Evaluating the Pros & Cons of Java FutureTask

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

Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand how Java FutureTask conveys a result from a computation running in a thread to thread(s) retrieving the result
- Recognize key methods in Java FutureTask
- Know what a Memoizer is & motivates how FutureTask can optimize its performance in concurrent programs
- Learn how to implement the Memoizer cache with FutureTask

- Recognize how the Memoizer class is applied to the PrimeChecker app to optimize prime # checking
- Evaluate the pros & cons of the Prime Checker app implementation & FutureTask

Evaluating the PrimeChecker App

Evaluating the PrimeChecker App

- The FutureTask version of the PrimeChecker app fixes a limitation with the previous version

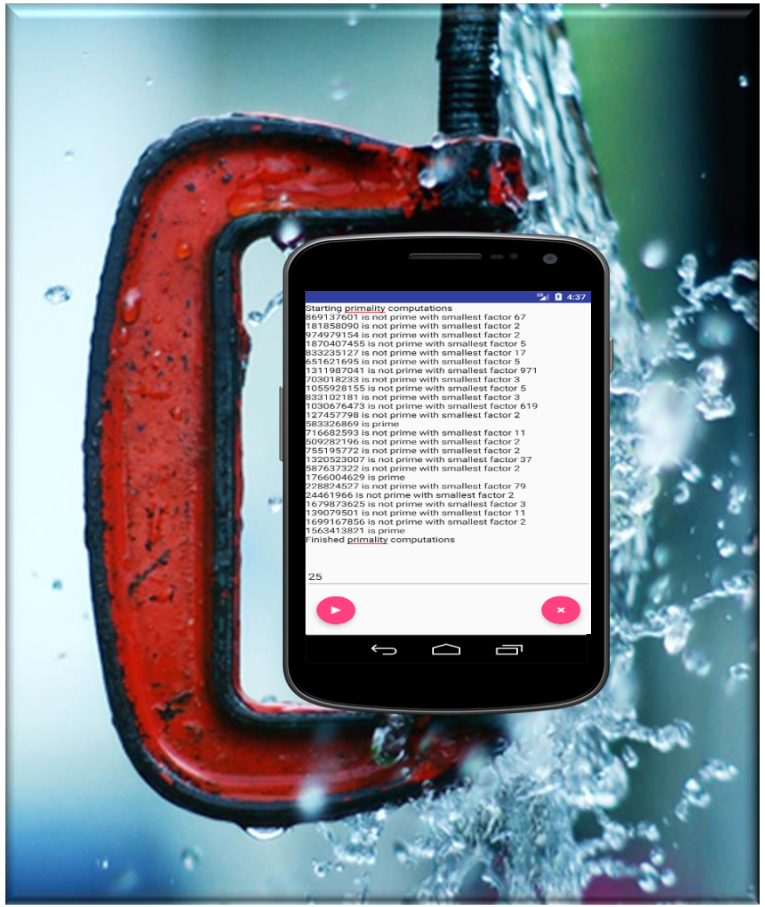
See earlier lessons on "Java ExecutorCompletionService"

Evaluating the PrimeChecker App

- The FutureTask version of the PrimeChecker app fixes a limitation with the previous version, e.g.
  - The Memoizer implementation no longer depends on ConcurrentHashMap features only available in Java 8 & beyond

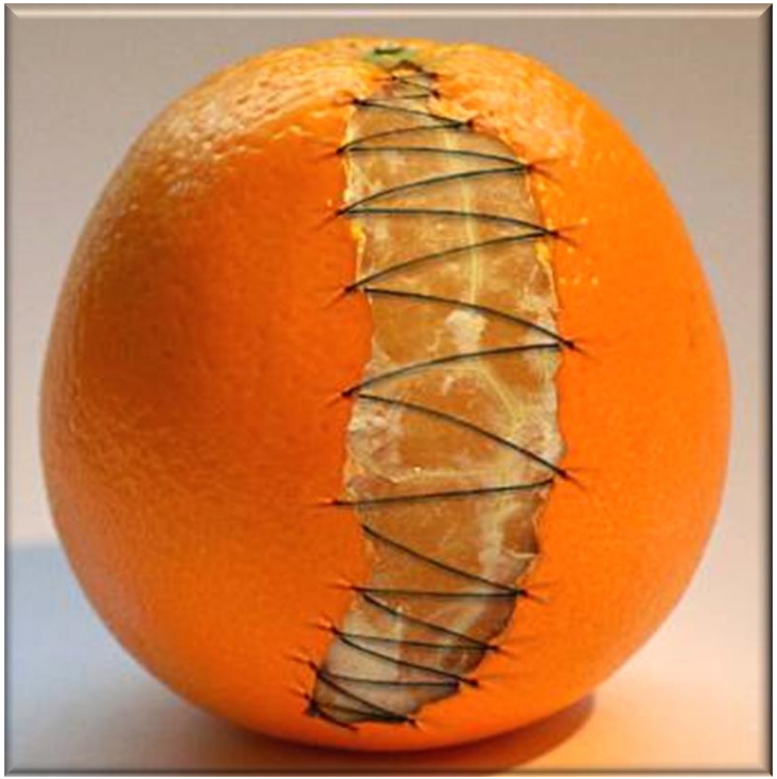
Evaluating the PrimeChecker App

- However, there is still a limitation



Evaluating the PrimeChecker App

- However, there is still a limitation, e.g.
  - If the Memoizer is used for a long period of time for a wide range of inputs it will continue to grow & never clean itself up!



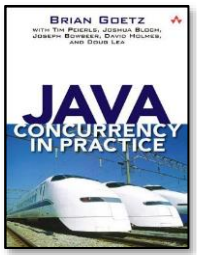
We fix this limitation in the lessons on the "*Java ScheduledExecutorService*"!

Evaluating Java FutureTask

Evaluating Java FutureTask

- Java 8's ConcurrentHashMap.computeIfAbsent() reduces need for FutureTask

```
private Future<V> computeValue(K key) {
  FutureTask<V> ft = new FutureTask<>(() -> mF.apply(key));
  Future<V> future = mCache.putIfAbsent(key,
                                        futureTask);
  if (future != null) return future;
  else { futureTask.run(); return futureTask; }
}
```



All threads block if value's not been completed by first task, & after it's completed, the blocked threads will unblock & any future threads calling the method won't block either

```
public V apply(final K key) {
  return mCache.computeIfAbsent(key, mFunction::apply);
}
```

See jcip.net

Evaluating Java FutureTask

- Java 8's ConcurrentHashMap.computeIfAbsent() reduces need for FutureTask

```
private Future<V> computeValue(K key) {
  FutureTask<V> ft = new FutureTask<>(() -> mF.apply(key));
  Future<V> future = mCache.putIfAbsent(key,
                                        futureTask);
  if (future != null) return future;
  else { futureTask.run(); return futureTask; }
}
```

All threads block if value's not been completed by first task, & after it's completed, the blocked threads will unblock & any future threads calling the method won't block either

```
public V apply(final K key) {
  return mCache.computeIfAbsent(key, mFunction::apply);
}
```

See ashkrit.blogspot.com/2014/12/what-is-new-in-java8-concurrenthashmap.html

Evaluating Java FutureTask

- Java 8's ConcurrentHashMap.computeIfAbsent() reduces need for FutureTask

```
private Future<V> computeValue(K key) {
  FutureTask<V> ft = new FutureTask<>(() -> mF.apply(key));
  Future<V> future = mCache.putIfAbsent(key,
                                        futureTask);
  if (future != null) return future;
  else { futureTask.run(); return futureTask; }
}
```

However, computeIfAbsent() only works if you're using Java 8 or beyond – otherwise you'll need to understand/use FutureTask!!

```
public V apply(final K key) {
  return mCache.computeIfAbsent(key, mFunction::apply);
}
```

End of Evaluating the Pros & Cons of Java FutureTask