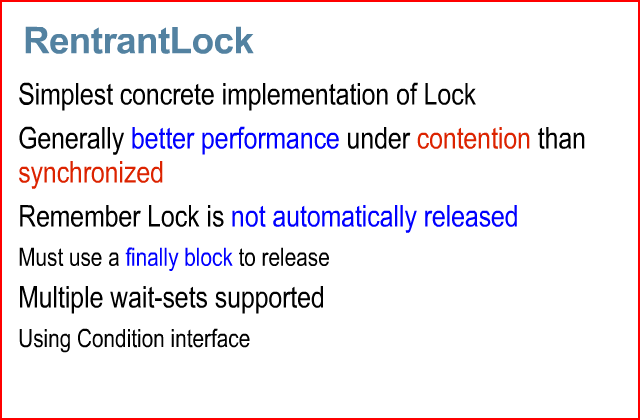


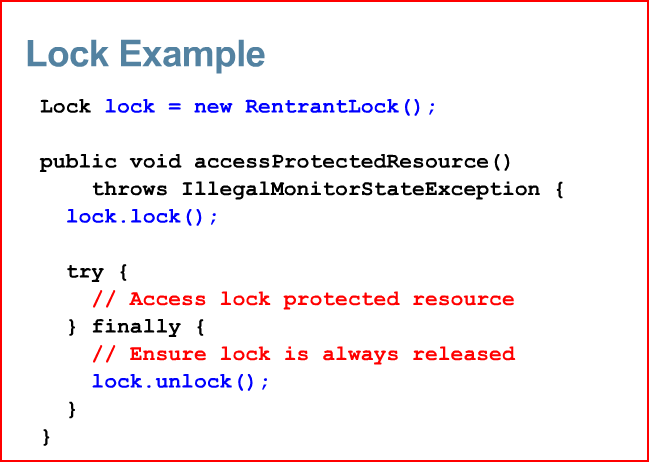


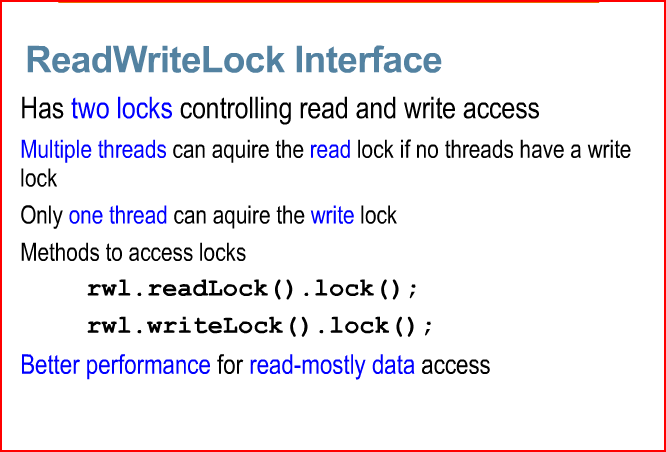


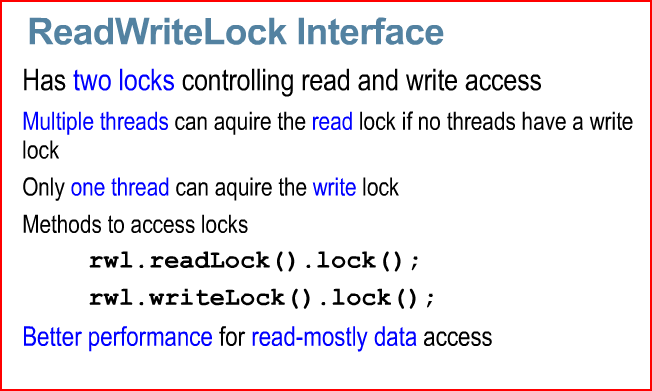


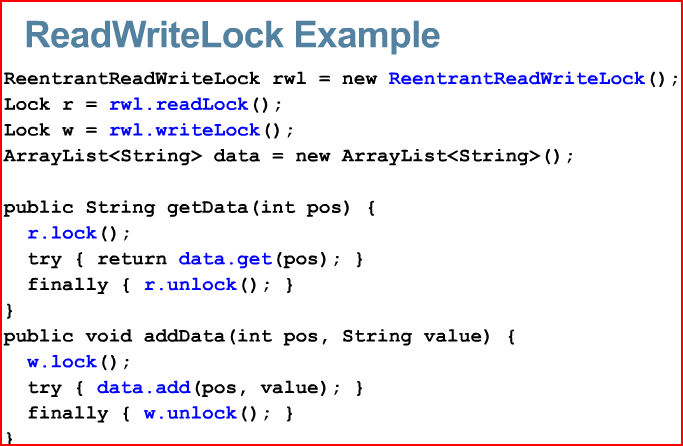


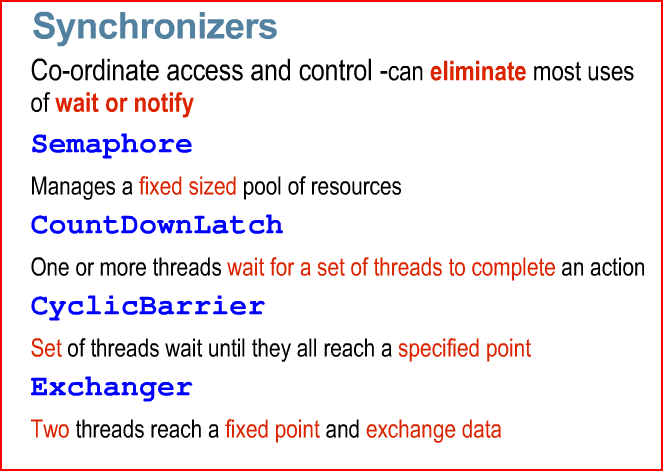


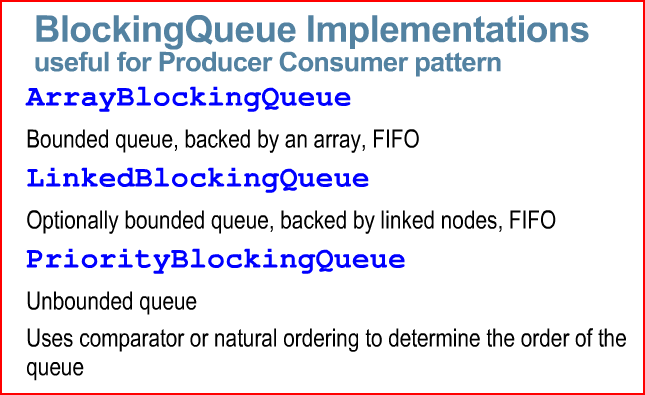




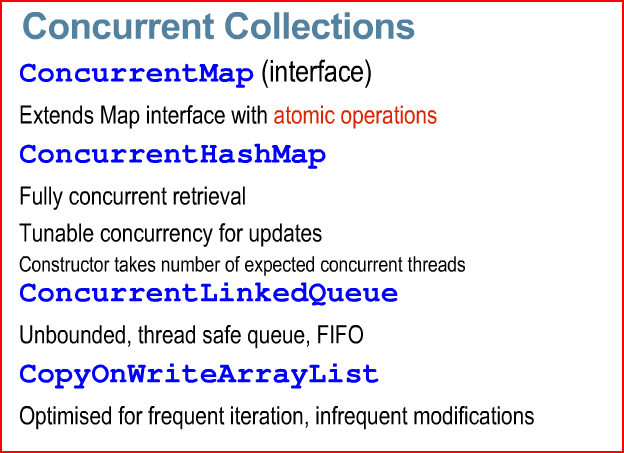








Set<String> mySet = Collections.newSetFromMap(new ConcurrentHashMap<String, Boolean>());



FOR SYNCHRNIZED SET

Set<String> mySet = Collections.newSetFromMap(new ConcurrentHashMap<String, Boolean>());

t looks like Java provides a concurrent Set implementation with its [ConcurrentSkipListSet](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListSet.html). A[SkipList Set](http://en.wikipedia.org/wiki/Skip_list) is just a special kind of set implementation. It still implements the Serializable, Cloneable, Iterable, Collection, NavigableSet, Set, SortedSet interfaces. This might work for you if you only need the Set interface.

Software that can do such things is known as *concurrent* software.

The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs. This lesson introduces the platform's basic concurrency support and summarizes some of the high-level APIs in the java.util.concurrent packages.

**Processes and Threads**

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

## Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space

## Threads

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process

**What is Executors?**

In large-scale applications, **it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as executors. The following subsections describe executors in detail.**

• Executor Interfaces define the three executor object types.

• Thread Pools are the most common kind of executor implementation.

• Fork/Join is a framework (new in JDK 7) for taking advantage of multiple processors.

*Executor Interfaces*

The java.util.concurrent package defines three executor interfaces:

• **Executor**, a simple interface that supports launching new tasks.

• **ExecutorService**, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.

• **ScheduledExecutorService**, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.  
  
Below is the example of **ExecutorService** using cachedThreadPool. This is using Callable instance, which does the task and return the result to calling programme.

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.E xecutors;

import java.util.concurrent.Future;

public class ThreadWithResultExample {

static ExecutorService exec = Executors.newCachedThreadPool();

public static void main(String...strings){

Future result = exec.submit(new Worker());

try {

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

} catch (InterruptedException e) {

e.printStackTrace();

} catch (ExecutionException e) {

e.printStackTrace();

}

exec.shutdown();

}

}

class Worker implements Callable {

@Override

public String call() throws Exception {

return (String) "result";

}

**COUNT DOWN LATCH**

<http://stackoverflow.com/questions/17827022/what-is-countdown-latch-in-java-multithreading>

CountDownLatch works in latch principle, main thread will wait until gate is open. One thread waits for n number of threads specified while creating CountDownLatch in Java.

Any thread, usually main thread of application, which calls CountDownLatch.await() will wait until count reaches zero or its interrupted by another thread. All other thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready.

As soon as count reaches zero, Thread awaiting starts running. One of the disadvantage of CountDownLatchis that it’s not reusable once count reaches to zero you cannot use CountDownLatch anymore.

edit:

Use CountDownLatch when one thread like main thread, require to wait for one or more thread to complete, before it can start processing.

Classical example of using CountDownLatch in Java is **any server side core Java application which uses** services architecture, where multiple services are provided by multiple threads and application can not start processing until all services have started successfully.

import java.util.concurrent.CountDownLatch;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

class Processor implements Runnable {

private CountDownLatch latch;

public Processor(CountDownLatch latch) {

this.latch = latch;

}

public void run() {

System.out.println("Started.");

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

latch.countDown();

}

}

// -----------------------------------------------------

public class App {

public static void main(String[] args) {

CountDownLatch latch = new CountDownLatch(3); // coundown from 3 to 0

ExecutorService executor = Executors.newFixedThreadPool(3); // 3 Threads in pool

for(int i=0; i < 3; i++) {

executor.submit(new Processor(latch)); // ref to latch. each time call new Processes latch will count down by 1

}

try

latch.await(); // wait untill latch counted down to 0

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("Completed.");

}

}

## BlockingQueue Usage

A BlockingQueue is typically used to have on thread produce objects, which another thread consumes. Here is a diagram that illustrates this principle:

|  |
| --- |
| A BlockingQueue with one thread putting into it, and another thread taking from it. |
| **A BlockingQueue with one thread putting into it, and another thread taking from it.** |

## Using Blocking Queue to implement Producer Consumer Pattern

*BlockingQueue* amazingly simplifies implementation of Producer-Consumer design pattern by providing outofbox support of blocking on put() and take(). Developer doesn't need to write confusing and critical piece of wait-notify code to implement communication. **BlockingQuue** is an interface and Java 5 provides different implantation like ArrayBlockingQueue and LinkedBlockingQueue , both implement FIFO order or elements, while ArrayLinkedQueue is bounded in nature LinkedBlockingQueue is optionally bounded. here is a complete **code example of Producer Consumer pattern** with BlockingQueue. Compare it with classic [wait notify](http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html) code, its much simpler and easy to understand.

import java.util.concurrent.BlockingQueue;

import java.util.concurrent.LinkedBlockingQueue;

import java.util.logging.Level;

import java.util.logging.Logger;

public class **ProducerConsumerPattern** {

    public static void main(String args[]){

**//Creating shared object**

     BlockingQueue sharedQueue = new LinkedBlockingQueue();

**//Creating Producer and Consumer Thread**

     Thread prodThread = new Thread(new Producer(sharedQueue));

     Thread consThread = new Thread(new Consumer(sharedQueue));

**//Starting producer and Consumer thread**

     prodThread.start();

     consThread.start();

    }

}

**//Producer Class in java**

class **Producer** implements **Runnable** {

    private final **BlockingQueue** sharedQueue;

    public Producer(BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        for(int i=0; i<10; i++){

            try {

                System.out.println("Produced: " + i);

                sharedQueue.put(i);

            } catch (InterruptedException ex) {

                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);

            }

        }

    }

}

**//Consumer Class in Java**

class Consumer implements Runnable{

    private final BlockingQueue sharedQueue;

    public Consumer (BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        while(true){

            try {

                System.out.println("Consumed: "+ sharedQueue.take());

            } catch (InterruptedException ex) {

                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);

            }

        }

    }

}

You see Producer Thread  produced number and Consumer thread consumes it in FIFO order because blocking queue allows elements to be accessed in FIFO.

That’s all on **How to use Blocking Queue to solve Producer Consumer problem** or **example of Producer consumer design pattern**. I am sure its much better than wait notify example but be prepare with both if you are going for any Java Interview as Interview may ask you both way.

Read more: <http://javarevisited.blogspot.com/2012/02/producer-consumer-design-pattern-with.html#ixzz35PlEwXuY>

# [what's the difference between CyclicBarrier/CountDownLatch and join in Java?](http://stackoverflow.com/questions/21808814/whats-the-difference-between-cyclicbarrier-countdownlatch-and-join-in-java)

<http://stackoverflow.com/questions/21808814/whats-the-difference-between-cyclicbarrier-countdownlatch-and-join-in-java>

The main difference between **CyclicBarrier** and **CountDownLatch** is that CyclicBarrier is reusable and CountDownLatch is not. You can reuse CyclicBarrier by calling reset() method which resets the barrier to its initial state.

CountDownLatch is good for one time event like application/module start-up time and CyclicBarrier can be used to in case of recurrent event e.g. concurrently (re-)calculating each time when the input data changed

**What is difference between CountDownLatch and CyclicBarrier in Java?**  
[CountDownLatch](http://javarevisited.blogspot.sg/2012/07/countdownlatch-example-in-java.html)and [CyclicBarrier in Java](http://javarevisited.blogspot.sg/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html)are two important concurrency utility which is added on Java 5 Concurrency API. Both are used to implement scenario, where one thread has to wait for other thread before starting processing but there is difference between them. Key point to mention, while answering this question is that CountDownLatch is not reusable once count reaches to zero, while CyclicBarrier can be reused even after barrier is broken. You can also see my previous article [difference between CyclicBarrier and CountDownLatch in Java](http://java67.blogspot.sg/2012/08/difference-between-countdownlatch-and-cyclicbarrier-java.html) for more detailed answer of this concurrency interview question and a real life example of where to use these concurrency utilities

Busy Spinning?

http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html

 It's a wait strategy, where one thread wait for a condition to become true, but instead of calling wait or sleep method and releasing CPU, it just spin. This is particularly useful if condition is going to be true quite quickly i.e. in millisecond or micro second. Advantage of not releasing CPU is that, all cached data

**Future Object :=**

public interface **Future<V>**

A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete.

* The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled. If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.

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

interface ArchiveSearcher { String search(String target); }

class App {

ExecutorService executor = ...

ArchiveSearcher searcher = ...

void showSearch(final String target)

throws InterruptedException {

Future<String> future

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

public String call() {

return searcher.search(target);

}});

displayOtherThings(); // do other things while searching

try {

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

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

}

}

The [FutureTask](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/FutureTask.html) class is an implementation of Future that implements Runnable, and so may be executed by an Executor. For example, the above construction with submit could be replaced by:

FutureTask<String> future =

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

public String call() {

return searcher.search(target);

}});

executor.execute(future);

Memory consistency effects: Actions taken by the asynchronous computation [*happen-before*](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility) actions following the corresponding Future.get() in another thread.

[**http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html**](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html)

**What is CyclicBarrier in Java**

CyclicBarrier in Java is a synchronizer introduced in JDK 5 on java.util.Concurrent package along with other concurrent utility like [Counting Semaphore](http://javarevisited.blogspot.sg/2012/05/counting-semaphore-example-in-java-5.html), [BlockingQueue](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html), [ConcurrentHashMap](http://javarevisited.blogspot.sg/2011/04/difference-between-concurrenthashmap.html) etc. CyclicBarrier is similar to CountDownLatch which we have seen in last article  [What is CountDownLatch in Java](http://javarevisited.blogspot.sg/2012/07/countdownlatch-example-in-java.html) and allows multiple threads to wait for each other (barrier) before proceeding. Difference between CountDownLatch and CyclicBarrier is a also very [popular multi-threading interview question](http://javarevisited.blogspot.sg/2011/07/java-multi-threading-interview.html) in Java. CyclicBarrier is a natural requirement for concurrent program because it can be used to perform final part of task once individual tasks  are completed. All threads which [wait](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html) for each other to reach barrier are called parties, CyclicBarrier is initialized with number of parties to be wait and threads wait for each other by calling CyclicBarrier.await() method which is a [blocking method in Java](http://javarevisited.blogspot.sg/2012/02/what-is-blocking-methods-in-java-and.html) and  blocks until all Thread or parties call await(). In general calling await() is shout out that Thread is waiting on barrier. await() is a blocking call but can be timed out or Interrupted by other thread. In this Java concurrency tutorial we will see *What is CyclicBarrier in Java*  and  an example of CyclicBarrier on which three Threads will wait for each other before proceeding further.

**import** java. util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
**import** java.util.logging.Level;  
**import** java.util.logging.Logger;  
  
/\*\*  
 \* Java program to demonstrate how to use CyclicBarrier in Java. CyclicBarrier is a

 \* new Concurrency Utility added in Java 5 Concurrent package.

 \*  
 \* @author Javin Paul  
 \*/  
**public** **class** CyclicBarrierExample {  
  
    *//Runnable task for each thread*  
    **private** **static** **class** Task **implements** [**Runnable**](http://javarevisited.blogspot.sg/2012/01/difference-thread-vs-runnable-interface.html) {  
  
        **private** **CyclicBarrier** barrier;  
  
        **public** Task(**CyclicBarrier** barrier) {  
            **this**.barrier = barrier;  
        }  
  
        @**Override**  
        **public** **void** run() {  
            **try** {  
                **System**.out.println(**Thread**.currentThread().getName() + " is waiting on barrier");  
                barrier.await();  
                **System**.out.println(**Thread**.currentThread().getName() + " has crossed the barrier");  
            } **catch** (**InterruptedException** ex) {  
                **Logger**.getLogger(CyclicBarrierExample.**class**.getName()).log(**Level**.SEVERE, **null**, ex);  
            } **catch** (**BrokenBarrierException** ex) {  
                **Logger**.getLogger(CyclicBarrierExample.**class**.getName()).log(**Level**.SEVERE, **null**, ex);  
            }  
        }

    }  
  
    **public** **static** **void** main(**String** args[]) {  
  
        *//creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()*  
        **final** **CyclicBarrier** cb = **new** **CyclicBarrier**(3, **new** **Runnable**(){  
            @**Override**  
            **public** **void** run(){  
                *//This task will be executed once all thread reaches barrier*  
                **System**.out.println("All parties are arrived at barrier, lets play");  
            }  
        });  
  
        *//starting each of thread*  
        **Thread** t1 = **new** **Thread**(**new** Task(cb), "Thread 1");  
        **Thread** t2 = **new** **Thread**(**new** Task(cb), "Thread 2");  
        **Thread** t3 = **new** **Thread**(**new** Task(cb), "Thread 3");  
  
        t1.start();  
        t2.start();  
        t3.start();  
        
    }  
}  
  
**Output:**  
**Thread** 1 is waiting on barrier  
**Thread** 3 is waiting on barrier  
**Thread** 2 is waiting on barrier  
All parties are arrived at barrier, lets play  
**Thread** 3 has crossed the barrier  
**Thread** 1 has crossed the barrier  
**Thread** 2 has crossed the barrier

Read more: <http://javarevisited.blogspot.com/2012/07/cyclicbarrier-example-java-5-concurrency-tutorial.html#ixzz3Le9yisJ2>

**http://stackoverflow.com/questions/11793067/how-does-concurrenthashmap-work-internally**

The ConcurrentHashMap is very similar to the java.util.HashTable class, except that ConcurrentHashMap offers better concurrency than HashTable or synchronizedMap does. ConcurrentHashMap does not lock the Map while you are reading from it. Additionally,ConcurrentHashMap does not lock the entire Map when writing to it. It only locks the part of the Map that is being written to, internally.

Another difference is that ConcurrentHashMap does not throw ConcurrentModificationExceptionif the ConcurrentHashMap is changed while being iterated. The Iterator is not designed to be used by more than one thread though whereas synchronizedMap may throw ConcurrentModificationException

**Object Level vs Class Level lock in Java**

**<https://howtodoinjava.com/java/multi-threading/object-vs-class-level-locking/>**

**Class level lock** prevents multiple threads to enter in synchronized block in any of all available instances of the class on runtime. This means if in runtime there are 100 instances of DemoClass, then only one thread will be able to execute demoMethod() in any one of instance at a time, and all other instances will be locked for other threads.

Class level locking should always be done **to make static data thread safe**. As we know that [**static**](https://howtodoinjava.com/java/basics/java-static-keyword/) keyword associate data of methods to class level, so use locking at static fields or methods to make it on class level.

## Object level lock vs class level lock – Important notes

1. Synchronization in Java guarantees that no two threads can execute a synchronized method, which requires same lock, simultaneously or concurrently.

**Difference between ExecutorService.submit and ExecutorService.execute()**

However, submit(Callable<T>) returns a Future object which allows a way for you to programatically cancel the running thread later as well as get the T that is returned when the Callable completes. See [JavaDoc of Future](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html) for more details

Future<?> future = executor.submit(longRunningJob);

...

//long running job is taking too long

future.cancel(true);

Moreover, if future.get() == null and doesn't throw any exception then Runnable executed successfully

The difference is that execute simply starts the task without any further ado, whereas submit returns a Future object to manage the task. You can do the following things with the Future object:

* Cancel the task prematurely, with the cancel method.
* Wait for the task to finish executing, with get.

The Future interface is more useful if you submit a Callable to the pool. The return value of the call method will be returned when you call Future.get. If you don't maintain a reference to the Future, there is no difference.