The Java Fork-Join Pool: Structure & Functionality (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

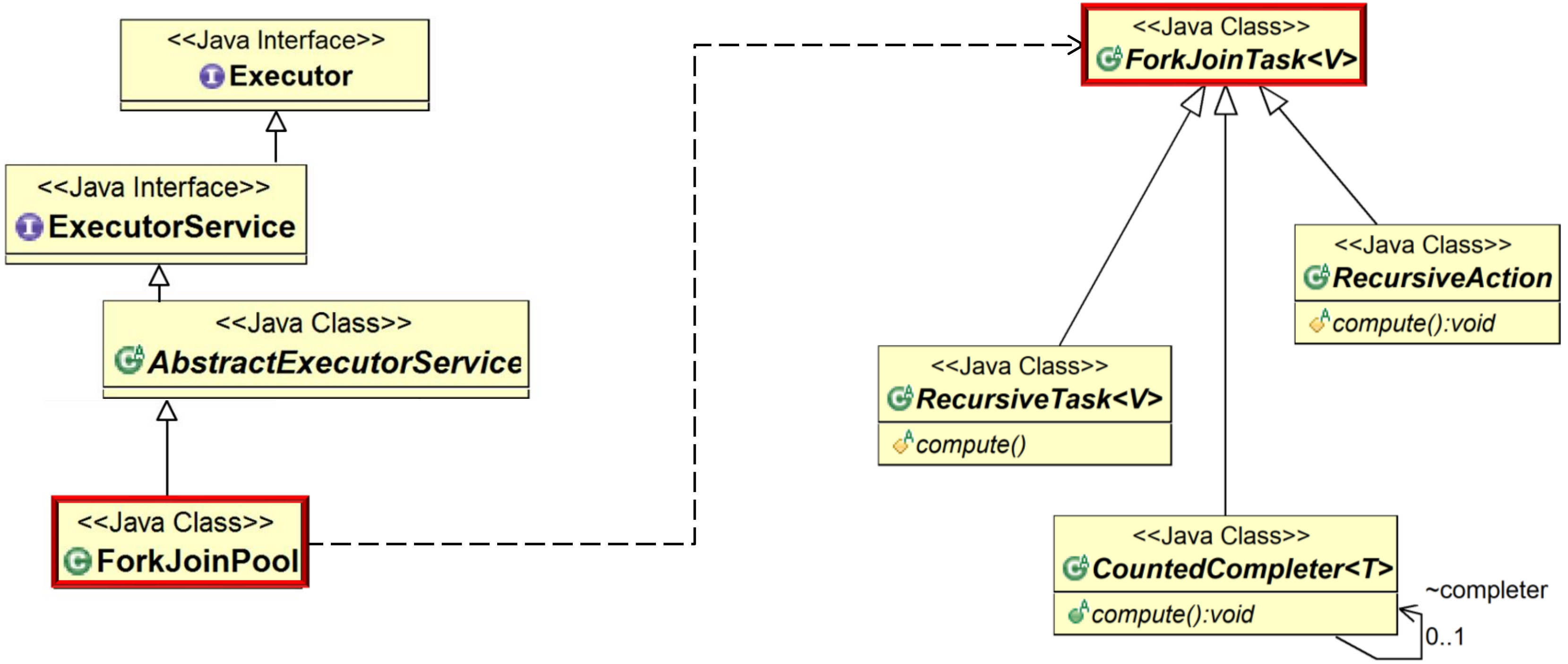
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

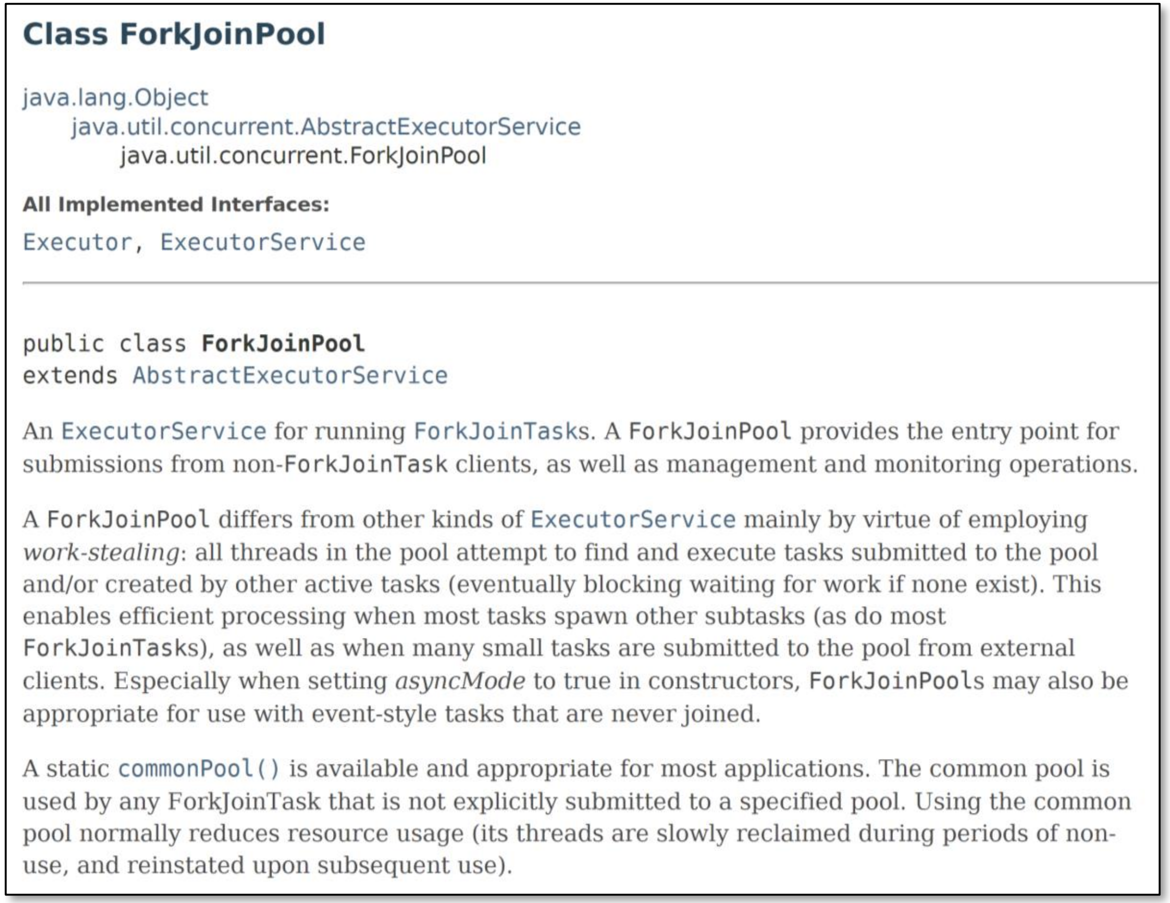
- Understand how the Java fork-join framework processes tasks in parallel
- Recognize the structure & functionality of the fork-join framework



The Structure & Functionality of the Fork-Join Framework

The Structure & Functionality of the Fork-Join Framework

- ForkJoinPool implements the ExecutorService interface



Class ForkJoinPool

java.lang.Object
    java.util.concurrent.AbstractExecutorService
        java.util.concurrent.ForkJoinPool

All Implemented Interfaces:

Executor, ExecutorService

```
public class ForkJoinPool
extends AbstractExecutorService
```

An ExecutorService for running ForkJoinTasks. A ForkJoinPool provides the entry point for submissions from non-ForkJoinTask clients, as well as management and monitoring operations.

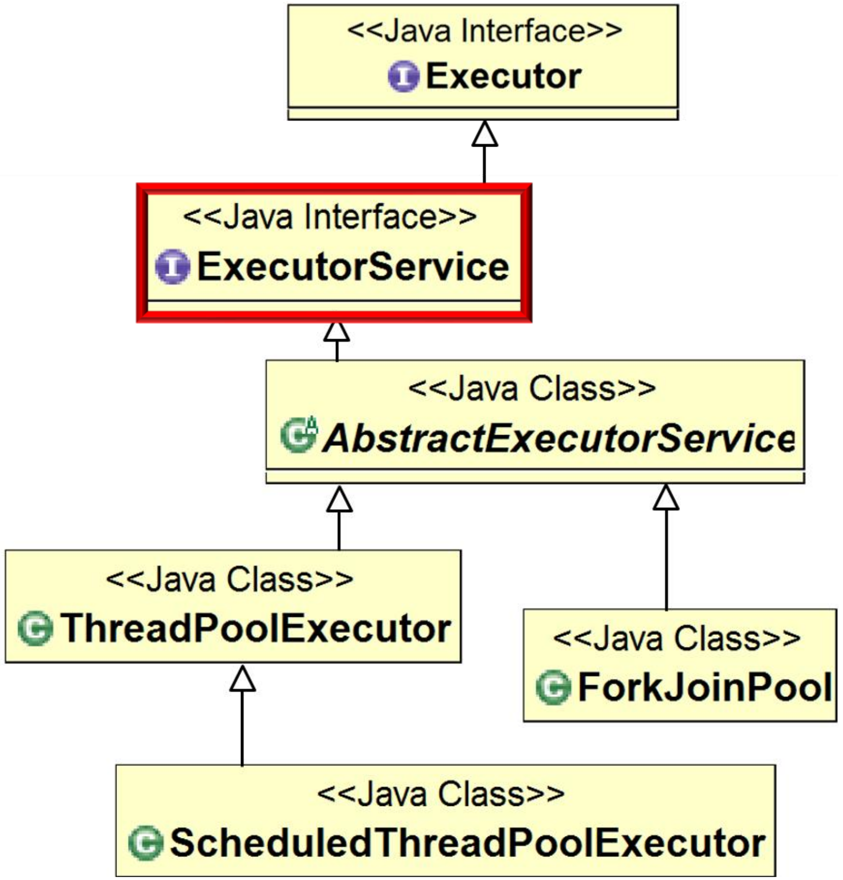
A ForkJoinPool differs from other kinds of ExecutorService mainly by virtue of employing *work-stealing*: all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks (as do most ForkJoinTasks), as well as when many small tasks are submitted to the pool from external clients. Especially when setting *asyncMode* to true in constructors, ForkJoinPools may also be appropriate for use with event-style tasks that are never joined.

A static commonPool() is available and appropriate for most applications. The common pool is used by any ForkJoinTask that is not explicitly submitted to a specified pool. Using the common pool normally reduces resource usage (its threads are slowly reclaimed during periods of non-use, and reinstated upon subsequent use).

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

The Structure & Functionality of the Fork-Join Framework

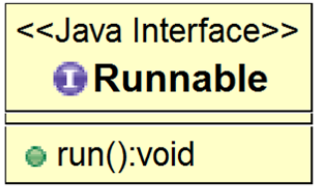
- ForkJoinPool implements the ExecutorService interface
  - This interface is the basis for Java Executor framework subclasses

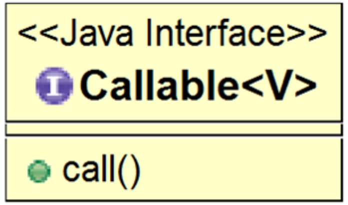


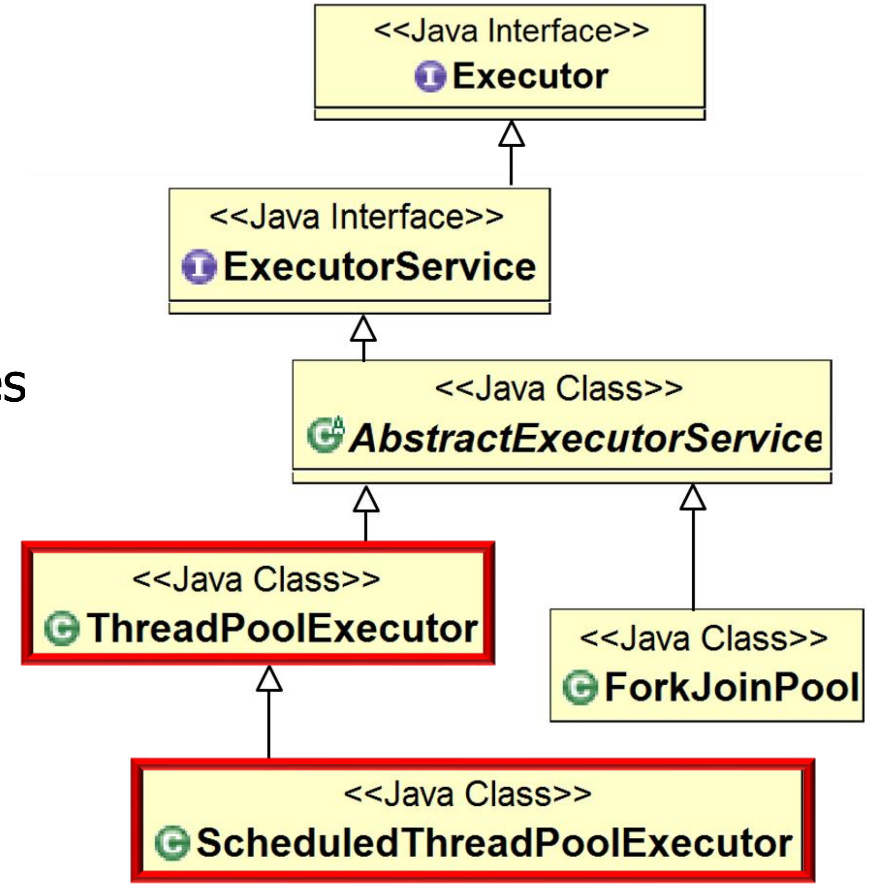
See docs.oracle.com/javase/tutorial/essential/concurrency/executors.html

The Structure & Functionality of the Fork-Join Framework

- ForkJoinPool implements the ExecutorService interface
  - This interface is the basis for Java Executor framework subclasses
  - Other implementations of Executor Service execute runnables or callables

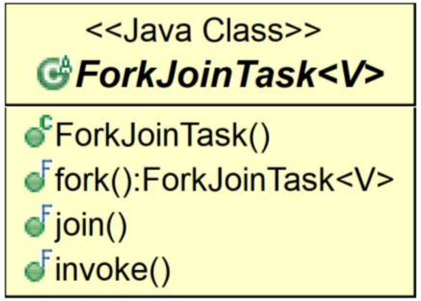


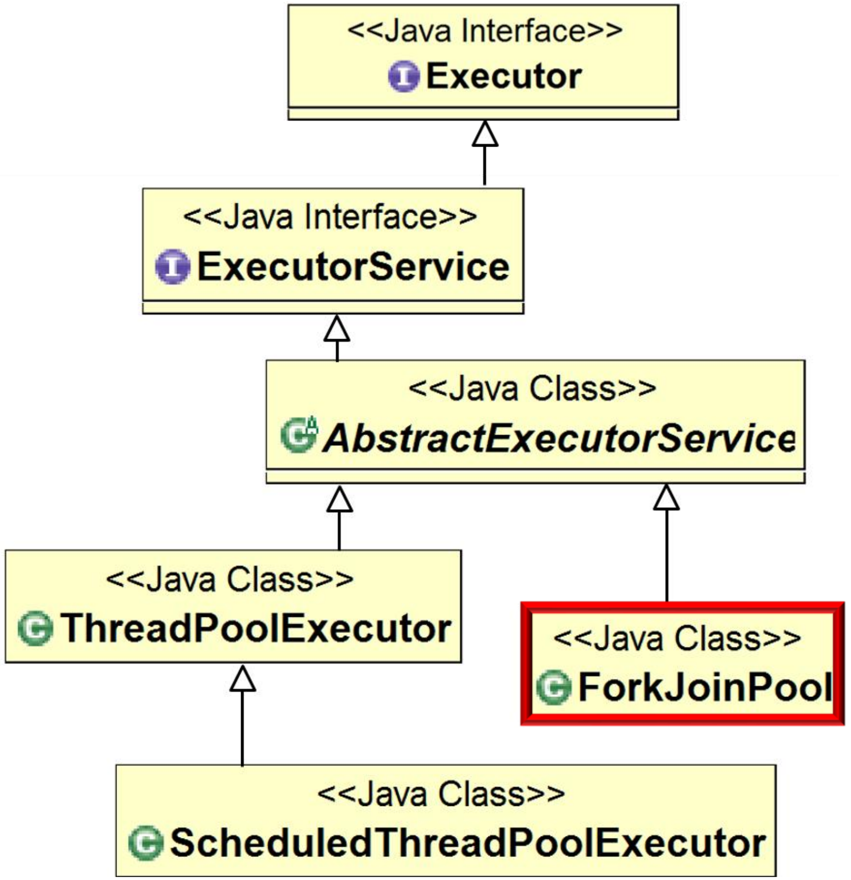




The Structure & Functionality of the Fork-Join Framework

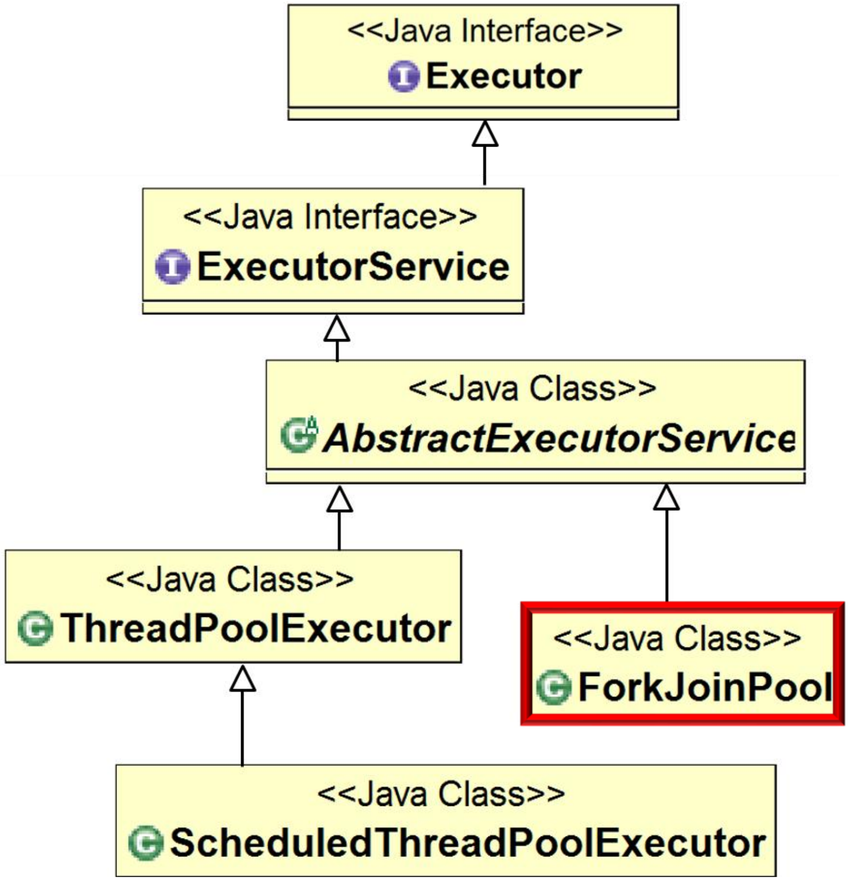
- ForkJoinPool implements the ExecutorService interface
  - This interface is the basis for Java Executor framework subclasses
  - Other implementations of Executor Service execute runnables or callables
  - In contrast, the ForkJoinPool executes ForkJoinTasks





The Structure & Functionality of the Fork-Join Framework

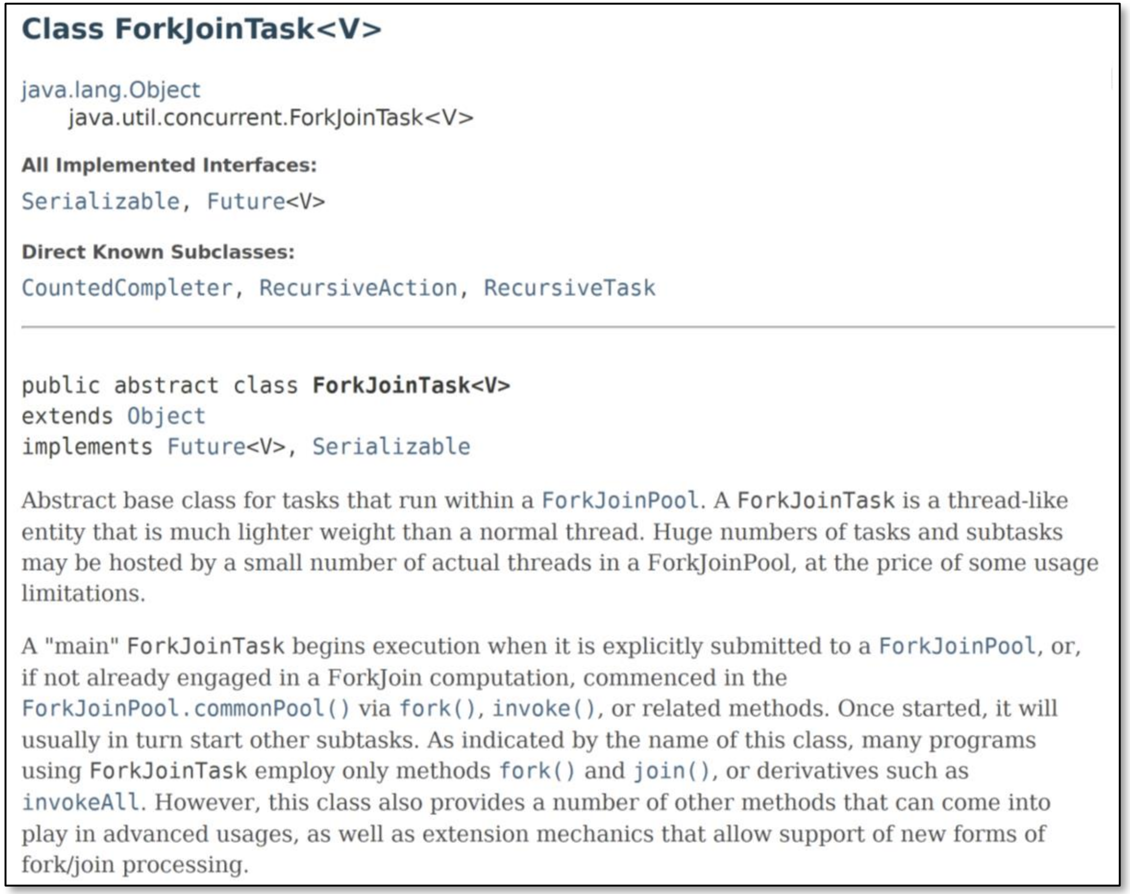
- ForkJoinPool implements the ExecutorService interface
  - This interface is the basis for Java Executor framework subclasses
  - Other implementations of Executor Service execute runnables or callables
  - In contrast, the ForkJoinPool executes ForkJoinTasks



It can also execute runnables & callables, but that's not its main purpose

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask associates a chunk of data along with a computation on that data



Class ForkJoinTask<V>

java.lang.Object
    java.util.concurrent.ForkJoinTask<V>

All Implemented Interfaces:

`Serializable`, `Future<V>`

Direct Known Subclasses:

`CountedCompleter`, `RecursiveAction`, `RecursiveTask`

```
public abstract class ForkJoinTask<V>
extends Object
implements Future<V>, Serializable
```

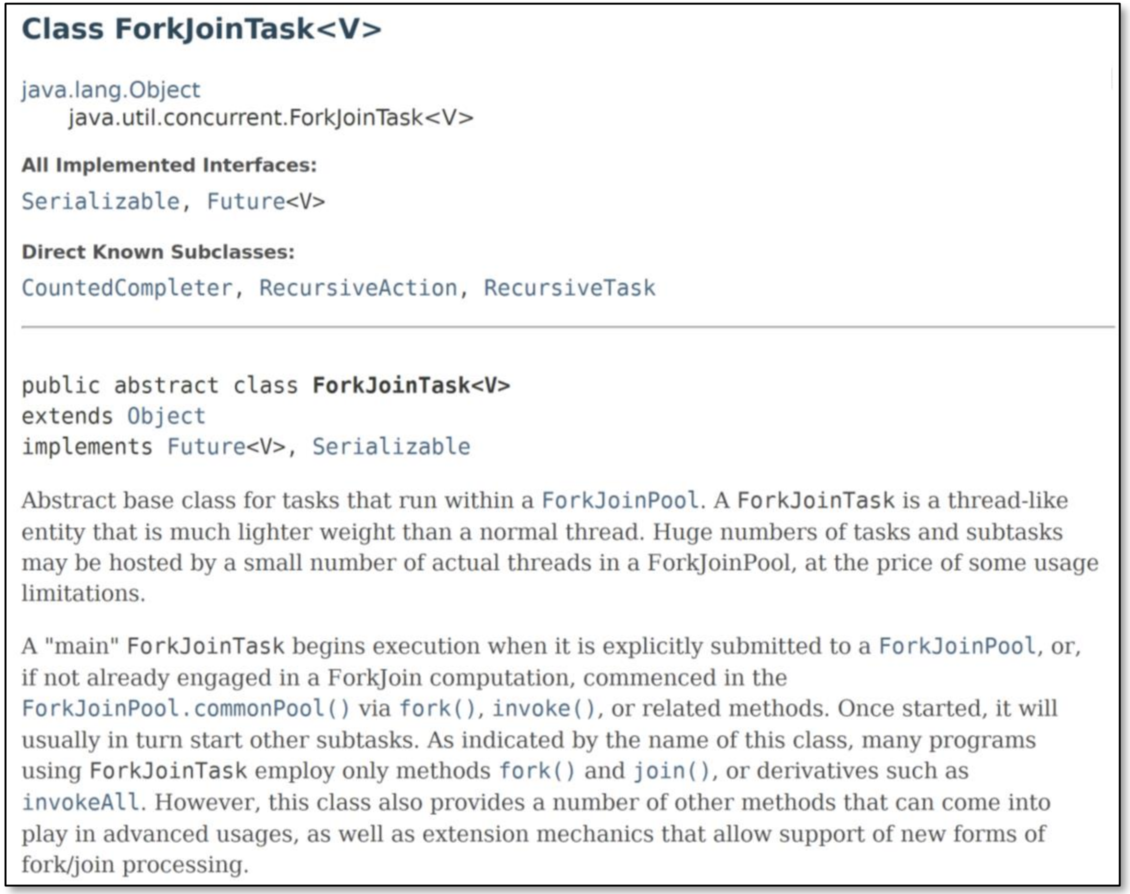
Abstract base class for tasks that run within a `ForkJoinPool`. A `ForkJoinTask` is a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a ForkJoinPool, at the price of some usage limitations.

A "main" `ForkJoinTask` begins execution when it is explicitly submitted to a `ForkJoinPool`, or, if not already engaged in a ForkJoin computation, commenced in the `ForkJoinPool.commonPool()` via `fork()`, `invoke()`, or related methods. Once started, it will usually in turn start other subtasks. As indicated by the name of this class, many programs using `ForkJoinTask` employ only methods `fork()` and `join()`, or derivatives such as `invokeAll`. However, this class also provides a number of other methods that can come into play in advanced usages, as well as extension mechanics that allow support of new forms of fork/join processing.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinTask.html

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask associates a chunk of data along with a computation on that data
  - This enables fine-grained data parallelism



Class ForkJoinTask<V>

java.lang.Object
    java.util.concurrent.ForkJoinTask<V>

All Implemented Interfaces:

`Serializable`, `Future<V>`

Direct Known Subclasses:

`CountedCompleter`, `RecursiveAction`, `RecursiveTask`

```
public abstract class ForkJoinTask<V>
extends Object
implements Future<V>, Serializable
```

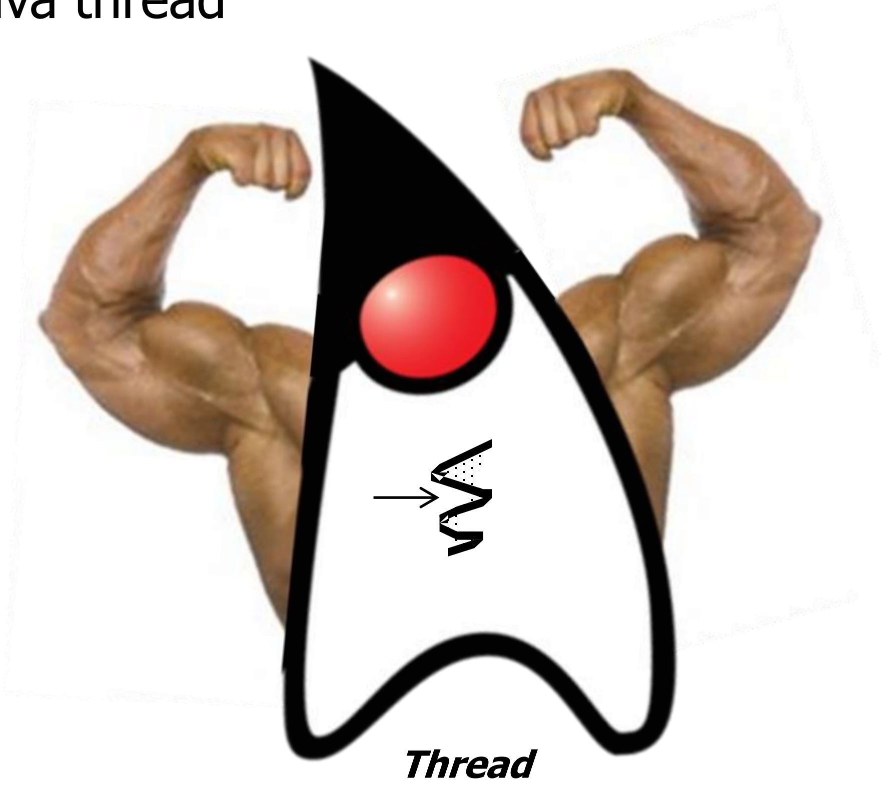
Abstract base class for tasks that run within a `ForkJoinPool`. A `ForkJoinTask` is a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a ForkJoinPool, at the price of some usage limitations.

A "main" `ForkJoinTask` begins execution when it is explicitly submitted to a `ForkJoinPool`, or, if not already engaged in a ForkJoin computation, commenced in the `ForkJoinPool.commonPool()` via `fork()`, `invoke()`, or related methods. Once started, it will usually in turn start other subtasks. As indicated by the name of this class, many programs using `ForkJoinTask` employ only methods `fork()` and `join()`, or derivatives such as `invokeAll`. However, this class also provides a number of other methods that can come into play in advanced usages, as well as extension mechanics that allow support of new forms of fork/join processing.

See www.dre.Vanderbilt.edu/~schmidt/PDF/DataParallelismInJava.pdf

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask is lighter weight than a Java thread



e.g., it doesn't maintain its own run-time stack, registers, thread-local storage, etc.

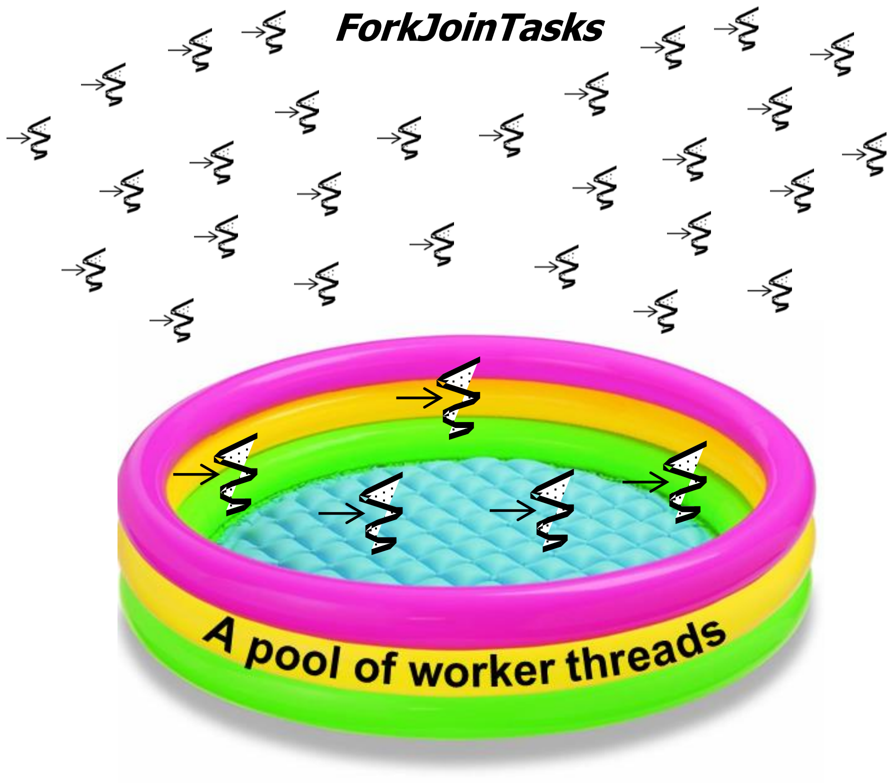
The Structure & Functionality of the Fork-Join Framework

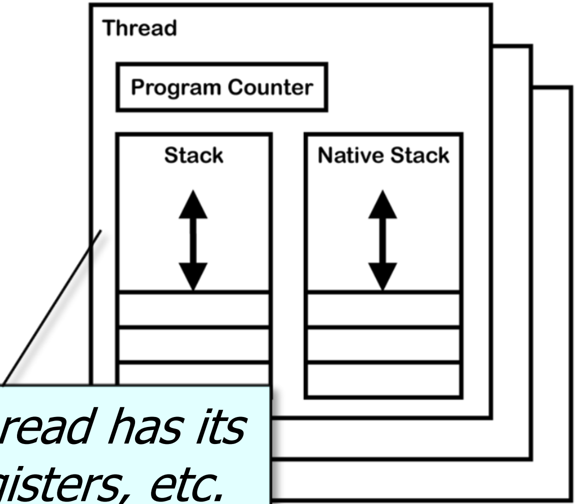
- A ForkJoinTask is lighter weight than a Java thread
  - A large # of ForkJoinTasks can thus run in a small # of worker threads in a fork-join pool

ForkJoinTasks
A pool of worker threads

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask is lighter weight than a Java thread
  - A large # of ForkJoinTasks can thus run in a small # of worker threads in a fork-join pool





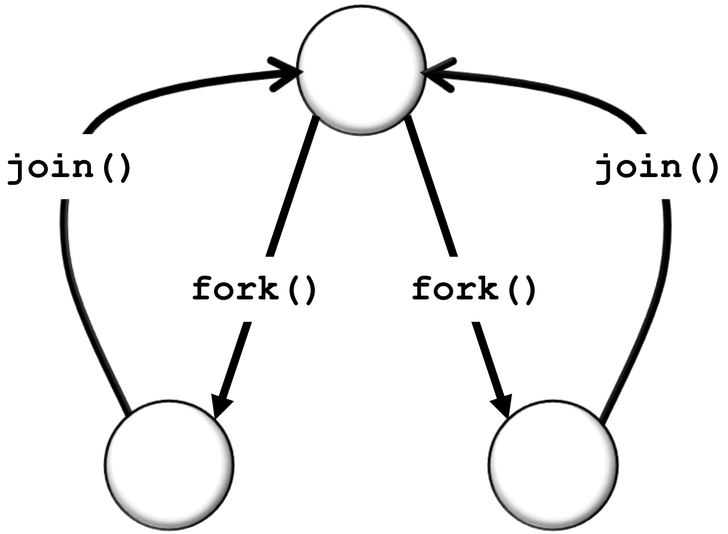
Each worker thread has its own stack, registers, etc.

See blog.jamesdbloom.com/JVMInternals.html

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask has two methods that control parallel processing/merging

Parent ForkJoinTask



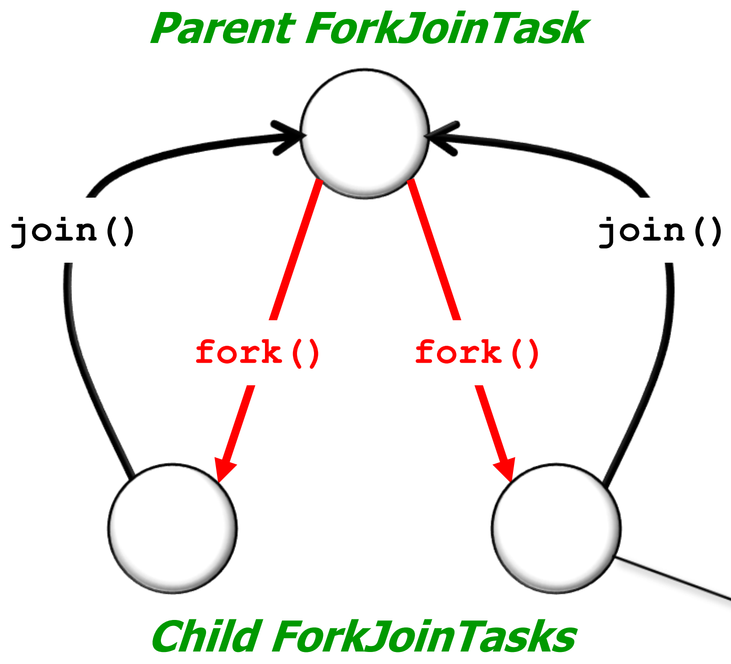
Child ForkJoinTasks

ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns result of computation when it is done

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask has two methods that control parallel processing/merging

ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns result of computation when it is done

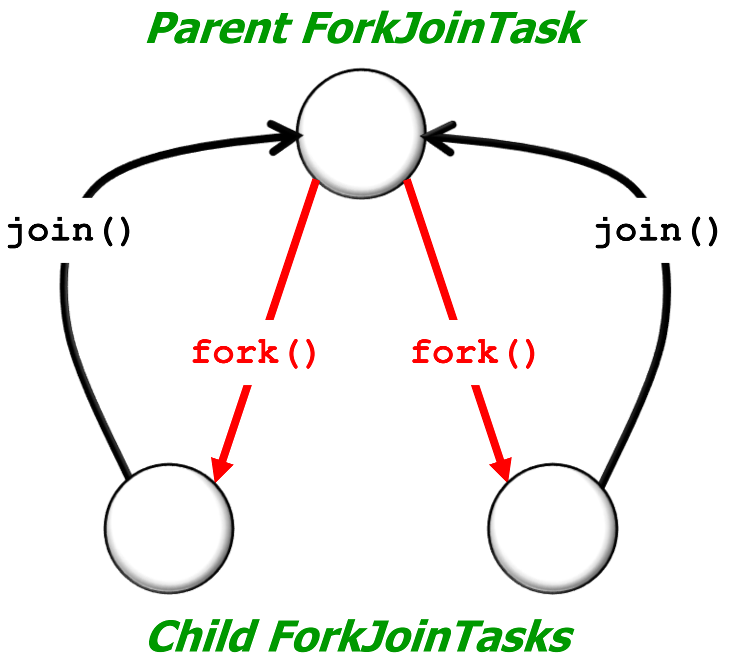


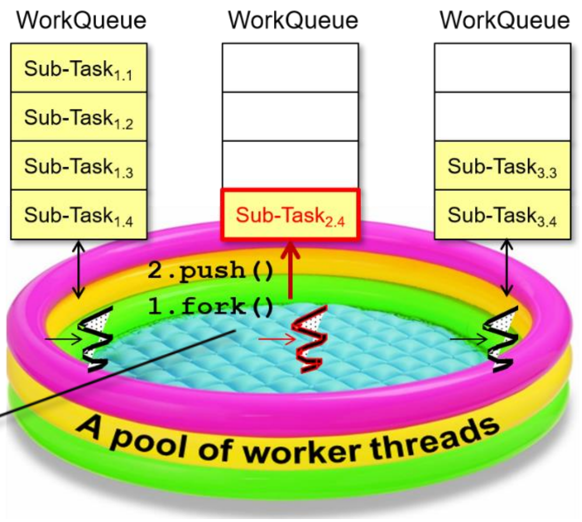
fork() is a lightweight variant of Thread.start() that creates a child ForkJoinTask

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask has two methods that control parallel processing/merging

ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns result of computation when it is done



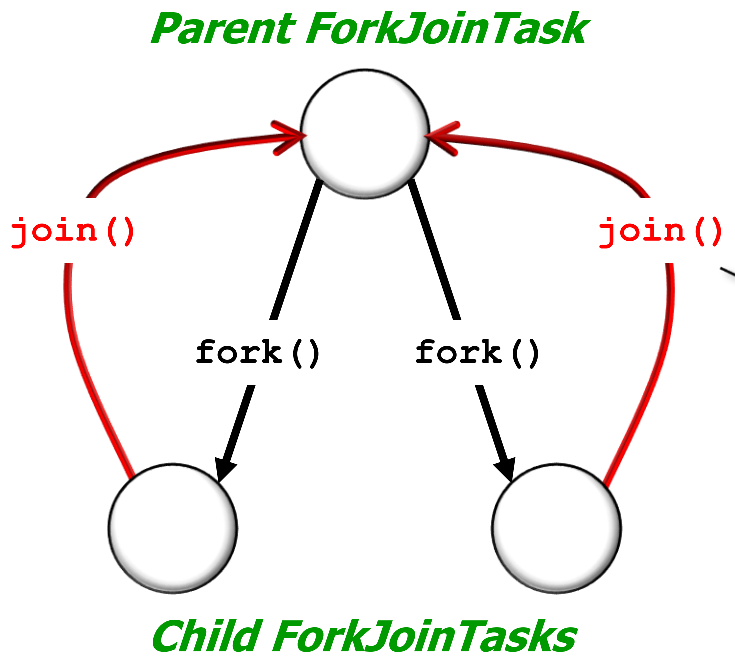


fork() doesn't run the task, but places it on a work queue in the calling worker thread

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask has two methods that control parallel processing/merging

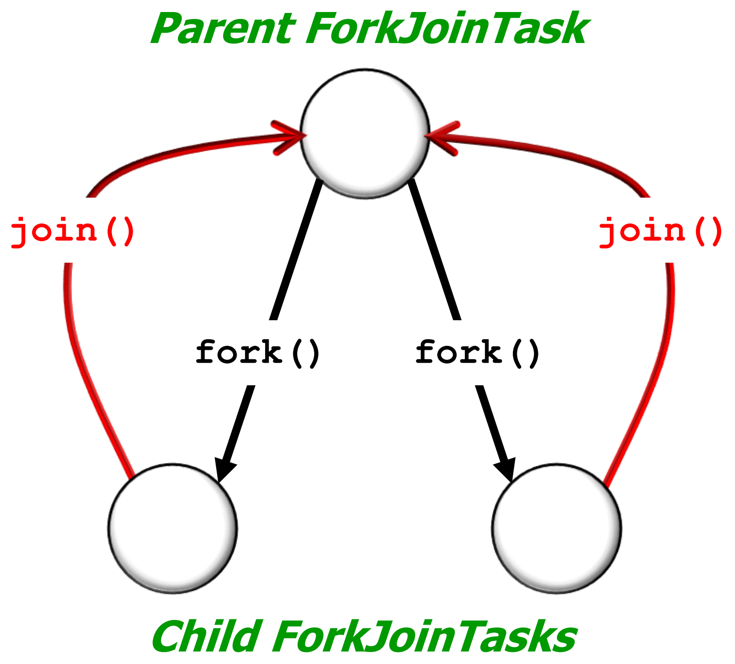
ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns result of computation when it is done



join() returns the result of a child task to the parent task that forked it

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask has two methods that control parallel processing/merging

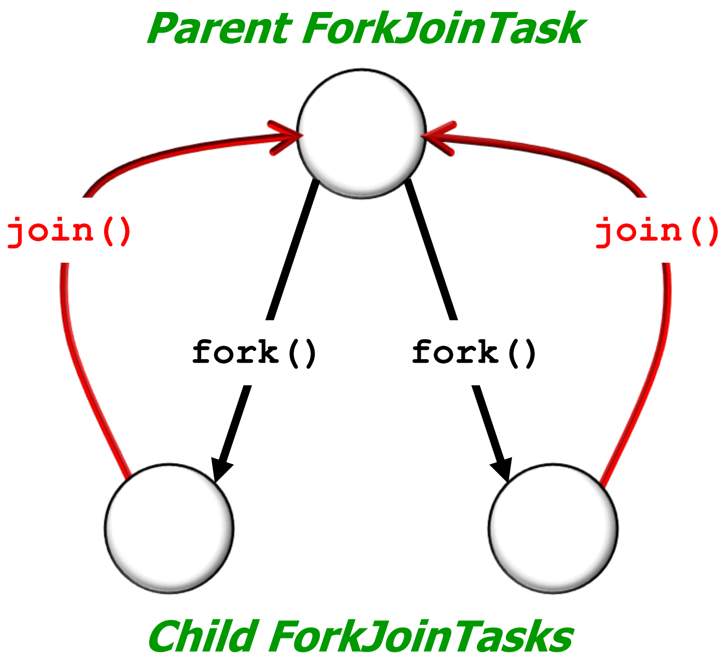


ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns result of computation when it is done

- Unlike Thread.join(), ForkJoinTask.join() doesn't simply block the calling thread

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask has two methods that control parallel processing/merging



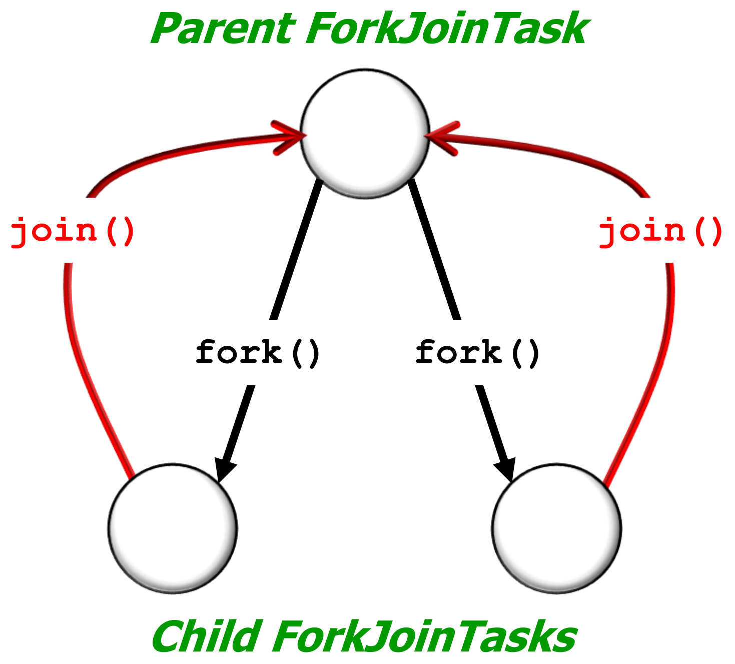
ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns result of computation when it is done

- Unlike Thread.join(), ForkJoinTask.join() doesn't simply block the calling thread
  - It uses a worker thread to run tasks

"Pitch in" via the "Collaborative Jiffy Lube" model of processing!

The Structure & Functionality of the Fork-Join Framework

- A ForkJoinTask has two methods that control parallel processing/merging



ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns result of computation when it is done

- Unlike Thread.join(), ForkJoinTask.join() doesn't simply block the calling thread
  - It uses a worker thread to run tasks
  - When a worker thread encounters a join() it processes other tasks until it notices the target sub-task is done

End of the Java Fork-Join Pool: Structure & Functionality (Part 1)