Applying the Java FutureTask-based Memoizer to the PrimeChecker App

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

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
Applying the Memoizer to Optimize Prime # Checking
Applying the Memoizer to Optimize Prime # Checking

- This app applies the Java ExecutorCompletionService framework & the FutureTask-based Memoizer to check if N random #'s are prime

See github.com/douglascraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutorServiceFutureTask
Applying the Memoizer to Optimize Prime # Checking

- This app applies the Java ExecutorCompletionService framework & the FutureTask-based Memoizer to check if $N$ random #'s are prime
- This app is “embarrassingly parallel” & compute-bound

Applying the Memoizer to Optimize Prime # Checking

- MainActivity checks primality of "count" random #'s via an ExecutorService w/a thread pool & the PrimeCallable class.
Applying the Memoizer to Optimize Prime # Checking

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

```
mExecutorService = Executors.newFixedThreadPool (Runtime.getRuntime().availableProcessors());
```

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html#newFixedThreadPool)
Applying the Memoizer to Optimize Prime # Checking

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

```java
mExecutorService = Executors.newFixedThreadPool
  (Runtime.getRuntime()
    .availableProcessors());
```

Pool size tuned to # of processor cores

See docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#availableProcessors
Applying the Memoizer to Optimize Prime # Checking

• MainActivity also uses a memoizer to optimize primality checking of the random #'s

See earlier parts of this lesson on "Application to Memoizer" & "Implementing a Memoizer"
PrimeCallable uses a Function object to extensibly determine if a number is prime.

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

    PrimeCallable(Long primeCandidate,
                  Function<Long, Long> primeChecker) {
        mPrimeCandidate = primeCandidate;
        mPrimeChecker = primeChecker;
    }
```

See PrimeExecutorServiceFutureTask/app/src/main/java/vandy/mooc/prime/activities/PrimeCallable.java
PrimeCallable uses a Function object to extensibly determine if a number is prime.

```java
class PrimeCallable
    implements Callable<PrimeResult> {
    long mPrimeCandidate;
    mFunction<Long, Long> mPrimeChecker;
    
    PrimeCallable(Long primeCandidate, 
                  Function<Long, Long> primeChecker) {
        mPrimeCandidate = primeCandidate;
        mPrimeChecker = primeChecker;
    }
}
```

PrimeCallable implements Callable so it can be submitted & run by the Java ExecutorCompletionService framework.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html)
PrimeCallable uses a Function object to extensibly determine if a number is prime.

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

    PrimeCallable(Long primeCandidate, Function<Long, Long> primeChecker) {
        mPrimeCandidate = primeCandidate;
        mPrimeChecker = primeChecker;
    }
}
```

The function that checks primes is passed as a param & stored in a field.

Applying the Memoizer to Optimize Prime # Checking
Applying the Memoizer to Optimize Prime # Checking

• PrimeCallable uses a Function object to extensibly determine if a # is prime

```java
class PrimeCallable implements Callable<PrimeResult> {
    ... 
    PrimeResult call() { 
        return new PrimeResult
            (mPrimeCandidate,
             mPrimeChecker
             .apply(mPrimeCandidate));
    }
    ... 
}
```

This method runs in the context of a thread in the fixed-sized thread pool
• PrimeCallable uses a Function object to extensibly determine if a # is prime

class PrimeCallable
    implements Callable<PrimeResult> {
        ...

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

apply() returns 0 if the # is prime or smallest factor if it’s not

The apply() method call can be transparently optimized via the Memoizer
PrimeCallable uses a Function object to extensibly determine if a # is prime

```java
class PrimeCallable
    implements Callable<PrimeResult> { ...

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

    ...
```

The PrimeResult tuple matches the prime # candidate with result of checking for primality
Applying the Memoizer to Optimize Prime # Checking

- Memoizer caches results when processing a stream of PrimeCallables

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

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

See PrimeExecutorServiceFutureTask/app/src/main/java/vandy/mooc/prime/activities/MainActivity.java
Applying the Memoizer to Optimize Prime # Checking

- Memoizer caches results when processing a stream of PrimeCallable:

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

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

See PrimeExecutorServiceFutureTask/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java
Applying the Memoizer to Optimize Prime # Checking

- Memoizer caches results when processing a stream of PrimeCallables

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

It's easy to change the 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); ...
```
Applying the Memoizer to Optimize Prime # Checking

- Memoizer caches results when processing a stream of PrimeCallables

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

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

- Memoizer caches results when processing a stream of PrimeCallables

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

```java
ew Random()
    .longs(count,
        sMAX_VALUE - count,
        sMAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable(ranNum, mMemoizer))
    .forEach(callable ->
        mRetainedState.mExecutorCompService::submit);
```
Applying the Memoizer to Optimize Prime # Checking

- Memoizer caches results when processing a stream of PrimeCallables

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

Transforms random #'s into PrimeCallables

```java
new Random()
  .longs(count,
    sMAX_VALUE - count,
    sMAX_VALUE)
  .mapToObj(ranNum ->
    new PrimeCallable(ranNum, mMemoizer))
  .forEach(callable ->
    mRetainedState.mExecutorCompService::submit);
```
Applying the Memoizer to Optimize Prime # Checking

- Memoizer caches results when processing a stream of PrimeCallables

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

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

This memoizer can be used wherever a Function is expected

See blog.indrejk.io/articles/java-8-behavior-parameterization
Applying the Memoizer to Optimize Prime # Checking

- Memoizer caches results when processing a stream of PrimeCallables

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

```java
new Random()
    .longs(count, sMAX_VALUE - count, sMAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable(ranNum, mMemoizer))
    .forEach(callable ->
        mRetainedState.mExecutorCompService::submit);
```
Applying the Memoizer to Optimize Prime # Checking

- Memoizer caches results when processing a stream of PrimeCallables

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

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

There’s no need for a list of futures due to the ExecutorCompletionService


- 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();
```
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 the Memoizer to Optimize Prime # Checking
• CompletionRunnable gets results as futures complete

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(); ...
  }
}
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 the Memoizer to Optimize Prime # Checking
Applying the Memoizer to Optimize Prime # Checking

- 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 completed futures aren’t yet available
Applying the Memoizer to Optimize Prime # Checking

- 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
RetainedState maintains key concurrency state across runtime configuration changes

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

Apply the Memoizer to Optimize Prime # Checking

See android.jlelse.eu/handling-orientation-changes-in-android-7072958c442a
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 Applying the Java FutureTask-based Memoizer to the PrimeChecker App