# Concurrent Java

# Key Points Notes

Simon Roberts, October 2016

# Introduction To Java Threads

- Threads provide for separation of concerns, and (very important today) effective use of concurrent hardware.
- Computation requires program, data and CPU. The `run` method of a `Runnable` provides the entry point to code. Methods it calls all provide code. The data that the `run` method, and the methods it calls, have access to provide the data. The `Thread` class provides a “virtual CPU”
  - Data in method local variables are uniquely visible to one thread. Data on the heap can be shared between threads.
- Start a thread with `aThread.start()`. Threads should be *requested* to stop; avoid stopping them forcibly
  - `Thread.sleep()` suspends execution for a period of milliseconds. Like many other “blocking” behaviors it might throw an `InterruptedException`.
  - Cooperative shutdown is usually arranged by invoking the `interrupt` method on a thread. The thread’s code must be written to poll the `interrupted` method at convenient intervals, and/or catch `InterruptedException`. When interruption is determined, the thread code should shut down cleanly.

# Concurrent Access To Data

## Concurrency Problems

- Data written by one thread isn’t automatically visible in other threads, nor is it necessarily visible in the order we expect.
- Compound operations in one thread, such as read/modify/write may be visible in other threads while still only partially completed. These considerations are closely similar to those that give rise to the need for transactions in databases
- Controlling the timing of inter-thread operations. A thread often needs to wait until another thread has prepared data for it, or otherwise has completed necessary operations, prior to proceeding.

## The Java Memory Model

- The Java memory model specifies the rules by which data written by one thread are visible to another. The specification

describes this in terms of *happens-before* relationships.
    - If action A *happens-before* action B, and action B examines effects of action A, then such effects will be visible to action B.
- Eight *happens-before* relationships are defined (some directly, others by extension of other rules). For reference, these are:
    - "Program order". Statements listed as a; b; c; in that order, when executed by a single thread will behave such that a *happens-before* b, and b *happens-before* c. Note this does not preclude compiler reordering for efficiency, provided the observed effects are seen as expected.
    - "Transitive". If a *happens-before* b, and b *happens-before* c, then a *happens-before* c.
    - "Thread start". The call to `aThread.start()` *happens-before* any action of the thread that is started.
    - "Thread end". When any thread that determines a thread has completed (e.g. using `aThread.join()`) then the last action of the thread *happens-before* the action that determines the completion.
    - "Interruption". Calling `aThread.interrupt()` *happens-before* any action by the thread that determines the thread is interrupted.
    - "Finalization". The last action of a constructor of an object *happens-before* the start of the finalizer for that object.
    - "Monitor". The release of a monitor lock *happens-before* any subsequent acquisition of the lock on that same object.
    - "Volatile". A write to a `volatile` variable *happens-before* any subsequent read of that same variable.
- The keyword `synchronized` can be used with an object reference to create a mutual exclusion. The structure looks like this:

```
synchronized(this) {
  // thread holds the lock
} // thread releases lock on exit
```

- No thread can enter a synchronized region unless it holds the "monitor lock" associated with the object referred to in the argument to the `synchronized` keyword. More generally, no execution will occur in that block unless the thread holds the monitor lock. The synchronized block mechanism is reentrant, and is robust in the face of exceptions.
- The Java Memory Model prohibits word-tearing (though using a `BitSet` object in a concurrent fashion can suffer from this.)

### Thread Synchronization

- Timing between threads can be achieved using the `wait`, `notify`, and `notifyAll` methods. These can only be called while the thread holds the monitor lock for the object on which they are invoked.
  - Calling `wait` causes the thread to become “blocked waiting for notification” (a.k.a. “waiting”)
  - Calling `notify` will transition any one thread that is currently waiting to become “blocked waiting for monitor lock”. If no threads are waiting, the notify is lost. The order of wakeups among multiple waiting threads is unspecified.
  - Calling `notifyAll` causes all threads currently waiting to move to the “blocked waiting for monitor lock” state. This can be very unscalable, as many threads often wake for no useful reason.

## Library Lock Mechanisms

- The package `java.util.concurrent.locks` provides library classes that perform locking/mutual-exclusion behaviors. The simplest is the `ReentrantLock`.
  - ReentrantLock allows multiple condition variables. Create a condition variable using the call `aReentrantLock.newCondition()`. Use multiple conditions as necessary to ensure that any thread that wakes can fully utilize the reason the wakeup was issued.
  - API locking tools must be used with `try` / `finally` so that the finally reliably releases the lock, even if an unchecked exception should arise.
- Library locks can attempt to gain a lock in an interruptible fashion. By contrast, once a thread attempts to enter a `synchronized` block, the only way it can ever execute again is by succeeding in obtaining the lock.
- The main `java.util.concurrent` package provides a `Semaphore` class, which is harder to use correctly than the explicit locks, but works exactly as standard literature describes.
- The `java.util.concurrent.locks` package also provides `ReentrantReadWriteLock`, which provides for multiple readers or a single writer, and the `StampedLock` which provides for reader writer scenarios, but also allows optimistic locking, and the up and down-grading of locks (e.g. turning a read-lock into a write-lock). `ReentrantReadWriteLock` supports conditions on

write locks only, and the `StampedLock` does not provide support any conditions.

# Simple Models for Thread-Safe Designs

## Immutability

- Most “threadsafety” issues arise when multiple threads are sharing data, and at least one thread is modifying that data. A powerful technique to avoid problems is to ensure that all data are ***immutable***. This doesn’t avoid every problem:
  - To share data—even immutable data--there must be a variable that refers to the object of interest, and that variable must be mutable if it refers to an object created at runtime. The write to this variable must *happen-before* any other thread tries to read it.
  - Timing / synchronization issues might still exist
- Immutability is sufficiently useful that the language specification gives a special status (and a kind of happens-before relationship) for `final` fields in objects. Essentially, the write to a final field *happens-before* the completion of the constructor.

## Safe Publication

- Commonly an object is created in a single thread, and all mutations to the object happen in that thread while the object is unreachable from any other thread. After that, a write may be made that publishes the address of the created object somewhere and makes it available for use by other threads. If the write creates a happens-before relationship between the end of initialization and any subsequent read of the published variable then we say that we have ***safe publication***. (For example, the write might be to a `volatile` variable, or a threadsafe structure in the `java.util.concurrent` libraries)

## Pipeline Architecture

- The pipeline (a.k.a. “producer-consumer” and sometimes "actor") architecture builds on the safe publication idea.
- Objects are created and initialized in one thread (the “producer”) and then placed in a `BlockingQueue` that safely publishes that object to exactly one other thread (the “consumer”). After publication, the producer thread deletes its copy of the reference to the object. In this way, any given time, each object is visible in

only one thread, and hence the current owning thread may mutate it safely.
- Passing the object through the queue creates the necessary happens-before relationships that ensure the next thread to see the object sees the writes performed by the preceding thread.
- The queue provides a means of timing control, in that the object cannot be taken from the queue until after it has been written there.
- The producers and consumers typically run in infinite loops. The buffering effect of the queue creates a degree of control of CPU allocation; if the queue fills, the CPU(s) stop working on production and are available for consumption instead, and vice versa.

### Streams

- The Java 8 Stream API facilitates repetitive operations on large sets of independent data items. Each step of processing should avoid interacting with the source of data and any other "external" mutable data. Concurrency is handled by the stream infrastructure when a parallel stream is requested.

## Overview of Key java.util.concurrent Utilities

### Atomic Classes

- Atomic types provide indivisible read/modify/write operations. For example, if an `AtomicInteger` contains the value 4, and two threads attempt to add the values 5 and 6 respectively, the final value will definitely be 15, and the first thread will either read the value 9 or 15 (depending on if it "gets there first") while the second definitely reads 10 or 15. The Adder and Accumulator classes provide for higher scalability in situations with large numbers of threads updating a value that is only occasionally read.

### Threadsafe Collections

- The package `java.util.concurrent` provides several collection implementations that are threadsafe, provide guarantees of happens-before relationships between writes to them and reads from them, and in the general case provide

better scalability than simply using locking around accesses to a collection.

## Scalability Features

- Scalability describes the ability of software to make use of additional CPUs. This is critical in many modern systems—even more so than raw performance or computational efficiency, since it defines the upper limit on what can be done, even when unlimited money is spent on hardware. Scalability is described by Amdahl's law. Scalability is lost whenever one thread must wait for another, and therefore we must avoid using locks and similar any more than is critical to correctness. Many of the concurrent api classes aim to provide high scalability.
- Some copy-on-write structures are provided. These are intended for scalability in situations where highly concurrent reading occurs, with rare updates. A write starts by duplicating the entire structure to an initially-private copy (which is a read operation) followed by local update, and subsequent replacement of the original structure with the modified one. This allows lock-free, and highly scalable reading. However, writes will take effect in a somewhat delayed fashion as other reads can be concurrently reading "stale" data. Also, the write is computationally demanding.

## Synchronizers

- Several utilities provide specifically for making one or more threads wait until some condition arises. Among these are `CountDownLatch`, `CyclicBarrier`, and `Exchanger`.

## `ThreadLocal` Variables

- The `ThreadLocal` class creates and manages objects on a per-thread basis, such that any code can access an object that is guaranteed to be confined to that thread. This does not provide inter-thread communication.
- The `ThreadLocalRandom` class creates a thread-local random number generator. Because random number generation involves a time-consuming critical section, providing every thread with a generator of its own can be a great benefit to throughput in concurrent code that uses random numbers.

# Thread Pools

## Pool Concept

- Threads are a relatively expensive operating system resource, and shouldn't be created for short-term use then destroyed. Instead use a thread pool.
  - A thread pool is easy to create with a `BlockingQueue<Runnable>` being read by a small number of threads that simply loop forever pulling jobs off the queue and executing their `run` methods.
- Java has thread pool functionality built in, and represented by the `Executor` and `ExecutorService` interfaces.
  - `ExecutorService` is a sub-interface of `Executor` which is more commonly used as it provides additional functionality.
  - `ExecutorService` instances can be created by calling fairly complex constructors, or by using the `Executors` class which defines several useful static factory methods.
  - A `Runnable` can be sent to an `Executor` for execution via the `execute` method.
- The `ExecutorService` provides additional features:
  - The ability to get a returned value, or a checked exception, back from the task. For this, the job is specified using a `Callable<E>`, rather than a `Runnable`. The returned type of the job is the generic type variable `E`.
  - A `Callable` is submitted for execution using the `submit` method of the `ExecutorService`. This method returns a "handle" on the job embodied in an object that implements the interface `Future<E>`
  - Calling the `Future` object's `get` method allows us to retrieve the output value of the job, or get the exception that occurred. If the job has not completed, this call blocks. We can poll to see if the job is finished by calling the `isDone` method on the `Future`.
  - The `Future` also allows us to request cooperative shutdown, of the job. Calling the `cancel` method of the `Future` will remove a not-yet-started job from the queue, and can send an interrupt to a running job so as to request cooperative shutdown.

## Results And Job Control Without Pools

- The `FutureTask` class implements the `Future` interface and can be wrapped around either a `Callable` or `Runnable`. When the `FutureTask` object is executed (either directly by a thread, or by submitting it to an executor, the `FutureTask` provides job control and result-access behaviors using the `Future` aspects that it implements.

## A Pool For Large Computations

- The `ForkJoin` system is a framework for execution of large, computationally intensive jobs in a concurrent environment. The jobs should generally not perform blocking operations such as IO, as this “spoils” the scheduling efficiency.
- The jobs are implemented in the `compute` method of either the `RecursiveTask` or `RecursiveAction` sub-classes.
- A jobs should split itself into a number of smaller sub-jobs (the splitting process is sometimes called “sharding”). All the sub-jobs are executed by the fork-join pool, and the intermediate results are recombined by the code written for the purpose in the jobs.
- The job should submit sharded sub-jobs using the `fork` method, and then get the results back using the `join` method. While waiting for the sub-job to finish, this job should recursively call its own `compute` method which might further subdivide the job.
- In pseudo-code a RecursiveTask<Double> might implement the compute method like this:

```
protected Double compute() {
  if (size of job is small enough) {
    return simple linear computation
  } else {
    sub = new job from one half of data
    other = new job from other half of data
    // run this sub-job in other threads
    other.fork();
    // run the rest of this job in this thread
    partialResult1 = sub.compute();
    // get the result from the other threads
    partialResult2 other.join()
    // combine the results and return
    return partialResult1 + partialResult2
  }
}
```

# Introduction to `CompletableFuture`

## Background

- Although the pure threaded model creates readable code by providing for separation of concerns, it can be limiting in terms of scalability and does not intrinsically provide safe data communication between threads. Many threads can give rise to too many context switches, stale caches, and other problems. A popular model for limiting the number of threads is the “callback” model, but this can lead to code that’s deeply nested and hard to maintain.
- The “promise pipeline” model proposed by Barbara Liskov (sometimes simply called “promises”) facilitates separation of concerns, readable and maintainable source code, safe data transfer between jobs, and minimal thread usage. This model is embodied in the `CompletableFuture` API in Java 8.

## Data Flow In `CompletableFuture`

- A completable future system is constructed by building a pipeline that carries data (or possibly an exception) from one processing element to the next.
- Each step acts as either a source of data, a sink of data, or performs some kind of transformation. As might be expected, these processing stages are represented using `Supplier<E>`, `Consumer<E>`, and `Function<E,F>`.
- Methods in the completable future API serve to *build* the pipeline, which might start execution immediately. So for example a pipeline can be started by using:
  `CompletableFuture.supplyAsync(()->”Initial Data”)`
- Processing might continue with:
  `.thenApplyAsync(x -> “more data” + x)`
- Exceptions follow parallel paths and will skip over normal handling elements. Exceptions can be handled using special methods, including `exceptionally` and `handle`/`handleAsync`.
- The method `exceptionally` processes an exception with a `Function<? extends Throwable, T>`. The method is called only if an exception arises, and the flow processing continues down the “normal” path using the value returned by the function.

- The `handle` methods are called for both success and exception situations. The two arguments represent the success data and the exception, so in any given invocation, one of them will be `null`.

## Asynchrony With `CompletableFuture`

- Crucial to the usefulness of the completable future API is its ability to integrate with external behaviors that are intrinsically long-running, such as file access and network operations. In such situations, a `CompletableFuture` object is used as a rendezvous object providing both timing coordination and data transfer.
- In these situations, the handler method should perform these tasks:
  - Create a `CompletableFuture` object to serve as the rendezvous object.
  - Initialize and trigger the background operation (which is processed in another thread, quite possibly an OS thread)
  - Arrange that if the background operation completes normally, the `complete` method of the rendezvous object is invoked with the result data as argument.
  - Arrange that if the background operation fails, the `completeExceptionally` method of the rendezvous object is called, with a suitable exception as the argument.
  - Immediately after performing these configurations, return the rendezvous object to the caller.
- To invoke the asynchronous operation, the method `thenCompose` is used.
- For example, given this processing chain:

```
CompletableFuture.supplyAsync(()->"filename.txt")
  .thenCompose(x -> getFileContents(x))
  .exceptionally(t -> "Error: " + t.getMessage())
  .thenAccept(System.out::println);
```

then the `getFileContents` method may be implemented using the following pseudo-code skeleton:

```
public static CompletableFuture<String>
  getFileContents(String fname)
{
  CompletableFuture<String> cfs =
    new CompletableFuture<>();

  prepare async file read operation
  when read completes successfully {
    cfs.complete(dataReadFromFile);
  } or when read completes exceptionally {
    cfs.completeExceptionally(exceptionFromRead);
  }
  return cfs;
}
```

- Note that the `AsynchronousFileChannel` class, and related socket and server socket classes provide for asynchronous, callback based, IO operations that would be well suited to completing this skeleton. These classes are in the package `java.nio.channels`. Future releases of Java will probably include IO APIs built around the `CompletableFuture` API.