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 ExecutorCompletion Service
• Recognize key methods in the Java CompletionService interface
• Visualize the ExecutorCompletionService in action
• Be aware of how the Java ExecutorCompletion Service implements the CompletionService interface
• 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
• This PrimeChecker implementation fixes problems w/the earlier versions, e.g.
  • Futures are processed as they complete

    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 with the earlier versions, e.g.
  - Futures are processed as they complete
  - Memoizer enables transparent optimization without 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
This PrimeChecker implementation fixes problems with the earlier versions, e.g.

- Futures are processed as they complete
- Memoizer enables transparent optimization without 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);
...
```

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

• However, there are still some limitations
Evaluating this PrimeChecker App

- However, there are 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 problem in the upcoming lesson on the “Java ScheduledExecutorService”!
End of Java Executor CompletionService: Evaluating Pros & Cons