Java ExecutorCompletionService: Application to 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 Java 8-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 Java 8-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

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

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)

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

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

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

    PrimeResult call() {
        return new PrimeResult
            (mPrimeCandidate, mPrimeChecker.apply(mPrimeCandidate));
    }
```
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> {
    mFunction<Long, Long> mPrimeChecker;
    ...

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

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

This Function param is a new feature added since the earlier PrimeCheck example.
**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

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

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

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

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 prime.

```java
class PrimeCallable implements Callable<PrimeResult> {
    mFunction<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 function performs the prime # check.
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 smallest factor if it’s not.

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

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

- Memoizer caches results when processing a stream of PrimeCallable objects

```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(callable ->
        mRetainedState.mExecutorCompService::submit); ...
```

See [blog.indrek.io/articles/java-8-behavior-parameterization](http://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());
```

```java
new Random()
    .longs(count, 
     sMAX_VALUE - count,
     sMAX_VALUE)
    .mapToObj(ranNum ->
               new PrimeCallable(ranNum, mMemoizer))
    .forEach(callable ->
             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());
```

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

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html](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(callable ->
   mRetainedState.mExecutorCompService::submit); ...
```
Memoizer caches results when processing a stream of PrimeCallables

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

Transforms random #’s into PrimeCallables
Applying Memoizer to Check for Prime #'s

- Memoizer caches results when processing a stream of 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); ...
```

A Memoizer object can be used wherever a Function is expected
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(callable ->
        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
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(callable ->
             mRetainedState.mExecutorCompService::submit);
```
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

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

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

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

... 

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

A new thread is created/started to execute the CompletionRunnable

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

mRetainedState.mThread.start();
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](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](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
    }
}
```

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

```java
Object onRetainNonConfigurationInstance() {
    return mRetainedState;
}
```

See android.jlelse.eu/handling-orientation-changes-in-android-7072958c442a
End of Java Executor CompletionService: Application to PrimeChecker App