The Java ScheduledExecutor Service (Part 1)

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

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Recognize the features provided by the Java ScheduledExecutorService interface & its related interfaces/classes

### Interface ScheduledExecutorService

**All Superinterfaces:**
Executor, ExecutorService

**All Known Implementing Classes:**
ScheduledThreadPoolExecutor

```java
public interface ScheduledExecutorService
extends ExecutorService

An ExecutorService that can schedule commands to run after a given delay, or to execute periodically.

The schedule methods create tasks with various delays and return a task object that can be used to cancel or check execution. The scheduleAtFixedRate and scheduleWithFixedDelay methods create and execute tasks that run periodically until cancelled.

Commands submitted using the Executor.execute(Runnable) and ExecutorService.submit methods are scheduled with a requested delay of zero. Zero and negative delays (but not periods) are also allowed in schedule methods, and are treated as requests for immediate execution.
```
Overview of the ScheduledExecutor Service Interface
Overview of the ScheduledExecutorService Interface

- Extends ExecutorService to schedule commands to run after a given delay or to execute periodically

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html
Overview of the ScheduledExecutorService Interface

- ScheduleThreadPoolExecutor implements ScheduledExecutorService & is useful when multiple worker threads are needed or when the additional flexibility or capabilities of ThreadPoolExecutor is needed.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html
Overview of the ScheduledExecutorService Interface

- Key methods in the ScheduledExecutorService interface

```java
interface ScheduledExecutorService extends ExecutorService {
    public <V> ScheduledFuture<V>
        schedule(Callable<V> callable,
                 long delay, TimeUnit unit);

    public ScheduledFuture<?>
        schedule(Runnable command,
                 long delay, TimeUnit unit);

    public ScheduledFuture<?>
        scheduleAtFixedRate(Runnable
                          command, long initialDelay,
                          long period, TimeUnit unit);

    public ScheduledFuture<?>
        scheduleWithFixedDelay(Runnable
                           command, long initialDelay,
                           long delay, TimeUnit unit);
}
```

All schedule() methods accept relative delays & periods as arguments, not absolute times or dates.
Overview of the ScheduledExecutorService Interface

• Key methods in the ScheduledExecutorService interface
• Create tasks with various delays

```
interface ScheduledExecutorService
    extends ExecutorService {
    public <V> ScheduledFuture<V>
        schedule(Callable<V> callable,
                 long delay, TimeUnit unit);

    public ScheduledFuture<?>
        schedule(Runnable command,
                 long delay, TimeUnit unit);
...
```

Tasks can be two-way (callable) or one-way (Runnable)
Overview of the `ScheduledExecutorService` Interface

- Key methods in the `ScheduledExecutorService` interface
  - Create tasks with various delays
  - Returns an object that can cancel or check execution status

```java
interface ScheduledExecutorService extends ExecutorService {
    public <V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, TimeUnit unit);
    public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);
    ...
}
```

A `ScheduledFuture` represents pending completion of the task and whose `get()` method will return null upon completion.

```
get() always returns null on a `ScheduledFuture` returned for a `Runnable`!
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledFuture.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledFuture.html)
Overview of the ScheduledExecutorService Interface

• Key methods in the ScheduledExecutorService interface
• Create tasks with various delays
  • Returns an object that can cancel or check execution status
• Some clever design techniques may be needed to cancel the runnable command..

```java
class DelegatingRunnable implements Runnable {
    /**
     * The actual runnable delegated to by the run() hook method.
     */
    Runnable mActualRunnable;

    /**
     * Delegate to the underlying runnable.
     */
    public void run() {
        mActualRunnable.run();
    }
}
```

See next part of this lesson for details in the context of the TimedMemoizer class
Overview of the ScheduledExecutorService Interface

- Key methods in the ScheduledExecutorService interface
  - Create tasks with various delays
  - Create & execute tasks that run periodically until cancelled

```java
interface ScheduledExecutorService
    extends ExecutorService {

    public ScheduledFuture<?>
        scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit);

    public ScheduledFuture<?>
        scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit);

    ...

```
Overview of the ScheduledExecutorService Interface

- Key methods in the ScheduledExecutorService interface
  - Create tasks with various delays
  - Create & execute tasks that run periodically until cancelled
  - Next execution starts immediately if the last one takes longer than period

```java
interface ScheduledExecutorService extends ExecutorService {
    ...

    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit);

    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit);

    ...
}
```
Overview of the ScheduledExecutorService Interface

- Key methods in the ScheduledExecutorService interface
  - Create tasks with various delays
  - Create & execute tasks that run periodically until cancelled
  - Next execution starts immediately if the last one takes longer than period
  - Next execution starts after delay time between termination of one execution & commencement of next

```java
interface ScheduledExecutorService extends ExecutorService {
    ...
    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit);

    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit);
    ...
}
```
Overview of the ScheduledExecutorService Interface

- Key methods in the ScheduledExecutorService interface
  - Create tasks with various delays
  - Create & execute tasks that run periodically until cancelled
  - Next execution starts immediately if the last one takes longer than period
  - Next execution starts after delay time between termination of one execution & commencement of next
  - Again, these methods return an object that can cancel or check execution status

```java
interface ScheduledExecutorService extends ExecutorService {

    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit);

    public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit);

    ...

}```
End of the Java Scheduled ExecutorService (Part 1)
The Java Scheduled ExecutorService (Part 2)

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

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Recognize the features provided by the Java ScheduledExecutorService interface & its related interfaces/classes

• Know the key methods provided by the Java ScheduledExecutorService

• Learn how to program a “PrimeChecker” app using the Java ScheduledExecutorService interface
Overview of the PrimeChecker App
Overview of the PrimeChecker App

• This “embarrassingly parallel” app shows how to use the Java ExecutorCompletionService framework to determine if \( N \) random numbers are prime.

See [github.com/douglasraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeScheduledExecutorService](https://github.com/douglasraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeScheduledExecutorService)
Overview of the PrimeChecker App

- This app uses a Java ExecutorService implemented with a fixed-size thread pool tuned to the # of processor cores in the computing device.

```java
mExecutor = Executors.newFixedThreadPool (Runtime.getRuntime().availableProcessors());
```

```
public class PrimeCallable {
    private long value;

    public PrimeCallable(long value) {
        this.value = value;
    }

    public PrimeResult call() {
        // Prime check logic
        return new PrimeResult(isPrime(value));
    }
}
```

```
public class PrimeResult {
    private long primeValue;

    public PrimeResult(long primeValue) {
        this.primeValue = primeValue;
    }

    public boolean isPrime(long value) {
        // Prime check logic
        return value % 2 != 0 && value > 1;
    }
}
```

```
public class CompletionRunnable {
    private MainActivity activity;
    private int status;

    public CompletionRunnable(MainActivity activity, int status) {
        this.activity = activity;
        this.status = status;
    }

    public void run() {
        // Completion logic
        activity.updateStatus(status);
    }
}
```

```
public class RetainedState {
    private PrimeCallable callable;

    public RetainedState(PrimeCallable callable) {
        this.callable = callable;
    }
}
```

```
public class MainActivity {
    private RetainedState retainedState;

    public MainActivity() {
        retainedState = new RetainedState(new PrimeCallable(12));
    }

    public void onCreate(Bundle savedInstanceState) {
        // UI and thread setup
    }
}
```
PrimeCallable defines a two-way means of determining whether a number is prime by calling a function that returns 0 if it's prime or the smallest factor if it's not.

class PrimeCallable {
    mFunction<Long, Long> mPrimeChecker;
    ...

    PrimeCallable(Long primeCandidate,
                  Function<Long, Long> primeChecker) {
        mPrimeChecker = primeChecker;
    }

    PrimeResult call() {
        return new PrimeResult
             (mPrimeCandidate,
              mPrimeChecker.apply
               (mPrimeCandidate));
    }
}

The actual function to use for computing primality is passed as a parameter.
PrimeCallable is used to match the prime number candidate with the result of checking for primality by calling a function that returns 0 if it’s prime or smallest factor if it’s not.

```java
class PrimeCallable {
    mFunction<Long, Long>
    mPrimeChecker;
...

    PrimeCallable(Long primeCandidate, 
        Function<Long, Long> primeChecker) {
        mPrimeChecker = primeChecker;
    }

    PrimeResult call() {
        return new PrimeResult
            (mPrimeCandidate, 
             mPrimeChecker.apply
             (mPrimeCandidate));
    }
}
```

PrimeResult is used to match the prime number candidate with the result of checking for primality.
MainActivity creates a “memoizer” that is used to optimize checking the primality of “count” random numbers

```java
Function<Long, Long> primeMemoizer =
    new TimedMemoizer<>
    (PrimeCheckers::bruteForceChecker,
     count * 500);

new Random()
    .longs(count, 0,
        Integer.MAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable
        (ranNum,
            primeMemoizer))
    .forEach(callable ->
        mRetainedState.
            mExecutorService
                ::submit);
```

TimedMemoizer enables transparent optimization of the PrimeCallable class & automatically times out cache entries after count * 0.5 seconds.
• MainActivity creates a thread that waits for all future results in the background so it doesn’t block the UI thread

    mRetainedState.mCompletionRunnable =
    new CompletionRunnable(this, count);

    mRetainedState.mThread =
    new Thread(mRetainedState.
    mRetainedCompletionRunnable);
    mRetainedState.mThread.start();

CompletionRunnable is stored in a field so it can be updated during a runtime configuration change
Overview of the PrimeChecker App

• CompletionRunnable provides a means for getting the results as the futures complete...

    private class CompletionRunnable implements Runnable {
        int mCount;
        MainActivity mActivity; ...

        public void run() {
            for (int i = 0; i < mCount; ++i) {
                PrimeResult pr = ... 
                mExecutorCompletionService.take().get();

                if (pr.mSmallestFactor != 0) ... 
                else ...

                ... 
                mActivity.done();
            ...
        }
    }

The take().get() call does not block synchronously since the async processing associated with that future has completed...
Overview of the PrimeChecker App

RetainedState maintains key concurrency state across runtime configuration changes

class RetainedState {
    ExecutorCompletionService mExecutorCompletionService;
    ExecutorService mExecutorService;
    CompletionRunnable mCompletionRunnable;
    Thread mThread;
}

void onCreate(...) {
    mRetainedState = (RetainedState) getLastNonConfigurationInstance();
    if (mRetainedState != null) {
    
if (mRetainedState != null) { ... 

Object onRetainNonConfigurationInstance() {
    return mRetainedState; }

See developer.android.com/reference/android/app/Activity.html#onRetainNonConfigurationInstance()
Evaluating the PrimeChecker App
Evaluating the PrimeChecker App

- Java ScheduledExecutorService fixes a problem with the ExecutorCompletionService PrimeCheck app
Evaluating the PrimeChecker App

- Java ScheduledExecutorService fixes a problem with the ExecutorCompletionService PrimeCheck app, e.g.
- If a TimedMemoizer is used over time for a wide range of inputs it uses ScheduledExecutorService to clean itself up periodically

```
<<Java Class>>
TimedMemoizer<K,V>

TimedMemoizer(Function<K,V>,long)
apply(K)
computeValue(K):RefCountedFutureTask<V>
scheduleTimeout(K):void
getFutureValue(K,RefCountedFutureTask<V>)
```

```
RefCountedFutureTask<V>

RefCountedFutureTask(Callable<V>,long)
equals(Object):boolean
get()
```

This cleanup keeps memory usage from expanding indefinitely
Applying Scheduled ExecutorService to Memoizer
Applying ScheduledExecutorService to Memoizer

- TimedMemoizer uses ConcurrentHashMap, FutureTask, & ScheduledExecutorService to limit the amount of time a key/value is retained in the cache.
Applying ScheduledExecutorService to Memoizer

- TimedMemoizer uses ConcurrentHashMap, FutureTask, & ScheduledExecutorService to limit the amount of time a key/value is retained in the cache
- It defines multiple private methods to aid readability
RefCountedFutureTask extends FutureTask to keep track of the # of times a key is referenced within a given # of millisecs.
Applying ScheduledExecutorService to Memoizer

• Keep track of the # of times a key is referenced within a given # of millisecs

class RefCountedFutureTask<V> extends FutureTask<V> {
  /** Tracks # of times a key is referenced. */
  final AtomicLong mRefCount;

  /** Constructor initializes superclass & field. */
  RefCountedFutureTask(Callable<V> callable, int count) {
    super(callable); mRefCount = new AtomicLong(count);
  }

  /** Returns true if the ref counts are equal. */
  public boolean equals(Object obj) {
    if (getClass() != obj.getClass()) return false;
    else {
      RefCountedFutureTask<V> t = (RefCountedFutureTask<V>) obj;
      return mRefCount.get() == t.mRefCount.get();
    }
  }
}

...
Applying ScheduledExecutorService to Memoizer

- Keep track of the # of times a key is referenced within a given # of millisecs

```java
class RefCountedFutureTask<V> extends FutureTask<V> {
    ...
    /** Waits for the computation to complete, retrieves results, * & increments ref count atomically. */
    public V get() throws ... {
        // Block until value is computed.
        V value = super.get();

        // Increment ref count atomically.
        mRefCount.getAndIncrement();

        // Return the value;
        return value;
    }
}
```
TimedMemoizer uses RefCountedFutureTask to ensure only one call to the function is run when a key & value is first added to the cache.

```java
public class TimedMemoizer<K, V> implements Function<K, V> {
    /** Associate a key with a value produced by a function. */
    private final ConcurrentMap<K, RefCountedFutureTask<V>> mCache = new ConcurrentHashMap<>();

    /** This function produces a value based on the key. */
    private final Function<K, V> mF;

    /** The amount of time to retain a value in the cache. */
    private final long mTimeoutInMillisecs;

    /** Executor that removes expired keys after timeout. */
    private ScheduledExecutorService mScheduledExecutorService = Executors.newScheduledThreadPool(1);

    /** Constructor initializes the fields. */
    public Memoizer(Function<K, V> function, long timeout) {
        mF = function; mTimeoutInMillisecs = timeout;
    }
    ...
```
Applying ScheduledExecutorService to Memoizer

- Returns the value associated with the key in cache

```java
public class TimedMemoizer<K, V> implements Function<K, V> {
    ...
    public V apply(K key) {
        // Try to find the key in the cache.
        RefCountedFutureTask<V> future = mCache.get(key);

        // If the key isn't present then compute its value.
        if (future == null)
            future = computeValue(key);

        // Return the value of the future, blocking until it's
        // computed.
        return getFutureValue(key, future);
    }
    ...
```
Applying ScheduledExecutorService to Memoizer

- Returns the value associated with the key in cache

```java
class TimedMemoizer<K, V> implements Function<K, V> {
    ...
    private RefCountedFutureTask<V> computeValue(K key) {
        // The RefCountedFutureTask computes & stores the value.
        final RefCountedFutureTask<V> futureTask =
            new RefCountedFutureTask<>((() -> mFunction.apply(key), 0);

        // Try putting futureTask in cache as the value of key.
        RefCountedFutureTask<V> f = mCache.putIfAbsent(key,
            futureTask);

        // The value's already in the cache.
        if (f != null) return f;
        else {
            futureTask.run(); // Compute the value

            // Use ScheduledExecutorService to remove key from cache
            // if it times out & hasn't been accessed recently.
            scheduleTimeout(key);

            return futureTask; // Return the future.
        }
    }
```
public class TimedMemoizer<K, V> implements Function<K, V> {
    ...
    private void scheduleTimeout(K key) {
        // Only schedule timeouts under right conditions.
        if (!Thread.currentThread().isInterrupted()
            && mTimeoutInMillisecs > 0) {
            // A ref count of 1 indicates key hasn't been accessed
            // in mTimeoutInMillisecs.
            final RefCountedFutureTask<V> nonAccessedValue =
                new RefCountedFutureTask<>()
                    (() -> mFunction.apply(key), 1);

            // Create a runnable that will remove stale entries.
            final Runnable removeIfStale = new Runnable() {
                @Override
                public void run() {
                    ...
                }
            };
    }
    ...
    // Schedule removal of key from cache if it isn’t access recently

Applying ScheduledExecutorService to Memoizer

- Schedule removal of key from cache if it isn’t accessed recently

```java
public class TimedMemoizer<K, V> implements Function<K, V> {
    ...
    private void scheduleTimeout(K key) {
        ...
        // Remove key only if not accessed in mTimeoutInMillisecs.
        if (!mCache.remove(key, nonAccessedValue)) {
            // Reset key's value so it won't be viewed as accessed.
            mCache.replace(key, nonAccessedValue);

            // Reschedule runnable to run in mTimeoutInMillisecs.
            mScheduledExecutorService.schedule(this,
                mTimeoutInMillisecs, TimeUnit.MILLISECONDS);
        }
    }
    ...

    // Schedule runnable to execute after mTimeoutInMillisecs.
    mScheduledExecutorService.schedule(removeIfStale,
        mTimeoutInMillisecs, TimeUnit.MILLISECONDS);
}
```
Applying ScheduledExecutorService to Memoizer

- Another way to schedule removal of key from cache if it isn’t access recently

```java
public class TimedMemoizer<K, V> implements Function<K, V> {
    ...  
    private void scheduleAtFixedRateTimeout(K key) {
        // Only schedule timeouts under right conditions.
        if (!Thread.currentThread().isInterrupted() && mTimeoutInMillisecs > 0) {
            // A ref count of 1 indicates key hasn’t been accessed
            // in mTimeoutInMillisecs.
            final RefCountedFutureTask<V> nonAccessedValue =
                new RefCountedFutureTask<>(
                    () -> mFunction.apply(key), 1);

            // Decouples scheduling of runnable from logic invoked
            // when the runnable is dispatched.
            class DelegatingRunnable implements Runnable {
                // The actual runnable delegated to by run().
                Runnable mActualRunnable;

                // Delegate to the underlying runnable.
                public void run() { mActualRunnable.run(); }
            } ...
        }
    }
```
Applying ScheduledExecutorService to Memoizer

- Another way to schedule removal of key from cache if it isn't access recently

```java
public class TimedMemoizer<K, V> implements Function<K, V> {
    ...
    private void scheduleAtFixedRateTimeout(K key) {
        DelegatingRunnable dRun = new DelegatingRunnable();

        // Schedule dRun to execute after given timeout.
        final ScheduledFuture<?> cancellableFuture =
            mScheduledExecutorService.scheduleAtFixedRate(dRun, mTimeoutInMillisecs,
                mTimeoutInMillisecs, TimeUnit.MILLISECONDS);

        // Runnable remove the key & futureTask from the cache
        dRun.mActualRunnable = () -> {
            // Remove key if it hasn't been accessed recently.
            if (mCache.remove(key, nonAccessedValue))
                // Stop ScheduledExecutorService from dispatching dRun.
                cancellableFuture.cancel(true);
            else
                // Reset key's value so it won't be considered accessed.
                mCache.replace(key, nonAccessedValue);
        };
    }
}...
```
Applying ScheduledExecutorService to Memoizer

- Return the value of the future, blocking until it's computed

```java
public class TimedMemoizer<K, V> implements Function<K, V> {
    ...
    private V getFutureValue(K key,
        RefCountedFutureTask<V> future) {
        try {
            // Get the result of the future, which will block if the
            // futureTask hasn't finished running yet.
            return future.get();
        } catch (Exception e) {
            // Unilaterally remove the key from the cache when an
            // exception occurs.
            mCache.remove(key);

            // Rethrow the exception.
            throw e;
        }
    }
    ...
```
End of Overview of Java ScheduledExecutorService (Part 2)