Java ExecutorCompletionService: Evaluating Pros & Cons

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Understand how the Java CompletionService interface defines a framework for handling the completion of asynchronous tasks
• Know how to instantiate the Java ExecutorCompletionService
• Recognize key methods in the Java CompletionService interface
• Visualize the ExecutorCompletionService in action
• Be aware of how the Java ExecutorCompletionService implements the CompletionService interface
• Know how to apply the Java ConcurrentHashMap class to design a “memoizer”
• Master how to implement the Memoizer class with Java ConcurrentHashMap
• See how Java ExecutorCompletionService & Memoizer are integrated into the “PrimeChecker” app
• Evaluate the pros & cons of this PrimeChecker app implementation
Evaluating this PrimeChecker App
Evaluating this PrimeChecker App

• This PrimeChecker implementation fixes problems w/the earlier versions
Evaluating this PrimeChecker App

- This PrimeChecker implementation fixes problems with the earlier versions, e.g.
- Futures are processed as they complete

```java
private class CompletionRunnable implements Runnable {
    int mCount; ...

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

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

This benefit stems from ExecutorCompletionService's "async future" processing model
Evaluating this PrimeChecker App

- This PrimeChecker implementation fixes problems w/the earlier versions, e.g.
  - Futures are processed as they complete
  - Memoizer enables transparent optimization w/out changing PrimeCallable

```java
mMemoizer = new Memoizer<>
  (PrimeCheckers::bruteForceChecker,
   new ConcurrentHashMap());
new Random()
  .longs(count, sMAX_VALUE - count,
         sMAX_VALUE)
  .mapToObj(ranNum -> new PrimeCallable(ranNum, mMemoizer))
  .forEach(callable ->
    mRetainedState.mExecutorCompService::submit);
```

Memoizer can be used wherever a Function is expected
Evaluating this PrimeChecker App

- This PrimeChecker implementation fixes problems w/the earlier versions, e.g.
  - Futures are processed as they complete
  - Memoizer enables transparent optimization w/out changing PrimeCallable

```java
mMemoizer = new Memoizer<>
    (PrimeCheckers::efficientChecker,
     new ConcurrentHashMap());
new Random()
    .longs(count, sMAX_VALUE - count, 
           sMAX_VALUE)
    .mapToObj(ranNum -> new PrimeCallable(ranNum, mMemoizer))
    .forEach(callable ->
        mRetainedState.mExecutorCompService::submit); ...
```

bruteForceChecker() can easily be replaced with a different method reference
Evaluating this PrimeChecker App

• However, there are still limitations
Evaluating this PrimeChecker App

- However, there are still limitations, 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 upcoming lesson on the “Java ScheduledExecutorService”
• However, there are still limitations, 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!
  • This implementation of Memoizer depends on ConcurrentHashMap features available only with Java 8 & beyond

We fix this limitation in the upcoming lesson on the "Java FutureTask"
End of Java Executor
CompletionService:
Evaluating Pros & Cons