Applying the Java ExecutorCompletion Service to the PrimeChecker App

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
Applying Memoizer to Check for Prime #’s
Applying Memoizer to Check for Prime #’s

This app shows how Java’s ExecutorCompletionService framework & the modern Java-based Memoizer can be used to check if $N$ random #’s are prime.

See ex/M4/Primes/PrimeExecutorCompletionService
Applying Memoizer to Check for Prime #’s

- This app shows how Java’s ExecutorCompletionService framework & the modern Java-based Memoizer can be used to check if $N$ random #’s are prime
- This app is “embarrassingly parallel” & compute-bound

Applying Memoizer to Check for Prime #'s

- MainActivity checks primality of “count” random #'s via an ExecutorService w/a thread pool & the PrimeCallable class

```
mExecutor = Executors.newCachedThreadPool();
```

The executor service uses a cached (variable-sized) pool of threads

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool)
PrimeCallable defines a two-way means of determining whether a # is prime by calling a function that returns 0 if it’s prime or smallest factor if it’s not prime.

```java
class PrimeCallable implements Callable<PrimeResult> {
    Function<Long, Long> mPrimeChecker;
    ...

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

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

Applying Memoizer to Check for Prime #'s
PrimeCallable defines a two-way means of determining whether a # is prime by calling a function that returns 0 if it’s prime or smallest factor if it’s not prime.

```java
class PrimeCallable implements Callable<PrimeResult> {
    Function<Long, Long> mPrimeChecker;
    ...

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

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

Applying Memoizer to Check for Prime #'s

Implements Callable to run in a pool thread
PrimeCallable defines a two-way means of determining whether a # is prime by calling a function that returns 0 if it’s prime or smallest factor if it’s not

class PrimeCallable implements Callable<PrimeResult> {
    Function<Long, Long> mPrimeChecker;
    ...

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

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

The function computing primality is parameterized

Applying Memoizer to Check for Prime #'s

The function computing primality is parameterized

This Function param is a new feature added since the earlier PrimeCheck example
PrimeCallable defines a two-way means of determining whether a # is prime by calling a function that returns 0 if it’s prime or smallest factor if it’s not prime.

class PrimeCallable implements Callable<PrimeResult> {
    Function<Long, Long> mPrimeChecker;
    ...

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

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

Applying Memoizer to Check for Prime #'s

This hook method is called in a pool thread.
PrimeCallable defines a two-way means of determining whether a # is prime by calling a function that returns 0 if it’s prime or smallest factor if it’s not:

class PrimeCallable implements Callable<PrimeResult> {
    Function<Long, Long> mPrimeChecker;
    ...

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

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

    This function performs the prime # check.
PrimeCallable defines a two-way means of determining whether a # is prime by calling a function that returns 0 if it’s prime or smallest factor if it’s not prime.

class PrimeCallable implements Callable<PrimeResult> {
    Function<Long, Long> mPrimeChecker;
    ...
    PrimeCallable(Long primeCandidate, Function<Long, Long> pc) {
        mPrimeChecker = pc;
    }
    PrimeResult call() {
        return new PrimeResult(
            mPrimeCandidate, mPrimeChecker.apply(mPrimeCandidate)
        );
    }
}

Applying Memoizer to Check for Prime #'s

Match prime # candidate with primality check result
MainActivity creates a Memoizer that optimizes primality checking of “count” random #’s

See src/main/java/vandy/mooc/prime/utils/Memoizer.java
• Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>(
    PrimeCheckers::bruteForceChecker,
    new ConcurrentHashMap());

new Random()
    .longs(count,
        sMAX_VALUE - count,
        sMAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable(ranNum, mMemoizer))
    .forEach(mRetainedState.mExecutorCompService::submit); ...
```
• Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>(
    PrimeCheckers::bruteForceChecker,
    new ConcurrentHashMap());
```

This memoizer caches prime # results

```java
new Random()
  .longs(count,
    sMAX_VALUE - count,
    sMAX_VALUE)
  .mapToObj(ranNum ->
    new PrimeCallable(ranNum, mMemoizer))
  .forEach(mRetainedState.mExecutorCompService::submit);
```
Applying Memoizer to Check for Prime #'s

- Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>(
    PrimeCheckers::bruteForceChecker,
    new ConcurrentHashMap());
```

It's easy to change prime # checker from this..

```java
new Random()
    .longs(count, sMAX_VALUE - count, sMAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable(ranNum, mMemoizer))
    .forEach(mRetainedState.mExecutorCompService::submit); ...
```

See blog.indrek.io/articles/java-8-behavior-parameterization
Applying Memoizer to Check for Prime #'s

- Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>
    (PrimeCheckers::efficientChecker,
     new ConcurrentHashMap());

new Random()
    .longs(count,
             sMAX_VALUE - count,
             sMAX_VALUE)
    .mapToObj(ranNum ->
               new PrimeCallable(ranNum, mMemoizer))
    .forEach(mRetainedState.mExecutorCompService::submit);
```

See blog.indrek.io/articles/java-8-behavior-parameterization
Applying Memoizer to Check for Prime #'s

- Memoizer caches results when processing a stream of PrimeCallables

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

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html
Applying Memoizer to Check for Prime #'s

- Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>
(PrimeCheckers::efficientChecker,
 new ConcurrentHashMap());
```

```java
new Random()
 .longs(count,
      sMAX_VALUE - count,
      sMAX_VALUE)
 .mapToObj(ranNum ->
      new PrimeCallable(ranNum, mMemoizer))
 .forEach(mRetainedState.mExecutorCompService::submit); ... 
```

Generates “count” random #'s between $sMAX\_VALUE - count$ & $sMAX\_VALUE$
Applying Memoizer to Check for Prime #'s

- Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>()
    (PrimeCheckers::efficientChecker,
    new ConcurrentHashMap());

new Random()
    .longs(count,
        sMAX_VALUE - count,
        sMAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable(ranNum, mMemoizer))
    .forEach(mRetainedState.mExecutorCompService::submit); ...
```
Applying Memoizer to Check for Prime #'s

- Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>(
    PrimeCheckers::efficientChecker,
    new ConcurrentHashMap());

new Random()
    .longs(count, sMAX_VALUE - count, sMAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable(ranNum, mMemoizer))
    .forEach(mRetainedState.mExecutorCompService::submit); ...
```

See blog.indrek.io/articles/java-8-behavior-parameterization
Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>(
    (PrimeCheckers::efficientChecker,
    new ConcurrentHashMap());

new Random()
    .longs(count,
        sMAX_VALUE - count,
        sMAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable(ranNum, mMemoizer))
    .forEach(mRetainedState.mExecutorCompService::submit); ...
```

Submit a value-returning task for execution for each prime callable

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html#submit](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html#submit)
Memoizer caches results when processing a stream of PrimeCallables

```java
mMemoizer = new Memoizer<>(
    PrimeCheckers::efficientChecker,
    new ConcurrentHashMap());
```

```java
new Random()
    .longs(count,
        sMAX_VALUE - count,
        sMAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable(ranNum, mMemoizer))
    .forEach(mRetainedState.mExecutorCompService::submit); ...
```

There’s no need for a list of futures due to the ExecutorCompletionService
Applying Memoizer to Check for Prime #’s

- MainActivity creates a thread to wait for all future results in the background so the UI thread doesn’t block...

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

CompletionRunnable is stored in a field so it can be updated during a runtime configuration change

```java
mRetainedState.mThread = new Thread (mRetainedState.mCompletionRunnable);
mRetainedState.mThread.start();
```
Applying Memoizer to Check for Prime #'s

- MainActivity creates a thread to wait for all future results in the background so the UI thread doesn’t block

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

mRetainedState.mThread = new Thread(mRetainedState.mCompletionRunnable);

mRetainedState.mThread.start();
```

A new thread is created/started to execute the CompletionRunnable
Applying Memoizer to Check for Prime #'s

- CompletionRunnable gets results as futures complete

```java
class CompletionRunnable implements Runnable {
    int mCount;
    MainActivity mActivity; ...

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

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

See `src/main/java/vandy/mooc/prime/activities/PrimeCallable.java`
Applying Memoizer to Check for Prime #'s

- CompletionRunnable gets results as futures complete

```java
class CompletionRunnable implements Runnable {
    int mCount;
    MainActivity mActivity; ...

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

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

See [docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html](http://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html)
Applying Memoizer to Check for Prime #’s

- CompletionRunnable gets results as futures complete

```java
class CompletionRunnable implements Runnable {
    int mCount;
    MainActivity mActivity; ...

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

            if (pr.mSmallestFactor != 0) ... else ...
            ...
            mActivity.done(); ...
```

Iterate thru all results
Applying Memoizer to Check for Prime #’s

- CompletionRunnable gets results as futures complete

```java
class CompletionRunnable implements Runnable {
    int mCount;
    MainActivity mActivity; ...

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

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

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

get() doesn’t block, though take() may block if no completed futures are yet available
Applying Memoizer to Check for Prime #’s

- CompletionRunnable gets results as futures complete
  ```java
  class CompletionRunnable implements Runnable {
      int mCount;
      MainActivity mActivity; ...
  }
  
  public void run() {
      for (int i = 0; i < mCount; ++i) {
          PrimeResult pr = ...
          mExecutorCompService .take().get();

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

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

Process & output results
Applying Memoizer to Check for Prime #'s

- RetainedState maintains key concurrency state across runtime configuration changes

```java
class RetainedState {
    ExecutorCompletionService mExecutorCompService;
    ExecutorService mExecutorService;
    CompletionRunnable mCompletionRunnable;
    Thread mThread;
    Memoizer<Long, Long> mMemoizer;
}
```

See android.jlelse.eu/handling-orientation-changes-in-android-7072958c442a
Applying Memoizer to Check for Prime #'s

- RetainedState maintains key concurrency state across runtime configuration changes

```java
void onCreate(...) {
    mRetainedState = (RetainedState)
    getLastNonConfigurationInstance();

    if (mRetainedState != null) {
        ... // update configurations
    }
}
```

Object onRetainNonConfigurationInstance() {
    return mRetainedState;
}

Android’s activity framework dispatches these hook methods to save & restore state when runtime configuration changes occur.

See android.jlelse.eu/handling-orientation-changes-in-android-7072958c442a
End of Applying the Java ExecutorCompletionService to the PrimeChecker App