The Java ExecutorCompletion Service (Part 3)
Douglas C. Schmidt
d.schmidt@vanderbilt.edu
Vanderbilt University
Institute for Software Integrated Systems
Nashville, Tennessee, USA
www.dre.vanderbilt.edu/~schmidt
Learning Objectives in this Part of the Lesson

• Understand how the Java CompletionService interface defines a framework for handling the completion of asynchronous tasks

• Recognize the key methods in the Java CompletionService interface

• Know how the ExecutorCompletionService implements CompletionService

• Learn how to program a “PrimeChecker” app using the Java ExecutorCompletionService interface
Overview of the PrimeChecker App
Overview of the PrimeChecker App

• This “embarrassingly parallel” app shows how to use the Java ExecutorCompletionService framework to determine if $N$ random numbers are prime

See github.com/douglasraignschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutorCompletionService
This app uses a Java ExecutorService implemented with a fixed-size thread pool tuned to the # of processor cores in the computing device.

```java
mExecutor = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
```
Overview of the PrimeChecker App

- 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 the smallest factor if it's not.

```java
class PrimeCallable {
    mFunction<Long, Long>
    mPrimeChecker;
    ...
}

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

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

The actual function to use for computing primality is passed as a parameter.
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 {
    mFunction<Long, Long> mPrimeChecker;
    ...
    PrimeCallable(Long primeCandidate, Function<Long, Long> primeChecker) {
        mPrimeChecker = primeChecker;
    }
    PrimeResult call() {
        return new PrimeResult
        (mPrimeCandidate,
        mPrimeChecker.apply
        (mPrimeCandidate));
    }
}
```

PrimeResult is used to match the prime number candidate with the result of checking for primality.
Overview of the PrimeChecker App

- MainActivity creates a “memoizer” that is used to optimize checking the primality of “count” random numbers.

```java
Function<Long, Long> primeMemoizer =
    new Memoizer<>((PrimeCheckers::bruteForceChecker);

new Random()
    .longs(count, 0, Integer.MAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable
            (ranNum,
                primeMemoizer))
    .forEach(callable ->
        mRetainedState.
            mExecutorService
                ::submit);
...
```

The Memoizer enables transparent optimization of the PrimeCallable!
Overview of the PrimeChecker App

- MainActivity creates a thread that waits for all future results in the background so it doesn’t block the UI thread

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

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

    mRetainedState.mThread.start();

CompletionRunnable is stored in a field so it can be updated during a runtime configuration change.
Overview of the PrimeChecker App

- CompletionRunnable provides a means for getting the results as the futures complete

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

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

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

            ...  
            mActivity.done(); 
            ...

get() doesn’t block synchronously since async processing associated with that future is completed, though take() may block if completed futures are available
```
Overview of the PrimeChecker App

- RetainedState maintains key concurrency state across runtime configuration changes

```java
class RetainedState {
    ExecutorCompletionService mExecutorCompletionService;
    ExecutorService mExecutorService;
    CompletionRunnable mCompletionRunnable;
    Thread mThread;
}

void onCreate(...) {
    mRetainedState = (RetainedState)
        getLastNonConfigurationInstance();
    if (mRetainedState != null) { ...

    Object onRetainNonConfigurationInstance() {
        return mRetainedState; }

See developer.android.com/reference/android/app/Activity.html#onRetainNonConfigurationInstance()
Evaluating the PrimeChecker App
Evaluating the PrimeChecker App

- Java ExecutorCompletionService fixes problems with the ExecutorService PrimeCheck app
Evaluating the PrimeChecker App

- Java ExecutorCompletionService fixes problems with the ExecutorService PrimeCheck app, e.g.
  - Futures are processed as they complete

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

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

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

This responsiveness is a benefit of “asynchronous future processing”
Evaluating the PrimeChecker App

- Java ExecutorCompletionService fixes problems with the ExecutorService PrimeCheck app, e.g.
  - Futures are processed as they complete
  - “Memoization” caches & retrieves results efficiently once they’re computed

The Memoizer class enables transparent optimization without changing existing code in PrimeCallable.
Evaluating the PrimeChecker App

• However, there is still a limitation
Evaluating the PrimeChecker App

• However, there is 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’ll fix this problem in the upcoming lesson on the Java ScheduledExecutorService
End of Overview of Java ExecutorService (Part 3)
Discussion Questions

1. Which of the following are ways the Memoizer potentially benefits the PrimeChecker app?
   
a. It caches & retrieves primality results efficiently once they’re computed
b. It ensures that keys/values are automatically purged from the Memoizer if they aren’t used for a long period of time
c. It ensures that futures are processed immediately as they complete
d. It enables transparent optimization of primality checking without modifying the PrimeCallable source code

2. Under which of the following circumstances will the Memoizer actually speed up the PrimeChecker app?
   
a. When the number of duplicate prime numbers is low
b. When the number of duplicate prime numbers is high
c. When the function that computes primality is inefficient
d. When the function that computes primality is efficient