The Java ExecutorService

(Part 3)

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

• Recognize the more powerful features provided by the Java ExecutorService interface & its related interfaces/classes

• Know the key methods provided by the Java ExecutorService

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

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

See [github.com/douglascraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutorService](http://github.com/douglascraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutorService)
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());
```

An additional thread is used to wait for the futures to complete.
Overview of the PrimeChecker App

- 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 {
    mFunction<Long, Long>
    mPrimeChecker;
    ...

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

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

The goal is to burn non-trivial CPU time!!
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));
    }
}
```

PrimeResult is used to match the prime number candidate with the result of checking for primality.
MainActivity creates a list of futures that will store the results of concurrently checking the primality of “count” random numbers.

```java
Function<Long, Long> primeChecker = PrimeCheckers::bruteForceChecker;

List<Future<PrimeCallable.PrimeResult>> futures = new Random()
    .longs(count, 0, Integer.MAX_VALUE)
    .mapToObj(ranNum ->
        new PrimeCallable
            (ranNum,
             primeChecker))
    .map(callable ->
        mRetainedState.
        mExecutorService
        ::submit)
    .collect(toList());
```

This Java 8 stream is very concise!
• MainActivity then creates a thread that waits for all future results in the background so it doesn’t block the UI thread

... 

mRetainedState.mFutureRunnable = new FutureRunnable(this, futures);

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

mRetainedState.mThread.start();

Summary of the PrimeChecker App

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

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

```java
private class FutureRunnable
    implements Runnable {
    List<Future<PrimeCallable.PrimeResult>> mFutures;
    MainActivity mMainActivity; ...

    public void run() {
        for (Future<PrimeResult> f : mFutures) {
            ... 
            PrimeResult pr = f.get();
            if (pr.mSmallestFactor != 0) ...
            else ...
            ... 
            mMainActivity.done();
        ...
    }
```

The `f.get()` call blocks synchronously until the async processing associated with that future has completed.
RetainedState maintains key concurrency state across runtime configuration changes

```java
class RetainedState {
    ExecutorService mExecutorService;
    FutureRunnable mFutureRunnable;
    Thread mThread;
}
```

Object onRetainNonConfigurationInstance() {
    return mRetainedState;
}

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

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

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

- Java ExecutorService fixes many problems with the Executor PrimeCheck app, e.g.
  - Prime checker function isn’t hard-coded

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

    ... 

    public PrimeCallable(long PrimeCandidate,
            Function<Long, Long> primeChecker) { ... } 

    ... 

    new PrimeCallable(ranNum,
            beEfficient ?
            PrimeCheckers::efficientChecker : 
            PrimeCheckers::bruteForceChecker);
```

This decoupling makes it easy to change prime checker algorithms.
Evaluating the PrimeChecker App

- Java ExecutorService fixes many problems with the Executor PrimeCheck app, e.g.
  - Prime checker function isn’t hard-coded
  - Two-way semantics of callables decouple PrimeCallable & MainActivity

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

    public PrimeCallable(long PrimeCandidate,
                         Function<Long, Long> primeChecker) {
        ...
    }

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

This decoupling simplifies runtime configuration changes
Evaluating the PrimeChecker App

- Java ExecutorService fixes many problems with the Executor PrimeCheck app, e.g.
  - Prime checker function isn’t hard-coded
  - Two-way semantics of callables decouple PrimeCallable & MainActivity
  - Runtime configuration changes can be handled gracefully

```java
Long bruteForceChecker(Long n) {
    if (n > 3)
        for (long factor = 2; factor <= n / 2; ++factor)
            if ((factor % (n / 10)) == 0 && Thread.interrupted())
                break;
        else if (n / factor * factor == n)
            return factor;
    return 0;
}
```

Running tasks continue to execute & update the GUI until they done or are interrupted.
Evaluating the Prime_checker App

- Java ExecutorService fixes many problems with the Executor PrimeChecker app, e.g.
  - Prime checker function isn’t hard-coded
  - Two-way semantics of callables decouple PrimeCallable & MainActivity
  - Runtime configuration changes can be handled gracefully
  - Lifecycle operations enable interruption of running tasks

```java
void interruptComputations() {
    mRetainedState.mExecutorService.shutdownNow();

    mRetainedState.mThread.
        Interrupt();

    ... 

done();
```

Shutting down an executor service interrupts all threads running tasks
• However, there are still some limitations
Evaluating the PrimeChecker App

• However, there are still some limitations, e.g.
• f.get() blocks the thread, even if some other futures may have completed

... 

private class FutureRunnable 
    implements Runnable {
        List<Future<PrimeCallable.PrimeResult>> mFutures;
        MainActivity mActivity; ...

    public void run() {
        for (Future<PrimeResult> f : mFutures) {
            ... 
            PrimeResult pr = f.get();
            if (pr.mSmallestFactor != 0) ... 
            else ... 
            ... 
            mActivity.done();
        ...

This is a common problem with “synchronous future processing”
• However, there are still some limitations, e.g.
  • f.get() blocks the thread, even if some other futures may have completed
  • The prime check is applied each time even if the results have been computed previous

    public class PrimeCallable
        implements Callable<PrimeResult> {
            ...
            public PrimeCallable(long PrimeCandidate,
                Function<Long, Long> primeChecker) { ... }
            ...
            public PrimeResult call() {
                return new PrimeResult(mPrimeCandidate,
                        mPrimeChecker.apply(mPrimeCandidate));
            }
        }

Ideally PrimeCallable could be optimized without changing its source code.
End of Overview of Java ExecutorService (Part 3)