<http://www.deadcoderising.com/java8-writing-asynchronous-code-with-completablefuture/>

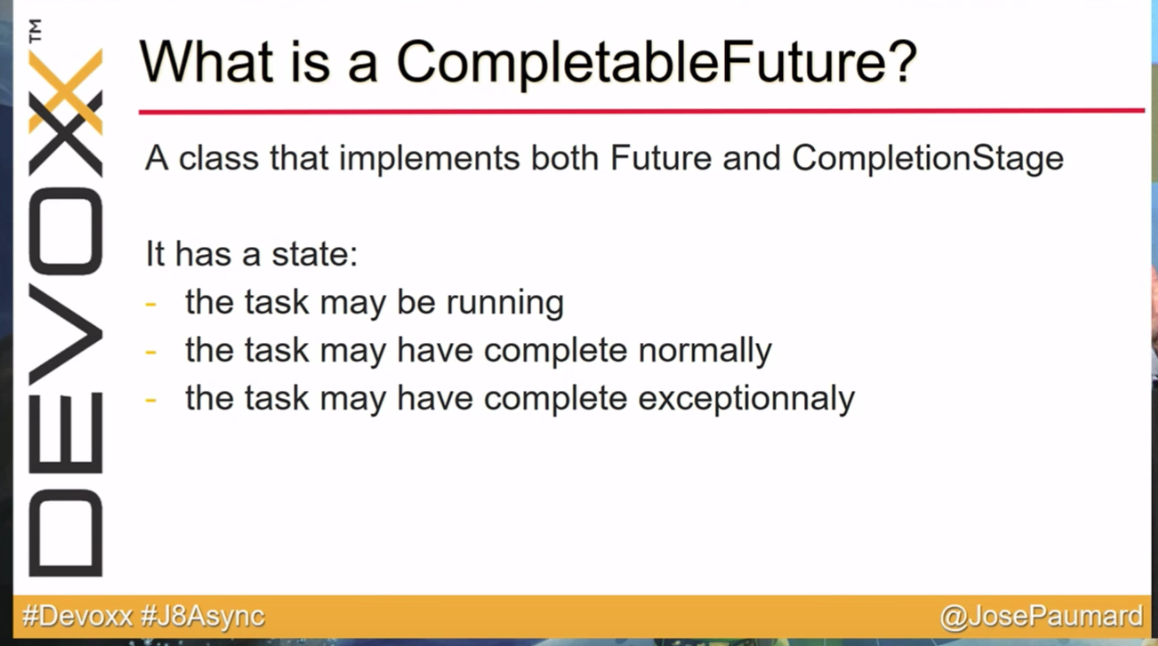
Beside implementing the Future interface, CompletableFuture also implements the CompletionStage interface.

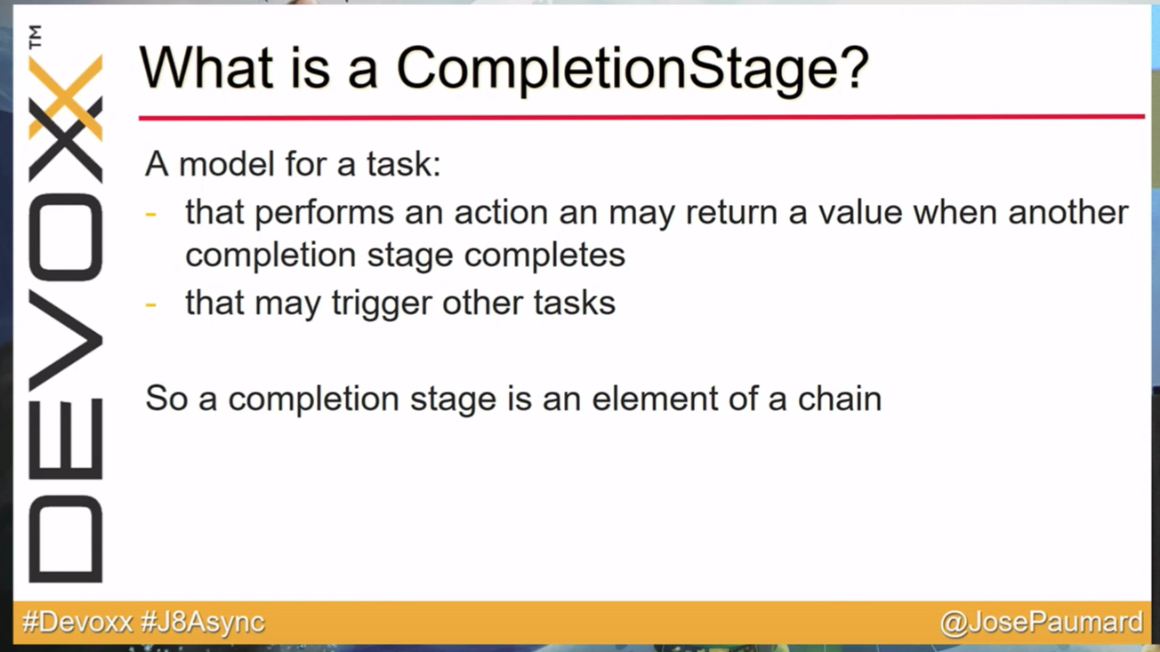
A CompletionStage is a promise. It promises that the computation eventually will be done.

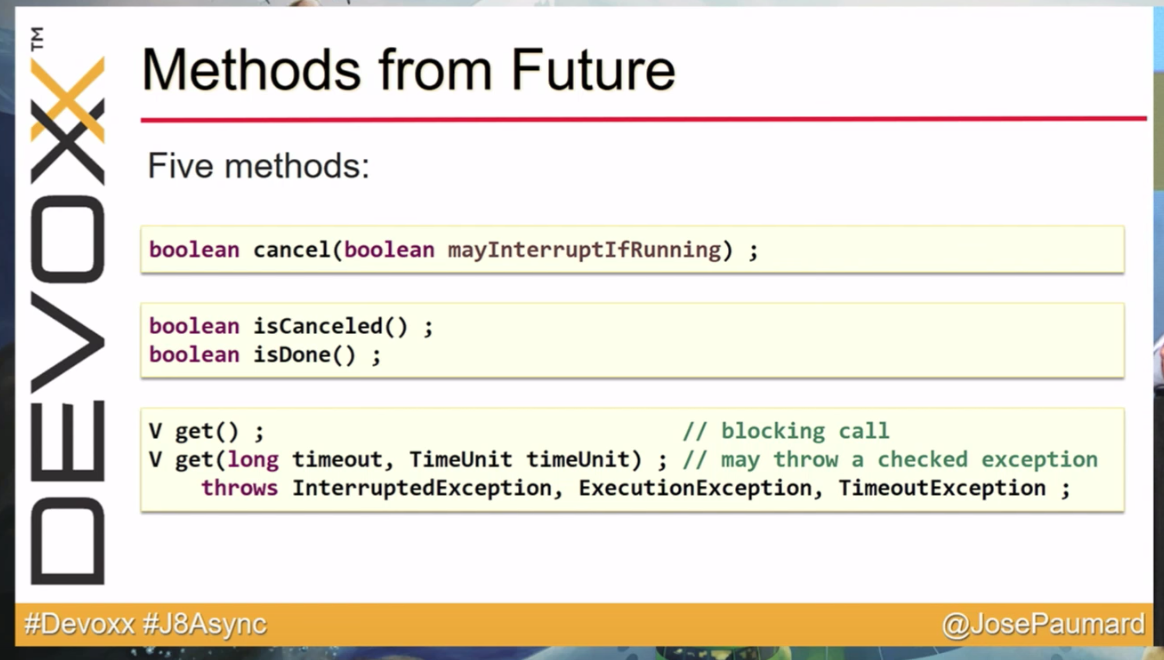
The great thing about the CompletionStage is that it offers a vast selection of methods that let you attach callbacks that will be executed on completion.

This way we can build systems in a non-blocking fashion.

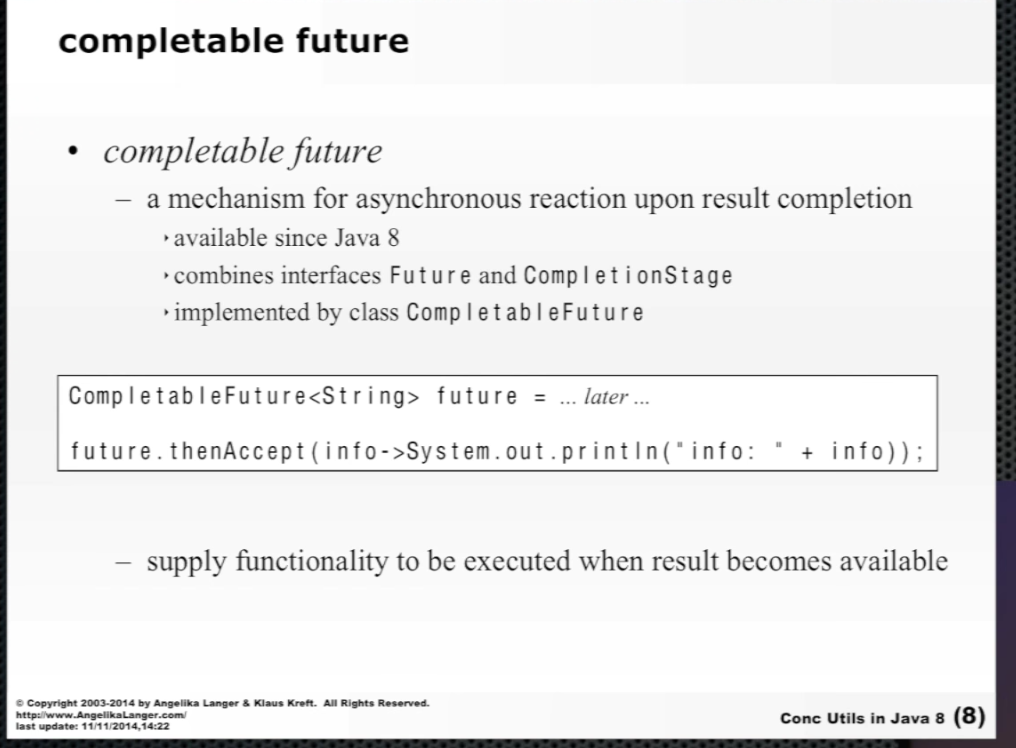
<https://www.youtube.com/watch?v=HdnHmbFg_hw>







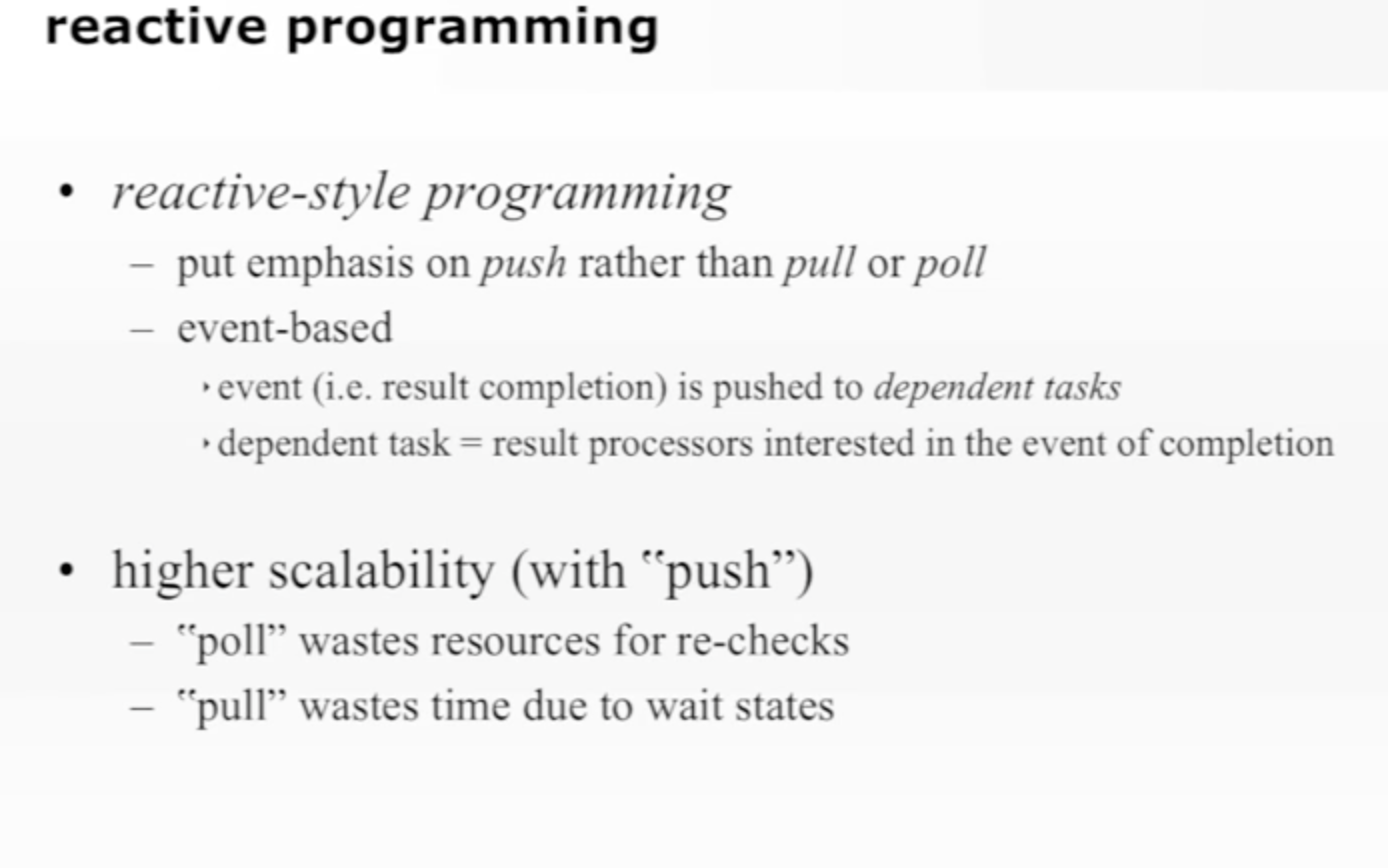
<https://www.youtube.com/watch?v=Q_0_1mKTlnY>

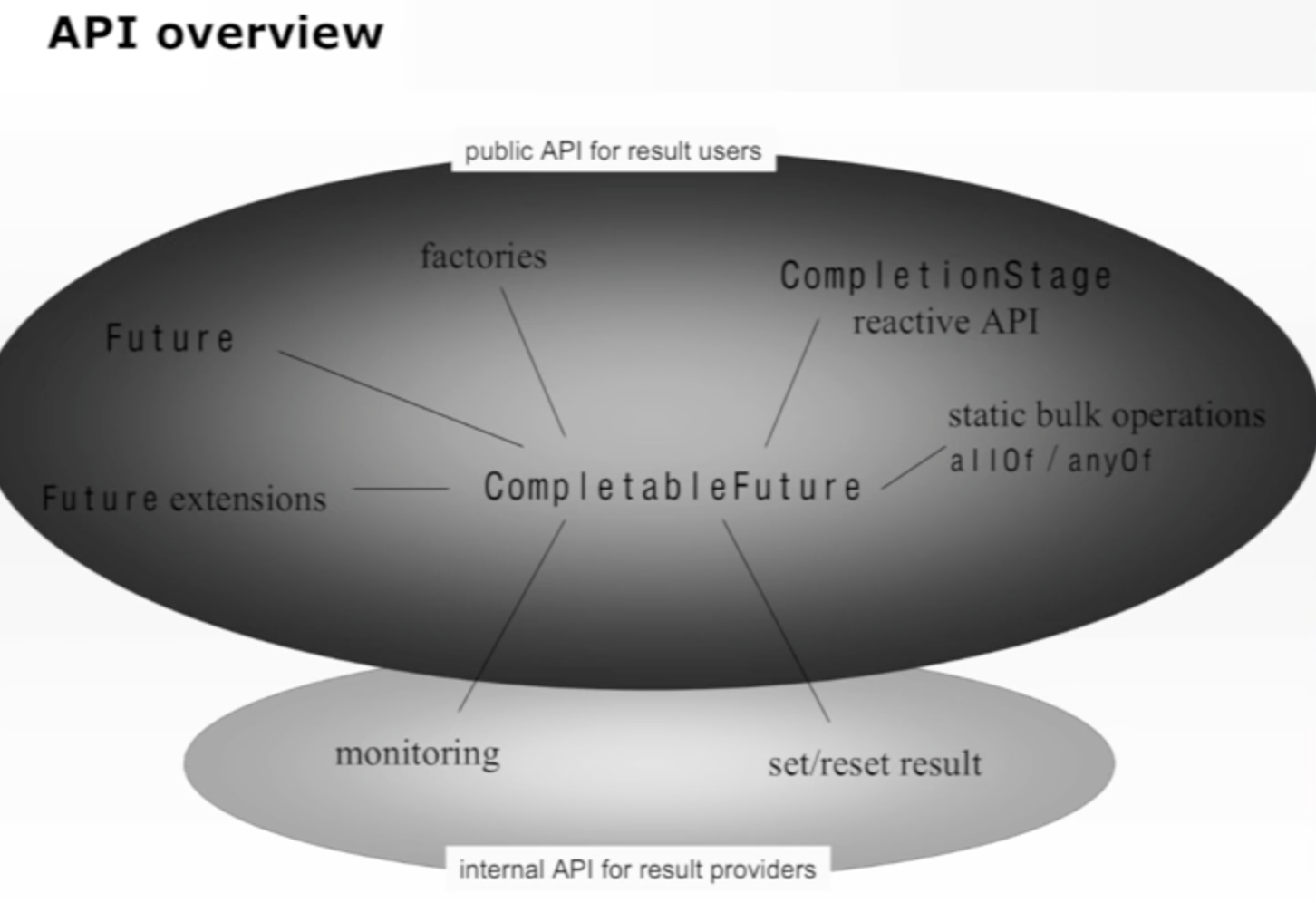


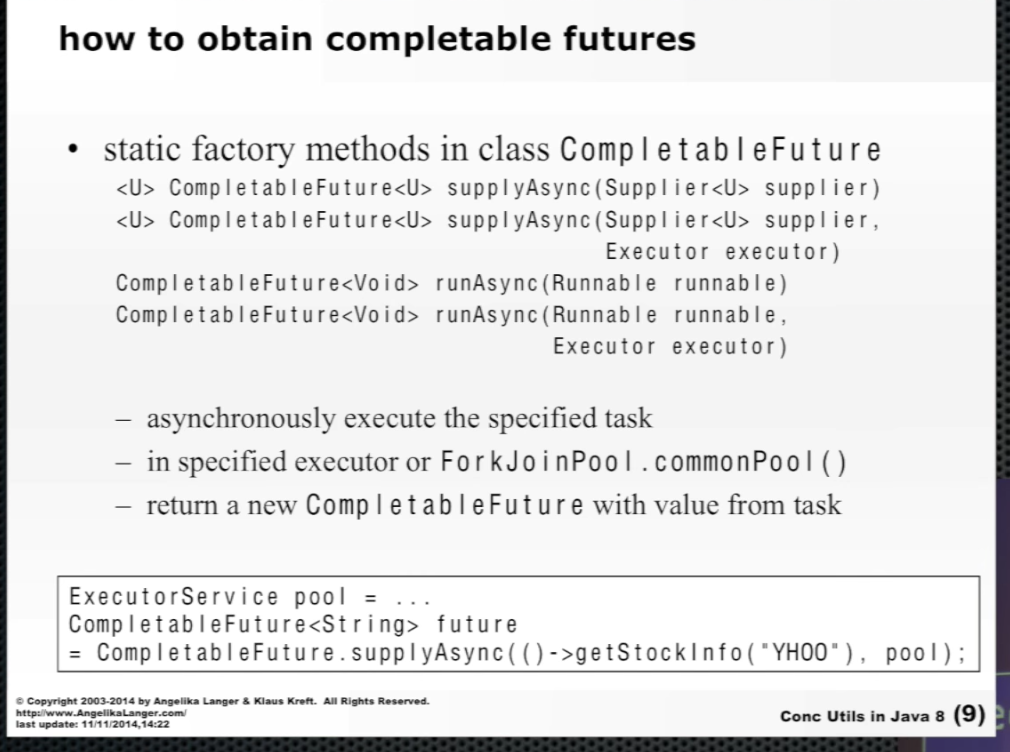
Instead of waiting for the thread to complete using future.get() method, with CompletableFuture, I can use thenAccept method which accepts a code that will executed right after the result is available in the CompletableFuture. This lambda passed to thenAccept method takes result as an input parameter.

Lambda passed to thenAccept directly without creating a task in thread pool, but if you use thenAcceptAsyn, it will create a task in the thread pool.

Lambda passed to thenAccept will also be executed as inside thread pool. It takes a Consumer as an argument. So, it will return CompletableFuture<Void>.







Instead of using, now you use a Supplier. Callable used to throw checked exception, Supplier doesn’t.

CompletableFuture.supplyAsync(supplier) method does something like below

CompletableFuture<U> d = **new** CompletableFuture<U>();  
executorService.execute(**new** AsyncSupply<U>(d, f));  
**return** d;

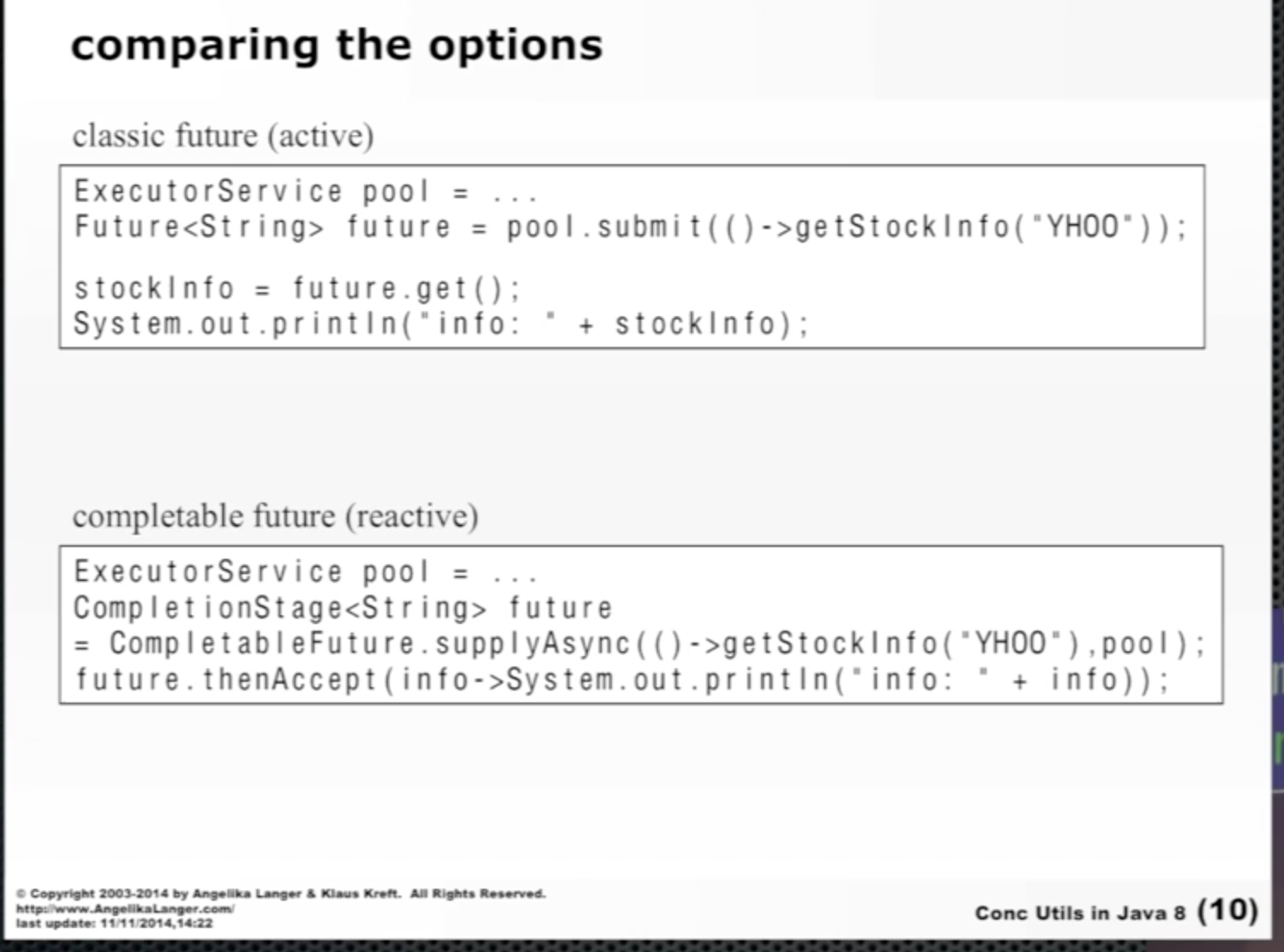
AysncSupply is a ForkAndjoinTask implementing Runnable.

If you don’t pass your own ExecutorService (thread pool), it uses ForkJoinPool.commonPool().

CompletableFuture.runAsync(runnable) method does something like below

CompletableFuture<Void> d = **new** CompletableFuture<Void>();  
e.execute(**new** AsyncRun(d, f));  
**return** d;

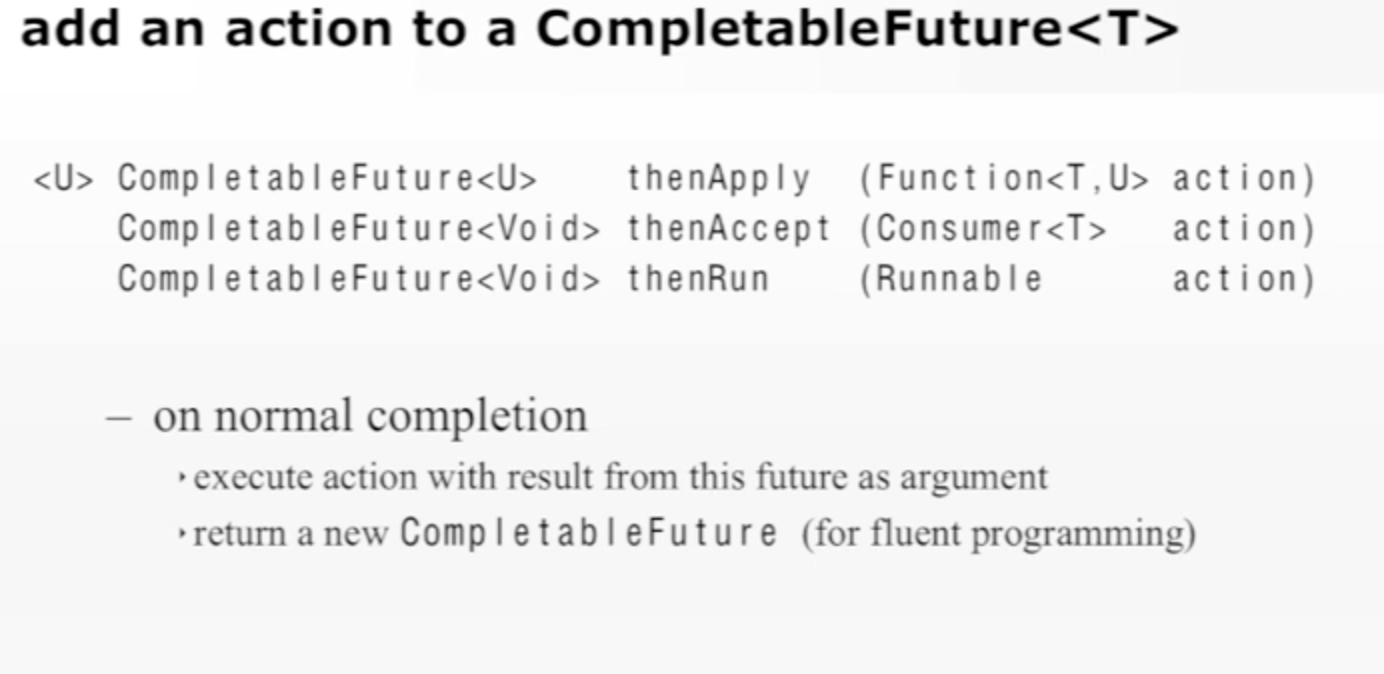
It returns CompletableFuture with nothing in it. This CompletableFuture is useful just to know whether the task in finished or not.





* whenComplete(BiConsumer<CompletableFuure’s result, CompletableFuture’s exception>)
* thenApply(Function< CompletableFuture’s result, some other object that you want as a result inside a new CompletableFuture>)
* thenAccept(Consumer<CompletableFuture’s result>)
* thenRun(Runnable)

These methods are from CompletionStage interface, which are overridden in CompletableFuture class.



+

CompletableFuture<T> **whenComplete**(BiConsumer<T, Throwable> action)

CompletableFuture<U> **thenCompose**(Function<T, CompletionStage<U>> fn)

CompletableFuture<V> **thenCombine**(CompletionStage<U> other, BiFunction<T,U,V> fn)

CompletableFuture<Void> **runAfterEither**(CompletionStage<?> other, Runnable action)

CompletableFuture<Void> **runAfterBoth**(CompletionStage<?> other, Runnable action)

CompletableFuture<Void> **acceptEither**(CompletionStage<T> other, Consumer<T> action)

CompletableFuture<U> **applyToEither**(CompletionStage<T> other, Function<T, U> fn)

CompletableFuture<T> **exceptionally**(Function<Throwable, T> fn)

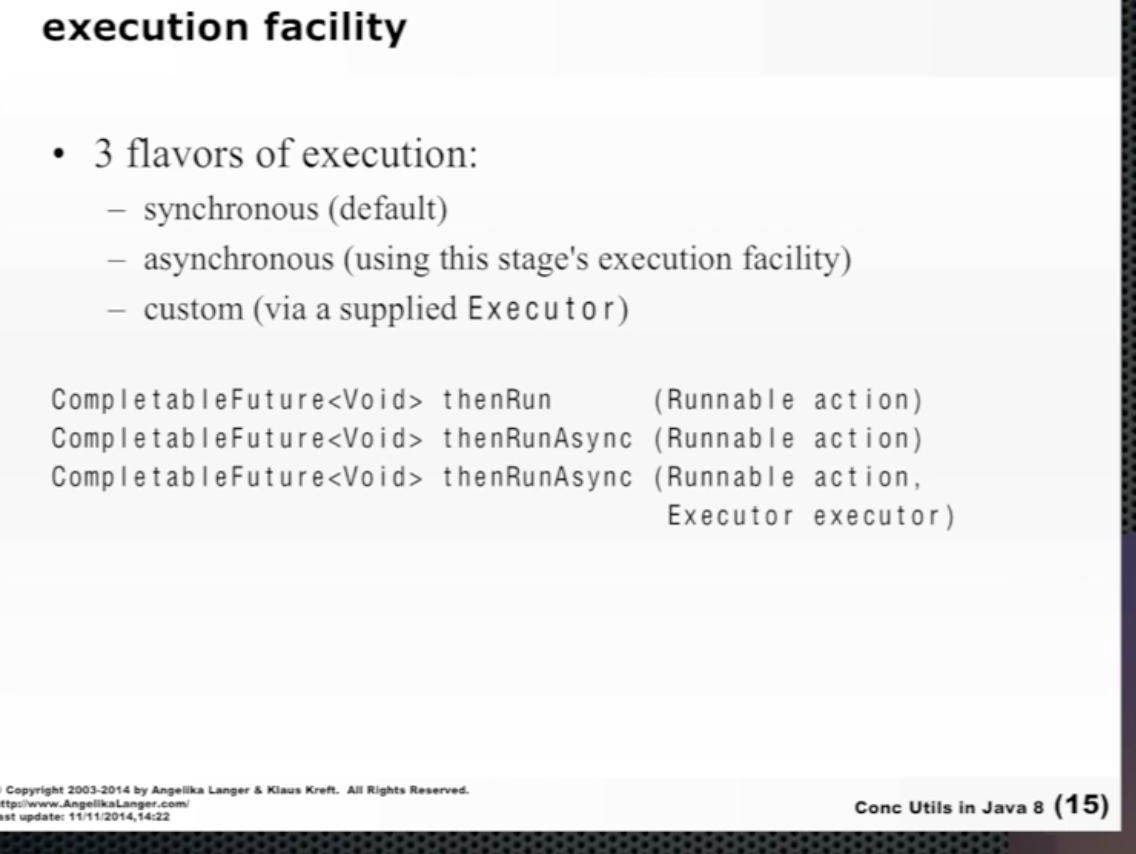
CompletableFuture<U> **handle**(BiFunction<T, Throwable, U> fn)

Here T is an output from CompletableFuture on which these reaction methods are called.

These are different flavors of actions that can be applied as reaction to CompletableFuture<T> (which is returned by methods like supplyAsync).

Each one of these reaction methods come with 3 different flavors.

* Sync
* Async (**will be submitted to ExecutorService type ForkJoinPool’s common thread pool**)
* Async **with your own thread pool**



There is one more method ‘thenCombine’ that actually combines two CompletableFutures and produce a result out of both of their result.

All of the above mentioned Reaction method are actually overridden from CompletionStage.

There are some methods which are actually part of CompletableFuture only, they are not inherited from anywhere else.

Method overridden from Future

**boolean** isDone()

Returns true if completed in any fashion: normally, exceptionally, or via cancellation.

T get() **throws** InterruptedException, ExecutionException

It’s **BLOCKING** method. Waits if necessary for this future to complete, and then returns its result.

T get(**long** timeout, TimeUnit unit)

It’s **BLOCKING** method till times out. Waits if necessary for at most the given time for this future to complete, and then returns its result, if available.

T getNow(T valueIfAbsent)

It doesn’t block. Returns the result value (or throws any encountered exception) if completed, else returns the given valueIfAbsent.

T join()

It is same as get(). It calls get() internally. Only difference is that if get() throws a checked exception, join() converts it into RuntimeException CompletionException.

Arbitrary-arity constructions methods

CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)

Returns a new CompletableFuture that is completed when all of the given CompletableFutures complete. If any of the given CompletableFutures complete exceptionally, then the returned CompletableFuture also does so, with a CompletionException holding this exception as its cause.

CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

Control (Result Producing) and status methods

**boolean** complete(T value)

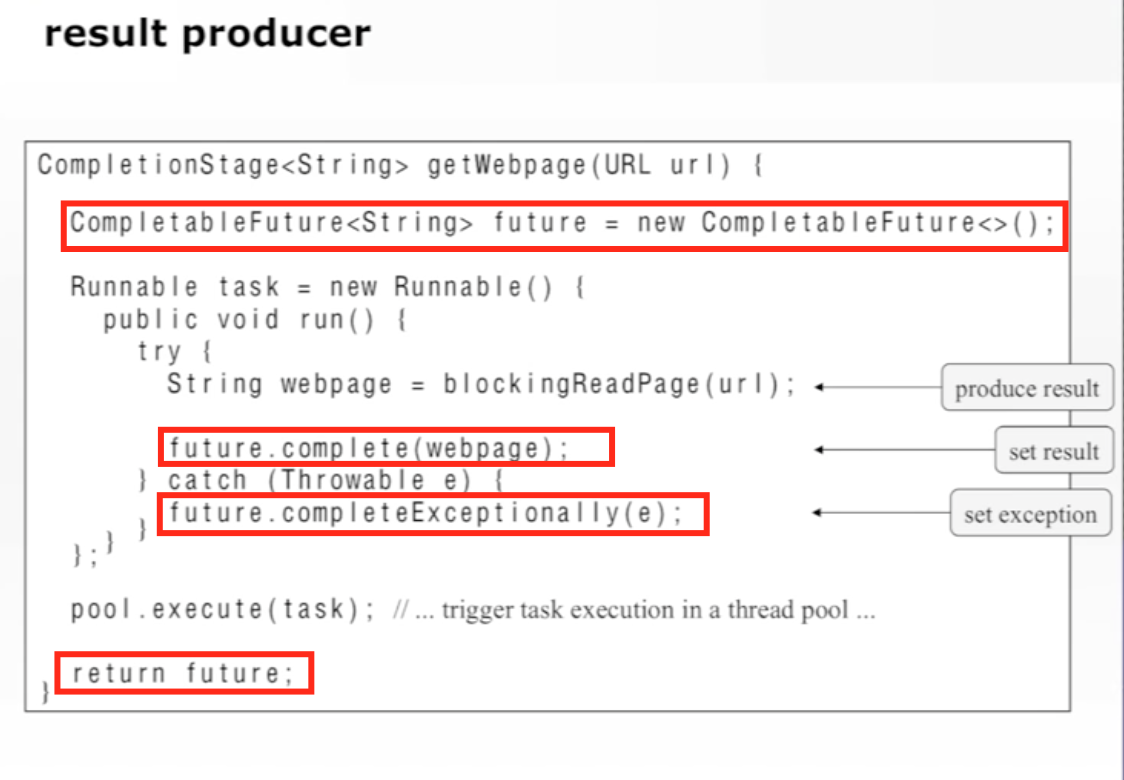
If not already completed, sets the value returned by get() and related methods to the given value.

**boolean** completeExceptionally(Throwable ex)

If not already completed, causes invocations of get() and related methods to throw the given exception.

**boolean** isCompletedExceptionally()

‘complete’ method is a very interesting method. It is mainly used when your client sends a request and you immediately have to send CompletableFuture back and run the task in background in another thread and populate returned empty CompletableFuture with the result.



In this example, client calls getWebpage method. It takes time to find the actual result, but meanwhile you can return an empty CompletableFuture and later on when the result is available, you can populate that empty CompletableFuture with the result.

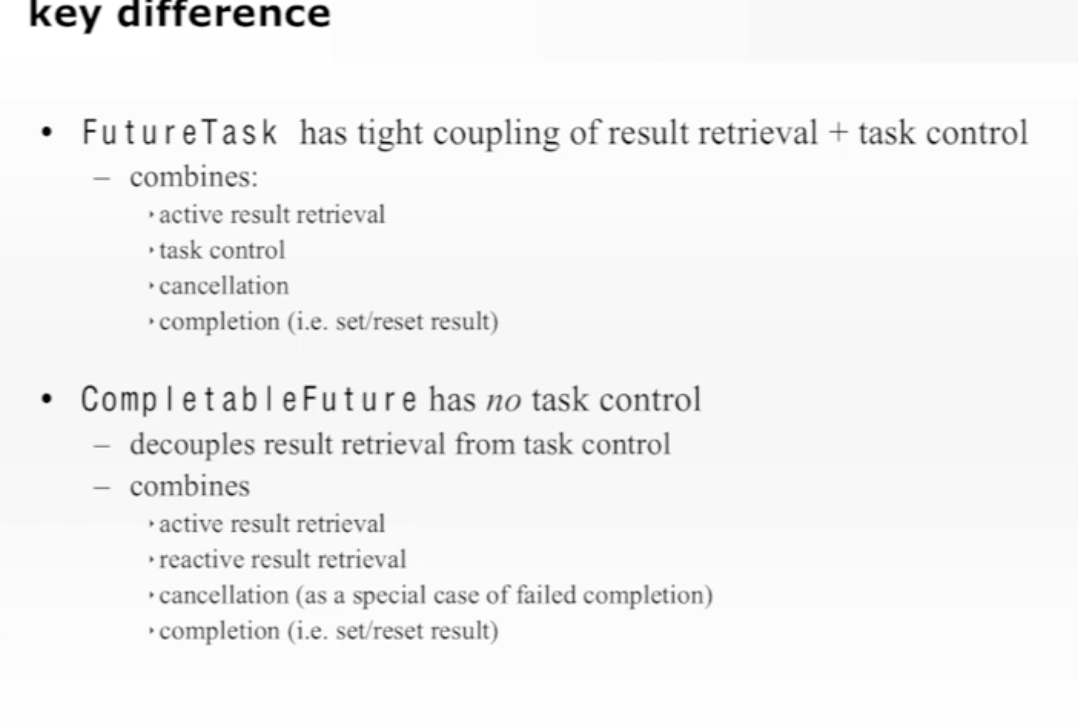
**boolean** cancel(**boolean** mayInterruptIfRunning)

If not already completed, completes this CompletableFuture with aCancellationException.

**boolean** isCancelled()

‘cancel’ method is very interesting. In case of Future, when you call future.cancel(true), it actually interrupts the running FutureTask and FutureTask will stop running. FutureTask has a reference of Future object.

But CompletableFuture decouples Task running and the result. It doesn’t know that result is created due to which task. In this case, if you call completableFuture.cancel(true), it will just populate exception in result, but actually will not interrupt the running task. This makes cancel method of CompletableFuture not so useful. So, it’s a bad idea to call cancel on CompletableFuture.

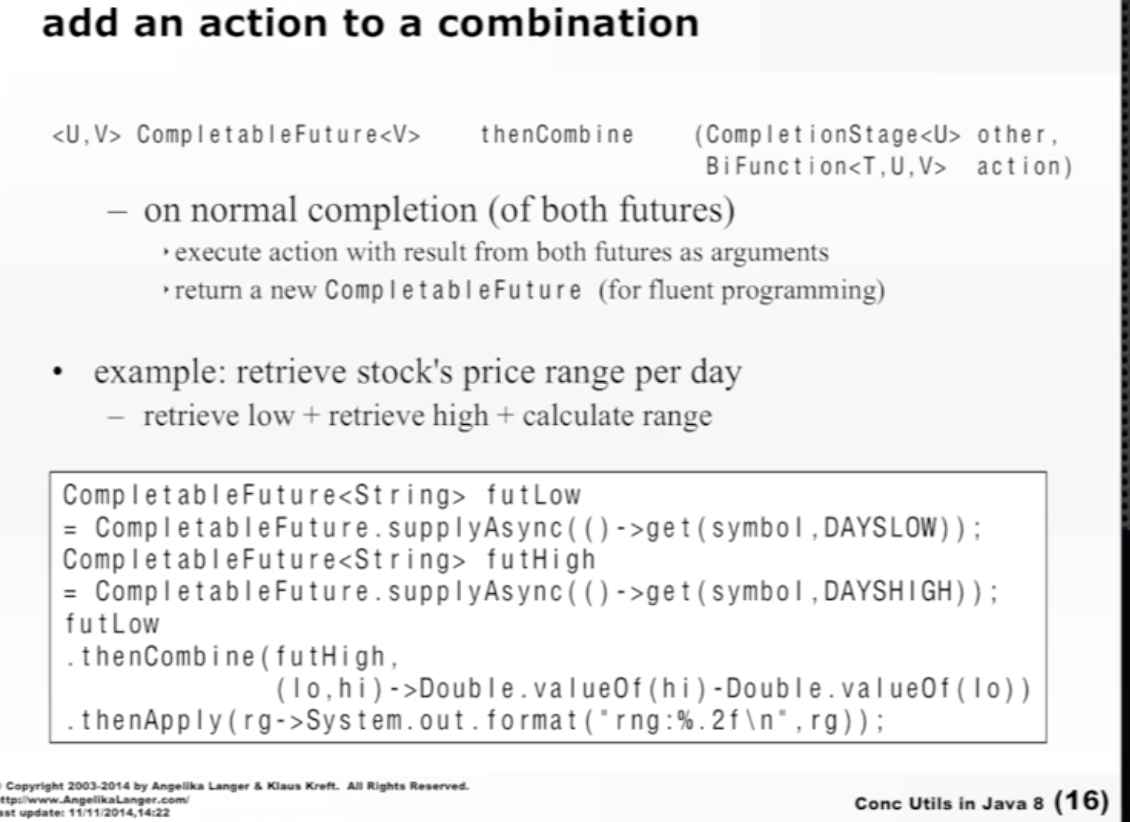


**void** obtrudeValue(T value)

**void** obtrudeException(Throwable ex)

If not already completed, then just force the result to be set as passed value/exception

**int** getNumberOfDependents()

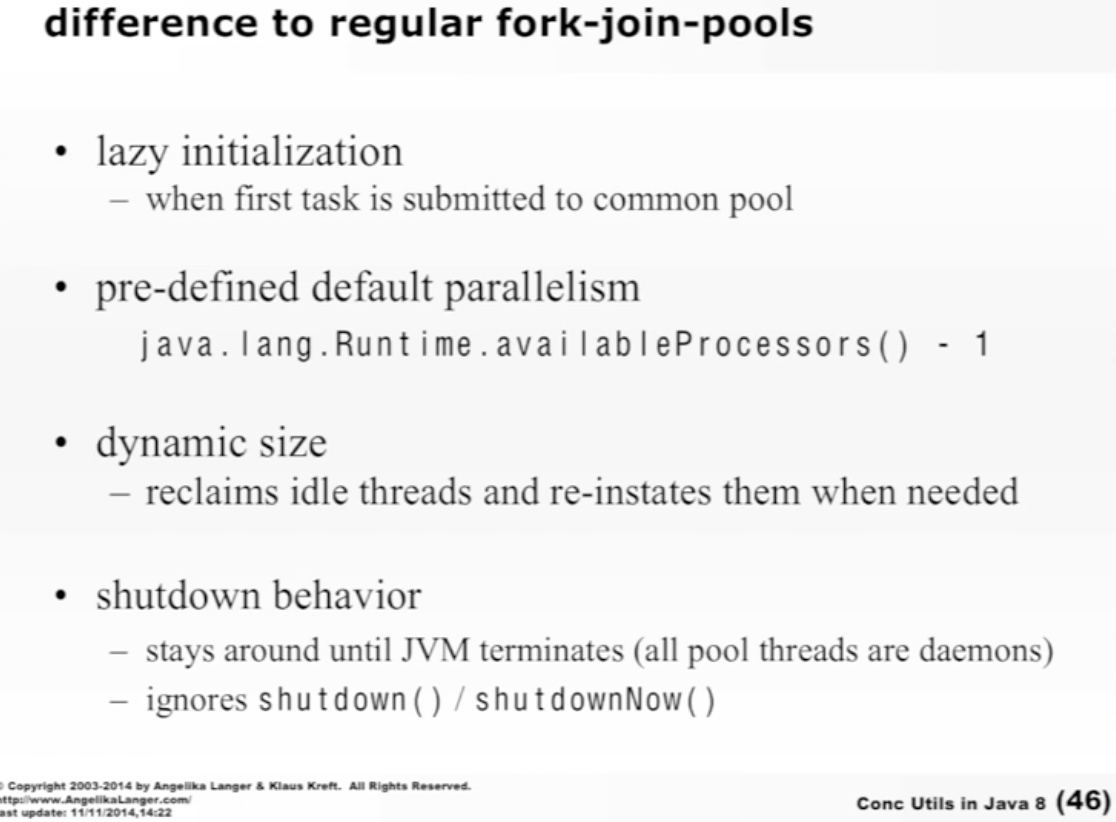


**ForkAndJoin Common Pool**

****

* Parallel Stream uses this common pool.
* CompletableFuture also uses this common pool unless another execution pool is provided by you.

It’s singleton.



Default number of max threads in common pool will be number of available processors – 1.

Its shutdown behavior is interesting. It’s a pool of **daemon** threads. So when JVM ends (Main thread), all daemon threads will end.

Below example shows that as soon as main thread ends (jvm ends), all daemon threads will end.

This is actually a bit problematic sometimes. e.g. you have submitted a task using CompletableFuture and it still running in common pool, but what if your main thread exits. It will terminate entire common pool and your background task will stop running.

So, before terminating your main thread, you can check a some state of the common pool.

**public class** DaemonTest {  
  
 **public static void** main(String[] args) {  
 Runnable r = () -> {  
 **new** WorkerThread().start();  
  
 **try** {  
 Thread.*sleep*(7500);  
 } **catch** (InterruptedException e) {  
 *// handle here exception* }  
 System.***out***.println(**"Parent Thread of Worker Thread is ending"**) ;  
 };  
 **new** Thread(r).start();  
  
 **try** {  
 Thread.*sleep*(15000);  
 } **catch** (InterruptedException e) {  
 *// handle here exception* }  
 System.***out***.println(**"Main Thread ending"**) ;  
 }  
  
}  
  
**class** WorkerThread **extends** Thread {  
  
 **public** WorkerThread() {  
 *// When false, (i.e. when it's a user thread),  
 // the Worker thread continues to run.  
 // When true, (i.e. when it's a daemon thread),  
 // the Worker thread terminates when the main  
 // thread terminates.* setDaemon(**true**);  
 }  
  
 **public void** run() {  
 **int** count = 0;  
  
 **while** (**true**) {  
 System.***out***.println(**"Hello from Worker "**+count++);  
  
 **try** {  
 *sleep*(5000);  
 } **catch** (InterruptedException e) {  
 *// handle exception here* }  
 }  
 }  
}

O/P:

Hello from Worker 0

Hello from Worker 1

parent Thread ending

Hello from Worker 2

Main Thread ending

**“class** ForkJoinPool **extends** AbstractExecutorService”

ForkJoinPool is a special type of ExecutorService.

It creates threads as per available processors.

Each thread(worker) in this pool has a workqueue (type of deque), which is mainly used for work-stealing.

If one thread has lot many tasks in its workqueue and other thread in the pool is idle, then it can steal the task from busy thread’s workqueue and start executing it. This is to make sure that all processors are properly utilized.

**@Contented**

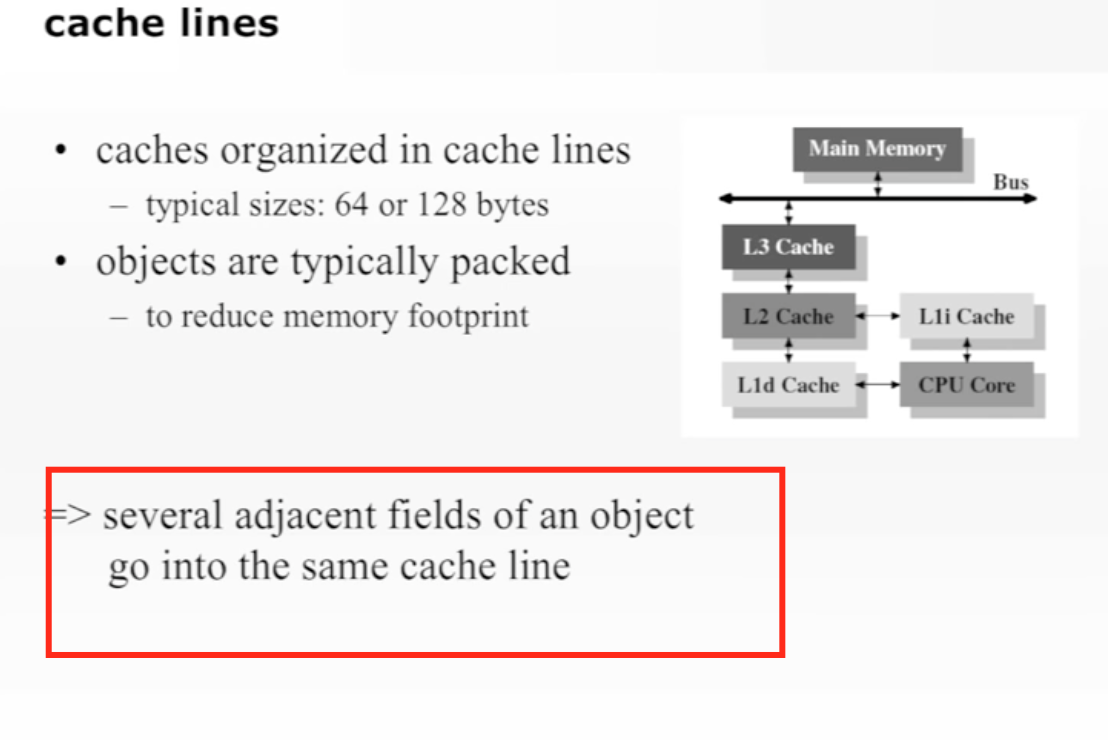
In every processor (CPU), there are cache line where the objects to be used are copied from main memory. This is true for the thread also. Thread has its own caches also.

When any object’s filed value is changed,

If it is a cold field(non-volatile), then jvm takes its own time to propagate that change to main memory and all other threads caches. This sometimes end up in a situation where other threads might be using old object for some time. To resolve this, we make the field volatile. But there is a problem with volatile field.

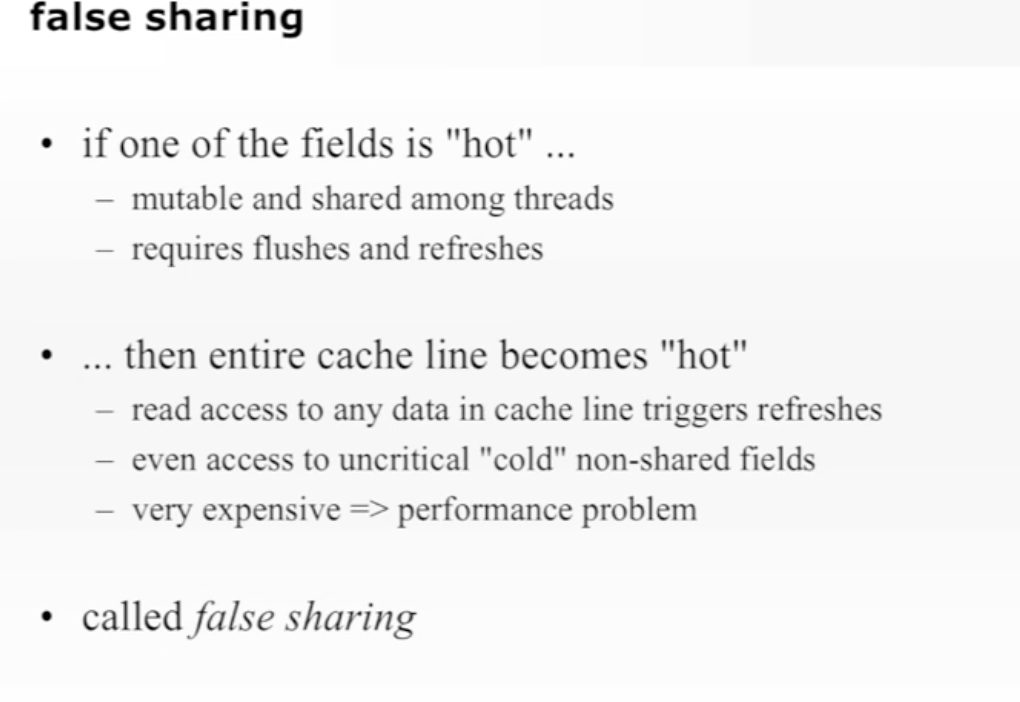
If it is a volatile (hot) field, each time you change the field, change happens in main memory and read also happens from main memory, so that change in value is available to all threads right away.

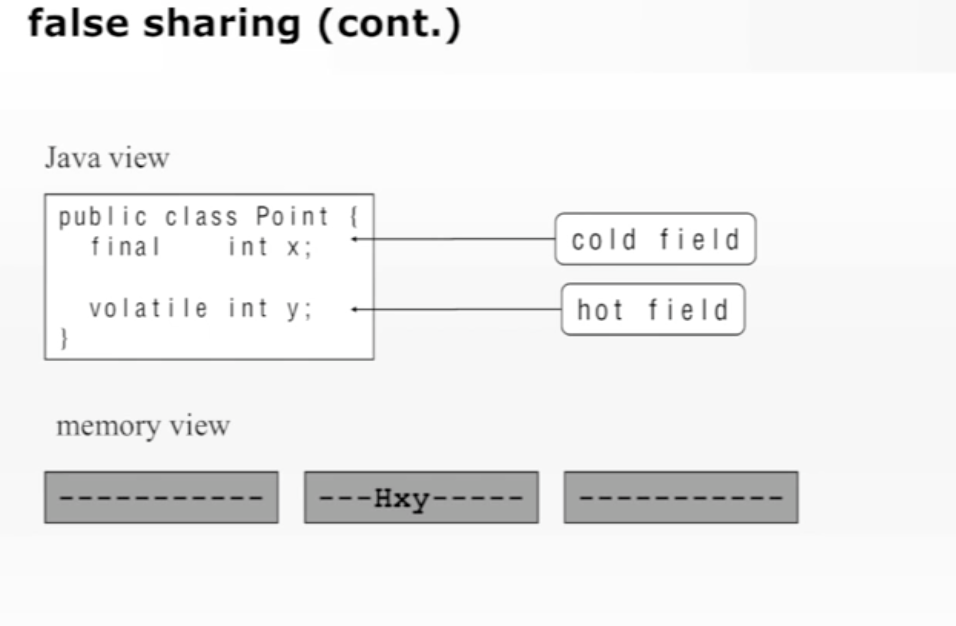
But it has a major problem.



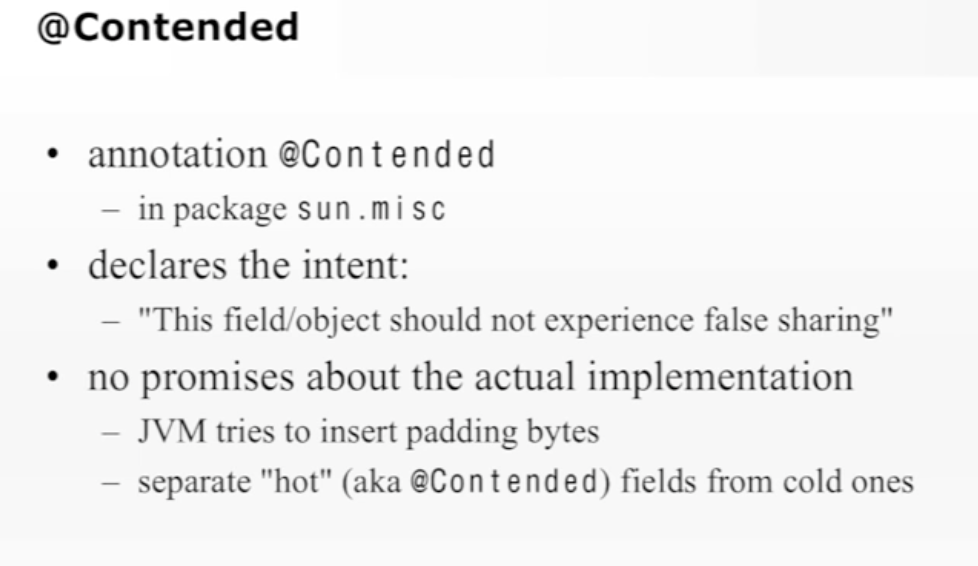


To save memory, compiler puts all the fields of the object densely packed together. If one of the field is volatile (hot), then instead of just flushing that field’s value alone in main memory, it flushes the entire cache line. There can be many objects in one cache line. This may create performance problems.





@Contended tries to solve this problem of flushing the entire cache line to main memory. If you mark hot field with @Contented, then all contended hot fields are kept in a separate cache line.



It can be used on class, field etc. @Contended has a very very low level usage. You may not need to use it.

ForkJoinPool uses @Contented.

