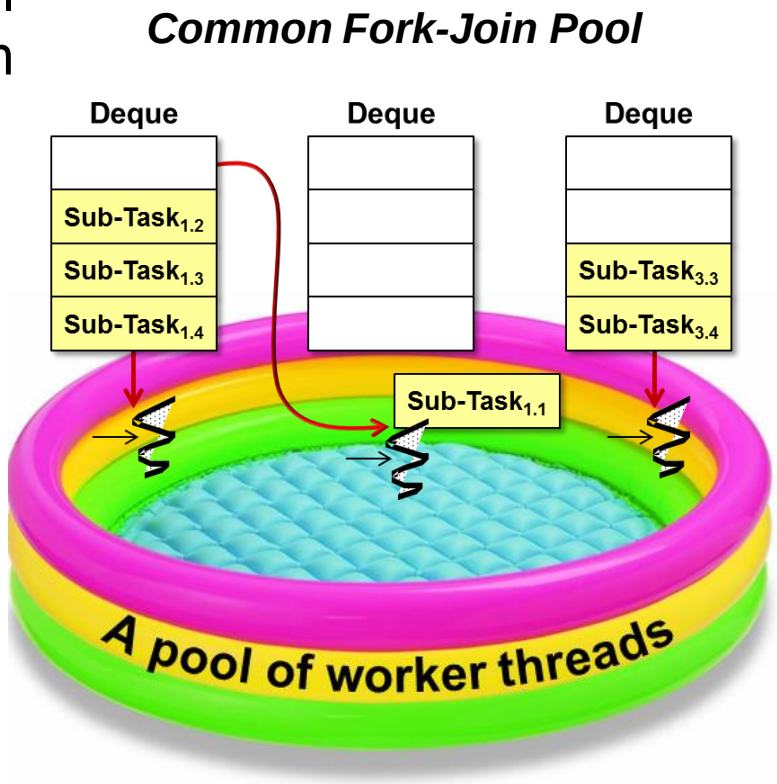
The Java Fork-Join Pool

Maximizing Core Utilization With the Common Fork-Join Pool

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand how the common fork-join pool helps to maximize processor core utilization



Overview of the Common Fork-Join Pool

Overview of the Common Fork-Join Pool

- A static common pool is available and appropriate for most programs.

commonPool

```
public static ForkJoinPool commonPool()
```

Returns the common pool instance. This pool is statically constructed; its run state is unaffected by attempts to `shutdown()` or `shutdownNow()`. However this pool and any ongoing processing are automatically terminated upon program `System.exit(int)`. Any program that relies on asynchronous task processing to complete before program termination should invoke `commonPool().awaitQuiescence`, before exit.

Returns:

the common pool instance

Since:

1.8

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

Overview of the Common Fork-Join Pool

- A static common pool is available and appropriate for most programs.
  - This pool is used by any ForkJoin Task that's not submitted to a specified pool within a process.

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

Overview of the Common Fork-Join Pool

- A static common pool is available and appropriate for most programs.
  - This pool is used by any ForkJoin Task that's not submitted to a specified pool within a process.
  - It helps optimize resource utilization since it's aware of which cores are used globally within a process.

Overview of the Common Fork-Join Pool

- A static common pool is available and appropriate for most programs.
  - This pool is used by any ForkJoin Task that's not submitted to a specified pool within a process.
  - It helps optimize resource utilization since it's aware of which cores are used globally within a process.
    - Goal is to maximize processor core utilization via work stealing.

See earlier lessons on "*The Java Fork-Join Pool Internals: Work Stealing*"

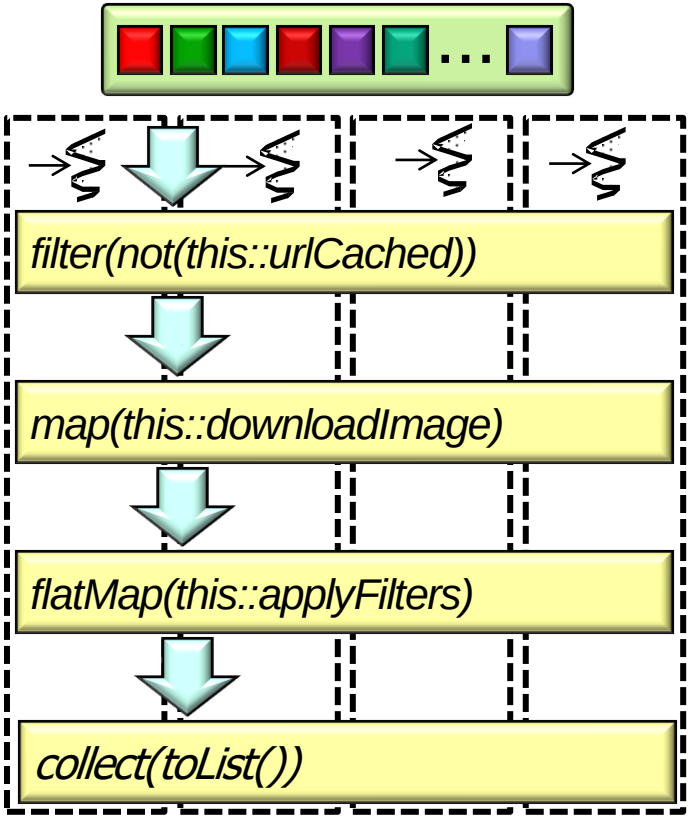
Overview of the Common Fork-Join Pool

- A static common pool is available and appropriate for most programs.
  - This pool is used by any ForkJoin Task that's not submitted to a specified pool within a process.
  - It helps optimize resource utilization since it's aware of which cores are used globally within a process.
    - Goal is to maximize processor core utilization via work stealing.
    - This "global" vs. "local" resource management trade-off is common in computing and other domains.

See blog.tsia.com/blog/local-or-global-resource-management-which-model-is-better

Overview of the Common Fork-Join Pool

- A static common pool is available and appropriate for most programs.
  - This pool is used by any ForkJoin Task that's not submitted to a specified pool within a process.
  - It helps optimize resource utilization since it's aware of which cores are used globally within a process.
  - This pool is also used by the Java parallel streams framework.



See dzone.com/articles/common-fork-join-pool-and-streams

Overview of the Common Fork-Join Pool

- By default, the common ForkJoinPool has one less thread than the number of cores.

```
ForkJoinPool makeCommonPool() {
  ...
  parallelism = Runtime
    .getRuntime()
    .availableProcessors() - 1;
  ...
```

Sets 'parallelism' to three on a quad-core processor.

See docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#availableProcessors

Overview of the Common Fork-Join Pool

- By default, the common ForkJoinPool has one less thread than the number of cores.

```
ForkJoinPool makeCommonPool() {
  ...
  parallelism = Runtime
    .getRuntime()
    .availableProcessors() - 1;
  ...
```

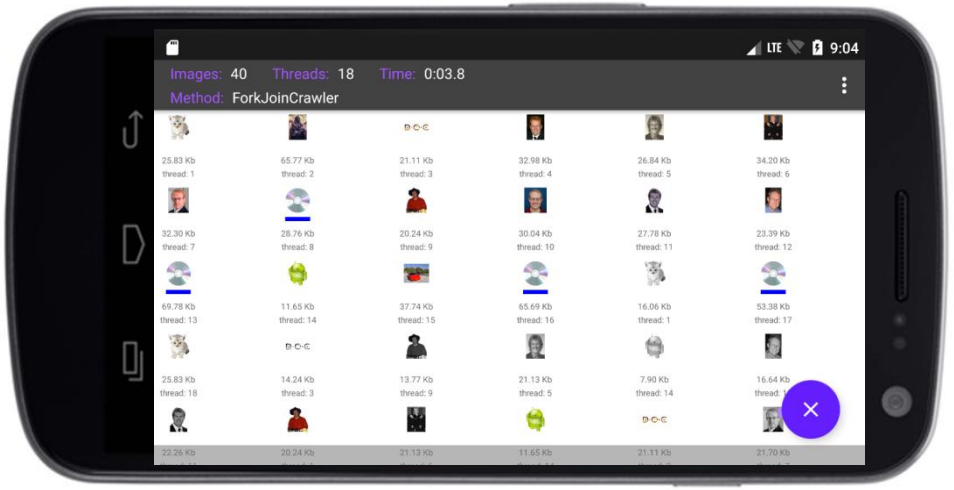
Returns three on a quad-core processor

```
System.out.println
   ("The parallelism in the"
    + "common fork-join pool is "
    + ForkJoinPool
        .getCommonPoolParallelism());
```

See github.com/douglascraigschmidt/LiveLessons/blob/master/SearchForkJoin

Overview of the Common Fork-Join Pool

- By default, the common ForkJoinPool has one less thread than the number of cores.

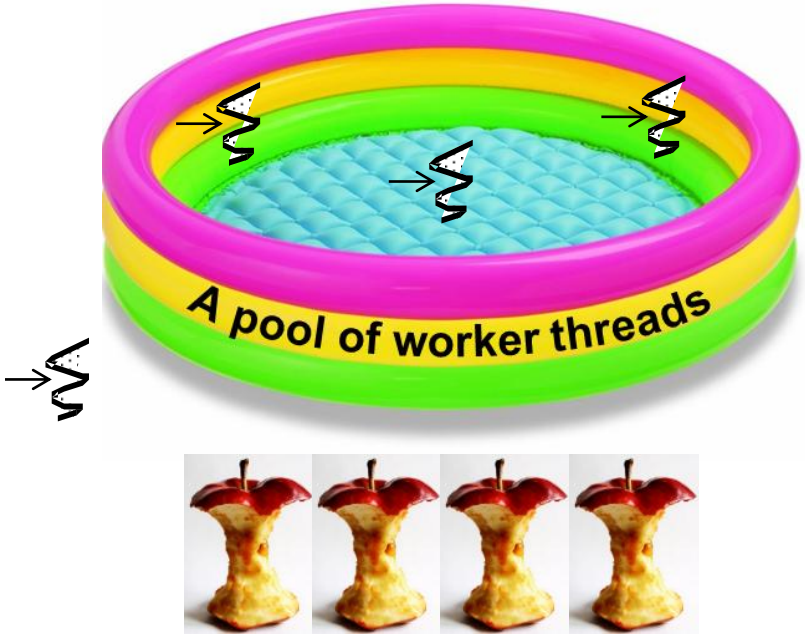


Invoking thread; e.g., the main (UI) thread

A program can leverage all cores since the invoking thread is also included.

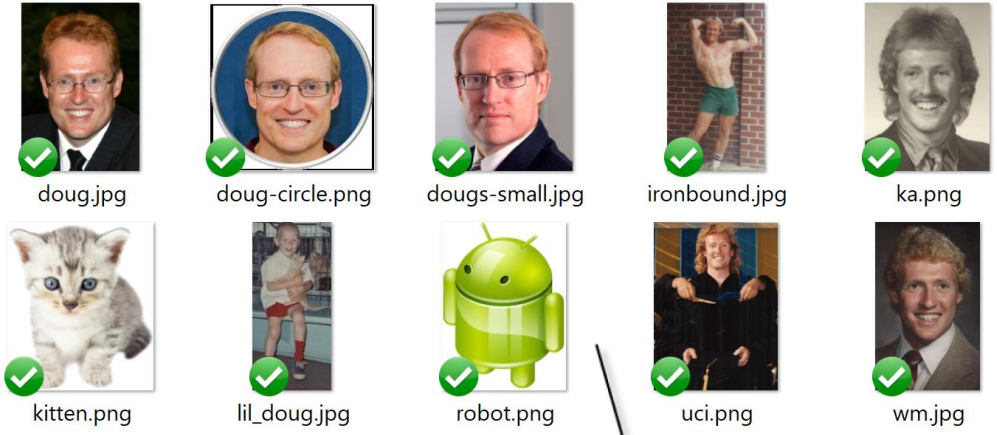
Overview of the Common Fork-Join Pool

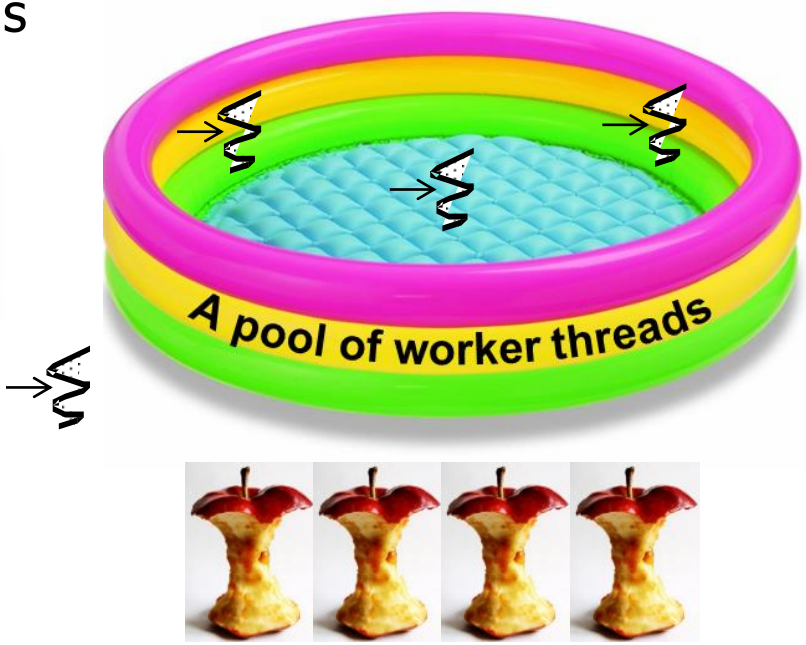
- The default number of threads in the fork-join pool may be inadequate.



Overview of the Common Fork-Join Pool

- The default number of threads in the fork-join pool may be inadequate.
  - Problems occur when blocking operations are used in the common fork-join pool.



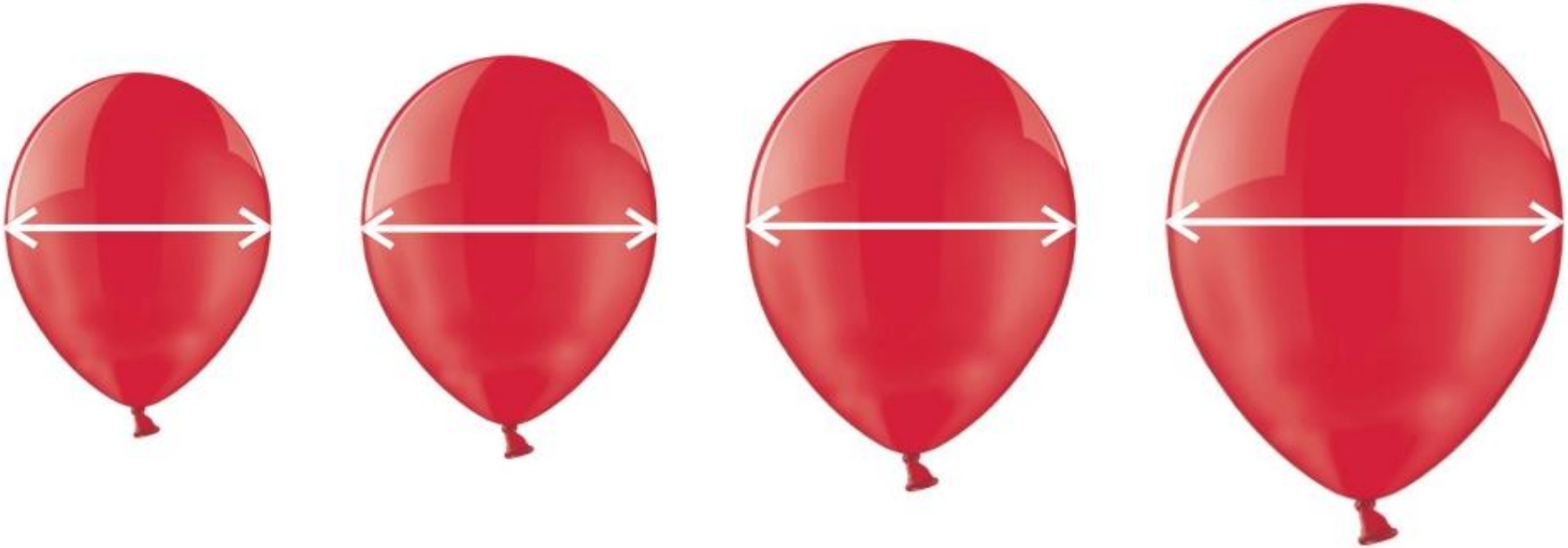


Downloading more images than number of cores

These problems may range from underutilization of processor cores to deadlock.

Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.



Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property

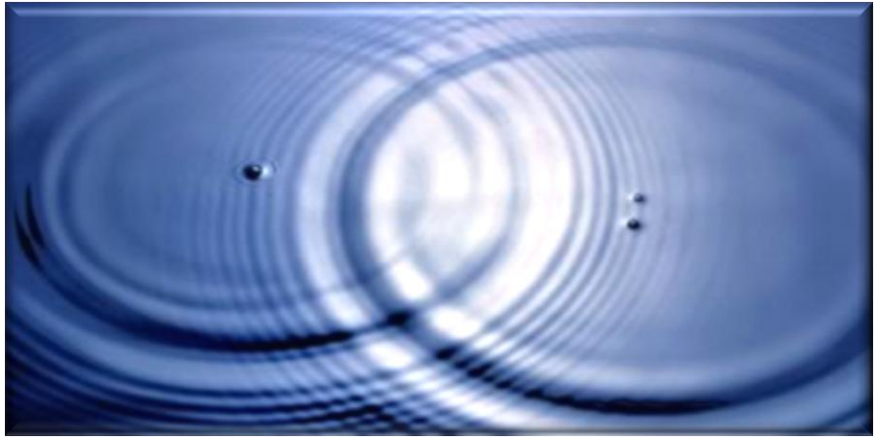
```
int desiredThreads = "10";
System.setProperty
   ("java.util.concurrent." +
    "ForkJoinPool.common." +
    "parallelism",
    desiredThreads);
```

It's hard to estimate total number of threads to set in common fork-join pool.

Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property
    - Modifying this property affects all common fork-join usage in a process!

```
int desiredThreads = "10";
System.setProperty
   ("java.util.concurrent." +
    "ForkJoinPool.common." +
    "parallelism",
    desiredThreads);
```



Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property
    - Modifying this property affects all common fork-join usage in a process!
    - This property can be changed only before the common fork-join pool is initialized
      - It's initialized "on-demand" the first time it's used

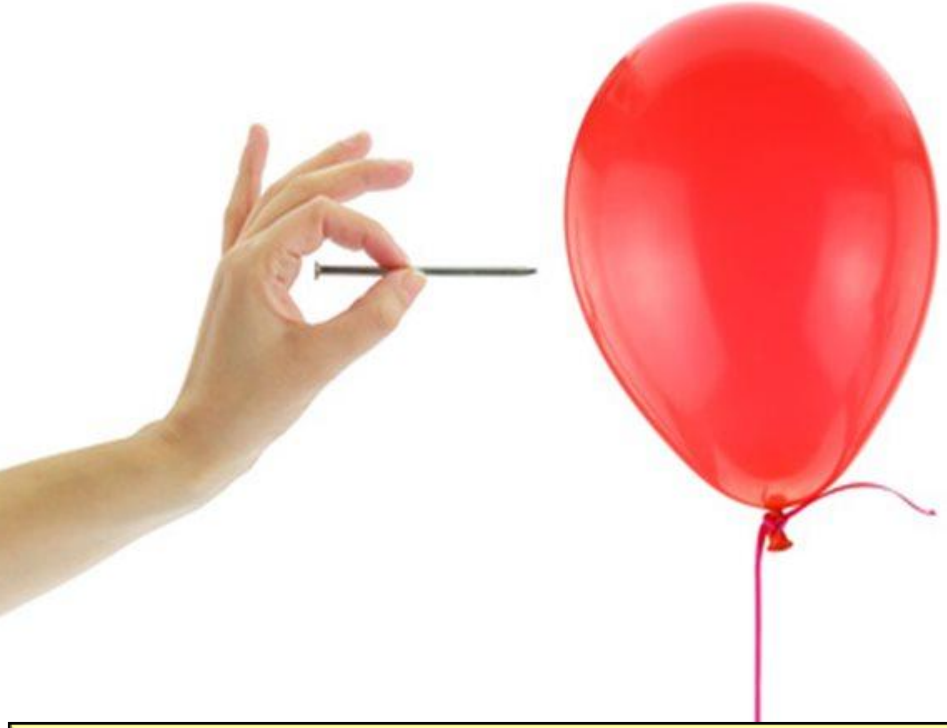
```
int desiredThreads = "10";
System.setProperty
   ("java.util.concurrent." +
    "ForkJoinPool.common." +
    "parallelism",
    desiredThreads);
```

See en.wikipedia.org/wiki/Lazy_initialization

Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property

```
int desiredThreads = "10";
System.setProperty
   ("java.util.concurrent." +
    "ForkJoinPool.common." +
    "parallelism",
    desiredThreads);
```



Another approach is thus needed to increase the fork/join pool size automatically

Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property
  - By using a ManagedBlocker

Interface ForkJoinPool.ManagedBlocker

Enclosing class:
`ForkJoinPool`

`public static interface ForkJoinPool.ManagedBlocker`

Interface for extending managed parallelism for tasks running in `ForkJoinPools`.

A `ManagedBlocker` provides two methods. Method `isReleasable()` must return `true` if blocking is not necessary. Method `block()` blocks the current thread if necessary (perhaps internally invoking `isReleasable` before actually blocking). These actions are performed by any thread invoking `ForkJoinPool.managedBlock(ManagedBlocker)`. The unusual methods in this API accommodate synchronizers that may, but don't usually, block for long periods. Similarly, they allow more efficient internal handling of cases in which additional workers may be, but usually are not, needed to ensure sufficient parallelism. Toward this end, implementations of method `isReleasable` must be amenable to repeated invocation.

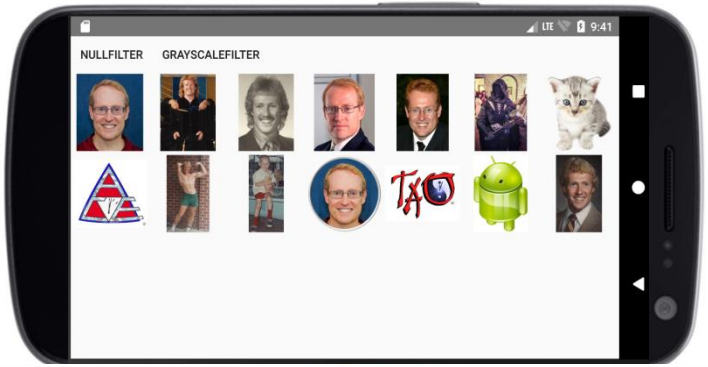
See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.ManagedBlocker.html

Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property
  - By using a ManagedBlocker
    - Temporarily add worker threads to the common fork-join pool

Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property
  - By using a ManagedBlocker
    - Temporarily add worker threads to the common fork-join pool
    - Useful when tasks block on I/O and/or synchronizers



ManageBlockers can only be used with the common fork-join pool.

Overview of the Common Fork-Join Pool

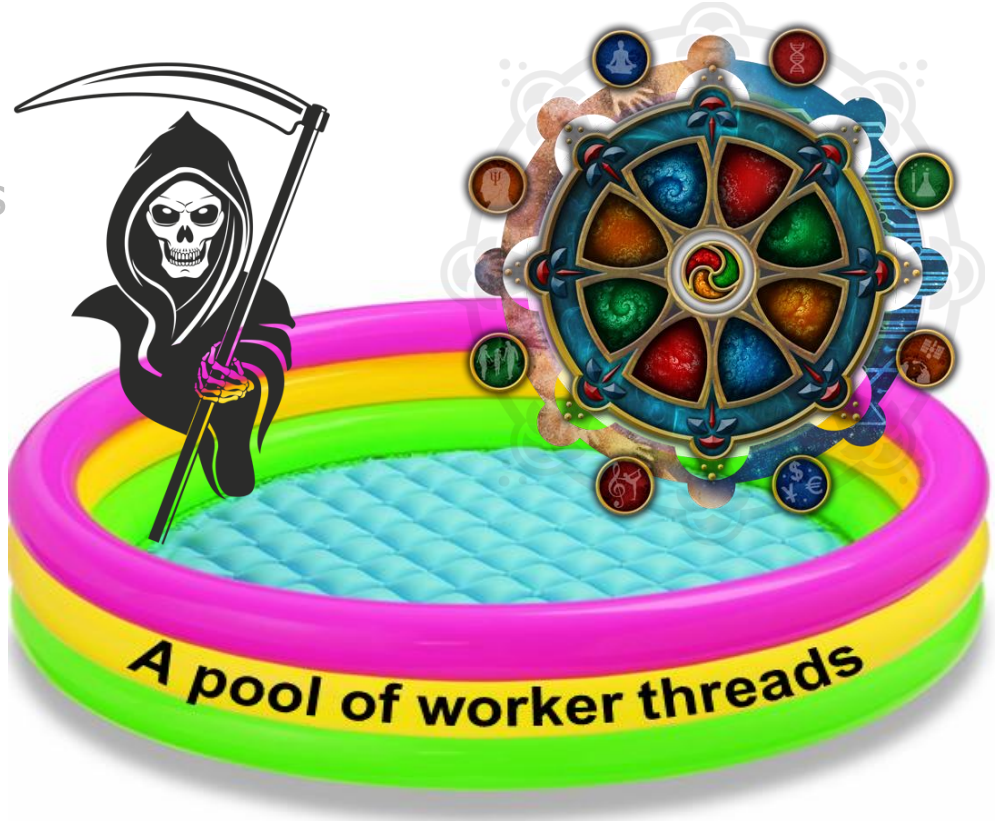
- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property
  - By using a ManagedBlocker
    - Temporarily add worker threads to the common fork-join pool
    - Useful when tasks block on I/O and/or synchronizers
    - It's helpful to encapsulate the ManagedBlocker mechanism

```
SupplierManagedBlocker<T> mb =
  new SupplierManagedBlocker<>
   (supplier);
...
ForkJoinPool.managedBlock(mb);
return mb.getResult();
```

See lesson on "*The Java Fork-Join Pool: Applying the ManagedBlocker Interface*"

Overview of the Common Fork-Join Pool

- The common pool size can be expanded and contracted programmatically.
  - By modifying a system property
  - By using a ManagedBlocker
    - Temporarily add worker threads to the common fork-join pool
    - Useful when tasks block on I/O and/or synchronizers
    - It's helpful to encapsulate the ManagedBlocker mechanism
    - ForkJoinPool reclaims threads during periods of non-use and reinstates them on later use



Java Fork-Join Pool Framework: Maximizing Core Utilization With the Common Fork-Join Pool

The End

The Java Fork-Join Pool

The ManagedBlocker Interface

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand how the common fork-join pool helps to maximize processor core utilization
- Recognize how the ManagedBlocker interface helps avoid starvation and improves performance

Interface ForkJoinPool.ManagedBlocker

Enclosing class:

ForkJoinPool

`public static interface` **`ForkJoinPool.ManagedBlocker`**

Interface for extending managed parallelism for tasks running in `ForkJoinPool`s.

A `ManagedBlocker` provides two methods. Method `isReleasable()` must return `true` if blocking is not necessary. Method `block()` blocks the current thread if necessary (perhaps internally invoking `isReleasable` before actually blocking). These actions are performed by any thread invoking `ForkJoinPool.managedBlock(ManagedBlocker)`. The unusual methods in this API accommodate synchronizers that may, but don't usually, block for long periods. Similarly, they allow more efficient internal handling of cases in which additional workers may be, but usually are not, needed to ensure sufficient parallelism. Toward this end, implementations of method `isReleasable` must be amenable to repeated invocation.

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

Overview of the ManagedBlocker Interface

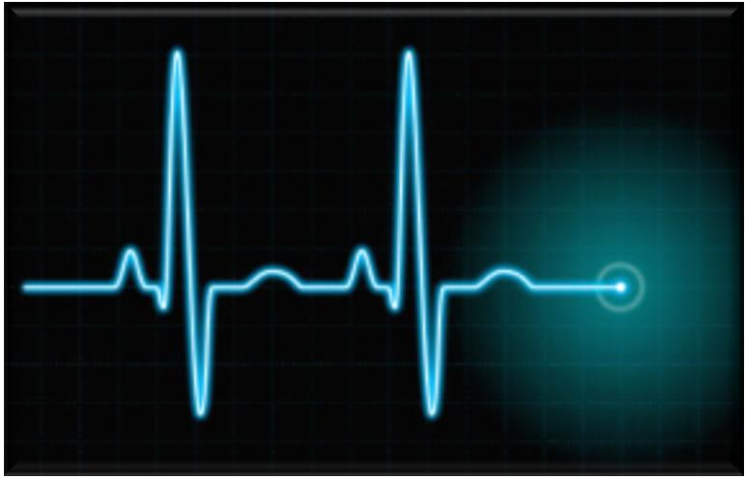
Overview of the ManagedBlocker Interface

- The Java fork-join framework is largely designed for tasks that "run to completion" without blocking.

See en.wikipedia.org/wiki/Run_to_completion_scheduling

Overview of the ManagedBlocker Interface

- ManagedBlocker handles cases where more worker threads may be needed to ensure liveness/responsiveness for blocking operations.



Interface ForkJoinPool.ManagedBlocker

Enclosing class:

`ForkJoinPool`

`public static interface ForkJoinPool.ManagedBlocker`

Interface for extending managed parallelism for tasks running in `ForkJoinPools`.

A `ManagedBlocker` provides two methods. Method `isReleasable()` must return `true` if blocking is not necessary. Method `block()` blocks the current thread if necessary (perhaps internally invoking `isReleasable` before actually blocking). These actions are performed by any thread invoking `ForkJoinPool.managedBlock(ManagedBlocker)`. The unusual methods in this API accommodate synchronizers that may, but don't usually, block for long periods. Similarly, they allow more efficient internal handling of cases in which additional workers may be, but usually are not, needed to ensure sufficient parallelism. Toward this end, implementations of method `isReleasable` must be amenable to repeated invocation.

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

Overview of the ManagedBlocker Interface

- ManagedBlocker handles cases where more worker threads may be needed to ensure liveness/responsiveness for blocking operations.
  - To automatically/temporarily increase common fork/join pool size

Interface ForkJoinPool.ManagedBlocker

Enclosing class:
`ForkJoinPool`

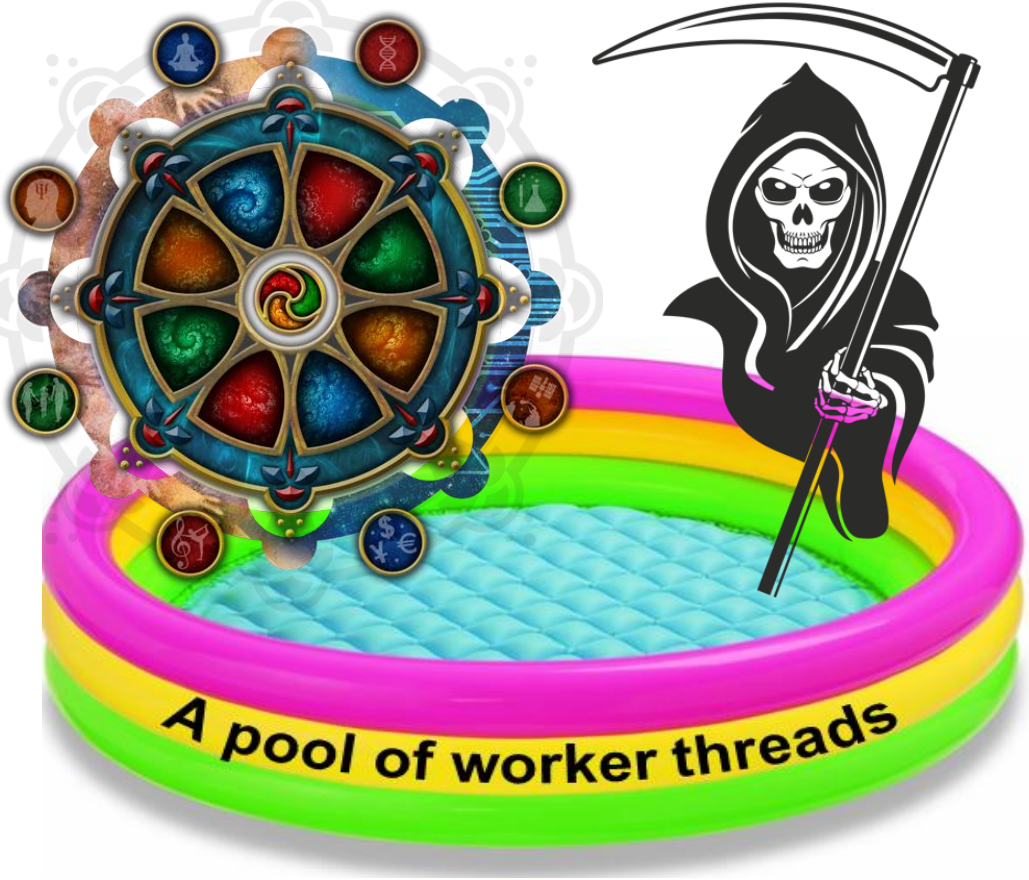
`public static interface ForkJoinPool.ManagedBlocker`

Interface for extending managed parallelism for tasks running in `ForkJoinPools`.

A `ManagedBlocker` provides two methods. Method `isReleasable()` must return `true` if blocking is not necessary. Method `block()` blocks the current thread if necessary (perhaps internally invoking `isReleasable` before actually blocking). These actions are performed by any thread invoking `ForkJoinPool.managedBlock(ManagedBlocker)`. The unusual methods in this API accommodate synchronizers that may, but don't usually, block for long periods. Similarly, they allow more efficient internal handling of cases in which additional workers may be, but usually are not, needed to ensure sufficient parallelism. Toward this end, implementations of method `isReleasable` must be amenable to repeated invocation.

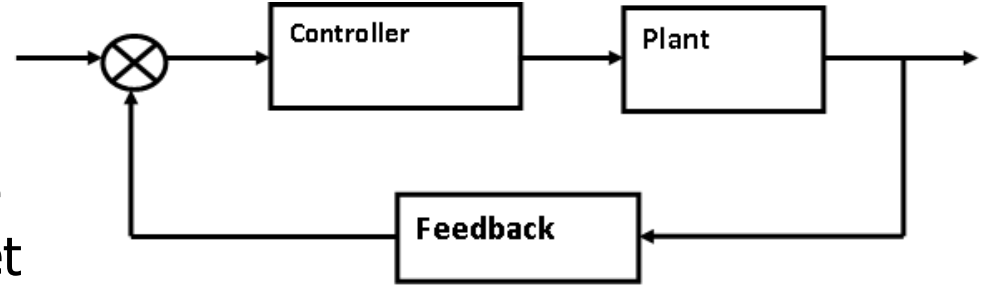
Overview of the ManagedBlocker Interface

- ForkJoinPool reclaims threads during periods of non-use and reinstates them on later use.



Overview of the ManagedBlocker Interface

- ForkJoinPool reclaims threads during periods of non-use and reinstates them on later use.
  - It also tries to create or activate threads to ensure that the target level of parallelism is met.



Overview of the ManagedBlocker Interface

- ManagedBlocker defines two methods.

```
interface ManagedBlocker {
  boolean isReleasable();

  boolean block();
}
```

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

Overview of the ManagedBlocker Interface

- ManagedBlocker defines two methods.
  - Returns true if blocking is unnecessary

```
interface ManagedBlocker {
  boolean isReleasable();

  boolean block();
}
```

Was able to acquire a lock or a message without blocking

Overview of the ManagedBlocker Interface

- ManagedBlocker defines two methods.
  - Returns true if blocking is unnecessary
  - Possibly blocks the calling thread

```
interface ManagedBlocker {
  boolean isReleasable();

  boolean block();
}
```

Waiting for a lock or I/O operation

Overview of the ManagedBlocker Interface

- ManagedBlocker defines two methods.
  - Returns true if blocking is unnecessary
  - Possibly blocks the calling thread
    - Returns true if no additional blocking is necessary

```
interface ManagedBlocker {
  boolean isReleasable();

  boolean block();
}
```

If isReleasable() would return true

How the Java Fork-Join Pool Applies ManagedBlocker

How the Java Fork-Join Pool Applies ManagedBlocker

- The ForkJoinPool class uses a ManagedBlocker internally.

```
class ForkJoinPool extends AbstractExecutorService {
  ...
  static void managedBlock(ManagedBlocker blocker) {
    ...
    while (!blocker.isReleasable()) {
      if (p.tryCompensate(p.ctl)) {
        ...
        do {}
        while (!blocker.isReleasable()
               && !blocker.block());
        ...
  }
  ...
```

See openjdk/7-b147/java/util/concurrent/ForkJoinPool.java

How the Java Fork-Join Pool Applies ManagedBlocker

- The ForkJoinPool class uses a ManagedBlocker internally.

```
class ForkJoinPool extends AbstractExecutorService {
  ...
  static void managedBlock(ManagedBlocker blocker) {
    ...
    while (!blocker.isReleasable()) {
      if (p.tryCompensate(p.ctl)) {
        ...
        do {}
        while (!blocker.isReleasable()
               && !blocker.block());
        ...
  }
  ...
```

Implements the ExecutorService interface

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

How the Java Fork-Join Pool Applies ManagedBlocker

- The ForkJoinPool class uses a ManagedBlocker internally.

```
class ForkJoinPool extends AbstractExecutorService {
  ...
  static void managedBlock(ManagedBlocker blocker) {
    ...
    while (!blocker.isReleasable()) {
      if (p.tryCompensate(p.ctl)) {
        ...
        do {}
        while (!blocker.isReleasable()
               && !blocker.block());
        ...
  }
  ...
```

This method activates a spare thread to ensure sufficient parallelism while the calling thread is blocked.

See [openjdk/7-b147/java/util/concurrent/ForkJoinPool.java]

How the Java Fork-Join Pool Applies ManagedBlocker

- The ForkJoinPool class uses a ManagedBlocker internally.

```
class ForkJoinPool extends AbstractExecutorService {
  ...
  static void managedBlock(ManagedBlocker blocker) {
    ...
    while (!blocker.isReleasable()) {
      if (p.tryCompensate(p.ctl)) {
        ...
        do {}
        while (!blocker.isReleasable()
               && !blocker.block());
        ...
  }
  ...
```

If there aren't enough live threads, create a spare thread to compensate for blocked joiners until they unblock.

How the Java Fork-Join Pool Applies ManagedBlocker

- The ForkJoinPool class uses a ManagedBlocker internally.

```
class ForkJoinPool extends AbstractExecutorService {
  ...
  static void managedBlock(ManagedBlocker blocker) {
    ...
    while (!blocker.isReleasable()) {
      if (p.tryCompensate(p.ctl)) {
        ...
        do {}
        while (!blocker.isReleasable()
               && !blocker.block());
        ...
  }
  ...
```

First attempt to acquire the resource without blocking

How the Java Fork-Join Pool Applies ManagedBlocker

- The ForkJoinPool class uses a ManagedBlocker internally.

```
class ForkJoinPool extends AbstractExecutorService {
  ...
  static void managedBlock(ManagedBlocker blocker) {
    ...
    while (!blocker.isReleasable()) {
      if (p.tryCompensate(p.ctl)) {
        ...
        do {}
        while (!blocker.isReleasable()
               && !blocker.block());
        ...
  }
  ...
```

May block the calling thread

The End

The Java Fork-Join Pool

Applying the ManagedBlocker Interface (Part I)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand how the common fork-join pool helps to maximize processor core utilization
- Recognize how the ManagedBlocker interface helps avoid starvation and improves performance
- Be able to apply the ManagedBlocker interface on blocking synchronizers and queues

```
class ManagedLocker implements
      ManagedBlocker {
  final ReentrantLock mLock;
  boolean mHasLock = false;

  ManagedLocker(ReentrantLock
                lock)
  { mLock = lock; }

  public boolean block() {
    if (!mHasLock)
      mLock.lock();
    return true;
  }
  ...
```

Applying the ManagedBlocker Interface

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a ReentrantLock (from Java docs).

```
class ManagedLocker implements ManagedBlocker {
  final ReentrantLock mLock;
  boolean mHasLock = false;

  ManagedLocker(ReentrantLock lock) { mLock = lock; }

  public boolean isReleasable()
  { return mHasLock || (mHasLock = mLock.tryLock()); }

  public boolean block() {
    if (!mHasLock) mLock.lock();
    return true;
  }
}
```

Handles a blocking synchronizer

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

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a ReentrantLock (from Java docs).

```
class ManagedLocker implements ManagedBlocker {
  final ReentrantLock mLock;
  boolean mHasLock = false;

  ManagedLocker(ReentrantLock lock) { mLock = lock; }

  public boolean isReleasable()
  { return mHasLock || (mHasLock = mLock.tryLock()); }

  public boolean block() {
    if (!mHasLock) mLock.lock();
    return true;
  }
}
```

Constructor stores the lock

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a ReentrantLock (from Java docs).

```
class ManagedLocker implements ManagedBlocker {
  final ReentrantLock mLock;
  boolean mHasLock = false;

  ManagedLocker(ReentrantLock lock) { mLock = lock; }

  public boolean isReleasable()
  { return mHasLock || (mHasLock = mLock.tryLock()); }

  public boolean block() {
    if (!mHasLock) mLock.lock();
    return true;
  }
}
```

Tries to acquire the lock (non-blocking)

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a ReentrantLock (from Java docs).

```
class ManagedLocker implements ManagedBlocker {
  final ReentrantLock mLock;
  boolean mHasLock = false;

  ManagedLocker(ReentrantLock lock) { mLock = lock; }

  public boolean isReleasable()
  { return mHasLock || (mHasLock = mLock.tryLock()); }

  public boolean block() {
    if (!mHasLock) mLock.lock();
    return true;
  }
}
```

Performs a blocking lock operation

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a BlockingQueue (from Java docs).

```
class QueueTaker<E> implements ManagedBlocker {
  final BlockingQueue<E> mQueue;
  volatile E mItem = null;

  QueueTaker(BlockingQueue<E> q) { mQueue = q; }

  public boolean isReleasable()
  { return mItem != null || (mItem = mQueue.poll()) != null; }

  public boolean block() throws InterruptedException
  { if (mItem == null) mItem = mQueue.take(); return true; }

  public E getItem() { return mItem; }
}
```

Handles a blocking queue

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

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a BlockingQueue (from Java docs).

```
class QueueTaker<E> implements ManagedBlocker {
  final BlockingQueue<E> mQueue;
  volatile E mItem = null;

  QueueTaker(BlockingQueue<E> q) { mQueue = q; }

  public boolean isReleasable()
  { return mItem != null || (mItem = mQueue.poll()) != null; }

  public boolean block() throws InterruptedException
  { if (mItem == null) mItem = mQueue.take(); return true; }

  public E getItem() { return mItem; }
}
```

The blocking queue

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a BlockingQueue (from Java docs).

```
class QueueTaker<E> implements ManagedBlocker {
  final BlockingQueue<E> mQueue;
  volatile E mItem = null;

  QueueTaker(BlockingQueue<E> q) { mQueue = q; }

  public boolean isReleasable()
  { return mItem != null || (mItem = mQueue.poll()) != null; }

  public boolean block() throws InterruptedException
  { if (mItem == null) mItem = mQueue.take(); return true; }

  public E getItem() { return mItem; }
}
```

Try to get an item (non-blocking)

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a BlockingQueue (from Java docs).

```
class QueueTaker<E> implements ManagedBlocker {
  final BlockingQueue<E> mQueue;
  volatile E mItem = null;

  QueueTaker(BlockingQueue<E> q) { mQueue = q; }

  public boolean isReleasable()
  { return mItem != null || (mItem = mQueue.poll()) != null; }

  public boolean block() throws InterruptedException
  { if (mItem == null) mItem = mQueue.take(); return true; }

  public E getItem() { return mItem; }
}
```

Block until an item is available

Applying the ManagedBlocker Interface

- This example applies a ManagedBlocker on a BlockingQueue (from Java docs).

```
class QueueTaker<E> implements ManagedBlocker {
  final BlockingQueue<E> mQueue;
  volatile E mItem = null;

  QueueTaker(BlockingQueue<E> q) { mQueue = q; }

  public boolean isReleasable()
  { return mItem != null || (mItem = mQueue.poll()) != null; }

  public boolean block() throws InterruptedException
  { if (mItem == null) mItem = mQueue.take(); return true; }

  public E getItem() { return mItem; }
}
```

Call after pool.managedBlock() completes to get item

The End

The Java Fork-Join Pool

Applying the ManagedBlocker Interface (Part II)

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Understand how the common fork-join pool helps to maximize processor core utilization
- Recognize how the ManagedBlocker interface helps avoid starvation and improves performance
- Be able to apply the ManagedBlocker interface on blocking synchronizers and queues
- Know how to encapsulate ManageBlocker and apply it on blocking I/O operations

```
public class BlockingTask {
  ...
  public static<T> T
    callInManagedBlocker
      (Supplier<T> supplier){
    ...
    ForkJoinPool.managedBlock
     (managedBlocker);
    ...
    return managedBlocker
      .getResult();
  }
  ...
```

Encapsulating ManagedBlocker With the BlockingTask Class

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
  ...
  public static<T> T callInManagedBlock(Supplier<T> supplier){



    SupplierManagedBlocker<T> managedBlocker =
      new SupplierManagedBlocker<T>(supplier);
    ...
    ForkJoinPool.managedBlock(managedBlocker);
    ...
    return managedBlocker.getResult();
  }
  ...
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex20

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
  ...
  public static<T> T callInManagedBlock(Supplier<T> supplier){
```

Enables the use of blocking suppliers with the common Java fork-join thread pool

```
    SupplierManagedBlocker<T> managedBlocker =
      new SupplierManagedBlocker<T>(supplier);
    ...
    ForkJoinPool.managedBlock(managedBlocker);
    ...
    return managedBlocker.getResult();
  }
  ...
```

See stackoverflow.com/q/37512662 for pros and cons of this approach.

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
  ...
  public static<T> T callInManagedBlock(Supplier<T> supplier){

    SupplierManagedBlocker<T> managedBlocker =
      new SupplierManagedBlocker<T>(supplier);
    ...
    ForkJoinPool.managedBlock(managedBlocker);
    ...
    return managedBlocker.getResult();
  }
  ...
```

Create a helper object to encapsulate the supplier

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
  ...
  public static<T> T callInManagedBlock(Supplier<T> supplier){



    SupplierManagedBlocker<T> managedBlocker =
      new SupplierManagedBlocker<T>(supplier);
    ...
    ForkJoinPool.managedBlock(managedBlocker);
    ...
    return managedBlocker.getResult();
  }
  ...
```

Submit ManagedBlocker to common ForkJoin thread pool

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

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
  ...
  public static<T> T callInManagedBlock(Supplier<T> supplier){



    SupplierManagedBlocker<T> managedBlocker =
      new SupplierManagedBlocker<T>(supplier);
    ...
    ForkJoinPool.managedBlock(managedBlocker);
    ...
    return managedBlocker.getResult();
  }
  ...
```

Return the result of the blocking call

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
  ...
  private static class SupplierManagedBlocker<T>
                  implements ForkJoinPool.ManagedBlocker {
    private final Supplier<T> mSupplier;

    private boolean mDone = false;

    private T mResult;


    private SupplierManagedBlocker(final Supplier supplier)
    { mSupplier = supplier; }
    ...
```

Blocking Supplier work with common fork/join pool

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
   ...
   private static class SupplierManagedBlocker<T>
                  implements ForkJoinPool.ManagedBlocker {
     private final Supplier<T> mSupplier;

     private boolean mDone = false;

     private T mResult;


     private SupplierManagedBlocker(final Supplier supplier)
     { mSupplier = supplier; }
     ...
```

Store supplier param for subsequent use

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
  ...
  private static class SupplierManagedBlocker<T>
                 implements ForkJoinPool.ManagedBlocker {
    private final Supplier<T> mSupplier;

    private boolean mDone = false;

    private T mResult;


    private SupplierManagedBlocker(final Supplier supplier)
    { mSupplier = supplier; }
    ...
```

Keeps track of whether blocking supplier is done

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
  ...
  private static class SupplierManagedBlocker<T>
                 implements ForkJoinPool.ManagedBlocker {
    private final Supplier<T> mSupplier;

    private boolean mDone = false;

    private T mResult;

    private SupplierManagedBlocker(final Supplier supplier)
    { mSupplier = supplier; }
    ...
```

Stores result obtained from the supplier for later use

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
   ...
   private static class SupplierManagedBlocker<T>
                  implements ForkJoinPool.ManagedBlocker {
      ...
      public boolean block()
      { mResult = mSupplier.get(); mDone = true; return true; }

      public boolean isReleasable()
      { return mDone; }

      public T getResult()
      { return mResult; }
}
```

Sets result via the blocking supplier's get() method

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
   ...
   private static class SupplierManagedBlocker<T>
                  implements ForkJoinPool.ManagedBlocker {
      ...
      public boolean block()
      { mResult = mSupplier.get(); mDone = true; return true; }

      public boolean isReleasable()
      { return mDone; }

      public T getResult()
      { return mResult; }
}
```

Indicates the result has been obtained

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
   ...
   private static class SupplierManagedBlocker<T>
                  implements ForkJoinPool.ManagedBlocker {
      ...
      public boolean block()
      { mResult = mSupplier.get(); mDone = true; return true; }

      public boolean isReleasable()
      { return mDone; }

      public T getResult()
      { return mResult; }
}
```

True if blocking supplier has finished, else false

There is no "non-blocking" behavior for BlockingTask.

Encapsulating ManagedBlocker With the BlockingTask Class

- BlockingTask integrates blocking suppliers with the common fork-join pool.

```
public class BlockingTask {
   ...
   private static class SupplierManagedBlocker<T>
                  implements ForkJoinPool.ManagedBlocker {
      ...
      public boolean block()
      { mResult = mSupplier.get(); mDone = true; return true; }

      public boolean isReleasable()
      { return mDone; }

      public T getResult()
      { return mResult; }
}
```

Returns supplier's result (called after pool.managedBlock() completes)

Encapsulating ManagedBlocker With the BlockingTask Class

- This example uses BlockingTask to ensure that there are enough threads in the common thread pool.

```
Image blockingDownload(URL url) {
  return BlockingTask
    .callInManagedBlock
      (() -> downloadImage(url));
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex20

Encapsulating ManagedBlocker With the BlockingTask Class

- This example uses BlockingTask to ensure that there are enough threads in the common thread pool.

```
Image blockingDownload(URL url) {
  return BlockingTask
    .callInManagedBlock
      (() -> downloadImage(url));
}
```

Transform a URL to an image by downloading each image via its URL

Encapsulating ManagedBlocker With the BlockingTask Class

- This example uses BlockingTask to ensure that there are enough threads in the common thread pool.

```
Image blockingDownload(URL url) {
  return BlockingTask
    .callInManagedBlock
      (() -> downloadImage(url));
}
```

This method call ensures the common fork-join thread pool is expanded to handle the blocking image download.

Encapsulating ManagedBlocker With the BlockingTask Class

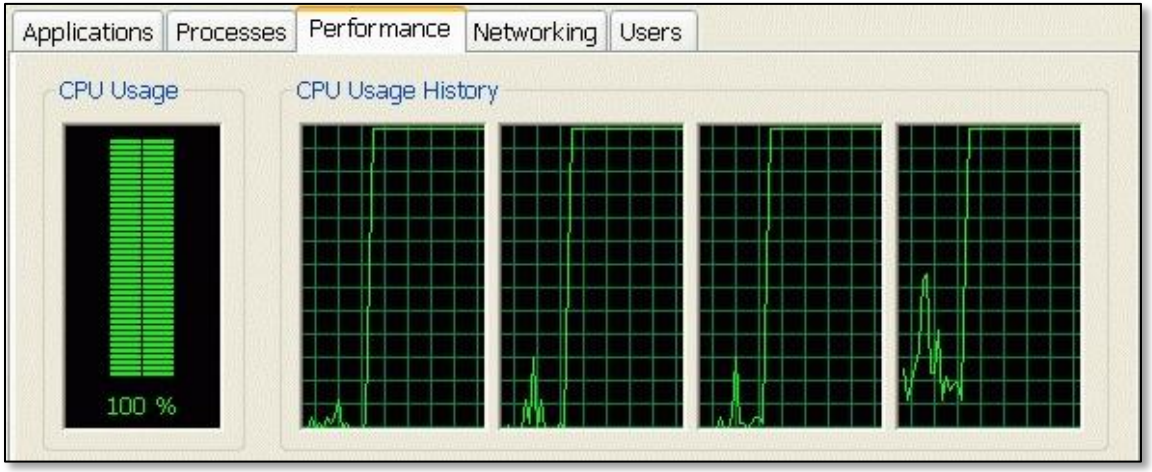
- This example uses BlockingTask to ensure that there are enough threads in the common thread pool.
  - Extra threads in the common fork-join pool are automatically terminated later.

```
Image blockingDownload(URL url) {
  return BlockingTask
    .callInManagedBlock
      (() -> downloadImage(url));
}
```

Encapsulating ManagedBlocker With the BlockingTask Class

- This example uses BlockingTask to ensure that there are enough threads in the common thread pool.
  - Extra threads in the common fork-join pool are automatically terminated later.
  - However, it's possible to saturate the CPU cores during bursty workloads.

```
Image blockingDownload(URL url) {
  return BlockingTask
    .callInManagedBlock
      (() -> downloadImage(url));
}
```



The End

The Java Fork-Join Pool

Overview of the Example Applications

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Compare and contrast different fork-join pool programming models

```
<T> List<T> applyAllIter(List<T> l, Function<T, T> op, ForkJoinPool fjp) {
  return fjp.invoke(new RecursiveTask<List<T>>() {
    protected List<T> compute() { ... }
   });
}

<T> List<T> applyAllSplit(List<T> l, Function<T, T> op, ForkJoinPool fjp) {
   class SplitterTask extends RecursiveTask<List<T>> { ... }
  return fjp.invoke(new SplitterTask(l));
 }

<T> List<T> applyAllSplitIndex(List<T> l,
                               Function<T, T> op, ForkJoinPool fjp) {
  T[] res = (T[]) Array.newInstance(l.get(0).getClass(), l.size());
  class SplitterTask extends RecursiveAction { ... }
  fjp.invoke(new SplitterTask(0, l.size()));
   return Arrays.asList(res);
}
```

Applying the Java Fork-Join Framework

Applying the Java Fork-Join Framework

- Several different Java fork-join programming models are applied on a common dataset.

```
public static <T> List<T> applyAllIter(List<T> list,
                                       Function<T, T> op,
                                       ForkJoinPool forkJoinPool)
    // Invoke a new task in the fork-join pool.
    return forkJoinPool.invoke((RecursiveTask) () -> {
            // Create a list to hold the forked tasks.
            List<ForkJoinTask<T>> forks =
                new LinkedList<>();

            // Create a list to hold the joined results.
            List<T> results =
                new LinkedList<>();

            // Iterate through list, fork all the tasks,
```

```
[1] Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex22

Applying the Java Fork-Join Framework

- Several different Java fork-join programming models are applied on a common dataset.
  - These models have different performance pros and cons.

```
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
```

```
public static <T> List<T> applyAllIter(List<T> list,
                                       Function<T, T> op,
                                       ForkJoinPool forkJoinPool)
    // Invoke a new task in the fork-join pool.
    return forkJoinPool.invoke((RecursiveTask) () -> {
            // Create a list to hold the forked tasks.
            List<ForkJoinTask<T>> forks =
                new LinkedList<>();

            // Create a list to hold the joined results.
            List<T> results =
                new LinkedList<>();

            // Iterate through list, fork all the tasks,
```

```
rting ForkJoinTest
lIter() steal count = 31
lSplitIndex() steal count = 16
lSplit() steal count = 21
lSplitIndexEx() steal count = 21
nting 4 results from fastest to slowest
lyAllSplitIndexEx() executed in 4575 ms
lyAllSplitIndex() executed in 5145 ms
lyAllSplit() executed in 5172 ms
lyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest
```

Some incur more "stealing," copy more data, make more method calls, etc.

Applying the Java Fork-Join Framework

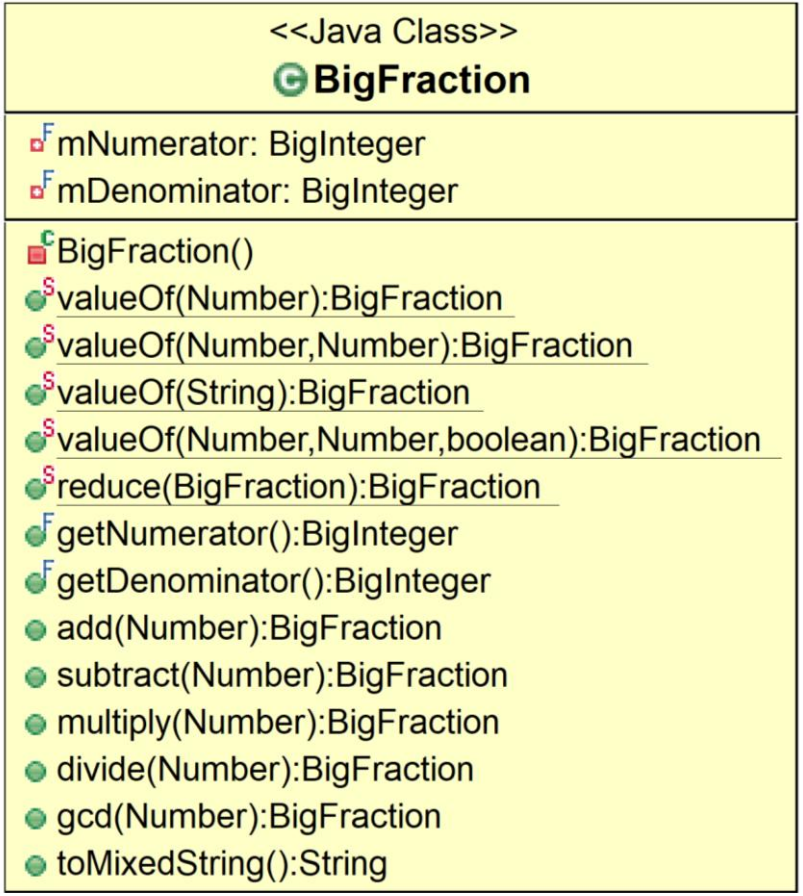
- Several different Java fork-join programming models are applied on a common dataset.
  - These models have different performance pros and cons.
  - Java functional programming and sequential streams features are used to simplify the code.

```
List<BigFraction> fractionList =
  Stream
    .generate(() ->
       makeBigFraction(new Random(),
                       false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

Function<BigFraction,
         BigFraction> op =
   bigFraction -> BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);
```

Applying the Java Fork-Join Framework

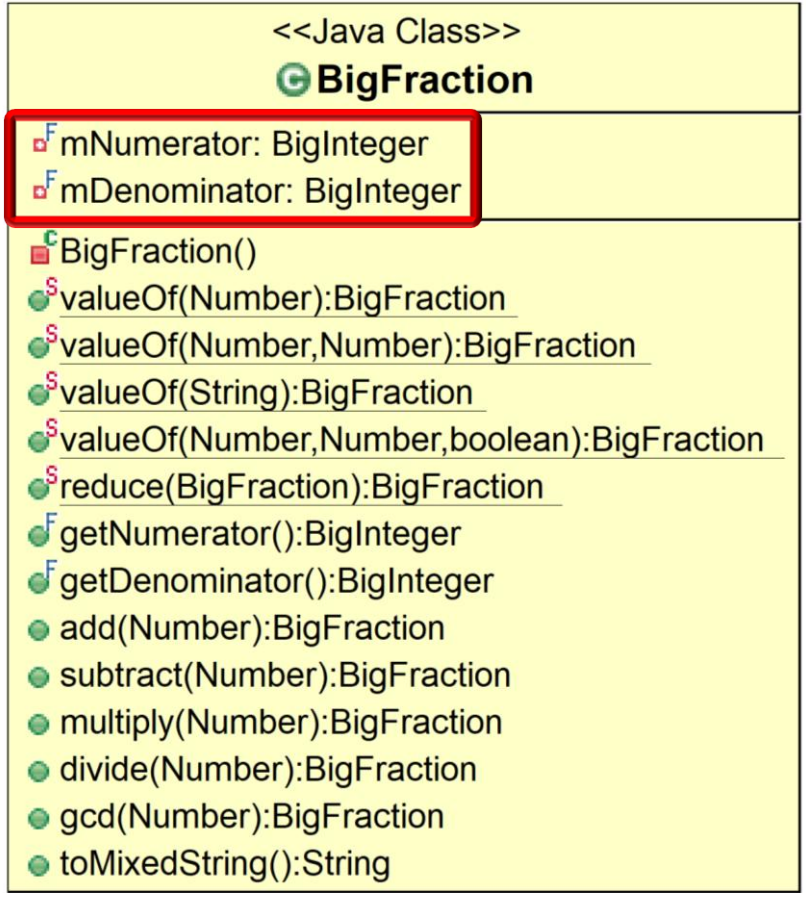
- These fork-join programming models all perform same operations on BigFractions.



See LiveLessons/blob/master/Java8/ex22/src/utils/BigFraction.java

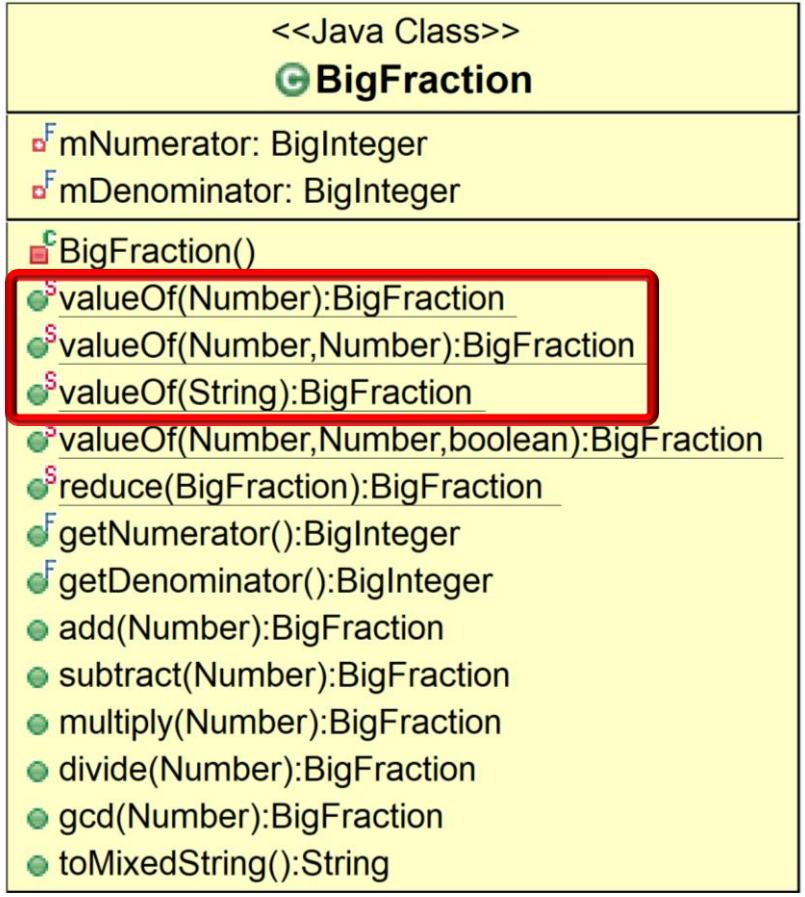
Applying the Java Fork-Join Framework

- These fork-join programming models all perform same operations on BigFractions.
  - Arbitrary-precision fraction has numerator and denominator using BigIntegers



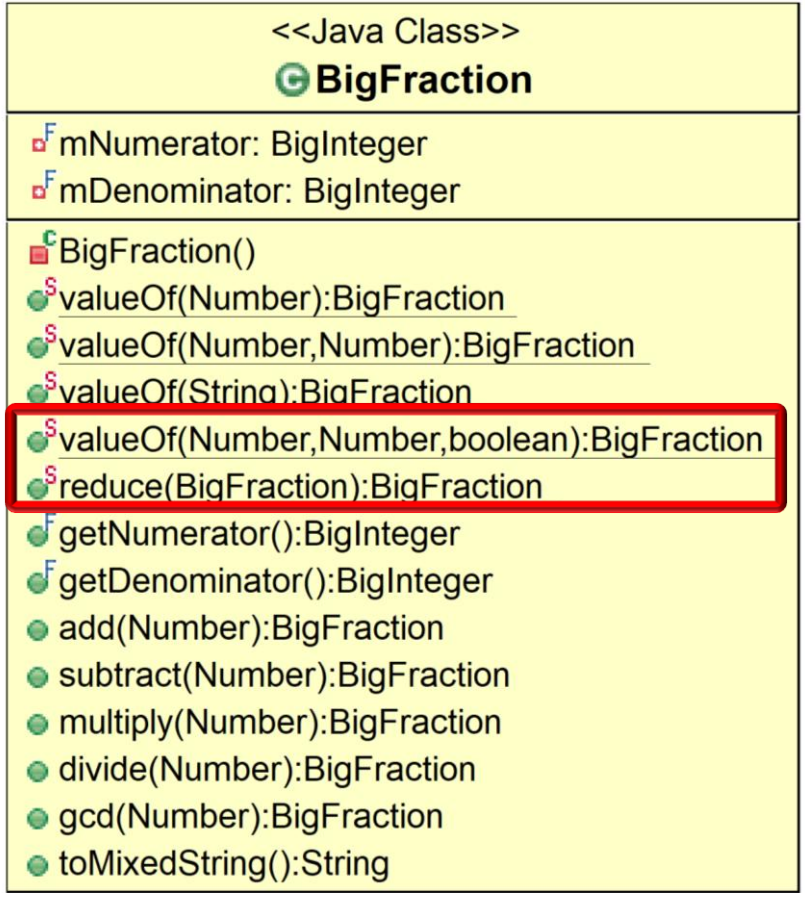
Applying the Java Fork-Join Framework

- These fork-join programming models all perform same operations on BigFractions.
  - Arbitrary-precision fraction has numerator and denominator using BigIntegers
  - Factory methods for creating "reduced" fractions
    - 44/55 → 4/5
    - 12/24 → 1/2
    - 144/216 → 2/3



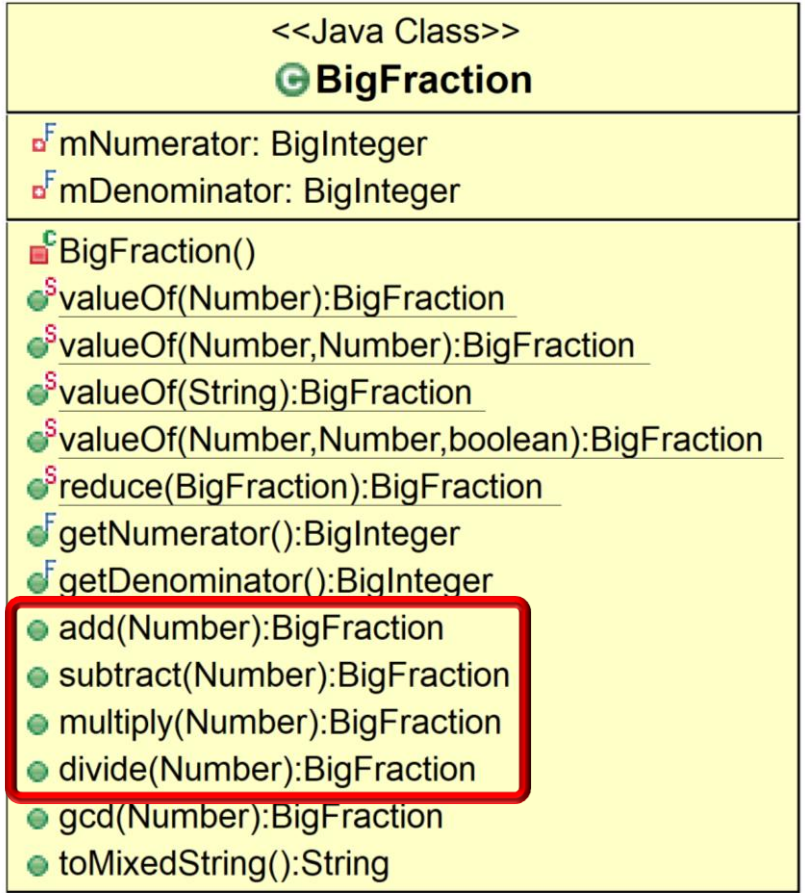
Applying the Java Fork-Join Framework

- These fork-join programming models all perform same operations on BigFractions.
  - Arbitrary-precision fraction has numerator and denominator using BigIntegers
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (and reducing them)
    - 12/24 (→ 1/2)



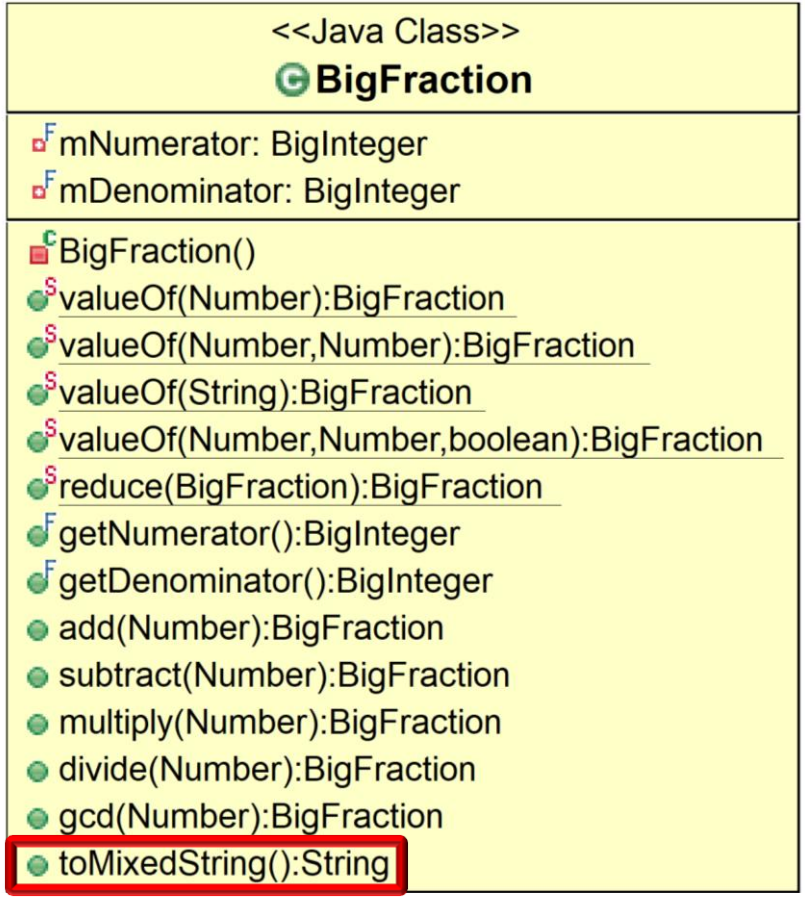
Applying the Java Fork-Join Framework

- These fork-join programming models all perform same operations on BigFractions.
  - Arbitrary-precision fraction has numerator and denominator using BigIntegers
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (and reducing them)
  - Arbitrary-precision fraction arithmetic
    - 18/4 × 2/3 = 3



Applying the Java Fork-Join Framework

- These fork-join programming models all perform same operations on BigFractions.
  - Arbitrary-precision fraction has numerator and denominator using BigIntegers
  - Factory methods for creating "reduced" fractions
  - Factory methods for creating "non-reduced" fractions (and reducing them)
  - Arbitrary-precision fraction arithmetic
  - Create a mixed fraction from an improper fraction
    - 18/4 → 4 1/2



See www.mathsisfun.com/improper-fractions.html

Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

See LiveLessons/blob/master/Java8/ex22/src/ex22.java

Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

Use a Java stream to generate random BigFractions up to sMAX_FRACTIONS

This is the primary use of Java streams in this example.

Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
BigFraction makeBigFraction(Random random, boolean reduced) {
  BigInteger numerator =
    new BigInteger(150000, random);

  BigInteger denominator =
    numerator.divide(BigInteger
                     .valueOf(random.nextInt(10) + 1));

  return BigFraction.valueOf(numerator,
                             denominator,
                             reduced);
}
```

Factory method that creates a large and random big fraction

See LiveLessons/blob/master/Java8/ex22/src/utils/BigFraction.java

Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
BigFraction makeBigFraction(Random random, boolean reduced) {
  BigInteger numerator =
    new BigInteger(150000, random);

  BigInteger denominator =
    numerator.divide(BigInteger
                     .valueOf(random.nextInt(10) + 1));

  return BigFraction.valueOf(numerator,
                             denominator,
                             reduced);
}
```

Make a random numerator uniformly distributed over range 0 to ($2^{150000} - 1$)

See docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html#BigInteger

Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
BigFraction makeBigFraction(Random random, boolean reduced) {
  BigInteger numerator =
    new BigInteger(150000, random);

  BigInteger denominator =
    numerator.divide(BigInteger
                     .valueOf(random.nextInt(10) + 1));

  return BigFraction.valueOf(numerator,
                             denominator,
                             reduced);
}
```

Make denominator by dividing numerator by a random number between 1 and 10

Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
BigFraction makeBigFraction(Random random, boolean reduced) {
  BigInteger numerator =
    new BigInteger(150000, random);

  BigInteger denominator =
    numerator.divide(BigInteger
                     .valueOf(random.nextInt(10) + 1));

  return BigFraction.valueOf(numerator,
                             denominator,
                             reduced);
}
```

Return a BigFraction with the numerator and denominator

Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

A function that reduces and multiplies a big fraction

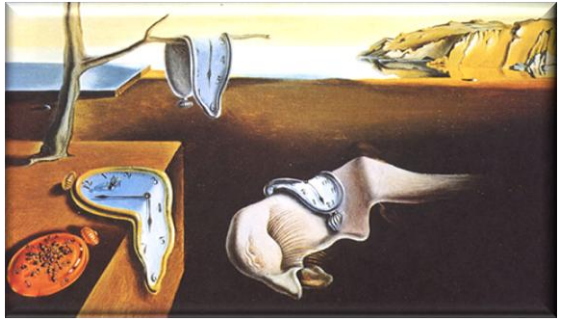
Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```



This function takes a surprisingly long time to run!

Applying the Java Fork-Join Framework

- The program reduces and multiplies big fractions using the Java fork-join framework.

```
public static void main(String[] argv) throws IOException {
  List<BigFraction> fractionList = Stream
    .generate(() -> makeBigFraction(new Random(), false))
    .limit(sMAX_FRACTIONS)
    .collect(toList());

  Function<BigFraction, BigFraction> op = bigFraction ->
    BigFraction
      .reduce(bigFraction)
      .multiply(sBigReducedFraction);

  testApplyAllIter(fractionList, op);
  testApplyAllSplit(fractionList, op);
  testApplyAllSplitIndex(fractionList, op); ...
```

Time various fork-join tests

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), and applyAllSplitIndex() helper methods.

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplitIndex
                     (List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), and applyAllSplitIndex() helper methods.

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplitIndex
                     (List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Each helper method uses a different means of applying the fork-join framework.

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), and applyAllSplitIndex() helper methods.

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }
```

Uses "work stealing" to disperse tasks to worker threads

```
void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }
```

```
void testApplyAllSplitIndex
                      (List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

See upcoming lesson on "*Java Fork-Join Pool: Implementing applyAllIter()*"

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), and applyAllSplitIndex() helper methods.

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }
```

Uses recursive decomposition to disperse tasks to worker threads

```
void testApplyAllSplitIndex
                     (List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

See upcoming lesson on "*Java Fork-Join Pool: Implementing applyAllSplit()*"

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), and applyAllSplitIndex() helper methods.

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }
```

Uses optimized recursive decomposition to disperse tasks to worker threads

```
void testApplyAllSplitIndex
                     (List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

See upcoming lesson on "*Java Fork-Join Pool: Implementing applyAllSplitIndex()*"

Applying the Java Fork-Join Framework

- Test the applyAllIter(), applyAllSplit(), and applyAllSplitIndex() helper methods.

```
void testApplyAllIter(List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllIter(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplit(List<BigFraction> fractionList,
                       Function<BigFraction, BigFraction> op)
{ applyAllSplit(fractionList, op, new ForkJoinPool()); }


void testApplyAllSplitIndex
                     (List<BigFraction> fractionList,
                      Function<BigFraction, BigFraction> op)
{ applyAllSplitIndex(fractionList, op, new ForkJoinPool()); }
```

Each helper method gets own fork-join pool sized to number of processor cores.

The End

The Java Fork-Join Pool

Implementing applyAllIter()

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Apply the fork-join framework in practice
- Examine the applyAllIter() method
  - This method uses "work stealing" to disperse tasks to worker threads.

```
<T> List<T> applyAllIter
   (List<T> list,
    Function<T, T> op,
    ForkJoinPool fjPool) {
   return fjPool
     .invoke(new
        RecursiveTask
          <List<T>>() {
        protected List<T>
          compute() {
                  ...
          }
        });
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex22

Implementing the applyAllIter() Method

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) {
  return fjPool
    .invoke(new RecursiveTask<List<T>>() {
              protected List<T> compute() {
                ...
              }
            });
}
```

See LiveLessons/blob/master/Java8/ex22/src/utils/ForkJoinUtils.java

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) {
  return fjPool
    .invoke(new RecursiveTask<List<T>>() {
              protected List<T> compute() {
                ...
              }
            });
}
```

These parameters are treated as "effectively final" variables in the anonymous inner class below.

See www.linkedin.com/pulse/java-8-effective-final-gaurhari-dass

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) {
  return fjPool
    .invoke(new RecursiveTask<List<T>>() {
              protected List<T> compute() {
                ...
              }
            });
}
```

Create an anonymous RecursiveTask instance whose compute() method iterative creates many sub-tasks

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) {
  return fjPool
    .invoke(new RecursiveTask<List<T>>() {
              protected List<T> compute() {
                ...
              }
            });
}
```

Code is verbose due to lack of functional interface (and thus can't use lambdas).

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) {
  return fjPool
    .invoke(new RecursiveTask<List<T>>() {
              protected List<T> compute() {
                ...
              }
            });
}
```

Invoke the task on the fork-join pool and then wait for and return the results

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

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T : list)
      forks.add(new RecursiveTask<T>() {
                protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Hook method implements the main fork-join task.

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Lists that hold the forked tasks and the results

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Iterate through input list, fork all the tasks, and add them to the forks list

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                  protected T compute() { return
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

This implementation relies on "work stealing" to disperse tasks to worker threads.

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Join all results of forked tasks and add to results list

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

Return the results list

Implementing the applyAllIter() Method

- Apply an "op" to all items in the list by calling fork-join methods iteratively.

```
<T> List<T> applyAllIter(List<T> list, Function<T, T> op,
                         ForkJoinPool fjPool) { ...
  protected List<T> compute() {
    List<ForkJoinTask<T>> forks = new LinkedList<>();
    List<T> res = new LinkedList<>();

    for (T t : list)
      forks.add(new RecursiveTask<T>() {
                protected T compute() { return op.apply(t); }
      }.fork());

    for (ForkJoinTask<T> task : forks) res.add(task.join());
    return res;
  } ...
```

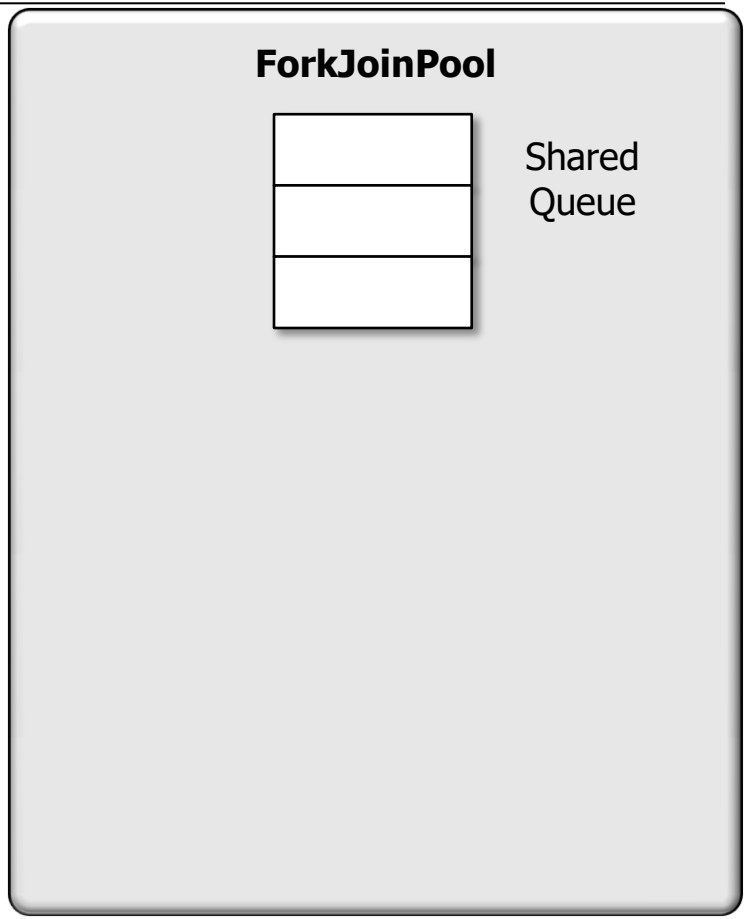
This implementation is very simple to program and understand since it's iterative.

Visualizing the applyAllIter() Method

Visualizing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  fjPool.invoke
    (new RecursiveTask<List<T>>() {
       protected List<T> compute() {
          ...
       }
  }
  ...
```

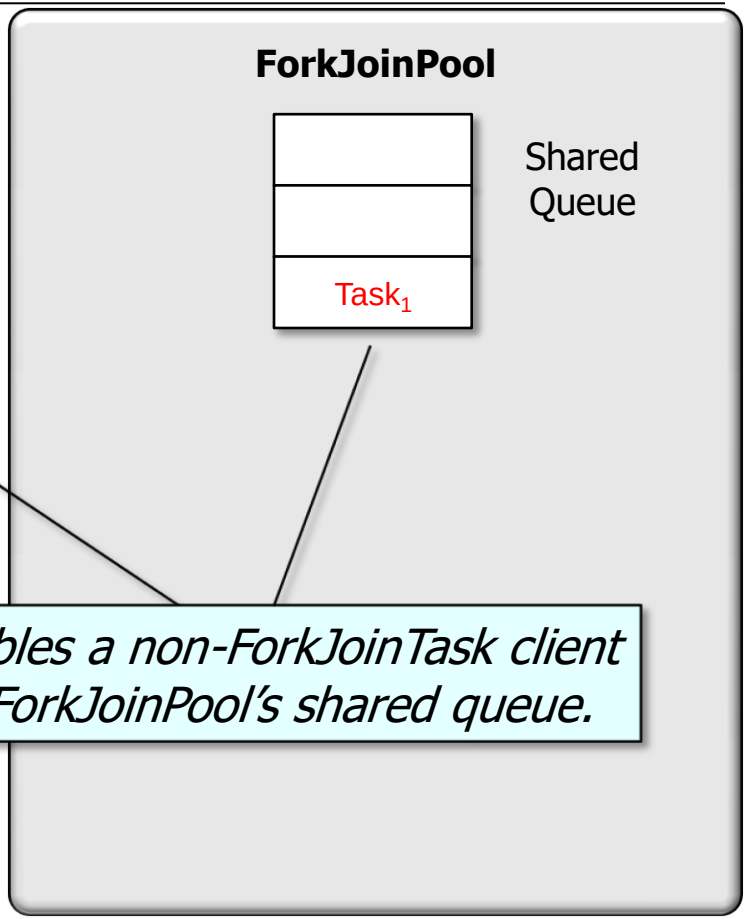


See LiveLessons/blob/master/Java8/ex22/src/utils/ForkJoinUtils.java

Visualizing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  fjPool.invoke
    (new RecursiveTask<List<T>>() {
       protected List<T> compute() {
          ...
       }
  }
  ...
```



The invoke() method enables a non-ForkJoinTask client to insert a task into the ForkJoinPool's shared queue.

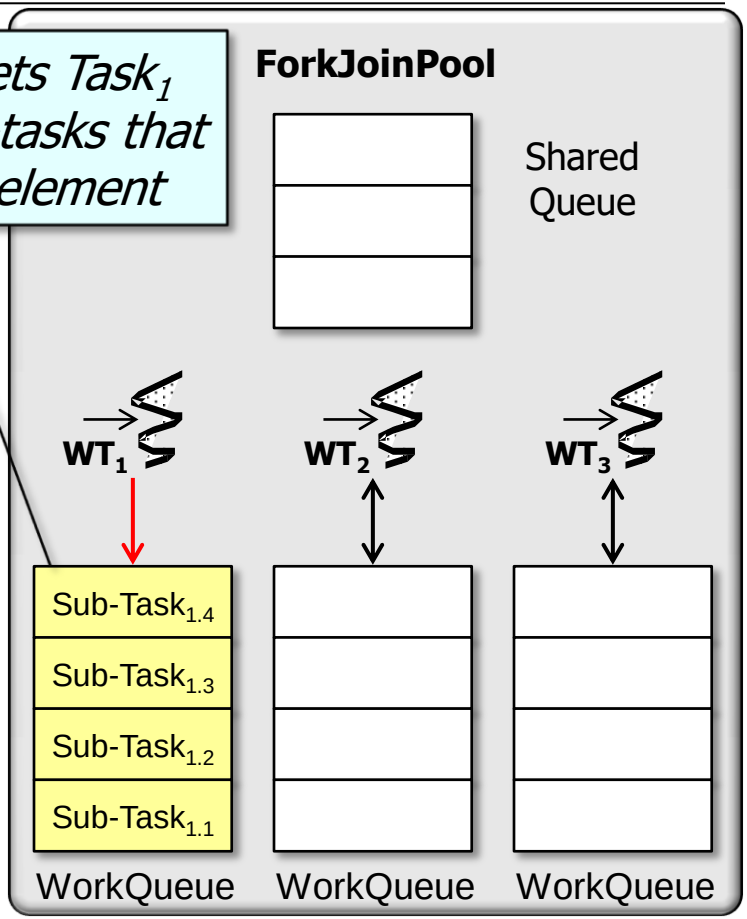
Visualizing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAll
          (List<T> li
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  for (T t : list)
    forks.add(new RecursiveTask<T>() {
      protected T compute()
      { return op.apply(t); }
    }.fork());

  for (ForkJoinTask<T> task : forks)
    results.add(task.join());
  ...
```

Worker thread WT_1 gets $Task_1$ and creates n new sub-tasks that run "op" on each list element



The highlighted code runs in the RecursiveTask's compute() method.

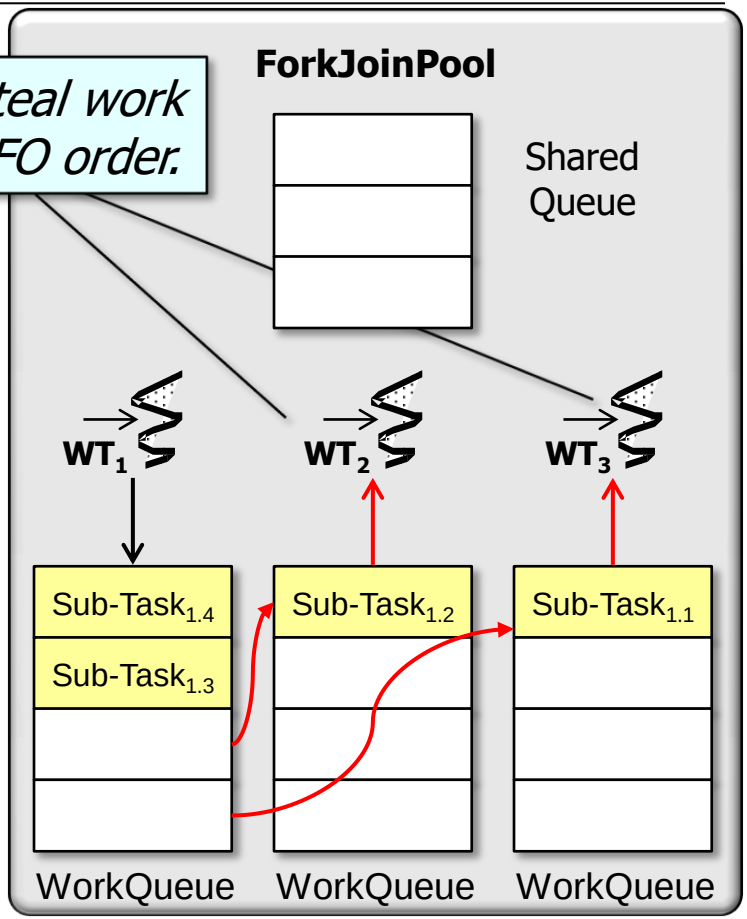
Visualizing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  for (T t : list)
    forks.add(new RecursiveTask<T>() {
      protected T compute()
      { return op.apply(t); }
    }.fork());

  for (ForkJoinTask<T> task : forks)
    results.add(task.join());
  ...
```

WT_2 and WT_3 steal work from WT_1 in FIFO order.



"Work stealing" overhead is high, but copying and method call overhead is low.

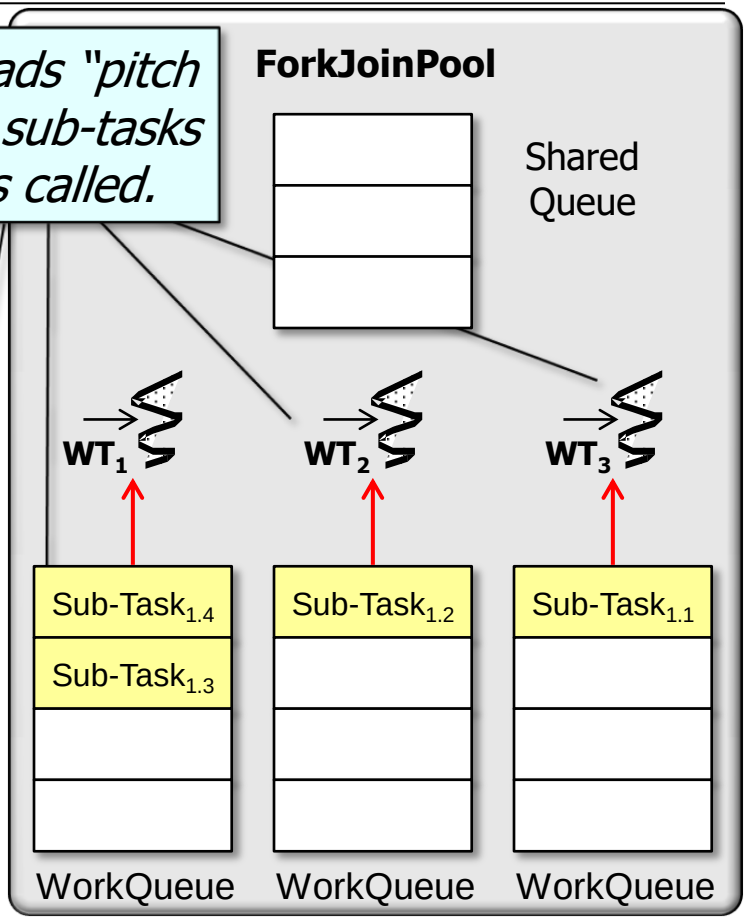
Visualizing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  for (T t : list)
    forks.add(new RecursiveTask<T>() {
      protected T compute()
      { return op.apply(t); }
    }.fork());

  for (ForkJoinTask<T> task : forks)
    results.add(task.join());
  ...
```

All worker threads "pitch in" to compute sub-tasks when join() is called.

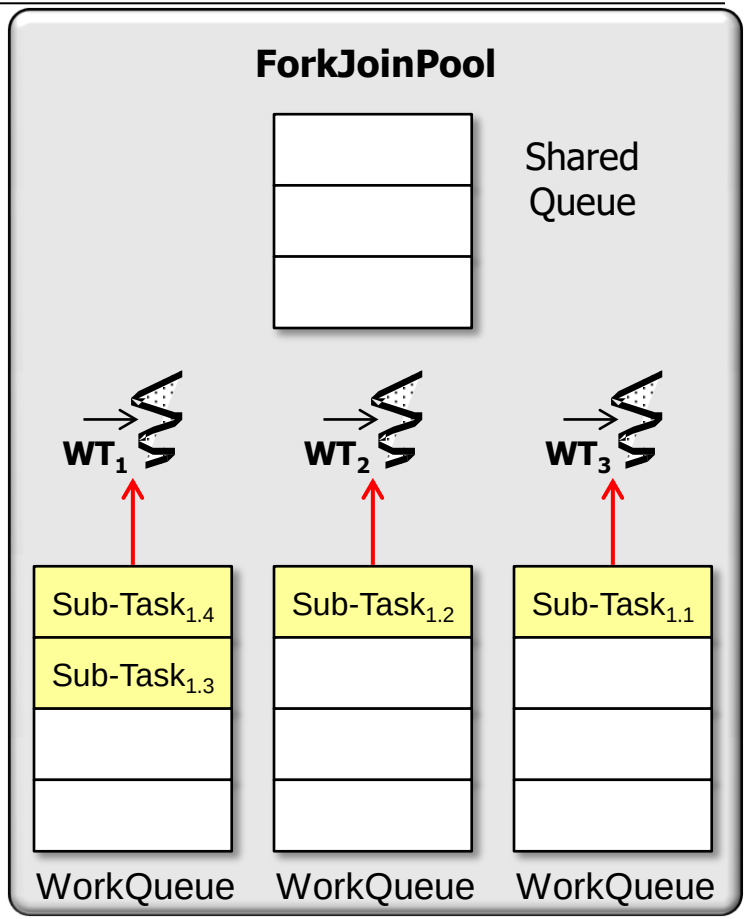


Visualizing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  for (T t : list)
    forks.add(new RecursiveTask<T>() {
      protected T compute()
      { return op.apply(t); }
    }.fork());

  for (ForkJoinTask<T> task : forks)
    results.add(task.join());
  ...
```



"Collaborative Jiffy Lube" model of processing!

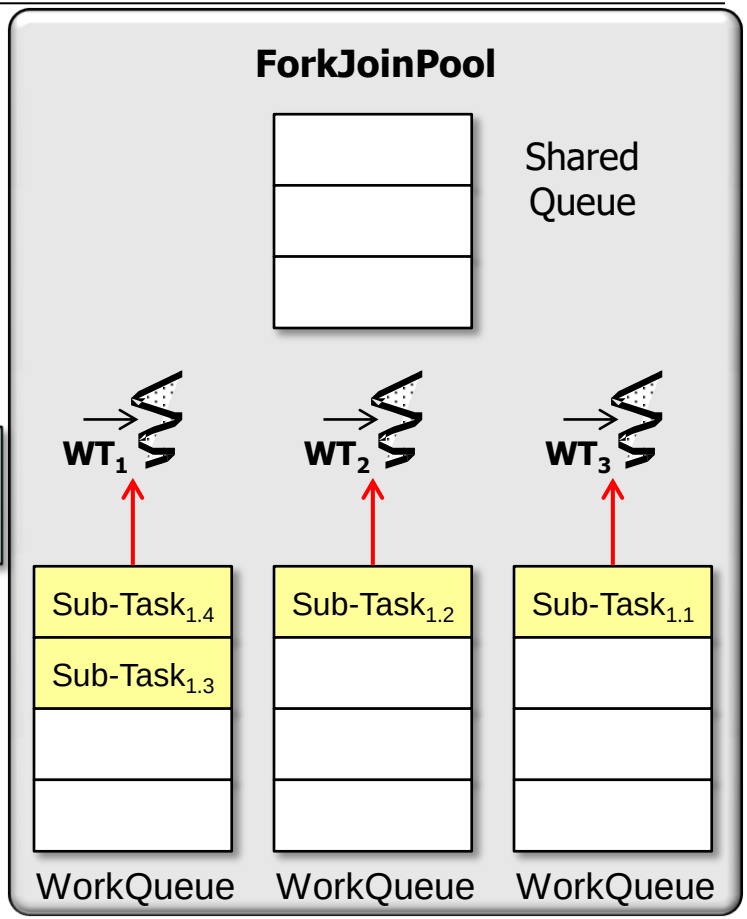
Visualizing the applyAllIter() Method

- Visualizing applyAllIter()

```
<T> List<T> applyAllIter
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  for (T t : list)
    forks.add(new
      protected T
      { return op.apply(t); }
    }.fork());

  for (ForkJoinTask<T> task : forks)
    results.add(task.join());
  ...
```

This loop implements "barrier synchronization."



See en.wikipedia.org/wiki/Barrier_(computer_science)

The End

The Java Fork-Join Pool

Implementing applyAllSplit()

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Apply the fork-join framework in practice
- Examine the applyAllIter() method
- Examine the applyAllSplit() method

```
<T> List<T> applyAllSplit
   (List<T> list,
    Function<T, T> op,
    ForkJoinPool fjPool) {
    class SplitterTask
    extends RecursiveTask
      <List<T>> { ... }

   return fjPool
      .invoke(new
         SplitterTask(list));
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex22

Implementing the Method applyAllSplit()

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) {
  class SplitterTask extends RecursiveTask<List<T>> { ... }




  return fjPool
    .invoke(new SplitterTask(list));
}
```

See LiveLessons/blob/master/Java8/ex22/src/utils/ForkJoinUtils.java

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) {
  class SplitterTask extends RecursiveTask<List<T>> { ... }

  return fjPool
    .invoke(new SplitterTask(list));
}
```

This task recursively partitions the list and runs each half in a ForkJoinTask.

This use case is an ideal application of the fork-join framework!

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) {
  class SplitterTask extends RecursiveTask<List<T>> { ... }
```

```
  return fjPool
    .invoke(new SplitterTask(list));
}
```

Invoke a new SplitterTask in the fork-join pool and then wait for and return the results

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    private List<T> mList;



    private SplitterTask(List<T> list) {
        mList = list;
    }
    ...

  } ...
```

This implementation recursively splits the list evenly in half and copies data.

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    private List<T> mList;



    private SplitterTask(List<T> list) {
        mList = list;
    }
    ...

  } ...
```

Constructor stores a reference to a portion of the original list.

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      if (mList.size() <= 1) {
        List<T> result =
          new ArrayList<>();

        for (T t : mList) result.add(op.apply(t));

        return result;
      } else {
        ...
    } ...
```

Recursively perform the computations in parallel on a subset of the list

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      if (mList.size() <= 1) {
        List<T> result =
          new ArrayList<>();

        for (T t : mList) result.add(op.apply(t));

        return result;
      } else {
        ...
    } ...
```

If the list has one or zero elements, create an empty ArrayList.

This if statement handles the base case for the recursion

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      if (mList.size() <= 1) {
        List<T> result =
          new ArrayList<>();

        for (T t : mList) result.add(op.apply(t));

        return result;
      } else {
        ...
    } ...
```

Apply "op" to the element in this list and add it to the result list

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      if (mList.size() <= 1) {
        List<T> result =
          new ArrayList<>();

        for (T t : mList) result.add(op.apply(t));

        return result;
      } else {
        ...
    } ...
```

Return the result list

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      ... else {
        int mid = mList.size() / 2;
        ForkJoinTask<List<T>> lt =
          new SplitterTask(mList.subList(0, mid)).fork();
        mList = mList.subList(mid, mList.size());
        List<T> rightResult = compute();
        List<T> leftResult = lt.join();
        leftResult.addAll(rightResult);
        return leftResult;
    } ...
```

Determine midpoint of the list

This else statement recursively and evenly partitions the list.

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      ... else {
        int mid = mList.size() / 2;
        ForkJoinTask<List<T>> lt =
          new SplitterTask(mList.subList(0, mid)).fork();
        mList = mList.subList(mid, mList.size());
        List<T> rightResult = compute();
        List<T> leftResult = lt.join();
        leftResult.addAll(rightResult);
        return leftResult;
    } ...
```

Create a new task to handle left-side of the list and fork it

This recursive decomposition disperses tasks to (other) worker threads.

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      ... else {
        int mid = mList.size() / 2;
        ForkJoinTask<List<T>> lt =
          new SplitterTask(mList.subList(0, mid)).fork();
        mList = mList.subList(mid, mList.size());
        List<T> rightResult = compute();
        List<T> leftResult = lt.join();
        leftResult.addAll(rightResult);
        return leftResult;
    } ...
```

Update mList to handle the right-side and compute results

compute() runs in the same task as its "parent" to minimize context switching.

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      ... else {
        int mid = mList.size() / 2;
        ForkJoinTask<List<T>> lt =
          new SplitterTask(mList.subList(0, mid)).fork();
        mList = mList.subList(mid, mList.size());
        List<T> rightResult = compute();
        List<T> leftResult = lt.join();
        leftResult.addAll(rightResult);
        return leftResult;
    } ...
```

Join with left-side results

compute() must be called before join()!

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      ... else {
        int mid = mList.size() / 2;
        ForkJoinTask<List<T>> lt =
          new SplitterTask(mList.subList(0, mid)).fork();
        mList = mList.subList(mid, mList.size());
        List<T> rightResult = compute();
        List<T> leftResult = lt.join();
        leftResult.addAll(rightResult);
        return leftResult;
    } ...
```

Combine left- and right-size and return results

Data may be copied since the lists are implemented via ArrayList.

Implementing the Method applyAllSplit()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplit(List<T> list, Function<T, T> op,
                          ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveTask<List<T>> {
    protected List<T> compute() {
      ... else {
        int mid = mList.size() / 2;
        ForkJoinTask<List<T>> lt =
          new SplitterTask(mList.subList(0, mid)).fork();
        mList = mList.subList(mid, mList.size());
        List<T> rightResult = compute();
        List<T> leftResult = lt.join();
        leftResult.addAll(rightResult);
        return leftResult;
    } ...
```

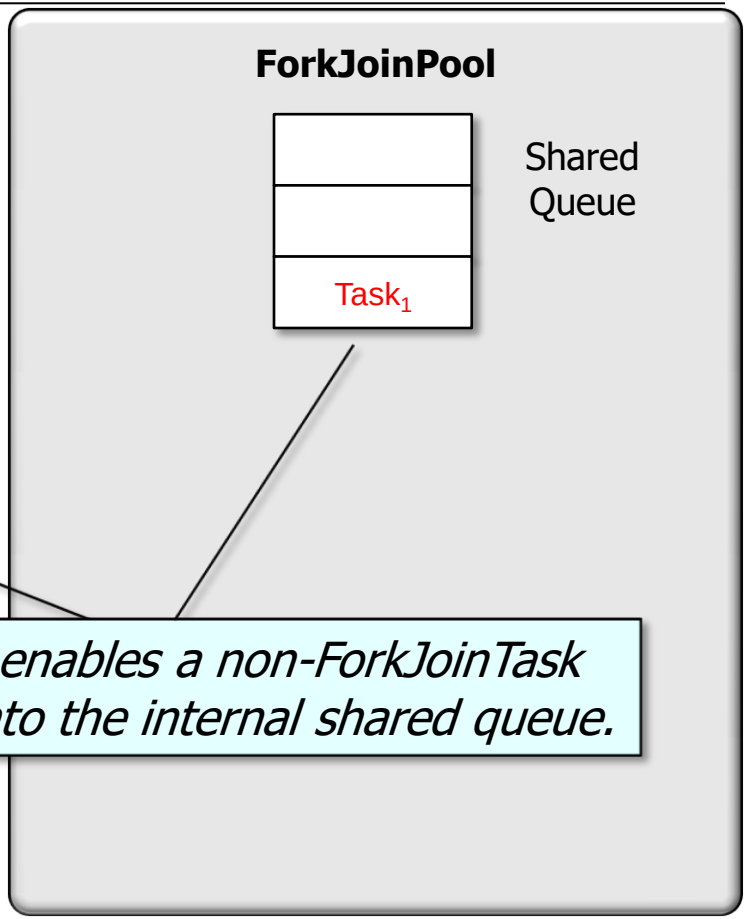
This implementation is harder to program and understand since it's recursive.

Visualizing the applyAllSplit() Method

Visualizing the applyAllSplit() Method

- Visualizing applyAllSplit()

```
<T> List<T> applyAllSplit
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  return fjPool
    .invoke(new SplitterTask
              (list));
}
```



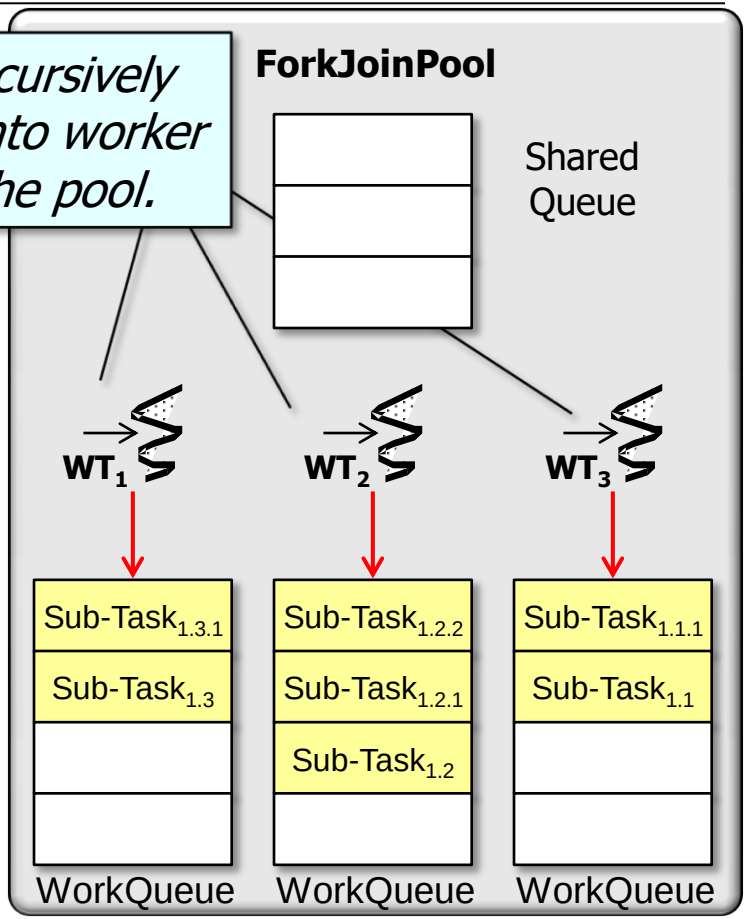
The invoke() method enables a non-ForkJoinTask client to insert a task into the internal shared queue.

Visualizing the applyAllSplit() Method

- Visualizing applyAllSplit()

```
<T> List<T> applyAllSplit
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) { ...
 class SplitterTask ... {
    protected List<T> compute() {
      ... else {
        ForkJoinTask<List<T>> lt =
          new SplitterTask
            (mList.subList(0, mid))
          .fork();
        ...
        List<T> rightResult =
          compute(); ...
```

Sub-tasks recursively decompose onto worker threads in the pool.



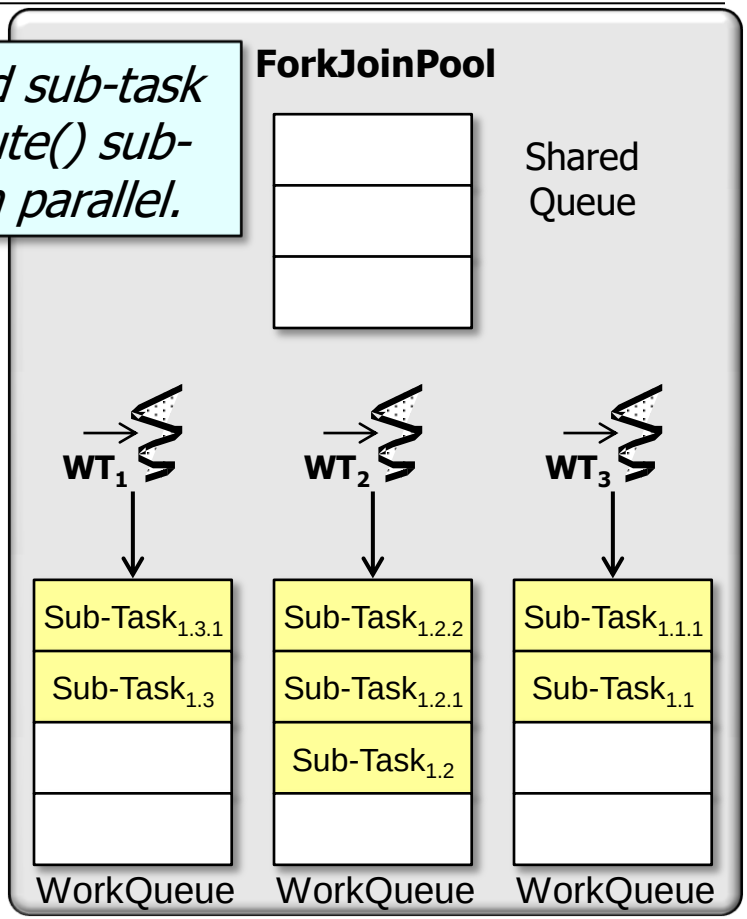
"Work stealing" overhead is lower, but copying and method call overhead is higher.

Visualizing the applyAllSplit() Method

- Visualizing applyAllSplit()

```
<T> List<T> applyAllSplit
          (List<T> list,-
           Function<T, T> op,
           ForkJoinPool fjPool) { ...
 class SplitterTask ... {
    protected List<T> compute() {
       ... else {
         ForkJoinTask<List<T>> lt =
           new SplitterTask
             (mList.subList(0, mid))
           .fork();
         ...
         List<T> rightResult =
           compute(); ...
```

The fork()'d sub-task and compute() sub-task run in parallel.



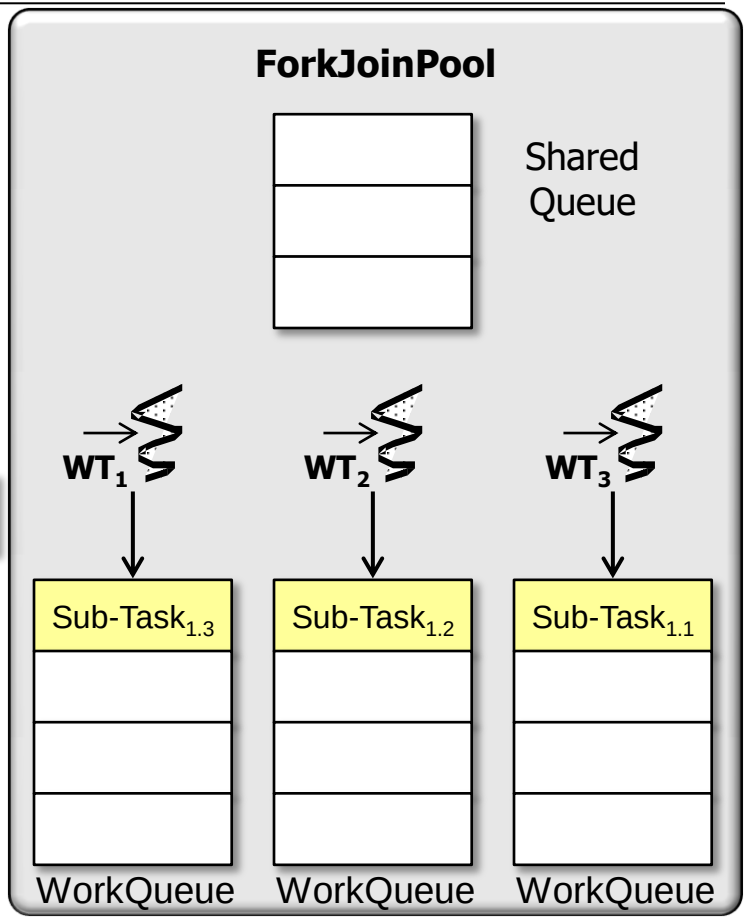
compute() runs in the same task as its "parent" to optimize performance.

Visualizing the applyAllSplit() Method

- Visualizing applyAllSplit()

```
<T> List<T> applyAllSplit
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) { ...
 class SplitterTask ... {
    protected List<T> compute() {
      ... else {
        ...
        List<T> leftResult =
          lt.join();
        leftResult
          .addAll(rightResult);
        return leftResult;
      }
```

join() returns a value



There is a "balanced tree" of join() calls, which scales better than applyAllIter().

The End

The Java Fork-Join Pool

Implementing applyAllSplitIndex()

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Apply the fork-join framework in practice
- Examine the applyAllIter() method
- Examine the applyAllSplit() method
- Examine the applyAllSplitIndex() method

```
<T> List<T> applyAllSplitIndex
   (List<T> list,
    Function<T, T> op,
    ForkJoinPool fjPool) {
   ...

   class SplitterTask
   extends RecursiveAction
   { ... }

   fjPool.invoke(new SplitterTask
     (0, list.size()));

   return Arrays.asList(results);
}
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex22

Implementing the Method applyAllSplitIndex()

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) {
  T[] results = (T[]) Array.newInstance(list.get(0).getClass(),
                                        list.size());


  class SplitterTask extends RecursiveAction { ... }


  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

See LiveLessons/blob/master/Java8/ex22/src/utils/ForkJoinUtils.java

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) {
  T[] results = (T[]) Array.newInstance(list.get(0).getClass(),
                                        list.size());
```

Create a new array to hold the results (yes, it's ugly…)

```
  class SplitterTask extends RecursiveAction { ... }


  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

See stackoverflow.com/a/18137953

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) {
  T[] results = (T[]) Array.newInstance(list.get(0).getClass(),
                                        list.size());
```

This task partitions list recursively and runs each half in a ForkJoinTask.

```
  class SplitterTask extends RecursiveAction { ... }


  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) {
  T[] results = (T[]) Array.newInstance(list.get(0).getClass(),
                                        list.size());


  class SplitterTask extends RecursiveAction { ... }


  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

Invoke a new SplitterTask in the fork-join pool and wait for results

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) {
  T[] results = (T[]) Array.newInstance(list.get(0).getClass(),
                                        list.size());


  class SplitterTask extends RecursiveAction { ... }


  fjPool.invoke(new SplitterTask(0, list.size()));

  return Arrays.asList(results);
}
```

Create and return a list from the array of results

These conversions to and from list → array → list incur overhead.

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndexEx(List<T> list,
                                 Function<T, T> op,
                                 ForkJoinPool fjPool,
                                 T[] results) {
```

An alternative—more elegant and more efficient—approach passes the results array as a parameter.

```
  class SplitterTask extends RecursiveAction {
    /* same implementation as follows next.. */
  }

  fjPool.invoke(new SplitterTask(0, list.size()));
}
```

This approach is cleaner and faster since it has no conversion overhead.

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    private int mLo;
    private int mHi;

    private SplitterTask(int lo, int hi) {
        mLo = lo;
        mHi = hi;
    }
    ...

  } ...
```

This task partitions list recursively and runs each half in a ForkJoinTask.

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    private int mLo;
    private int mHi;

    private SplitterTask(int lo, int hi) {
        mLo = lo;
        mHi = hi;
    }
    ...

  } ...
```

It uses indices to avoid the overhead of copy sublists.

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Recursively perform the computations in parallel

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Find midpoint in current range

This code recursively and evenly partitions the list.

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Apply "op" if there's just one element

This if statement handles the base case for the recursion.

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Create a new task to handle the left-hand side of the list and fork it

This implementation uses recursive decomposition to disperse tasks to worker threads.

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Compute the right-hand side in parallel with left-hand side

compute() runs in the same task as its "parent" to minimize context switching.

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

Join with left-hand side (this is a synchronization point)

See stackoverflow.com/questions/4800503/memory-visibility-in-fork-join

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

No need to merge left/right items since they are already in results!

Implementing the Method applyAllSplitIndex()

- Apply an "op" to all list items by recursively splitting via fork-join method calls.

```
<T> List<T> applyAllSplitIndex(List<T> list,
                               Function<T, T> op,
                               ForkJoinPool fjPool) { ...
  class SplitterTask extends RecursiveAction {
    protected void compute() {
      int mid = (mLo + mHi) >>> 1;
      if (mLo == mid)
        results[mLo] = op.apply(list.get(mLo));
      else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo, mLo = mid).fork();
        compute();
        lt.join();
      } ...
```

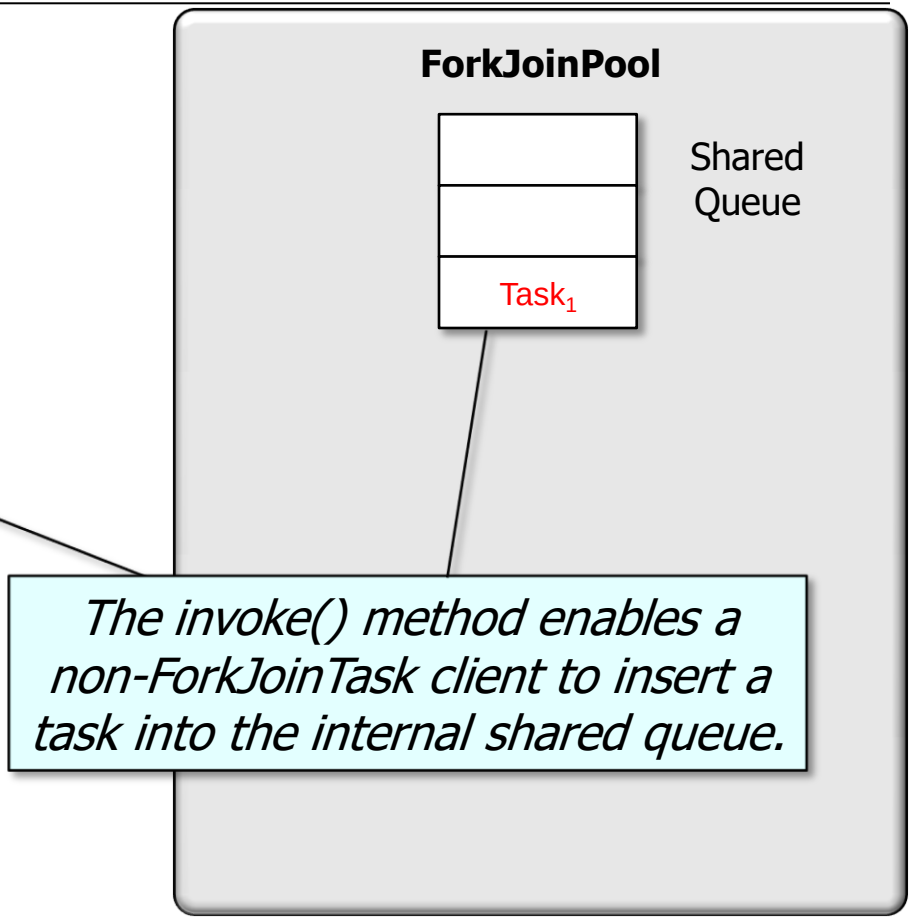
This implementation is also harder to program and understand since it's recursive.

Visualizing the applyAllSplitIndex() Method

Visualizing the applyAllSplitIndex() Method

- Visualizing applyAllSplitIndex()

```
<T> List<T> applyAllSplitIndex
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) {
  ...
  fjPool
    .invoke(new SplitterTask
              (0,
               list.size()));
  ...
```

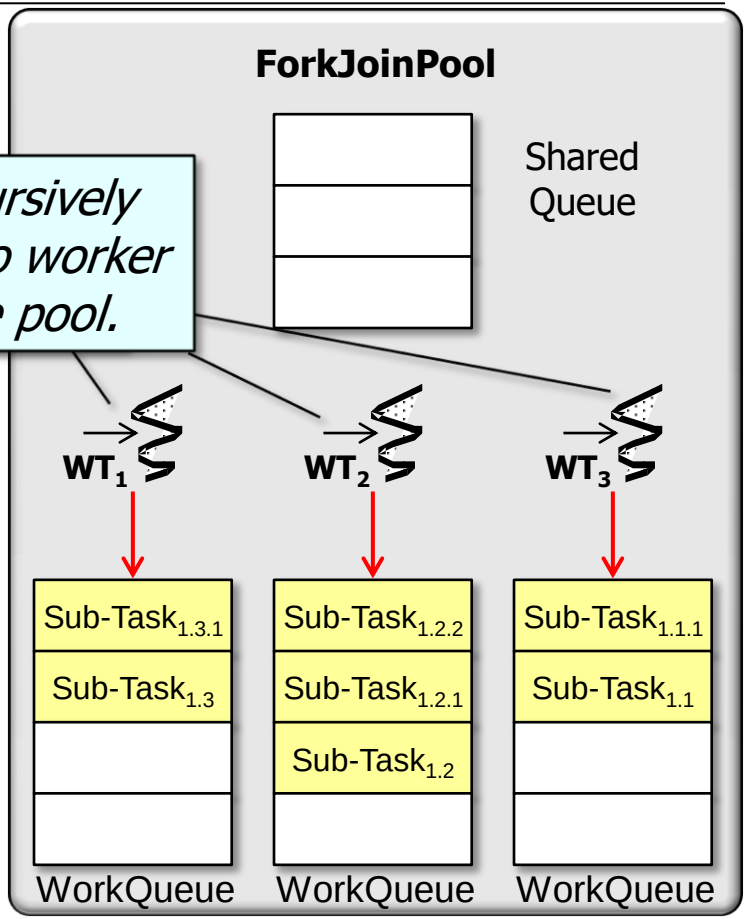


Visualizing the applyAllSplitIndex() Method

- Visualizing applyAllSplitIndex()

```
<T> List<T> applyAllSplitIndex
          (List<T> list,
           Function<T, T>
           ForkJoinPool fj
  class SplitterTask ... {
    protected void compute() {
      ... else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo,
                           mLo = mid)
          .fork();
        compute();
        lt.join();
      }
```

Sub-tasks recursively decompose onto worker threads in the pool.



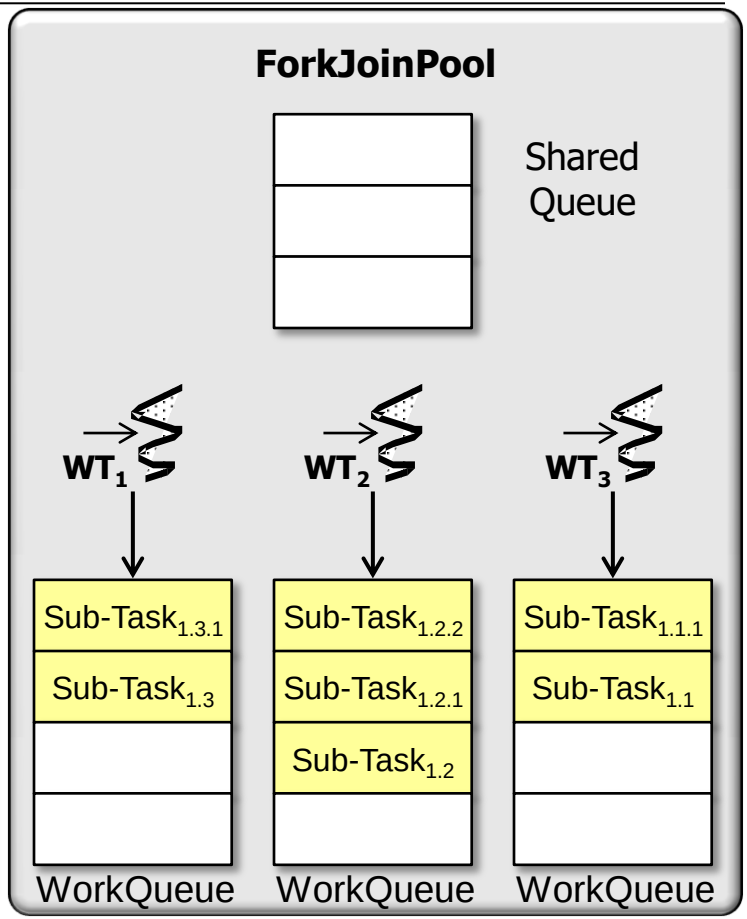
"Work stealing" and copying overhead is low, but method call overhead is high.

Visualizing the applyAllSplitIndex() Method

- Visualizing applyAllSplitIndex()

```
<T> List<T> applyAllSplitIndex
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) { ...
  class SplitterTask ... {
    protected void compute() {
      ... else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo,
                                mLo = mid)
          .fork();
        compute();
        lt.join();
      }
```

The fork()'d sub-task and compute() sub-task run in parallel.

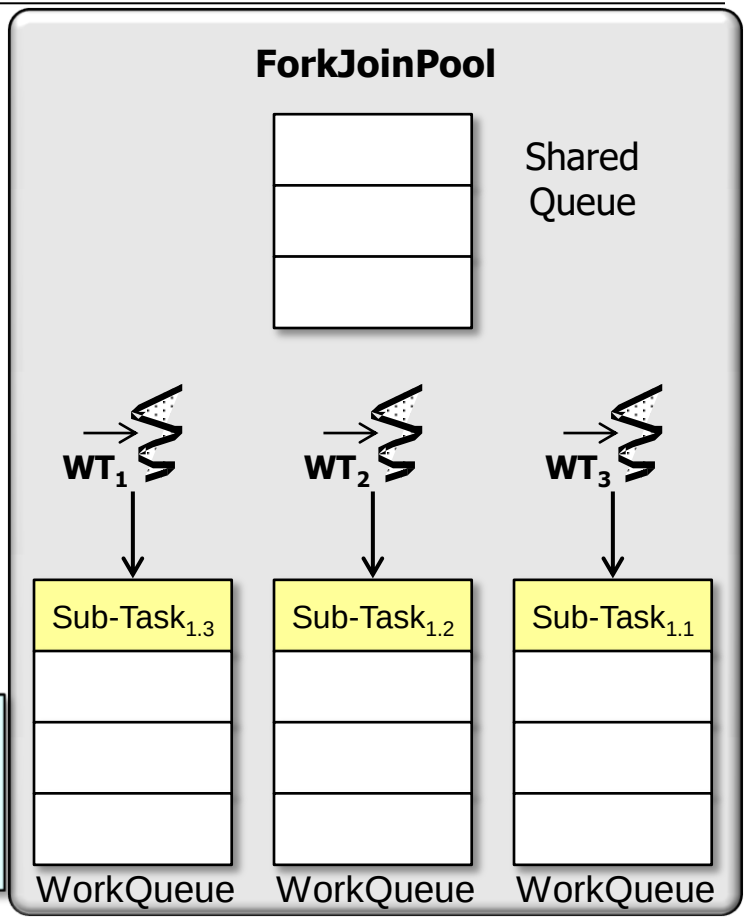


Visualizing the applyAllSplitIndex() Method

- Visualizing applyAllSplitIndex()

```
<T> List<T> applyAllSplitIndex
          (List<T> list,
           Function<T, T> op,
           ForkJoinPool fjPool) { ...
  class SplitterTask ... {
    protected void compute() {
      ... else {
        ForkJoinTask<Void> lt =
          new SplitterTask(mLo,
                                mLo = mid)
          .fork();
        compute();
        lt.join();
      }
```

join() returns no value and just serves as a synchronization point.



There is a "balanced tree" of join() calls.

The End

The Java Fork-Join Pool

Evaluating the Example Applications

Douglas C. Schmidt

Learning Objectives in This Part of the Lesson

- Apply the fork-join framework in practice
- Examine the applyAllIter() method
- Examine the applyAllSplit() method
- Examine the applyAllSplitIndex() method
- Evaluate the example applications of the fork-join pool framework

Evaluating the Example Applications

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.

```
<T> List<T> applyAllIter
                (List<T> list,
                 Function<T, T> op,
                 ForkJoinPool fjPool) {
  ...
  for (T t : list)
    forks.add
      (new RecursiveTask<T>() {
        protected T compute()
        { return op.apply(t); }
      }.fork());

  for (ForkJoinTask<T> task : forks)
    results.add(task.join());
  ...
```

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
    - But it incurs more work stealing

```
Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest
```

Tests were conducted on a 2.6 GHz 6-core Lenovo laptop with 64 MBs RAM.

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
    - But it incurs more work stealing
      - Which lowers performance

```
Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest
```

Evaluating the Example Applications

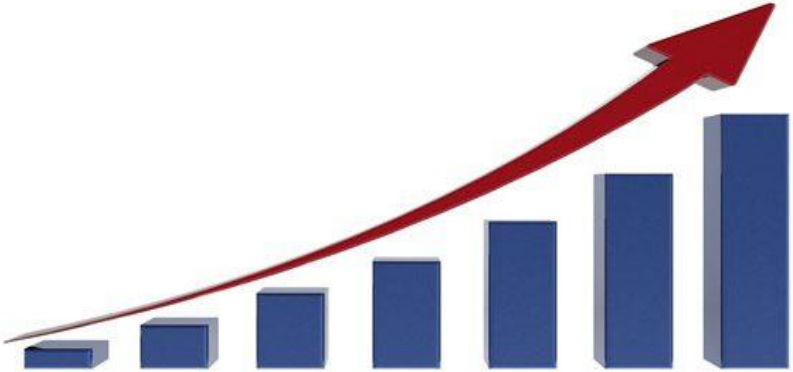
- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
  - Recursive decomposition incurs fewer "steals."

```
Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest
```

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
  - Recursive decomposition incurs fewer "steal."
    - Which improves performance

```
Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest
```



There are also other factors (e.g., less data copying) that improve performance.

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
  - Recursive decomposition incurs fewer "steal."
    - Which improves performance
    - But is more complicated to program

```
class SplitterTask extends
      RecursiveTask<List<T>> {
  protected List<T> compute() {
    ...
    int mid = mList.size() / 2;
    ForkJoinTask<List<T>> lt =
      new SplitterTask(mList.subList
                       (0, mid)).fork();
    mList = mList
      .subList(mid, mList.size());
    List<T> rightResult = compute();
    List<T> leftResult = lt.join();
    leftResult.addAll(rightResult);
    return leftResult;
  } ...
```

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
  - Recursive decomposition incurs fewer "steal."
    - Which improves performance
    - But is more complicated to program
    - And also does more "work" with respective to method calls, etc.

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
  - Recursive decomposition incurs fewer "steal."
  - RecursiveAction's overhead is lower than RecursiveTask's.

```
Starting ForkJoinTest
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 16
applyAllSplit() steal count = 21
applyAllSplitIndexEx() steal count = 21
[1] Printing 4 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 4575 ms
testApplyAllSplitIndex() executed in 5145 ms
testApplyAllSplit() executed in 5172 ms
testApplyAllIter() executed in 5599 ms
[1] Finishing ForkJoinTest
```

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
  - Recursive decomposition incurs fewer "steal."
  - RecursiveAction's overhead is lower than RecursiveTask's.
    - But RecursiveAction is also more idiosyncratic

```
<T> List<T> applyAllSplitIndex
                (List<T> list,
                 Function<T, T> op,
                 ForkJoinPool fjPool) {
  T[] results = (T[]) Array
    .newInstance
      (list.get(0).getClass(),
       list.size());
  ...
```

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
  - Recursive decomposition incurs fewer "steal."
  - RecursiveAction's overhead is lower than RecursiveTask's.
    - But RecursiveAction is also more idiosyncratic
      - Especially for generics

```
<T> List<T> applyAllSplitIndex
                (List<T> list,
                 Function<T, T> op,
                 ForkJoinPool fjPool) {
  T[] results = (T[]) Array
    .newInstance
      (list.get(0).getClass(),
       list.size());
  ...
```

Evaluating the Example Applications

- Each Java fork-join programming model has pros and cons.
  - Iterative fork()/join() is simple to program/understand.
  - Recursive decomposition incurs fewer "steal."
  - RecursiveAction's overhead is lower than RecursiveTask's.
    - But RecursiveAction is also more idiosyncratic
      - Especially for generics
      - Changing the API can help!

```
<T> List<T> applyAllSplitIndexEx
            (List<T> list,
             Function<T, T> op,
             ForkJoinPool fjPool,
             T[] results) {
...
```

}

Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams.

```
<T> List<T> applyParallelStream
                (List<T> list,
                 Function<T, T> op) {
    return list

                .parallelStream()

                .map(op)

                .collect(toList());

}
```

See earlier lessons on the "*Java Parallel Streams Framework*"

}

Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams.

```
<T> List<T> applyParallelStream
                (List<T> list,
                 Function<T, T> op) {
  return list
                .parallelStream()
                .map(op)
                .collect(toList());
}
```

The params and return value are similar.

However, the parallel streams framework uses the common fork-join pool.

}

Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams.

```
<T> List<T> applyParallelStream
                (List<T> list,
                 Function<T, T> op) {
    return list
```

Convert the list to a parallel stream

```
                .parallelStream()

                .map(op)

                .collect(toList());

}
```

}

Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams.

```
<T> List<T> applyParallelStream
                (List<T> list,
                 Function<T, T> op) {
    return list

                .parallelStream()

                .map(op)

                .collect(toList());

}
```

Apply "op" function to each element of the stream

}

Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams.

```
<T> List<T> applyParallelStream
                (List<T> list,
                 Function<T, T> op) {
    return list

                .parallelStream()

                .map(op)

                .collect(toList());

}
```

Convert the transformed stream back into a list and return it to the caller

}

Evaluating the Example Applications

- Ironically, the most concise solution involves the use of parallel streams.

```
applyAllIter() steal count = 31
applyAllSplitIndex() steal count = 18
applyAllSplit() steal count = 22
applyAllSplitIndexEx() steal count = 22
applyParallelStream() steal count = 22
[1] Printing 5 results from fastest to slowest
testApplyAllSplitIndexEx() executed in 3883 msecs
testApplyAllSplitIndex() executed in 3886 msecs
testApplyAllSplit() executed in 3980 msecs
testParallelStream() executed in 3987 msecs
testApplyAllIter() executed in 5402 msecs
```

```
<T> List<T> applyParallelStream
                (List<T> list,
                 Function<T, T> op) {
    return list
                .parallelStream()
                .map(op)
                .collect(toList());
}
```

The parallel stream version performs relatively well and is much easier to program!

The End