The Java Fork-Join Pool Framework (Part 1)

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

Professor of Computer Science

Institute for Software
Integrated Systems

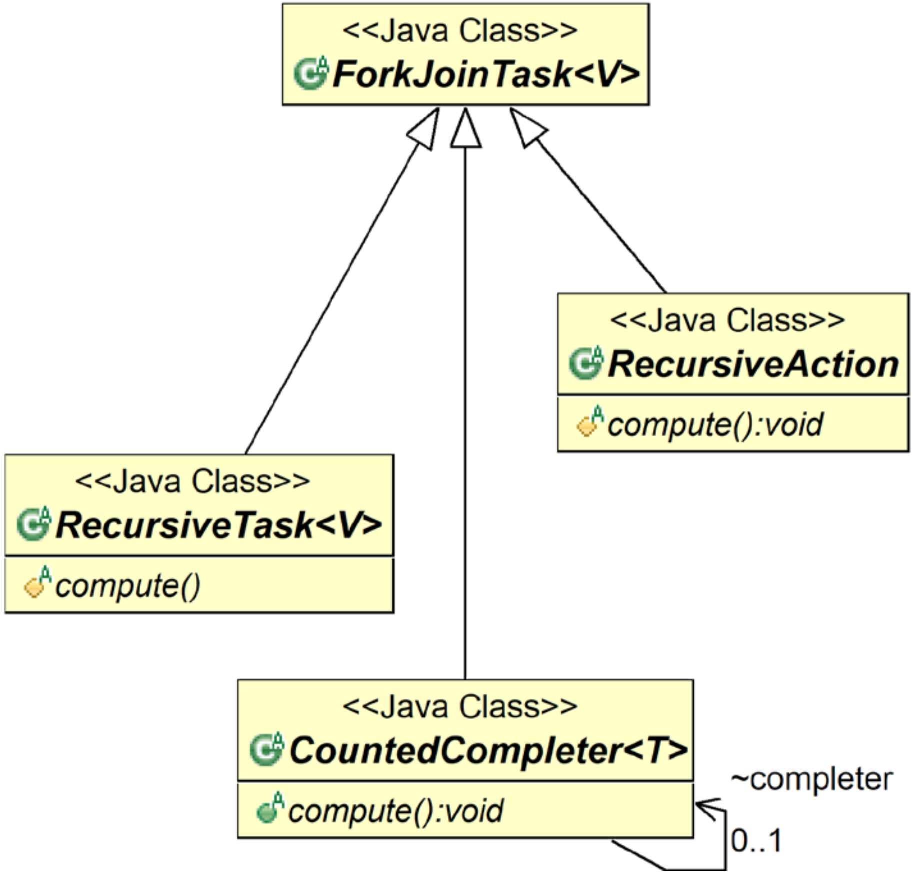
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand how the Java fork-join framework processes tasks in parallel

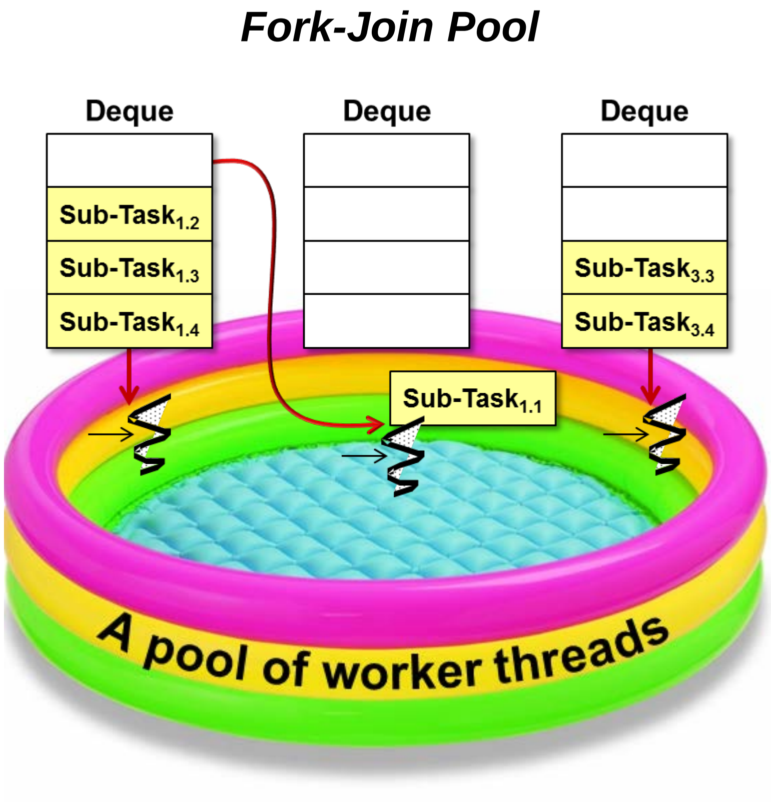
Learning Objectives in this Part of the Lesson

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



Learning Objectives in this Part of the Lesson

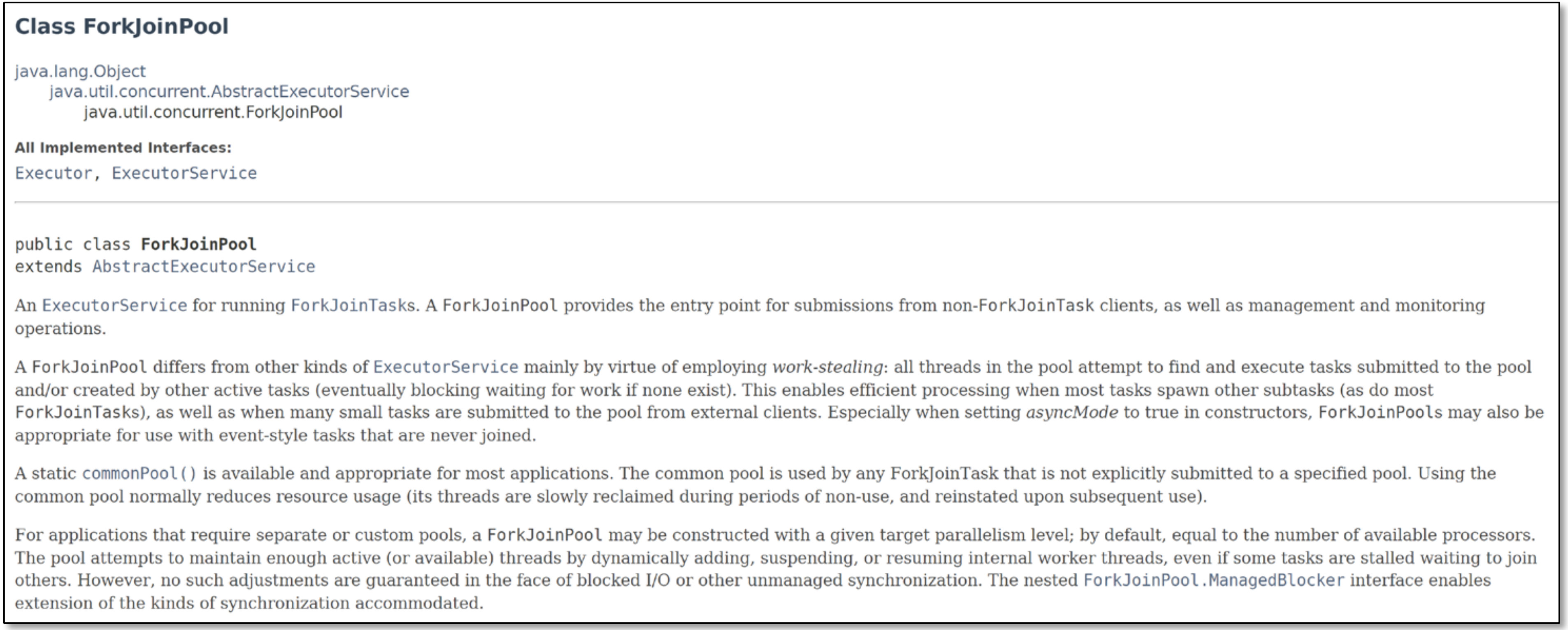
- Understand how the Java fork-join framework processes tasks in parallel
- Recognize the structure & functionality of the fork-join framework
- Know how the fork-join framework is implemented internally



Overview of the Java Fork-Join Pool Computation Model

Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool provides a high performance, fine-grained task execution framework for Java data parallelism



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 `ForkJoinTask`s. 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 `ForkJoinTask`s), as well as when many small tasks are submitted to the pool from external clients. Especially when setting *asyncMode* to true in constructors, `ForkJoinPool`s 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).

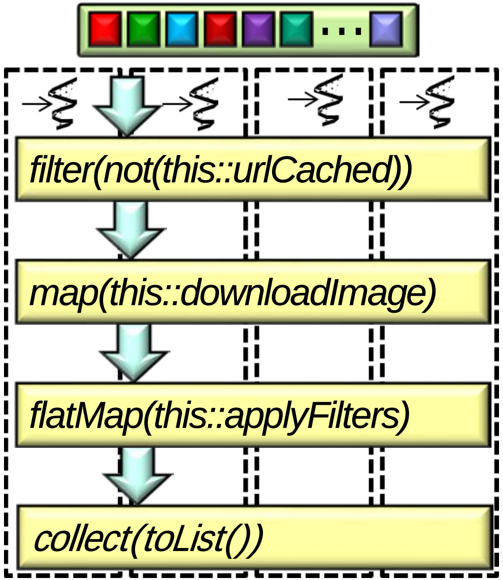
For applications that require separate or custom pools, a `ForkJoinPool` may be constructed with a given target parallelism level; by default, equal to the number of available processors. The pool attempts to maintain enough active (or available) threads by dynamically adding, suspending, or resuming internal worker threads, even if some tasks are stalled waiting to join others. However, no such adjustments are guaranteed in the face of blocked I/O or other unmanaged synchronization. The nested `ForkJoinPool.ManagedBlocker` interface enables extension of the kinds of synchronization accommodated.

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

Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool provides a high performance, fine-grained task execution framework for Java data parallelism
  - It provides a parallel computing engine for many higher-level frameworks

Parallel Streams



filter(not(this::urlCached))

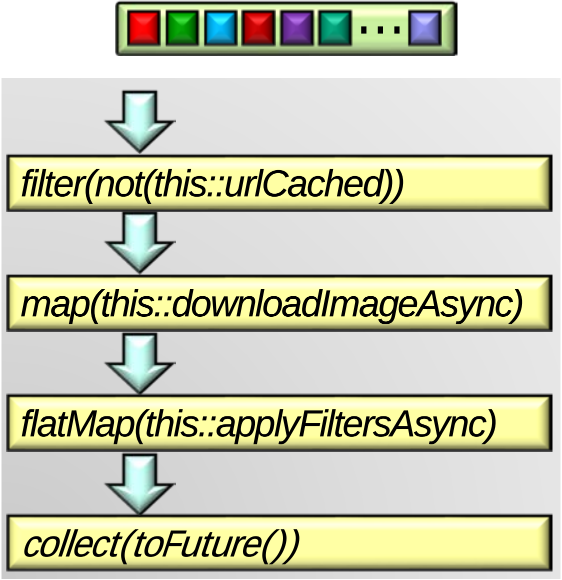
map(this::downloadImage)

flatMap(this::applyFilters)

collect(toList())

ForkJoinPool

Completable Futures



filter(not(this::urlCached))

map(this::downloadImageAsync)

flatMap(this::applyFiltersAsync)

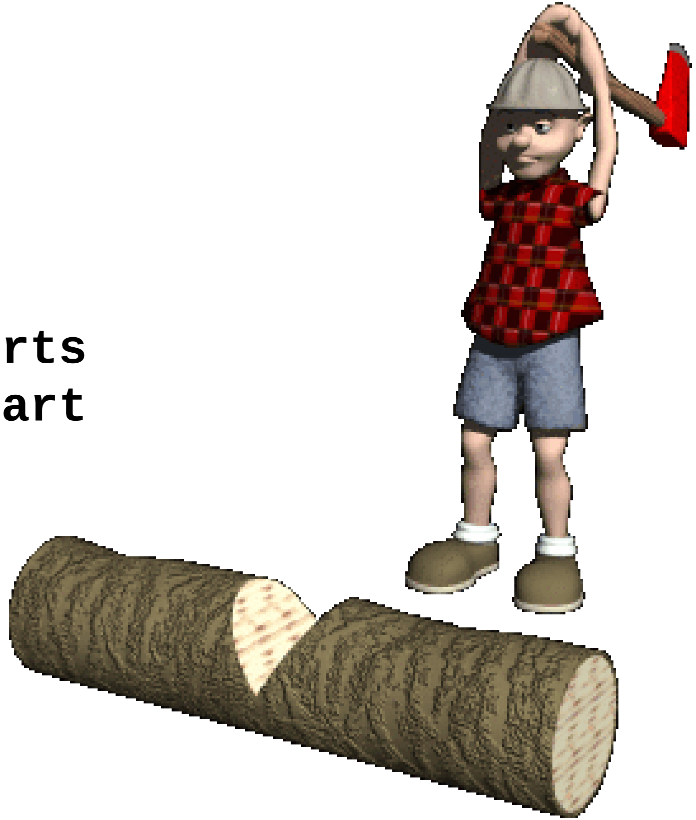
collect(toFuture())

See www.infoq.com/interviews/doug-lea-fork-join

Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer

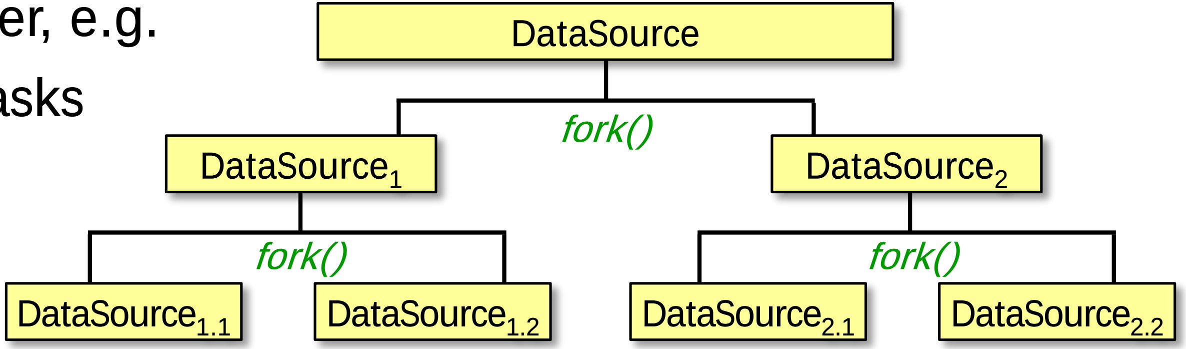
```
Result solve(Problem problem) {
  if (problem is small)
    directly solve problem
  else {
    split problem into independent parts
    fork new subtasks to solve each part
    join all subtasks
    compose result from subresults
  }
}
```



See en.wikipedia.org/wiki/Divide_and_conquer_algorithm

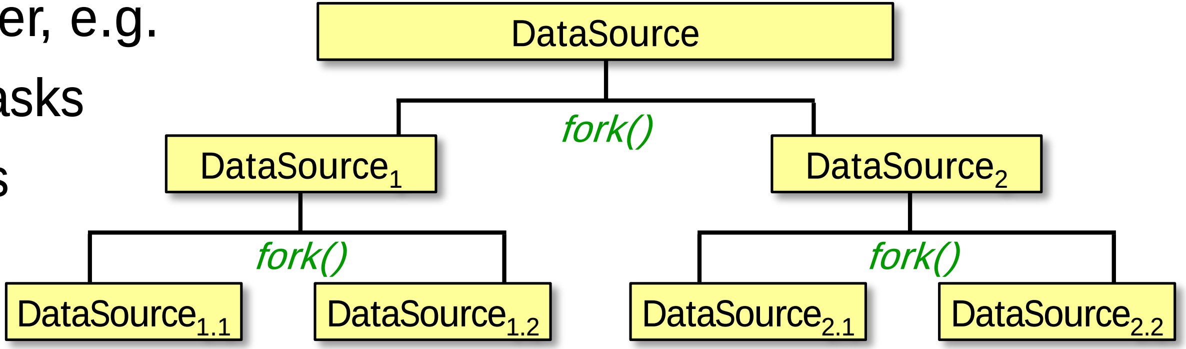
Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks



Overview of the Java Fork-Join Pool Computation Model

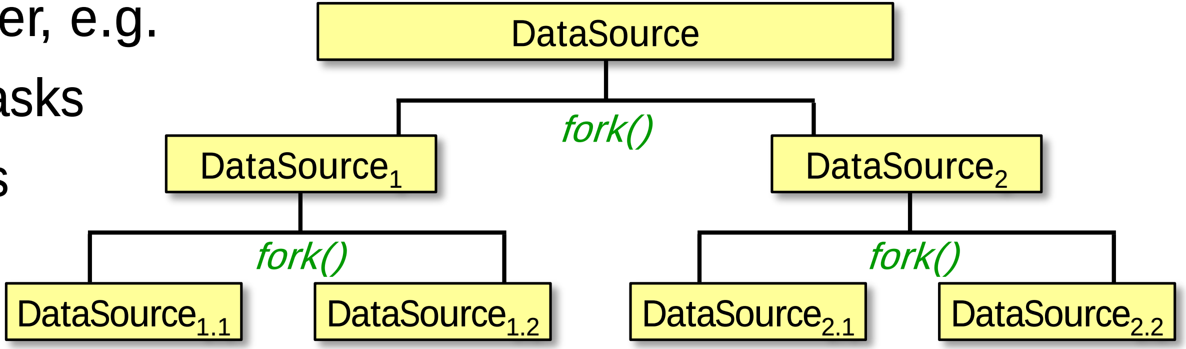
- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
    - A task creates sub-tasks by fork()'ing



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

Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
    - A task creates sub-tasks by fork()'ing



DataSource

fork()

DataSource$_1$

DataSource$_2$

fork()

fork()

DataSource$_{1.1}$

DataSource$_{1.2}$

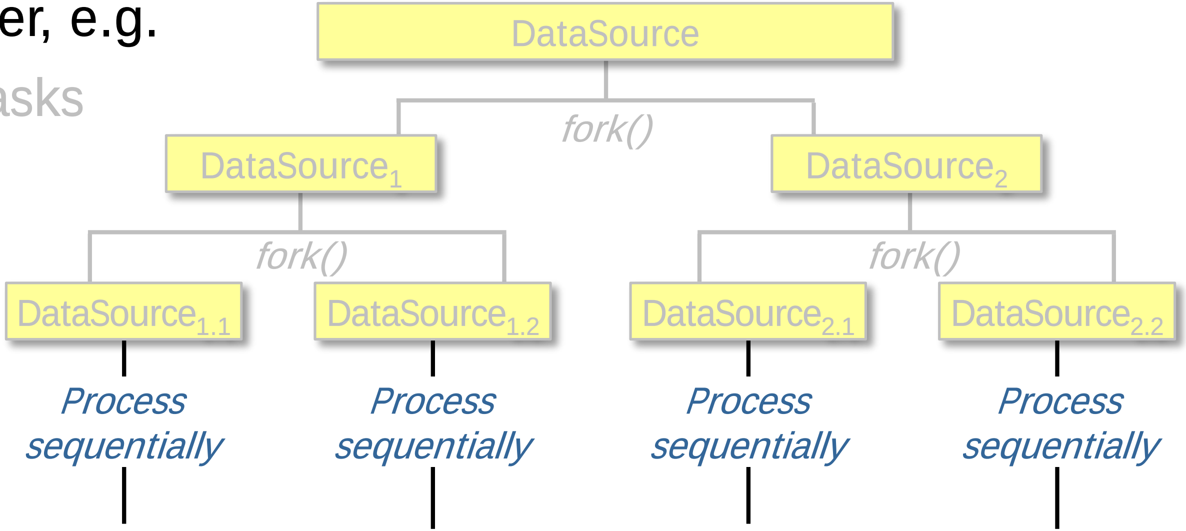
DataSource$_{2.1}$

DataSource$_{2.2}$

A (sub-)task only splits itself into (more) sub-tasks if the work is sufficiently big

Overview of the Java Fork-Join Pool Computation Model

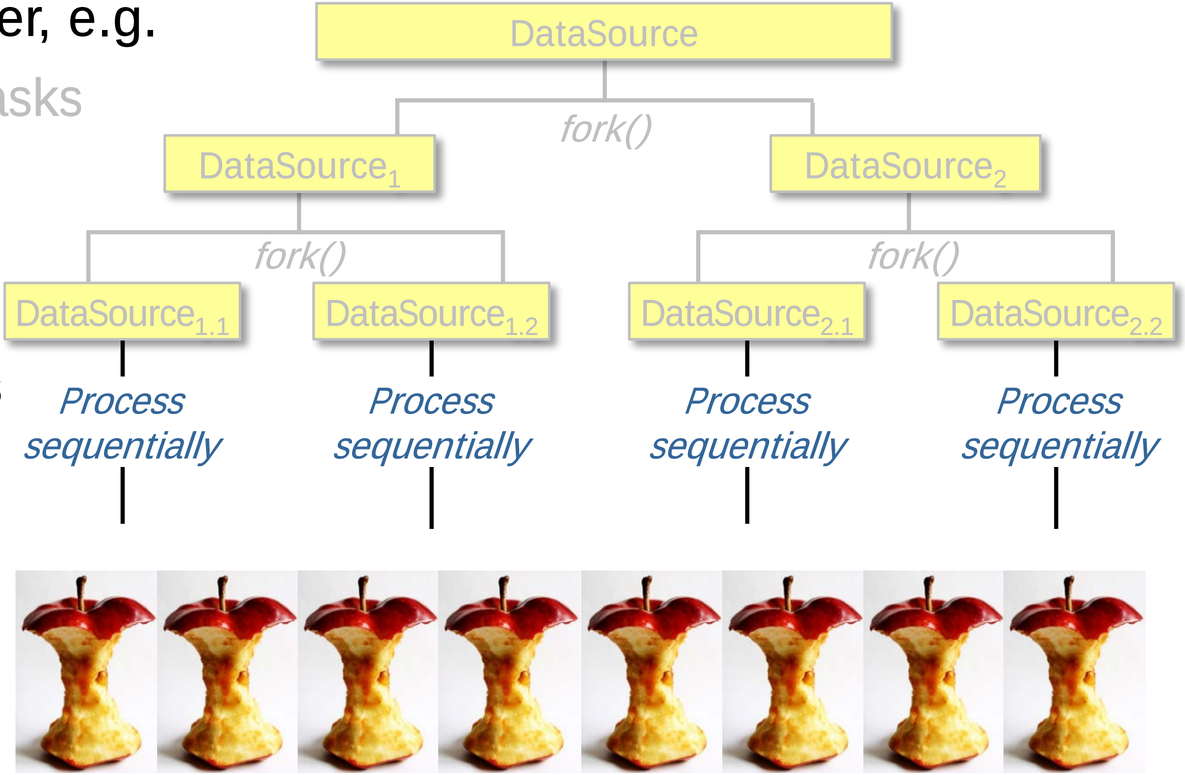
- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel



Implemented by fork-join framework, Java execution environment, OS, & hardware

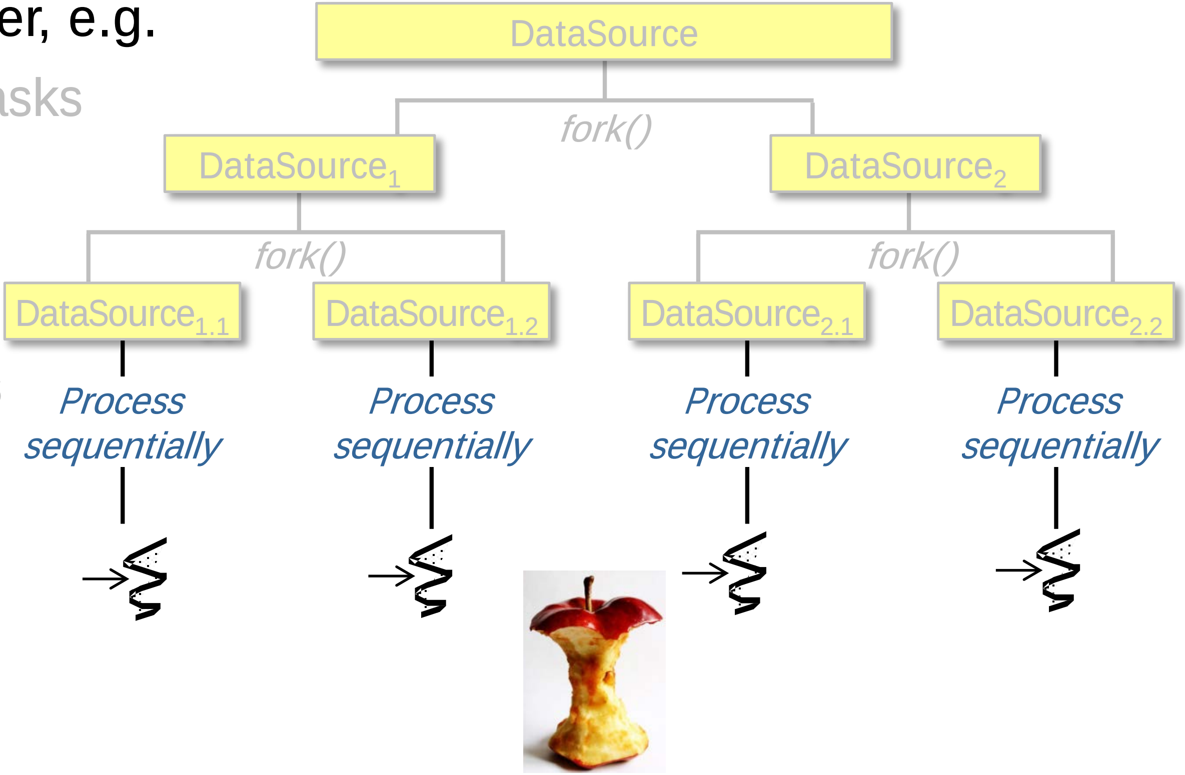
Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel
    - Sub-tasks can run in parallel on different cores



Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel
    - Sub-tasks can run in parallel on different cores
    - Sub-tasks can run concurrently in different threads on a single core



DataSource

fork()

DataSource$_1$

DataSource$_2$

fork()

fork()

DataSource$_{1.1}$

DataSource$_{1.2}$

DataSource$_{2.1}$

DataSource$_{2.2}$

Process sequentially

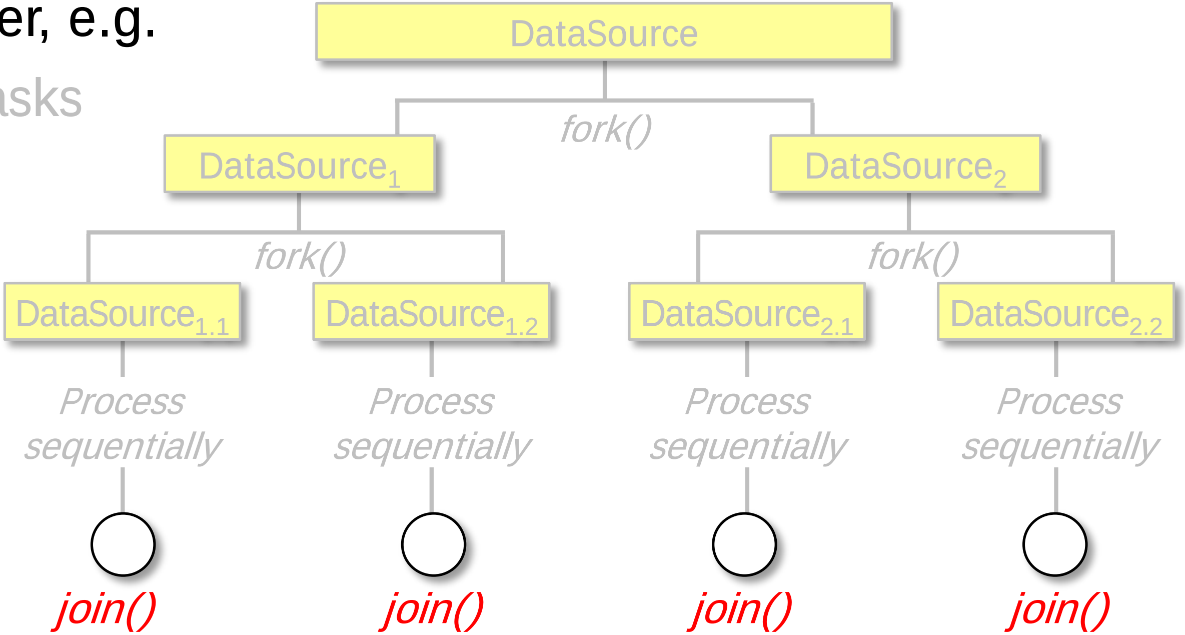
Process sequentially

Process sequentially

Process sequentially

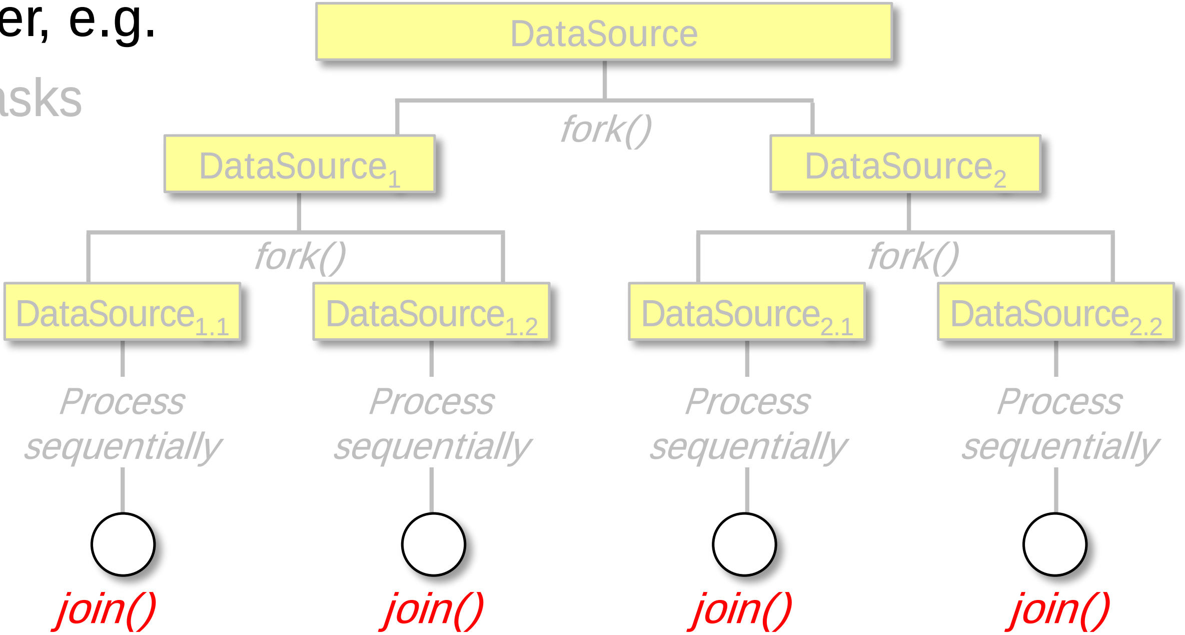
Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel
  - Waiting for them to complete



Overview of the Java Fork-Join Pool Computation Model

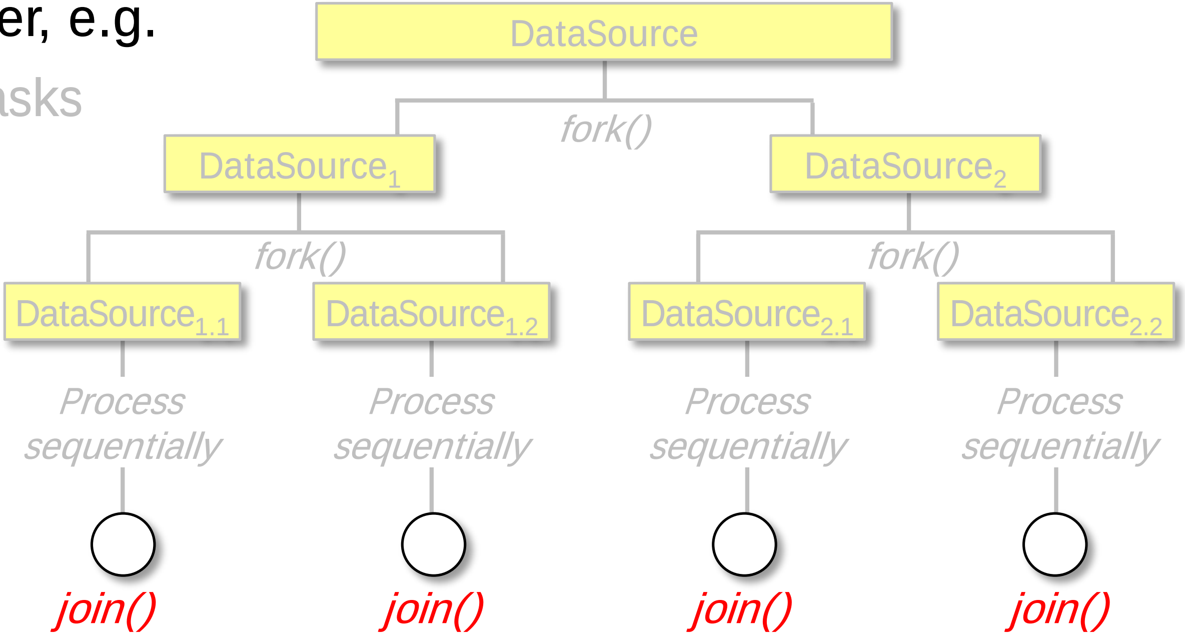
- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel
  - Waiting for them to complete
    - join() waits for a sub-task to finish



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

Overview of the Java Fork-Join Pool Computation Model

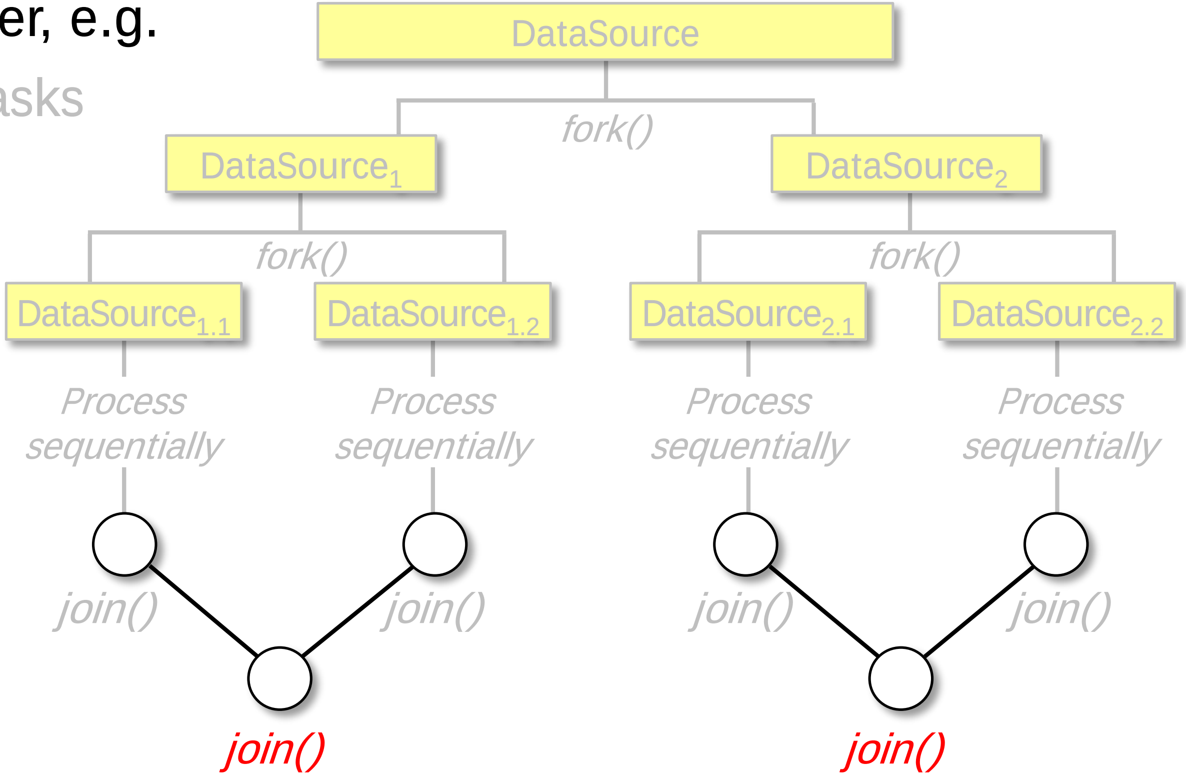
- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel
  - Waiting for them to complete
    - join() waits for a sub-task to finish



join() also plays a role in executing sub-tasks, as discussed shortly

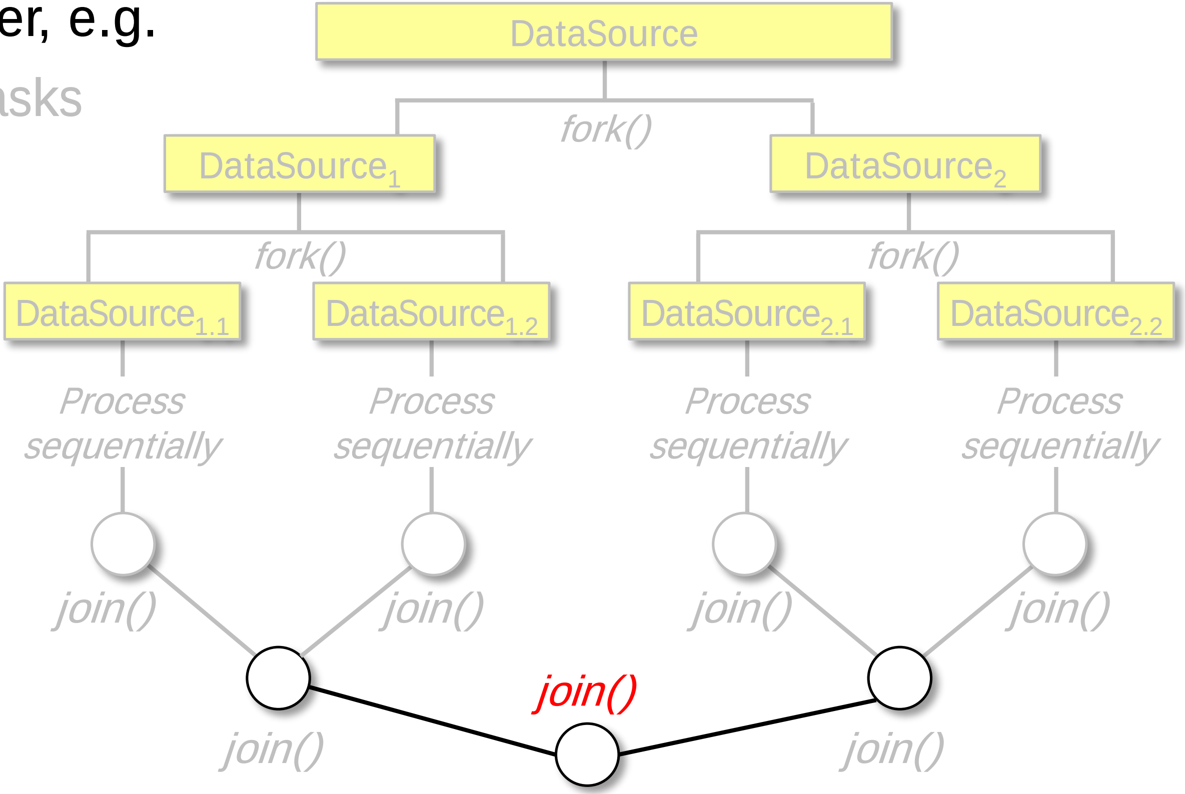
Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel
  - Waiting for them to complete
  - Merging the results



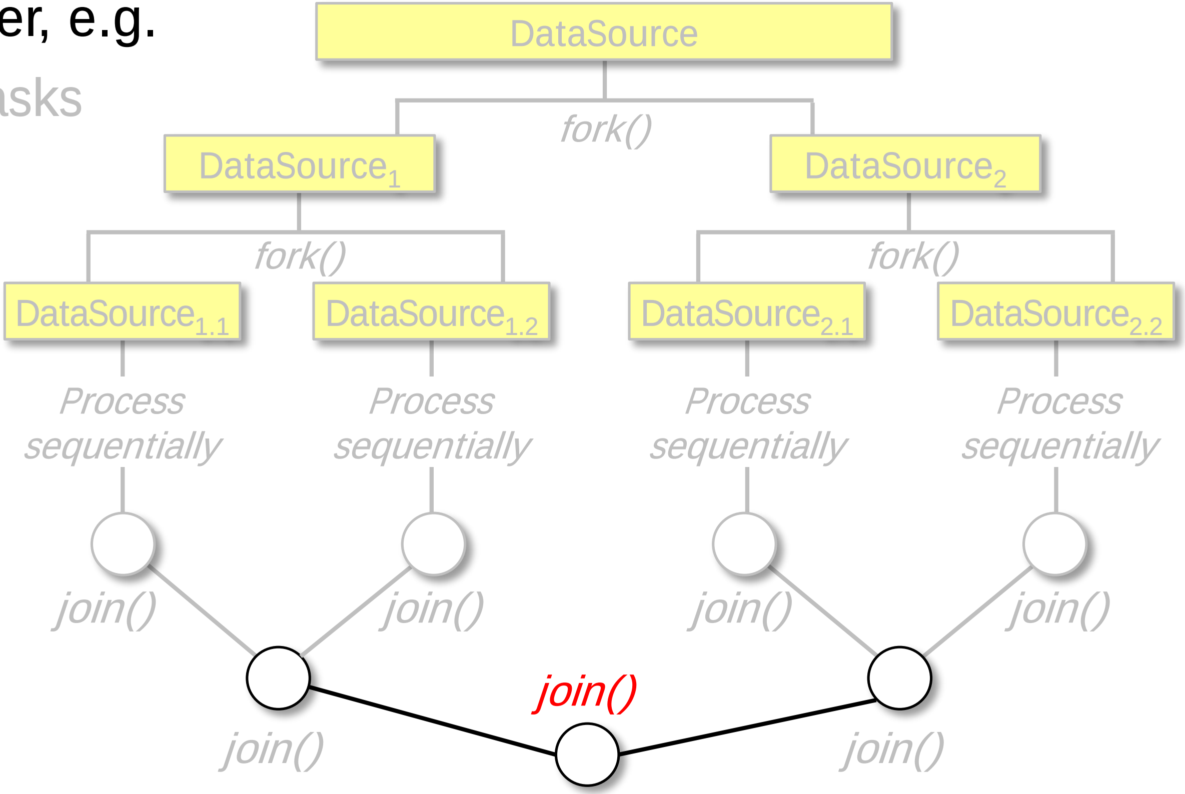
Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel
  - Waiting for them to complete
  - Merging the results
    - A task can use calls to join() to merge all sub -task results together



Overview of the Java Fork-Join Pool Computation Model

- The fork-join pool supports a style of parallel programming that solves problems by divide & conquer, e.g.
  - Splitting a task into sub-tasks
  - Solving the sub-tasks in parallel
  - Waiting for them to complete
  - Merging the results
    - A task can use calls to join() to merge all sub-task results together

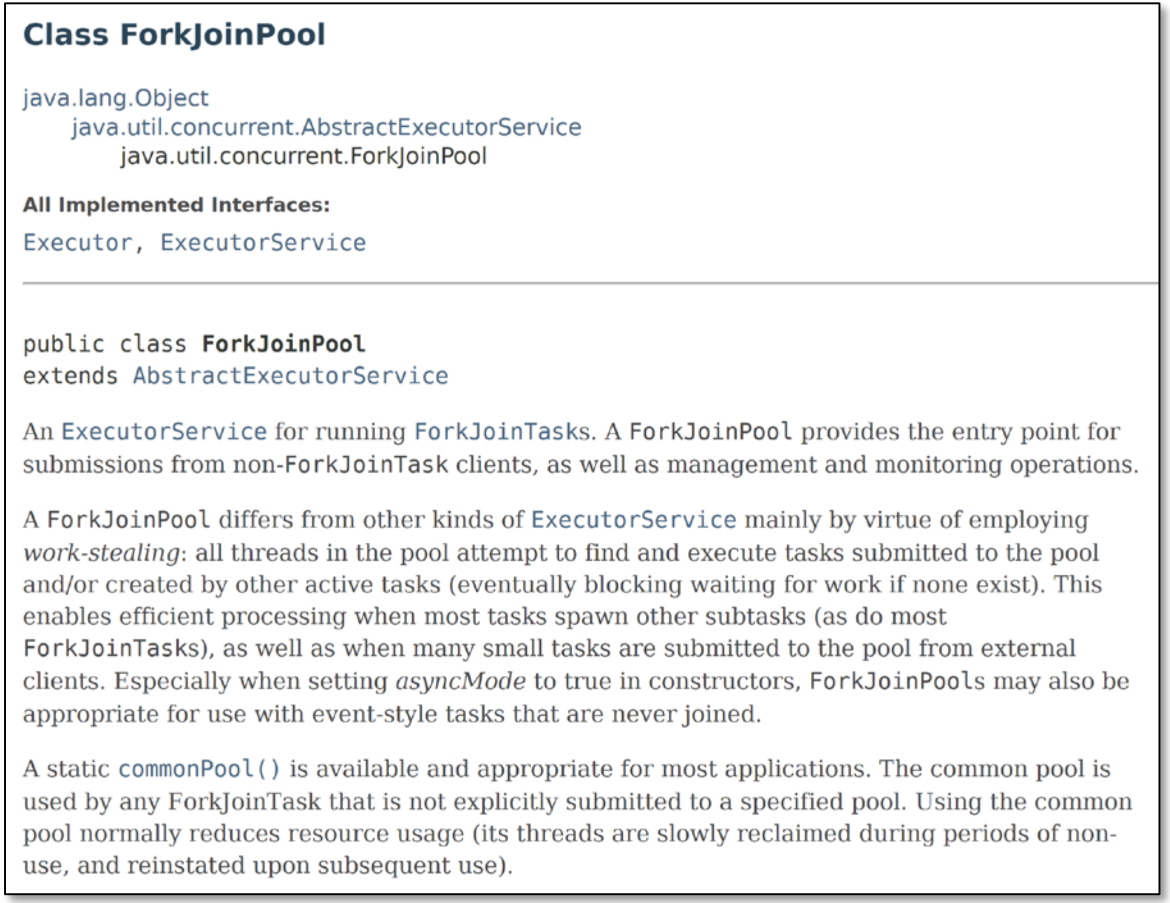


If a task does not return a result then it just waits for its sub-tasks to complete

The Fork-Join Framework Structure & Functionality

The Fork-Join Framework Structure & Functionality

- ForkJoinPool is an Executor Service implementation



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 `ForkJoinTask`s. A `ForkJoinPool` provides the entry point for submissions from non-`ForkJoinTask` clients, as well as management and monitoring operations.

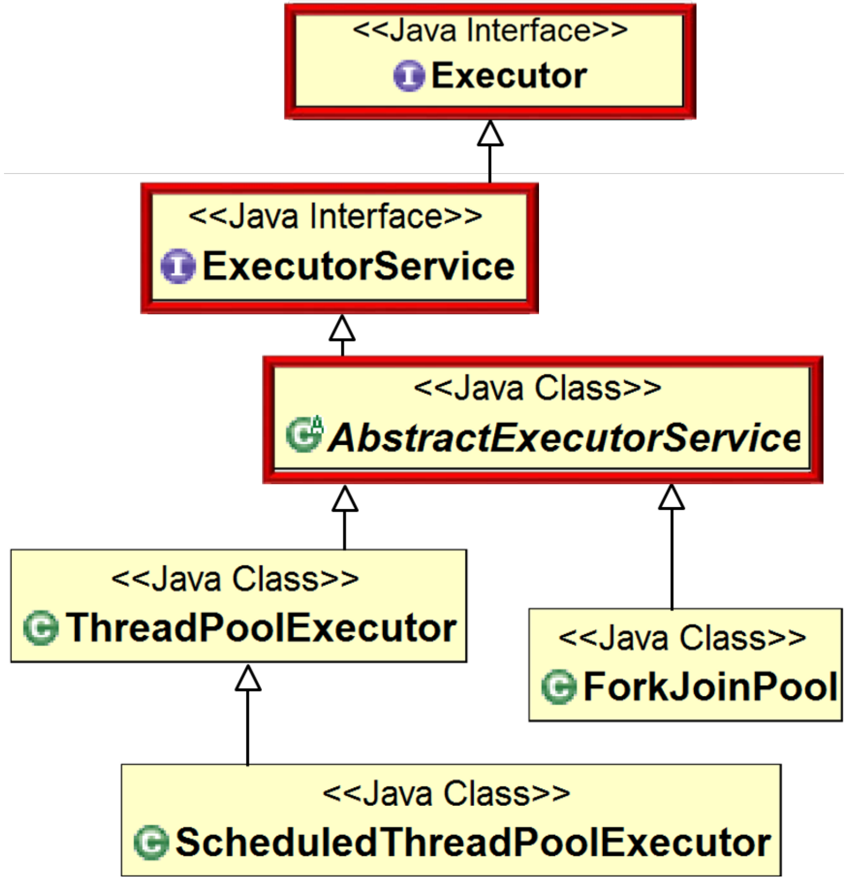
A `ForkJoinPool` differs from other kinds of `ExecutorService` mainly by virtue of employing *work-stealing*: all threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks (eventually blocking waiting for work if none exist). This enables efficient processing when most tasks spawn other subtasks (as do most `ForkJoinTask`s), as well as when many small tasks are submitted to the pool from external clients. Especially when setting *asyncMode* to true in constructors, `ForkJoinPool`s 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 Fork-Join Framework Structure & Functionality

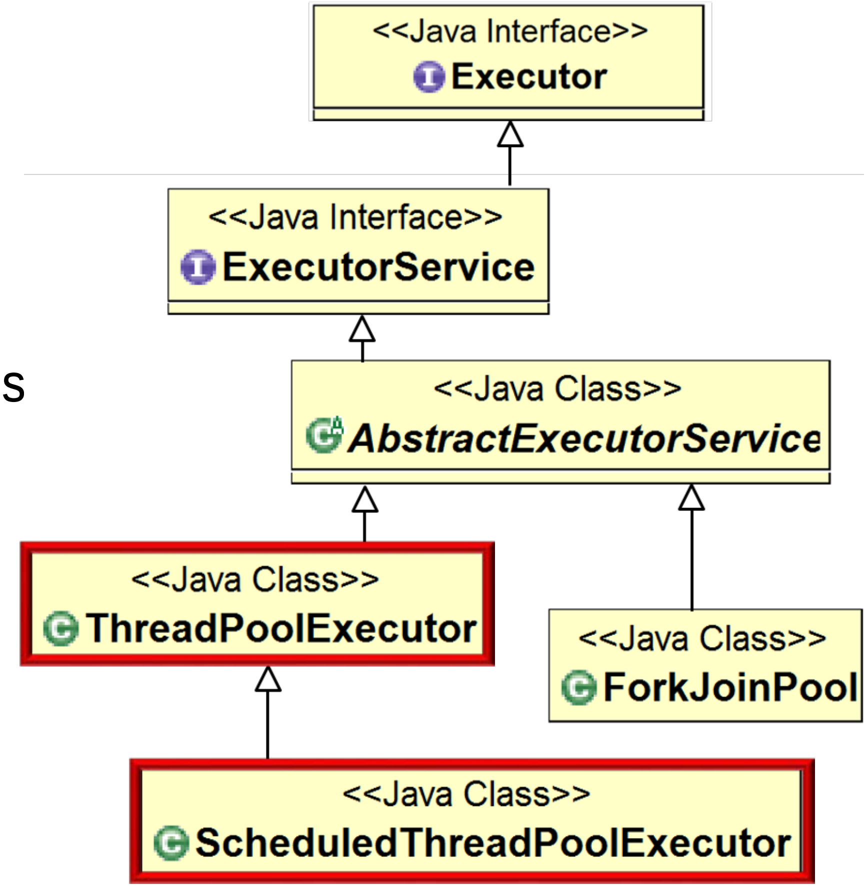
- ForkJoinPool is an Executor Service implementation
  - Executor Service is the basis for Java Executor framework subclasses



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

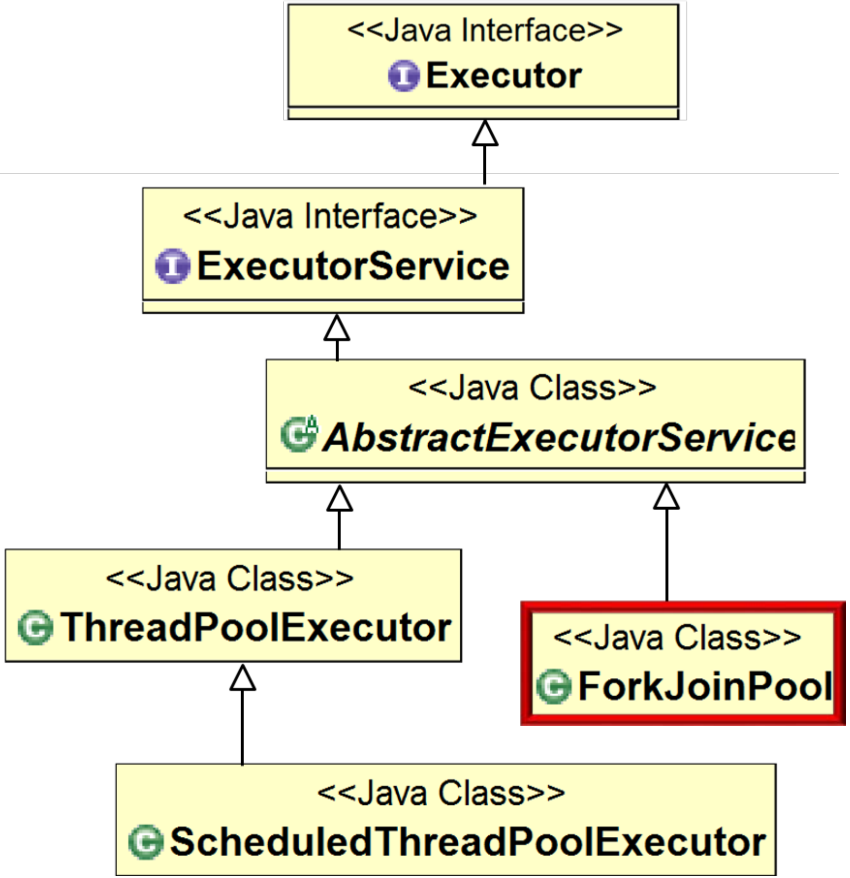
The Fork-Join Framework Structure & Functionality

- ForkJoinPool is an Executor Service implementation
  - Executor Service is the basis for Java Executor framework subclasses
  - Other implementations of Executor Service execute runnables or callables



The Fork-Join Framework Structure & Functionality

- ForkJoinPool is an Executor Service implementation
  - Executor Service 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 Fork-Join Framework Structure & Functionality

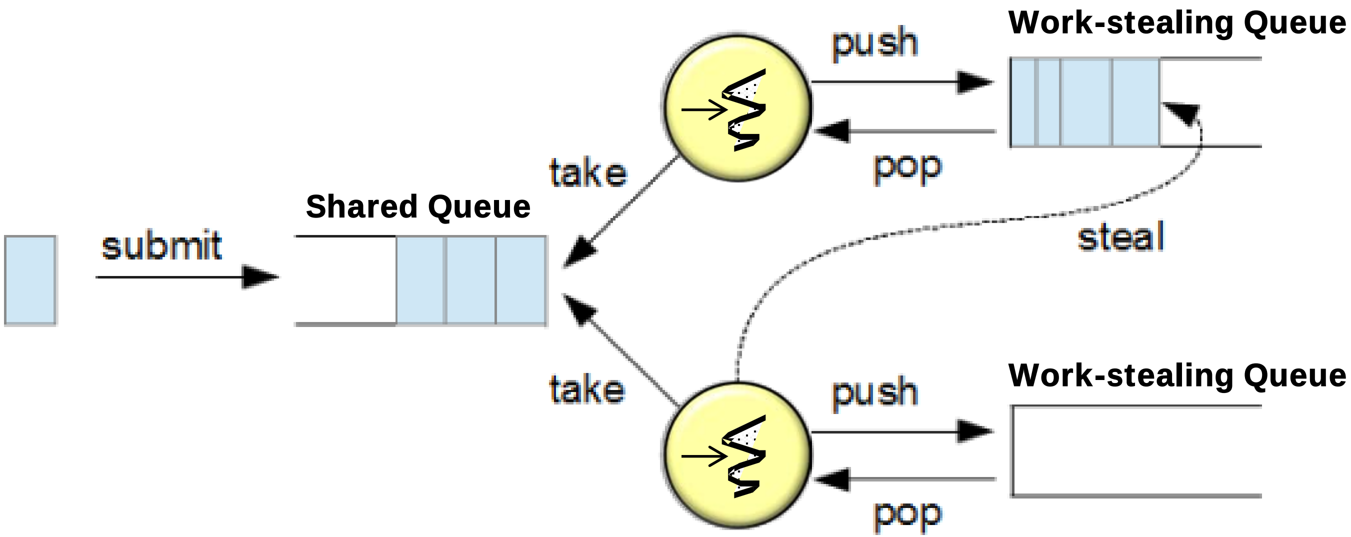
- ForkJoinPool enables non-ForkJoinTask clients to process ForkJoinTasks

void	execute(ForkJoinTask<T>) – Arrange async execution
T	invoke(ForkJoinTask<T>) – Performs the given task, returning its result upon completion
ForkJoinTask <T>	submit(ForkJoinTask) – Submits a ForkJoinTask for execution, returns a future

We'll discuss these methods later in this lesson

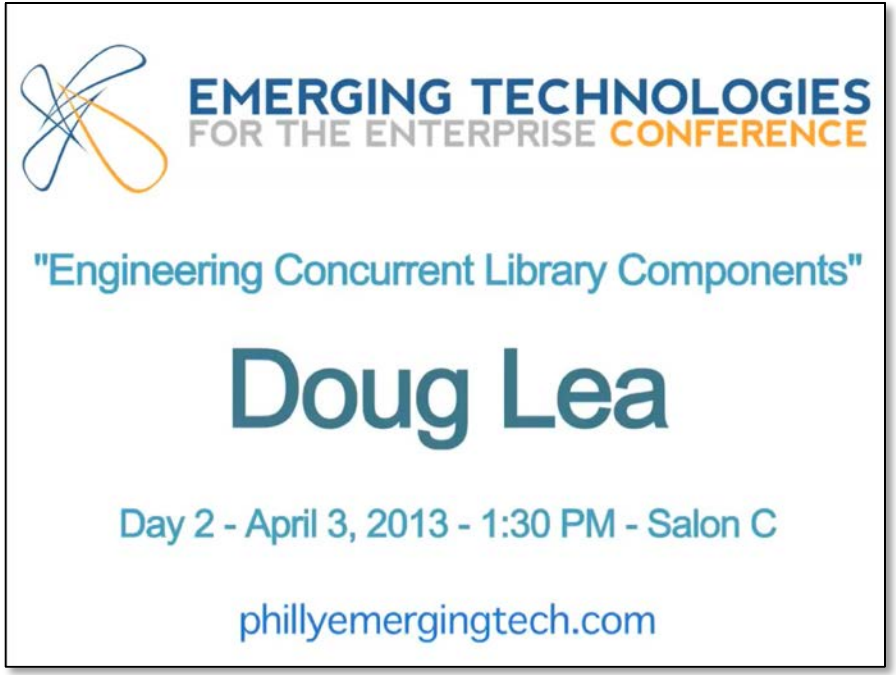
The Fork-Join Framework Structure & Functionality

- ForkJoinPool enables non-ForkJoinTask clients to process ForkJoinTasks
  - Clients insert new tasks onto a shared queued used to feed work-stealing queues managed by worker threads



Overview of Java Fork-Join Framework Internals

- ForkJoinPool enables non-ForkJoinTask clients to process ForkJoinTasks
  - Clients insert new tasks onto a shared queued used to feed work-stealing queues managed by worker threads
  - The goal is to maximize utilization of processor cores



EMERGING TECHNOLOGIES
FOR THE ENTERPRISE CONFERENCE

"Engineering Concurrent Library Components"

Doug Lea

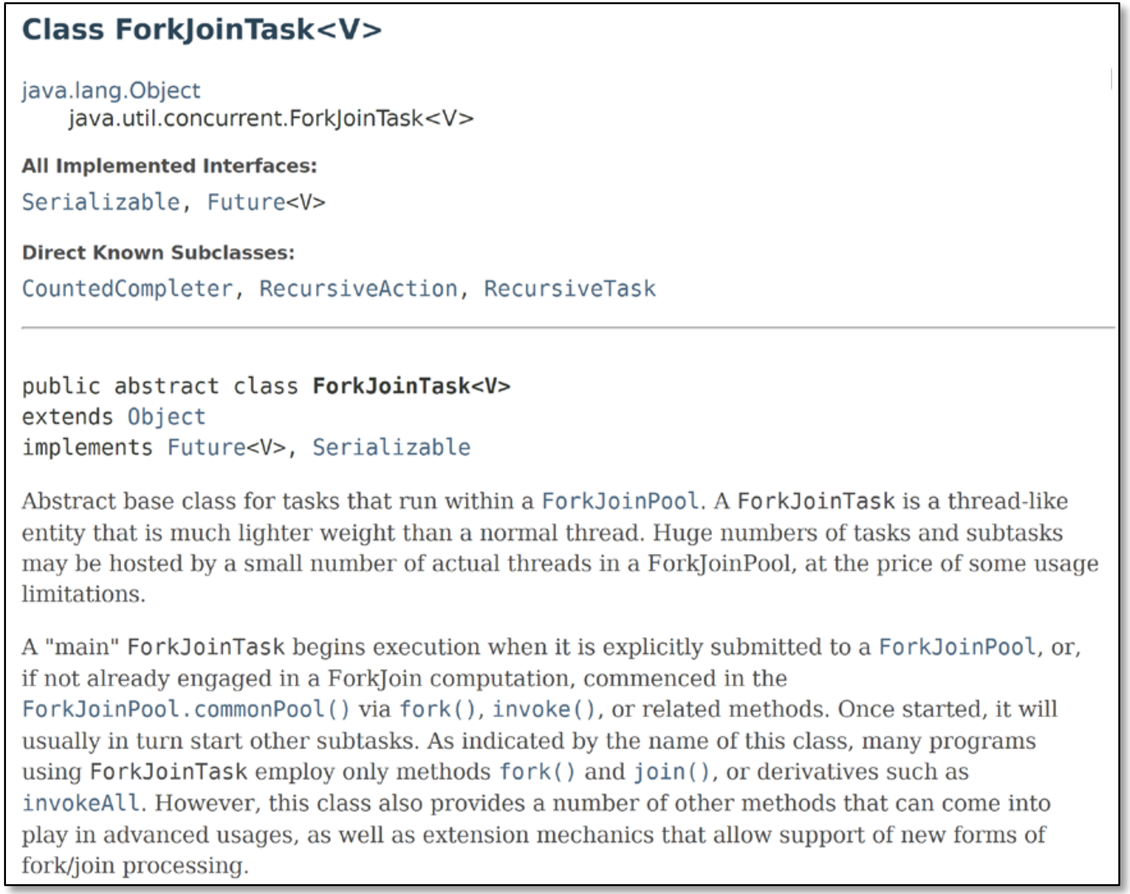
Day 2 - April 3, 2013 - 1:30 PM - Salon C

phillyemergingtech.com

See www.youtube.com/watch?v=sq0MX3fHkro

The Fork-Join Framework Structure & Functionality

- 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
```

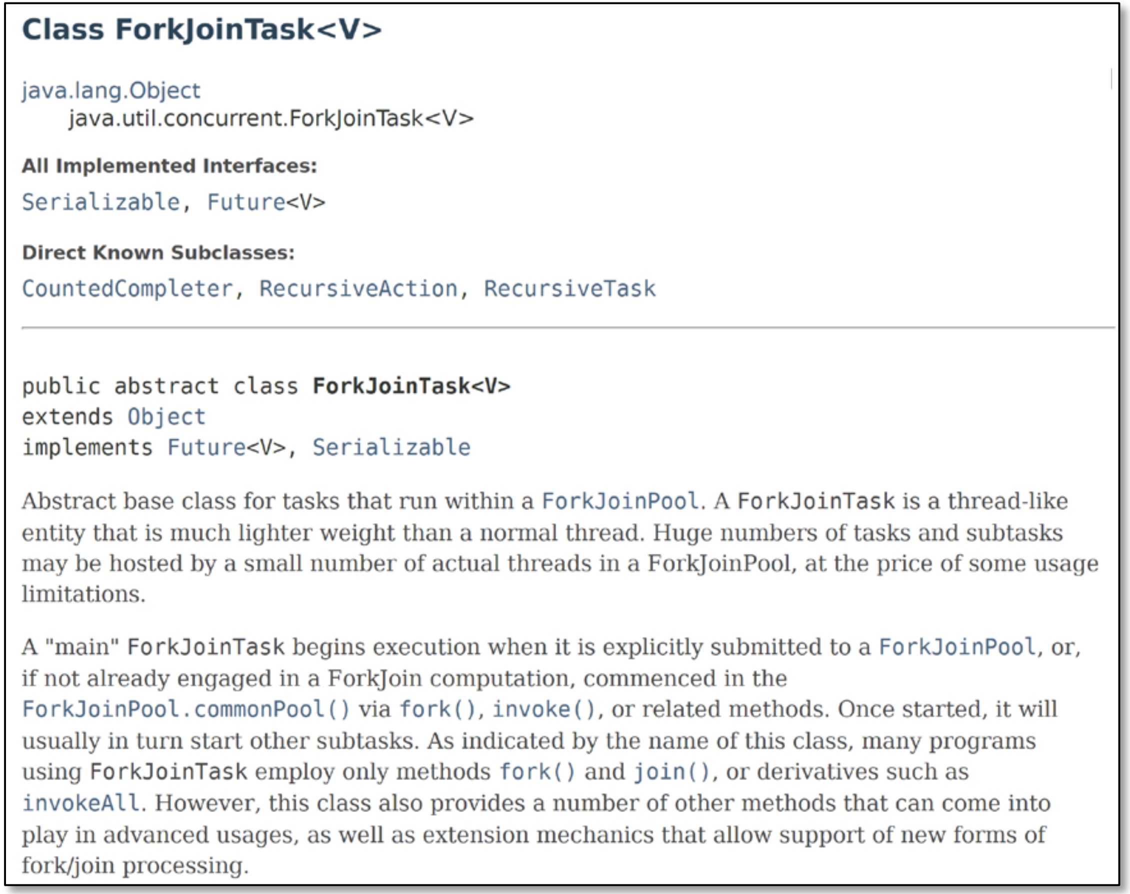
Abstract base class for tasks that run within a `ForkJoinPool`. A `ForkJoinTask` is a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a ForkJoinPool, at the price of some usage limitations.

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 Fork-Join Framework Structure & Functionality

- 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
```

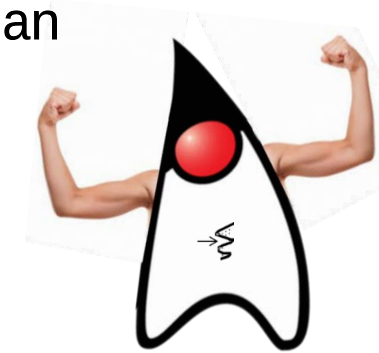
Abstract base class for tasks that run within a `ForkJoinPool`. A `ForkJoinTask` is a thread-like entity that is much lighter weight than a normal thread. Huge numbers of tasks and subtasks may be hosted by a small number of actual threads in a ForkJoinPool, at the price of some usage limitations.

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.

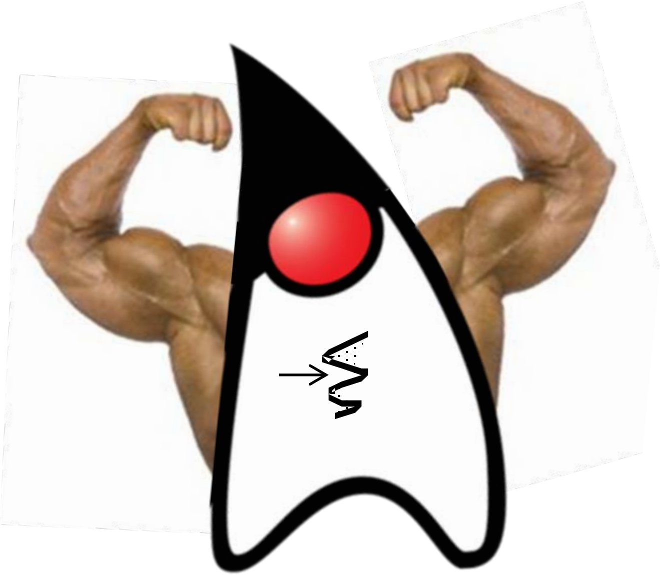
See www.dre.Vanderbilt.edu/~schmidt/PDF/DataParallelismInJava.pdf

The Fork-Join Framework Structure & Functionality

- A ForkJoinTask is lighter weight than a Java thread



ForkJoinTask

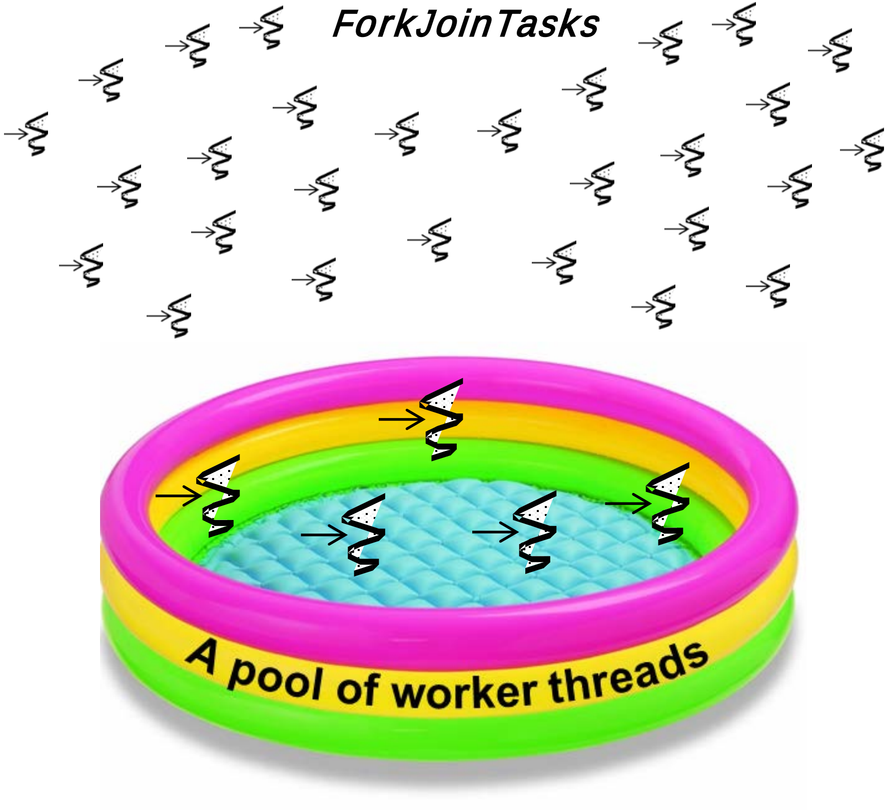


Thread

e.g., it doesn't maintain its own run-time stack

The Fork-Join Framework Structure & Functionality

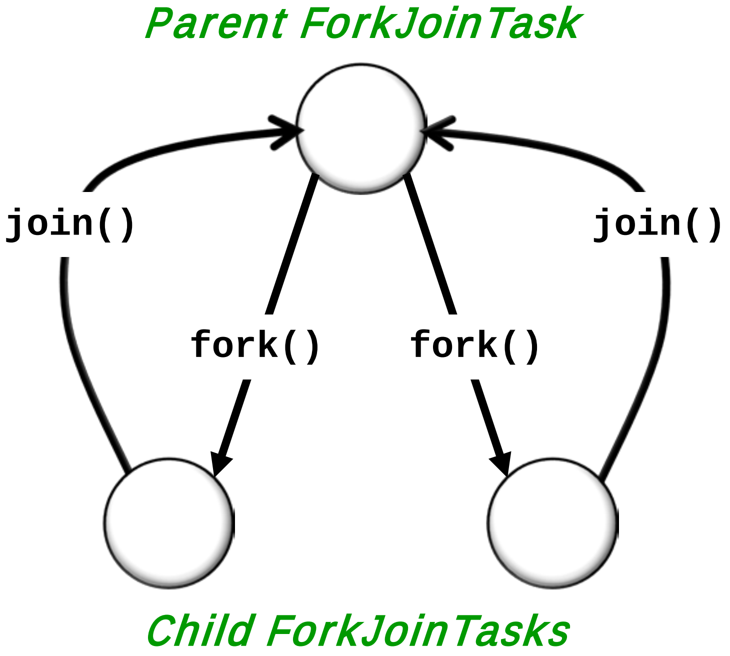
- A ForkJoinTask is lighter weight than a Java thread
  - A large # of ForkJoinTasks can thus run in a small # of worker threads in a fork-join pool



Each worker thread is a Java Thread object with its own stack, registers, etc.

The Fork-Join Framework Structure & Functionality

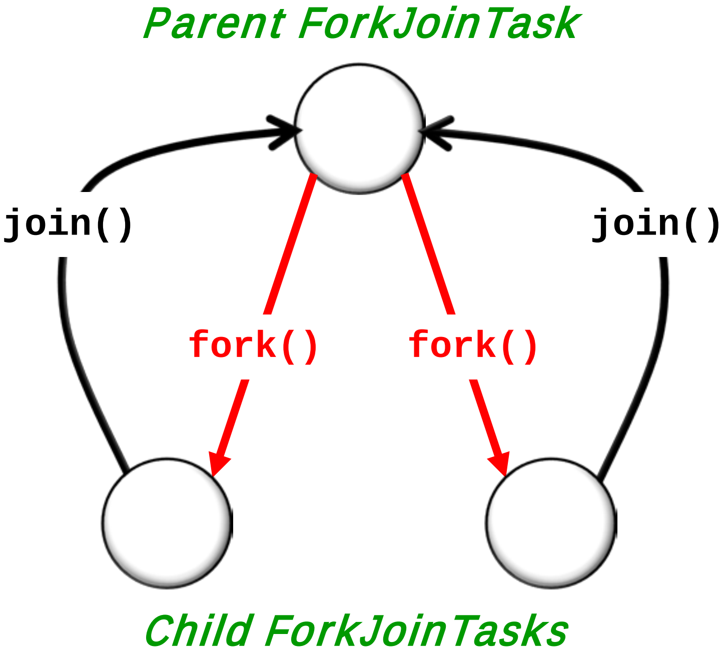
- A ForkJoinTask has two methods that control parallel processing/merging



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

The Fork-Join Framework Structure & Functionality

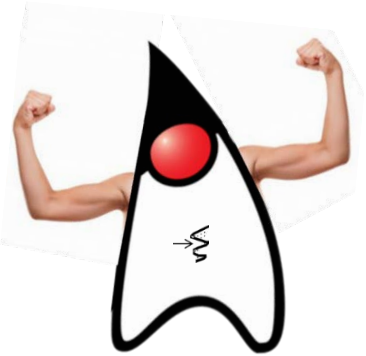
- A ForkJoinTask has two methods that control parallel processing/merging



Parent ForkJoinTask

Child ForkJoinTasks

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

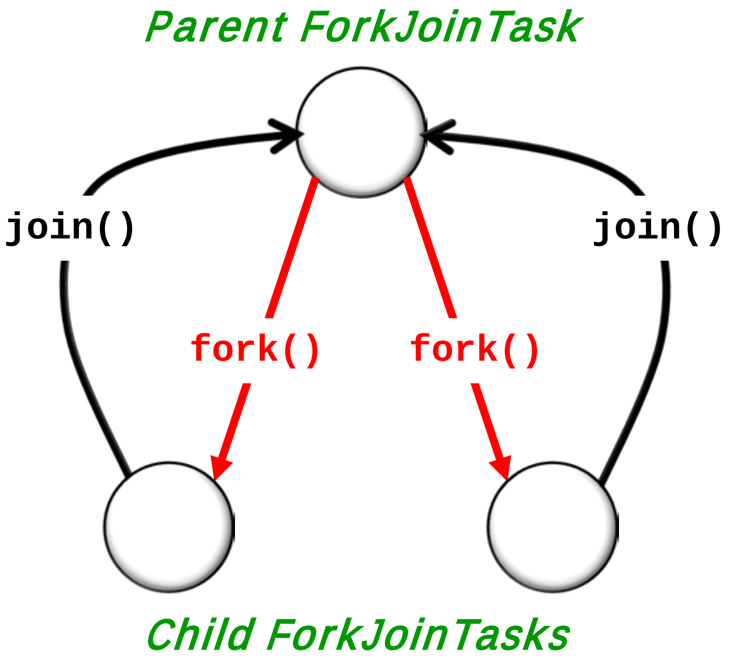


ForkJoinTask

fork() is akin to a lightweight version of Thread.start()

The Fork-Join Framework Structure & Functionality

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

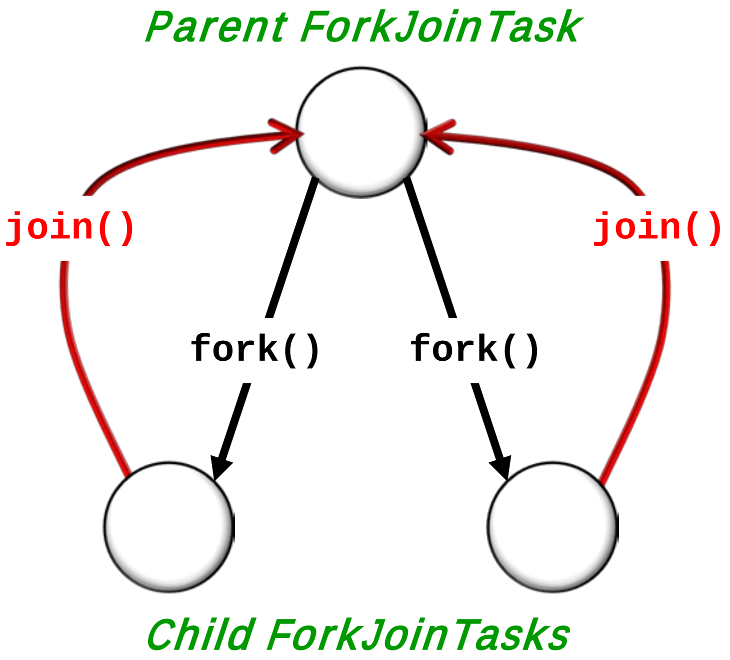


ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns the result of the computation when it is done

fork() does not run the task immediately, but instead places it on a work queue

The Fork-Join Framework Structure & Functionality

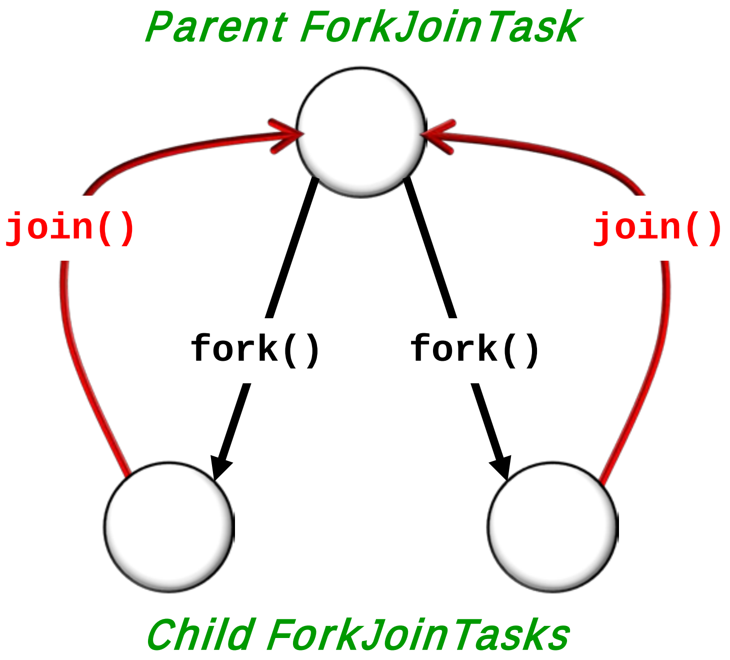
- A ForkJoinTask has two methods that control parallel processing/merging



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

The Fork-Join Framework Structure & Functionality

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

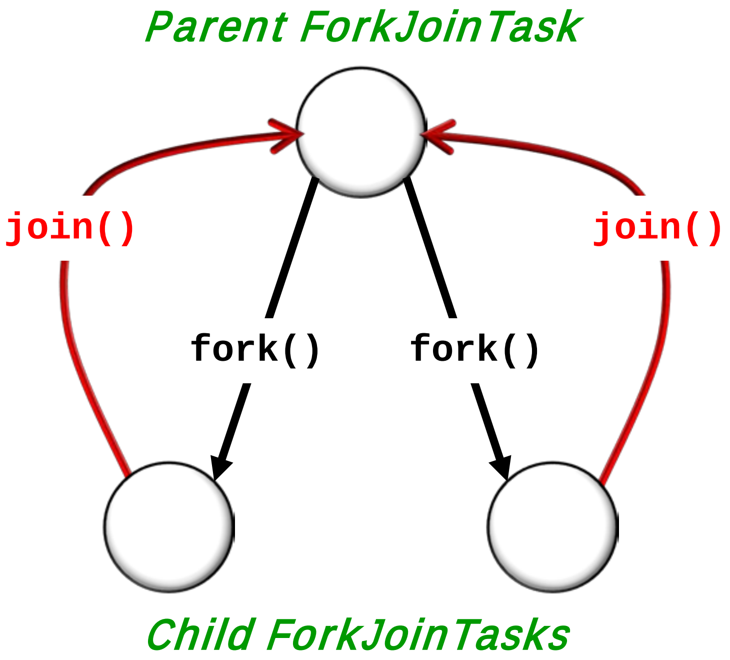


ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns the result of the computation when it is done

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

The Fork-Join Framework Structure & Functionality

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

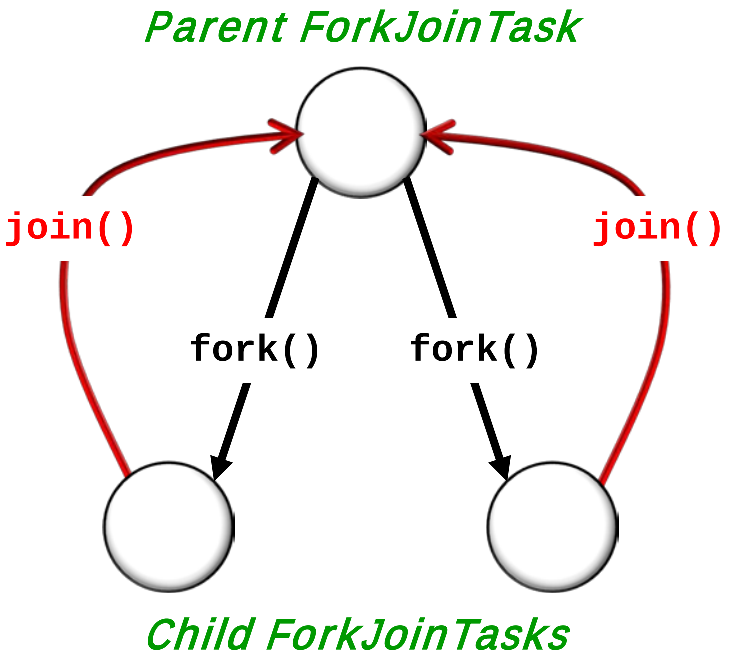


ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns the result of the computation when it is done

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

The Fork-Join Framework Structure & Functionality

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

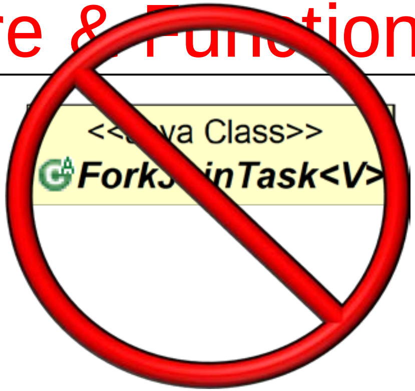


ForkJoinTask <T>	**fork**() – Arranges to asynchronously execute this task in the appropriate pool
V	**join**() – Returns the result of the computation when it is done

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

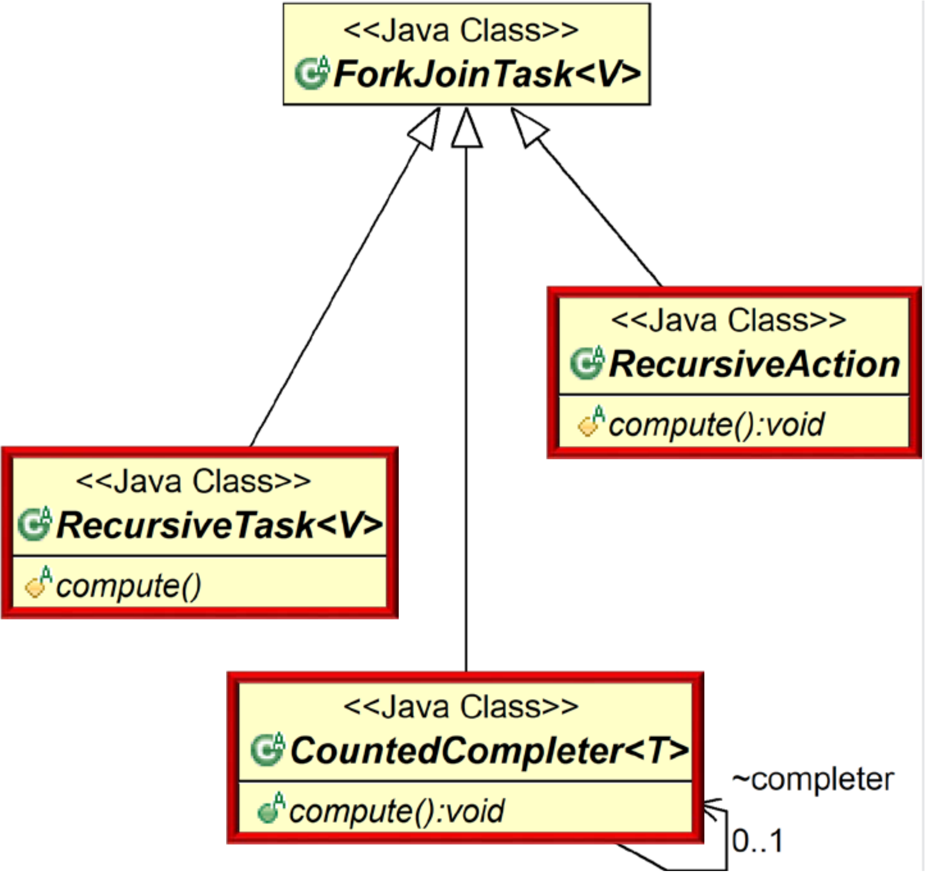
The Fork-Join Framework Structure & Functionality

- Programs rarely use the ForkJoinTask class directly



The Fork-Join Framework Structure & Functionality

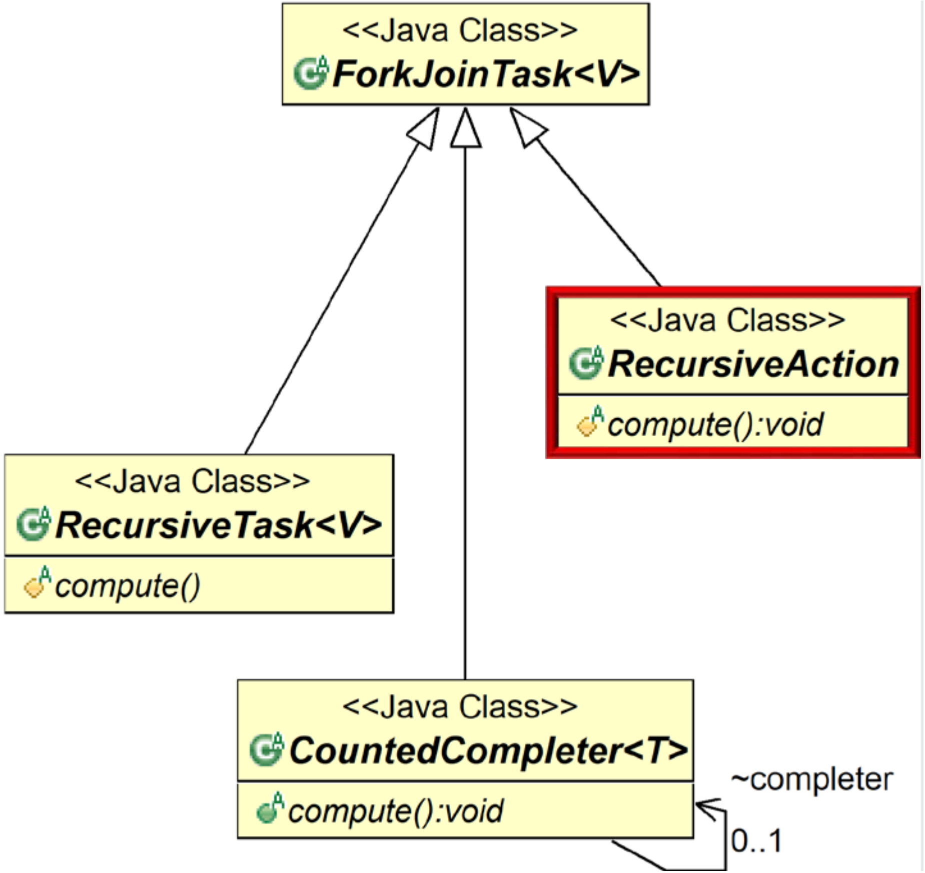
- Programs rarely use the ForkJoinTask class directly … but instead extend one of its subclasses & override compute()



See docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-tree.html

The Fork-Join Framework Structure & Functionality

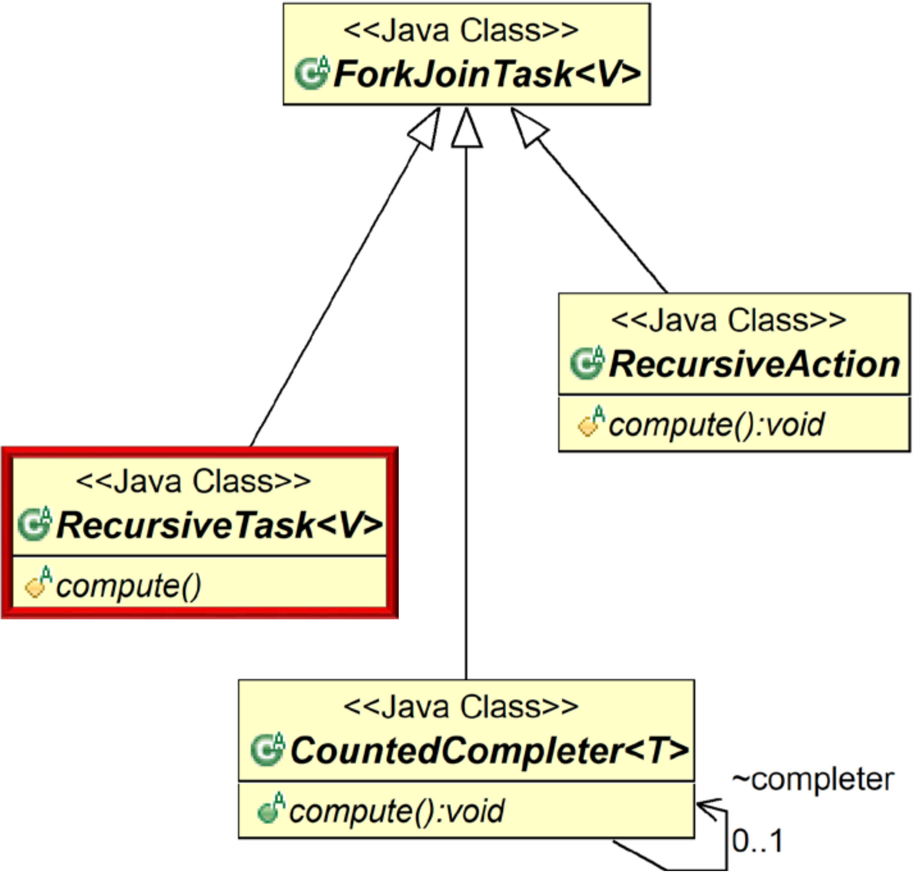
- Programs rarely use the ForkJoinTask class directly … but instead extend one of its subclasses & override compute()
  - **RecursiveAction**
    - Use for computations that do not return results



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

The Fork-Join Framework Structure & Functionality

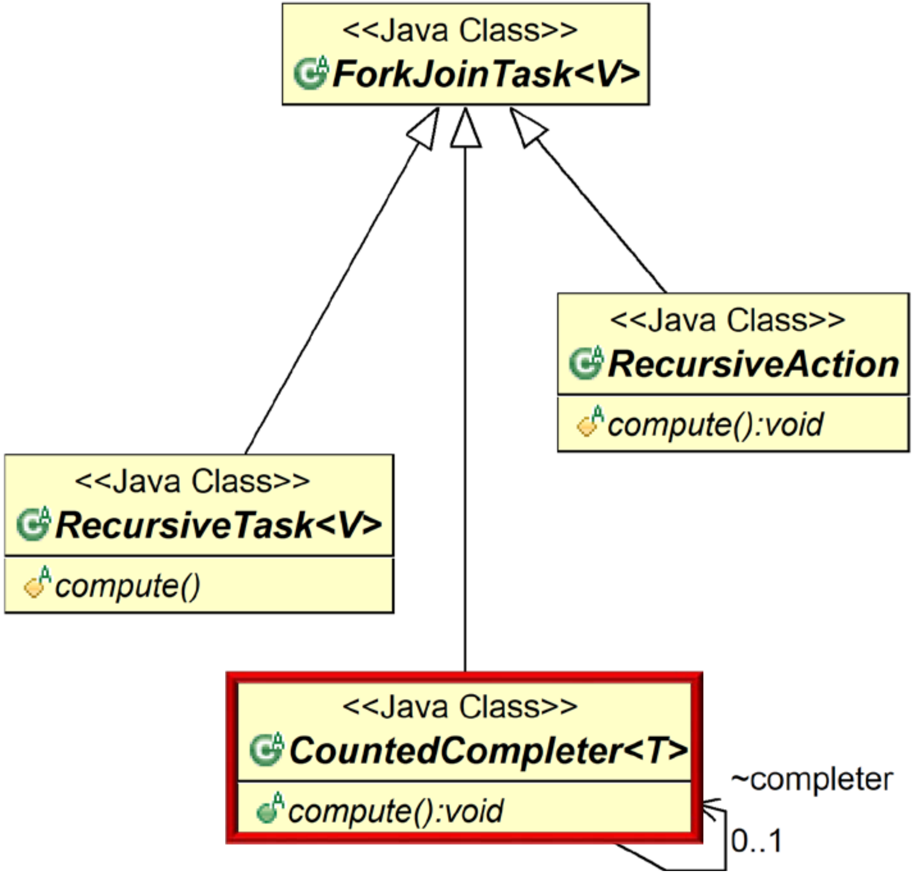
- Programs rarely use the ForkJoinTask class directly … but instead extend one of its subclasses & override compute()
  - RecursiveAction
  - **RecursiveTask**
    - Use for computations that do return results



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

The Fork-Join Framework Structure & Functionality

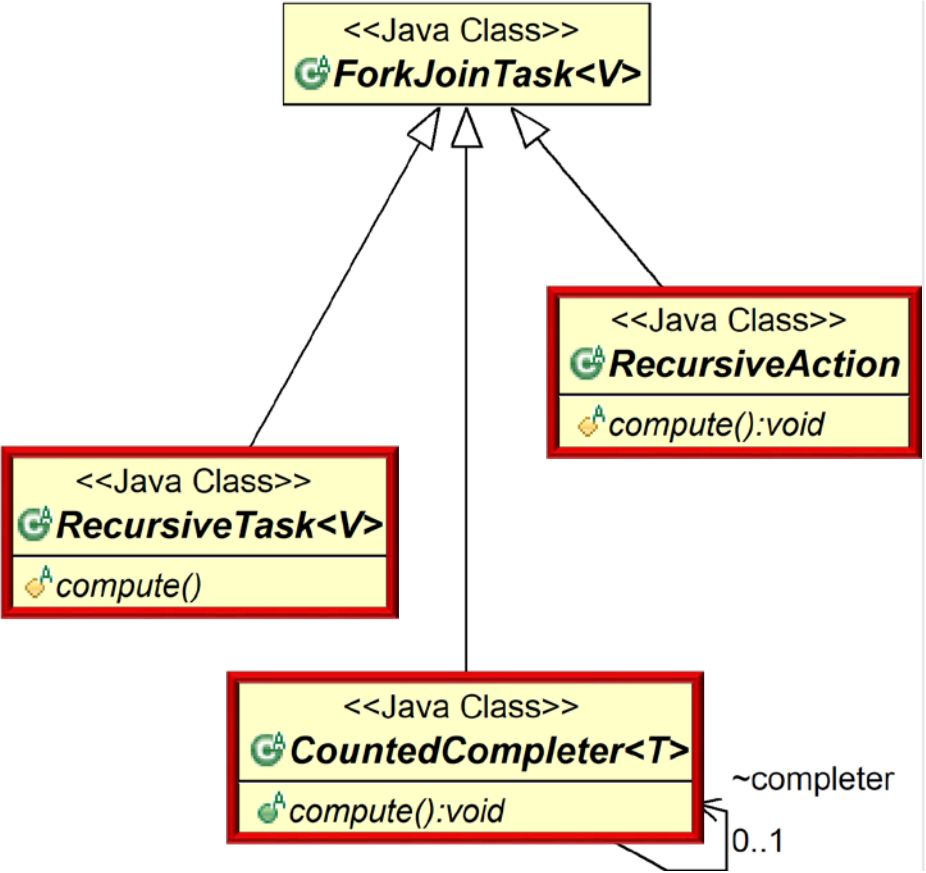
- Programs rarely use the ForkJoinTask class directly … but instead extend one of its subclasses & override compute()
  - RecursiveAction
  - RecursiveTask
  - **CountedCompleter**
    - Used for computations in which completed actions trigger other actions



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

The Fork-Join Framework Structure & Functionality

- Programs rarely use the ForkJoinTask class directly … but instead extend one of its subclasses & override compute()
  - **RecursiveAction**
  - **RecursiveTask**
  - **CountedCompleter**

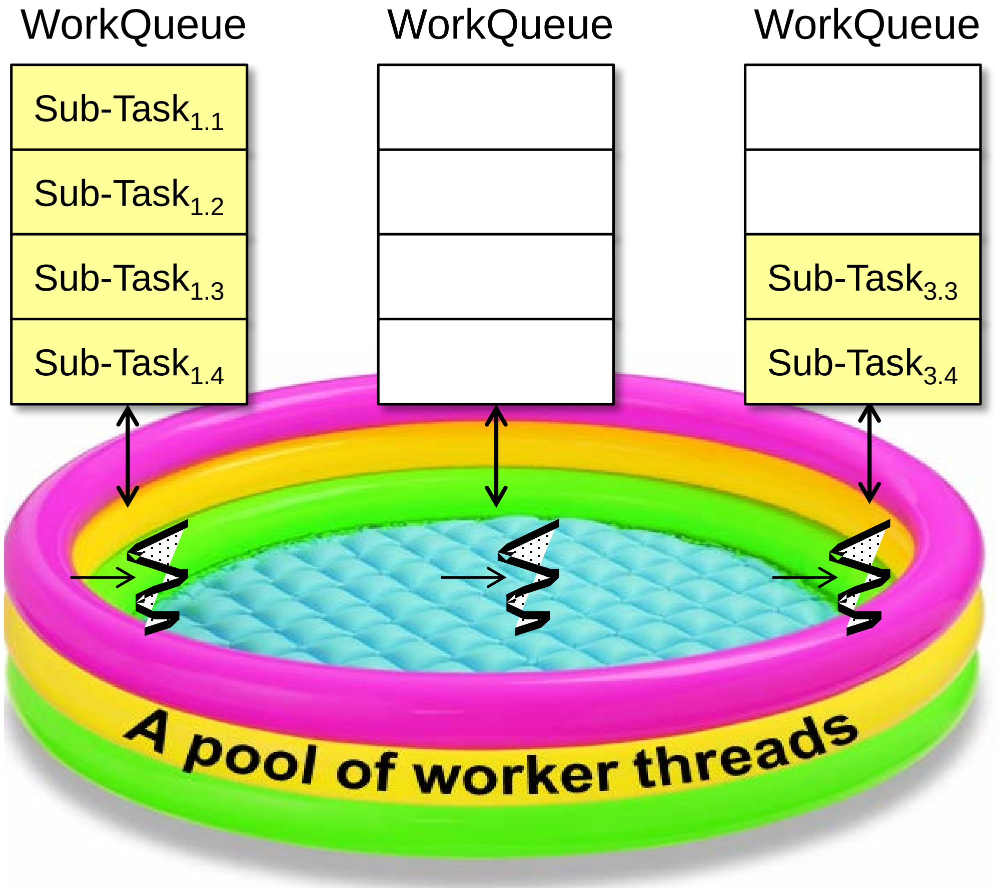


None of the classes are functional interfaces, so lambda expressions can't be used..

Overview of Java Fork-Join Framework Internals

Overview of Java Fork-Join Framework Internals

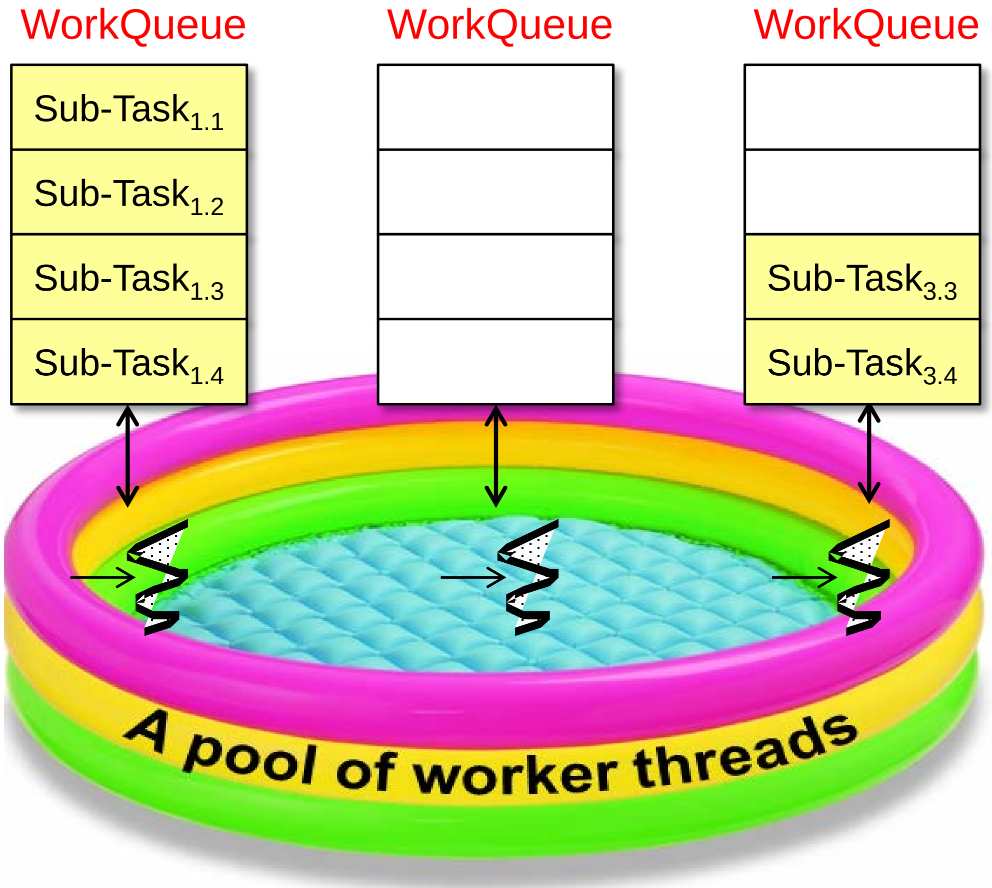
- Each worker thread in a fork-join pool maintains its own "double-ended queue" (deque)



See en.wikipedia.org/wiki/Double-ended_queue

Overview of Java Fork-Join Framework Internals

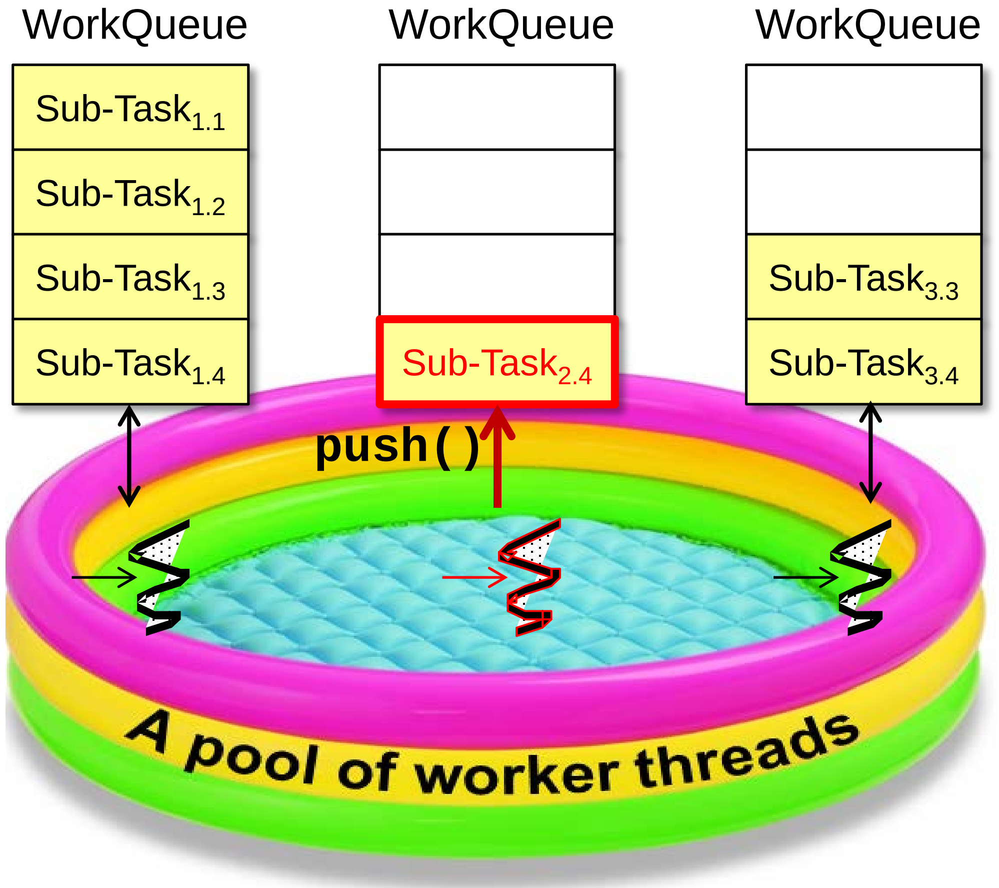
- Each worker thread in a fork-join pool maintains its own "double-ended queue" (deque)
  - The Java fork-join framework implements this deque via the WorkQueue class



See java8/util/concurrent/ForkJoinPool.java

Overview of Java Fork-Join Framework Internals

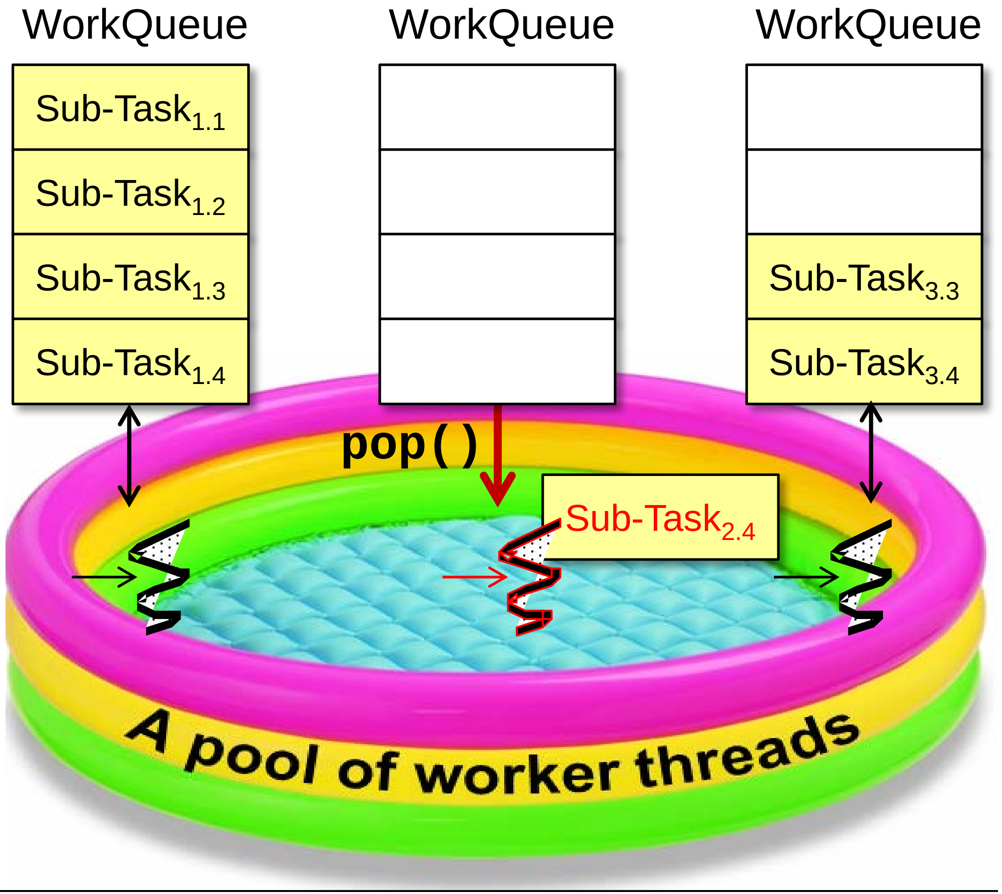
- Sub-tasks fork()'d in a task run by a worker thread are pushed onto the head of that worker's own deque



See gee.cs.oswego.edu/dl/papers/fj.pdf

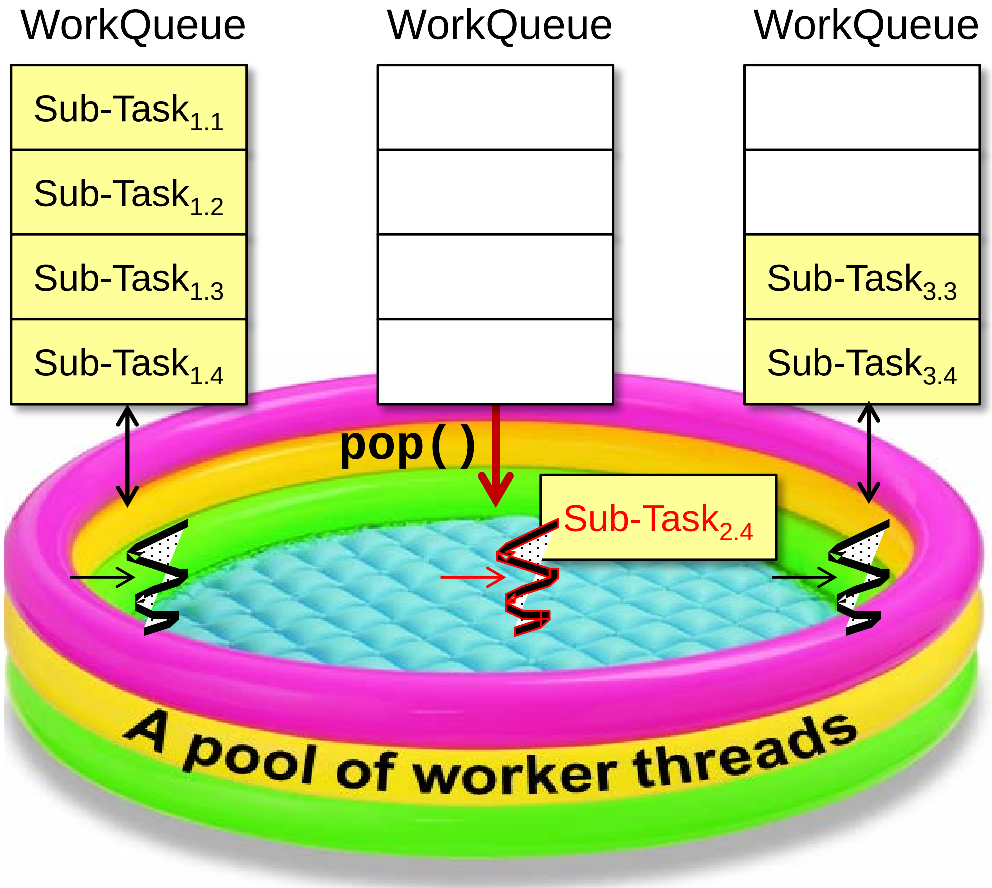
Overview of Java Fork-Join Framework Internals

- Sub-tasks fork()'d in a task run by a worker thread are pushed onto the head of that worker's own deque
  - A worker threads processes its own deque in LIFO order by popping (sub-)tasks from the from of its own deque



Overview of Java Fork-Join Framework Internals

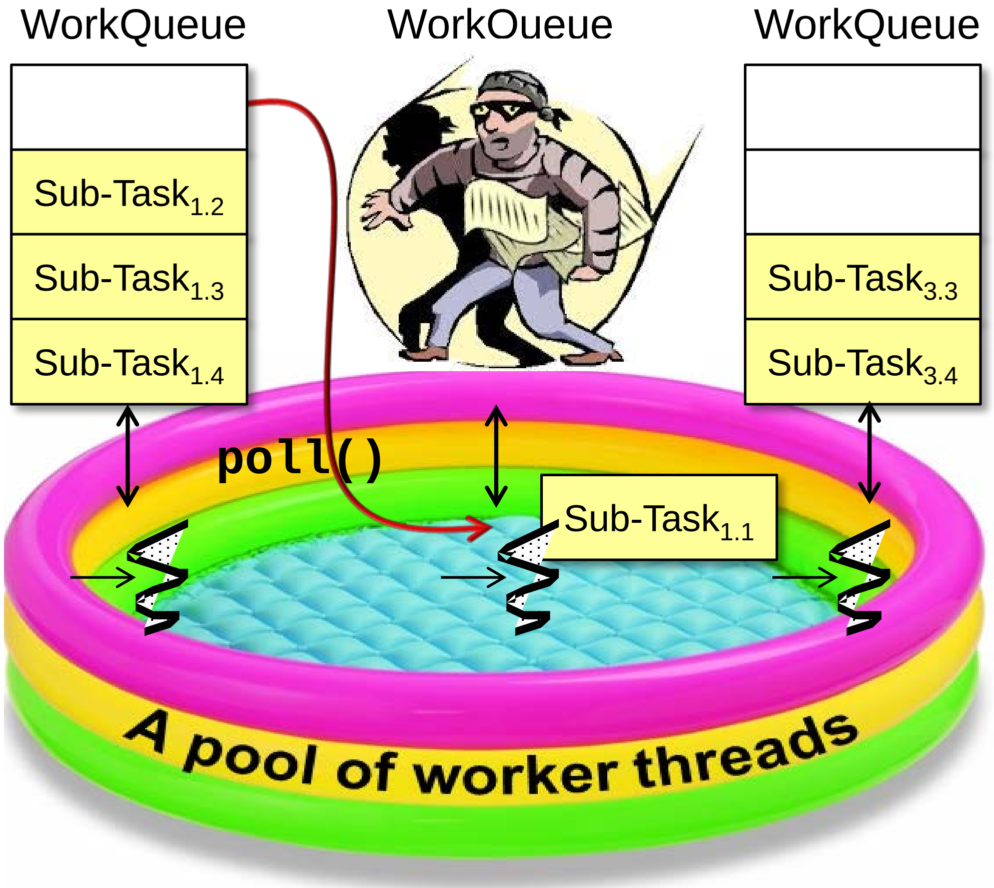
- Sub-tasks fork()'d in a task run by a worker thread are pushed onto the head of that worker's own deque
  - A worker threads processes its own deque in LIFO order by popping (sub-)tasks from the from of its own deque



"LIFO" pop/push enhances locality of reference & improves cache performance

Overview of Java Fork-Join Framework Internals

- To maximize core utilization, idle worker threads “steal” work from the tail of busy threads’ deques



WorkQueue

Sub-Task$_{1.2}$

Sub-Task$_{1.3}$

Sub-Task$_{1.4}$

WorkOueue

poll()

Sub-Task$_{1.1}$

WorkQueue

Sub-Task$_{3.3}$

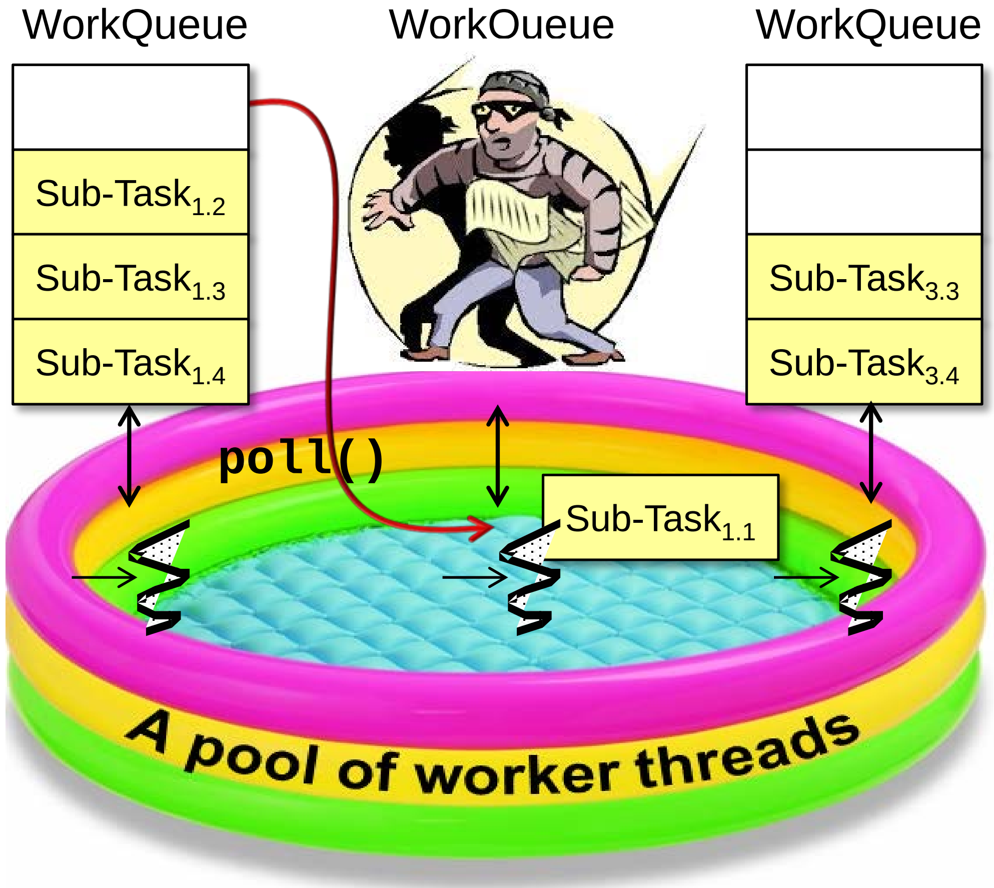
Sub-Task$_{3.4}$

A pool of worker threads

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

Overview of Java Fork-Join Framework Internals

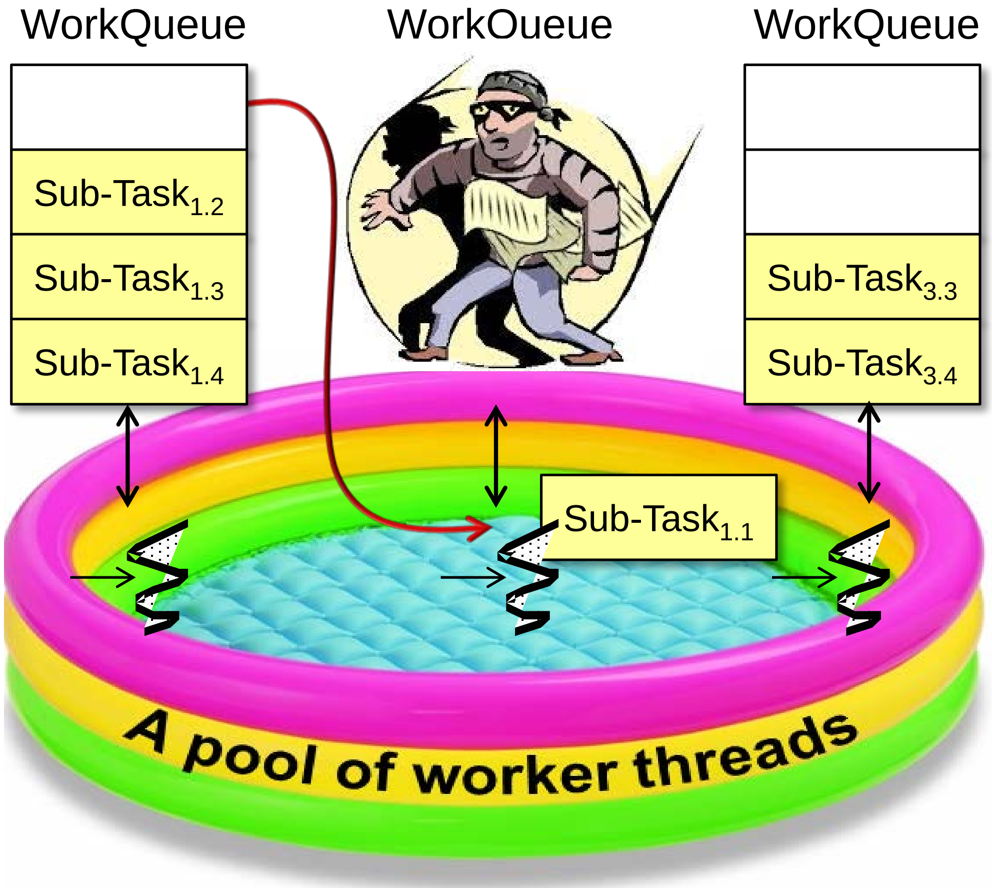
- To maximize core utilization, idle worker threads “steal” work from the tail of busy threads’ deques



Worker threads to steal from are selected randomly to lower contention

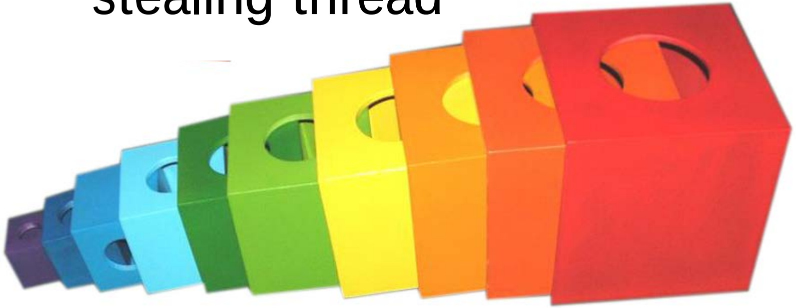
Overview of Java Fork-Join Framework Internals

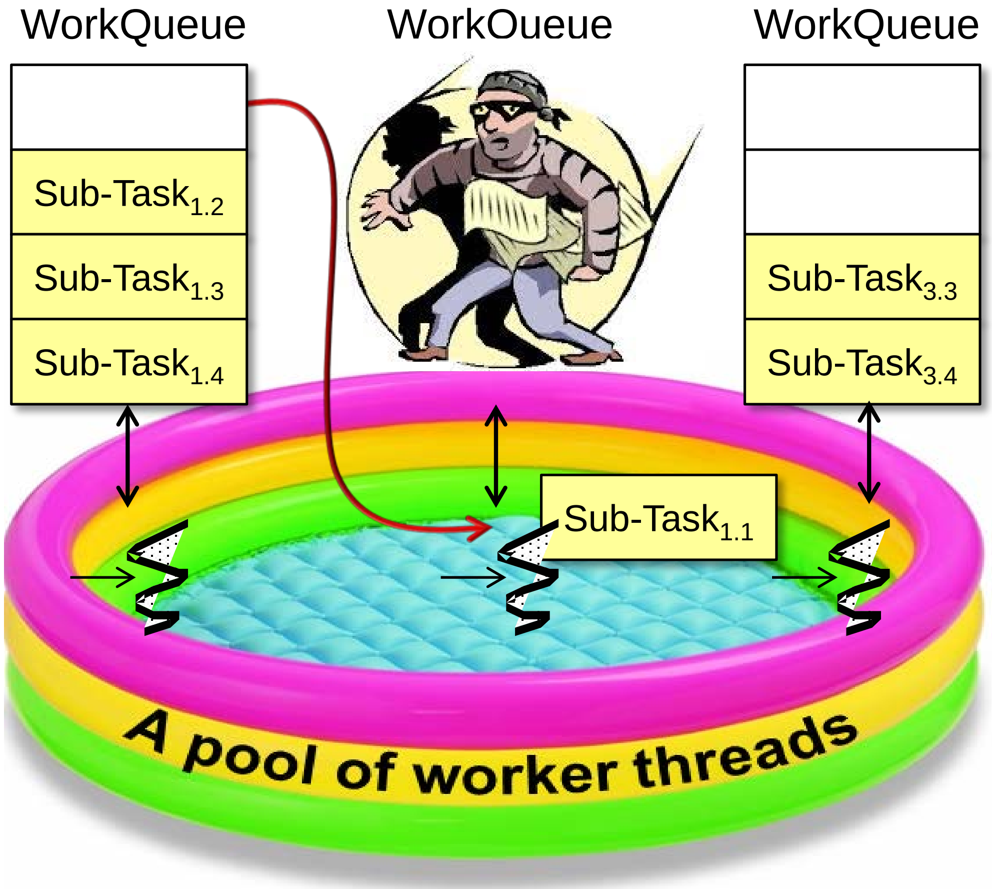
- To maximize core utilization, idle worker threads “steal” work from the tail of busy threads’ deques
  - Tasks are stolen in FIFO order since an older stolen task may provide a larger unit of work



Overview of Java Fork-Join Framework Internals

- To maximize core utilization, idle worker threads “steal” work from the tail of busy threads’ deques
  - Tasks are stolen in FIFO order since an older stolen task may provide a larger unit of work
    - Enables further recursive decompositions by the stealing thread





WorkQueue

Sub-Task$_{1.2}$

Sub-Task$_{1.3}$

Sub-Task$_{1.4}$

WorkOueue

Sub-Task$_{1.1}$

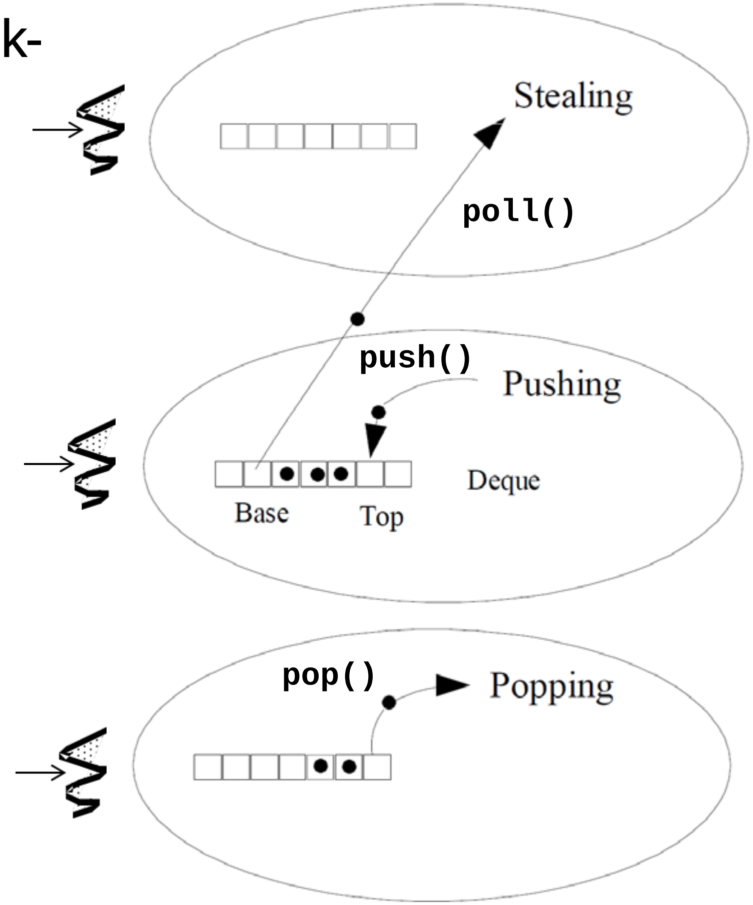
WorkQueue

Sub-Task$_{3.3}$

Sub-Task$_{3.4}$

Overview of Java Fork-Join Framework Internals

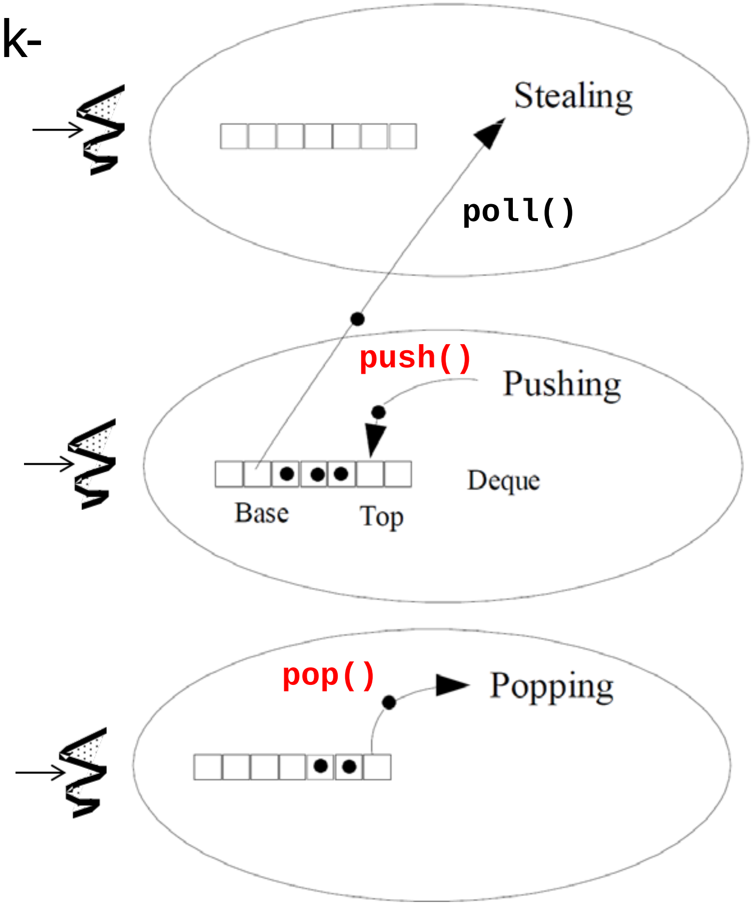
- The WorkQueue deque that implements work-stealing minimizes locking contention



See www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-deque.pdf

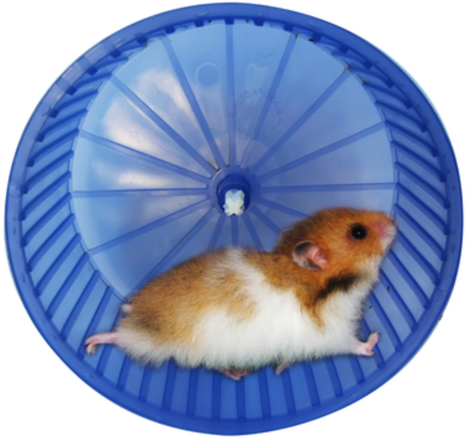
Overview of Java Fork-Join Framework Internals

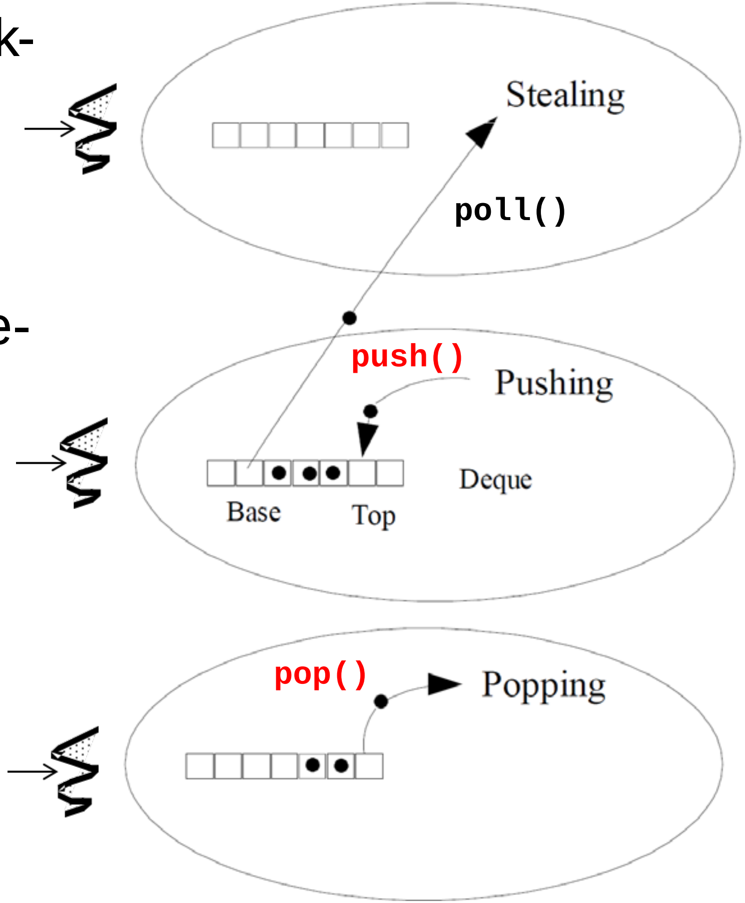
- The WorkQueue deque that implements work-stealing minimizes locking contention
  - push() & pop() are only called by the owning worker thread



Overview of Java Fork-Join Framework Internals

- The WorkQueue deque that implements work-stealing minimizes locking contention
    - push() & pop() are only called by the owning worker thread
        - These operations use wait-free "compare-and-swap" (CAS) operations

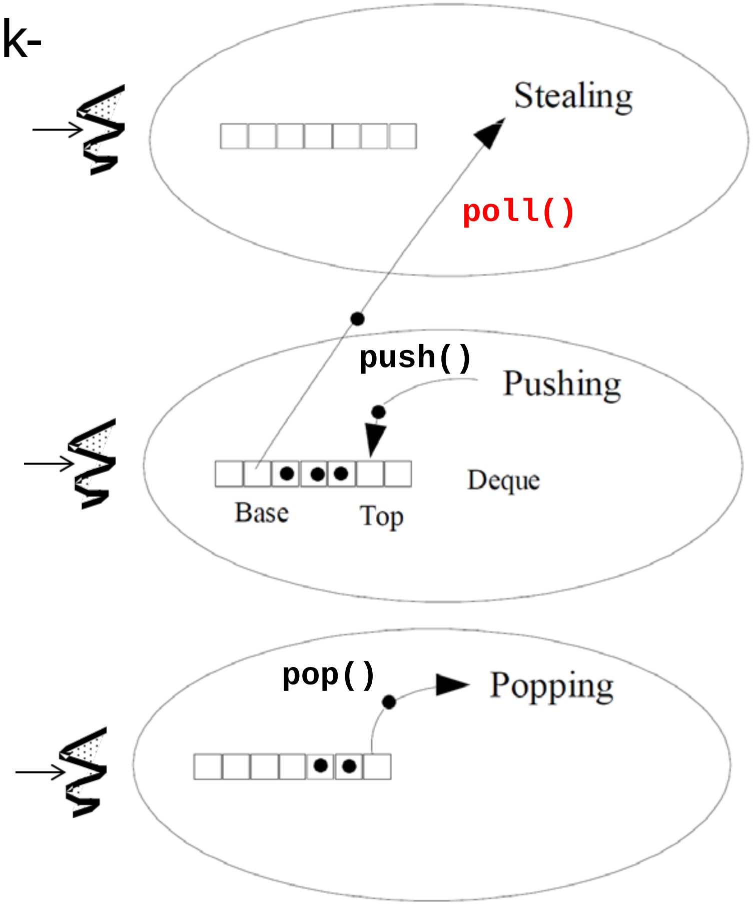




See en.wikipedia.org/wiki/Compare-and-swap

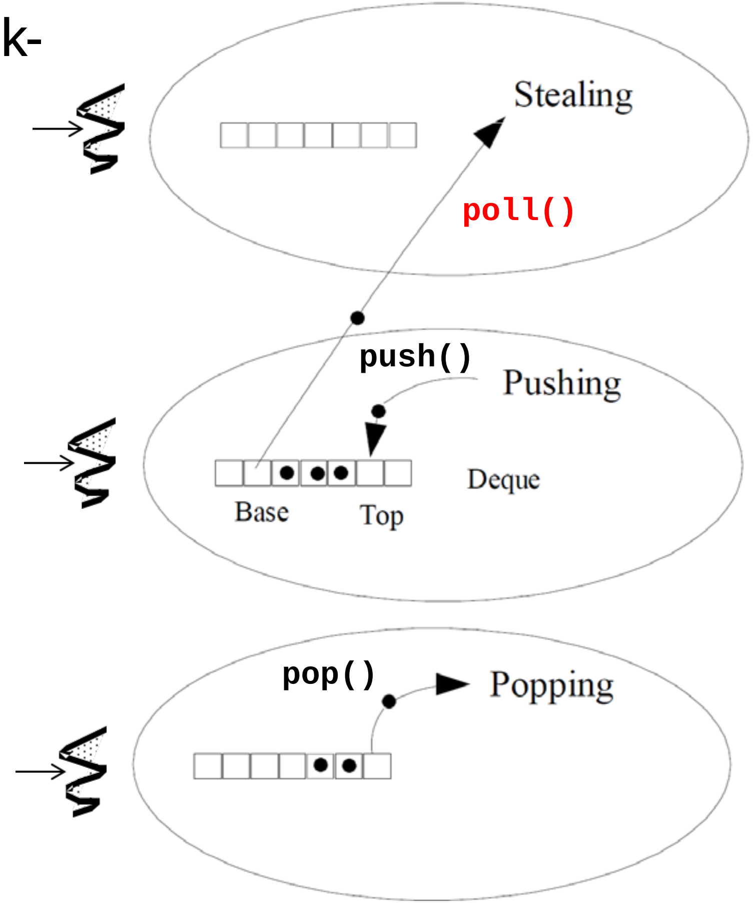
Overview of Java Fork-Join Framework Internals

- The WorkQueue deque that implements work-stealing minimizes locking contention
  - push() & pop() are only called by the owning worker thread
  - poll() may be called from another worker thread to "steal" a (sub-)task



Overview of Java Fork-Join Framework Internals

- The WorkQueue deque that implements work-stealing minimizes locking contention
  - push() & pop() are only called by the owning worker thread
  - poll() may be called from another worker thread to “steal” a (sub-)task
    - May not always be wait-free



See ForkJoinPool “Implementation Overview” comments for details..

End of the Java Fork-Join Pool Framework (Part 1)