The Java ExecutorService Interface

(Part 4)

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

• Understand how ThreadPoolExecutor implements the 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 to determine if \( N \) random numbers are prime.

See [github.com/douglasraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutorService](https://github.com/douglasraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutorService)
Overview of the PrimeChecker App

• This “embarrassingly parallel” app shows how to use the Java ExecutorService to determine if $N$ random numbers are prime
• It also shows how to handle runtime configuration changes in Android

See developer.android.com/guide/topics/resources/runtime-changes.html
Overview of the PrimeChecker App

- This “embarrassingly parallel” app shows how to use the Java ExecutorService to determine if \( N \) random numbers are prime
- It also shows how to handle runtime configuration changes in Android
- As well as thread interruptions

See docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html
Overview of the PrimeChecker App

• 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());
```

A fixed-size thread pool uses an unbounded queue to avoid deadlocks

See asznajder.github.io/thread-pool-induced-deadlocks
Overview of the PrimeChecker App

- 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());

...mThread = new Thread(...mRetainedFutureRunnable);
...mThread.start();
```

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

- PrimeCallable implements Callable & defines a two-way means of determining whether a # 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;
    }

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

See [PrimeExecutorService/app/src/main/java/vandy/mooc/prime/activities/PrimeCallable.java](https://example.com)
Overview of the PrimeChecker App

PrimeCallable implements Callable & defines a two-way means of determining whether a # 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;
    }

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

The function that computes primality is passed as a param & stored in a field
Overview of the PrimeChecker App

• PrimeCallable implements Callable & defines a two-way means of determining whether a # 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;
    }

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

The call() hook method applies the function, which returns 0 if it’s prime or smallest factor if it’s not
Overview of the PrimeChecker App

- PrimeCallable implements Callable & defines a two-way means of determining whether a # 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;
    }

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

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

- MainActivity creates a list of futures that will store the results of concurrently checking the primality of “count” random numbers

```java
List<Future<PrimeResult>> futures = ...
```

See PrimeExecutorService/app/src/main/java/vandy/mooc/prime/activities/MainActivity.java
Overview of the PrimeChecker App

- MainActivity creates a list of futures that will store the results of concurrently checking the primality of “count” random numbers.

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

This list of futures is initialized via a Java 8 sequential stream.
Overview of the PrimeChecker App

- MainActivity creates a list of futures that will store the results of concurrently checking the primality of “count” random numbers

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

Generates “count” random numbers ranging between 0 & MAX_VALUE

- MainActivity:
  - onCreate(Bundle):void
  - initializeViews():void
  - setCount(View):void
  - startOrStopComputations(View):void
  - startComputations(int):void
  - interruptComputations():void
  - done():void
  - println(String):void
  - onRetainNonConfigurationInstance():Object
  - onDestroy():void
Overview of the PrimeChecker App

- MainActivity creates a list of futures that will store the results of concurrently checking the primality of “count” random numbers

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

Transforms random #s into PrimeCallables
Overview of the PrimeChecker App

MainActivity creates a list of futures that will store the results of concurrently checking the primality of “count” random numbers.

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

Function<Long, Long> primeMemoizer = new Memoizer<>(PrimeCheckers ::efficientChecker);

This (stateful) memoizer transparently caches results of primality checking & integrates seamlessly into Java 8 stream.

See [PrimeExecutorService/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java](PrimeExecutorService/app/src/main/java/vandy/mooc/prime/utils/Memoizer.java)
Overview of the PrimeChecker App

- MainActivity creates a list of futures that will store the results of concurrently checking the primality of “count” random numbers.

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

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit)
Overview of the PrimeChecker App

- MainActivity creates a list of futures that will store the results of concurrently checking the primality of “count” random numbers

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

Collects the results into a list of futures to PrimeResults
Overview of the PrimeChecker App

- MainActivity creates a background thread that waits for all future results to avoid blocking the UI thread.

```java
class FutureRunnable implements Runnable {
    List<Future<PrimeResult>> mFutures;
    MainActivity mActivity;

    public void run() {
        ...
    }
}
```

*FutureRunnable executes in a background thread in the Executor Service & gets the results of all completed futures.*

See [developer.android.com/training/articles/perf-anr.html](http://developer.android.com/training/articles/perf-anr.html)
Overview of the PrimeChecker App

- MainActivity creates a background thread that waits for all future results to avoid blocking the UI thread

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

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

mRetainedState.mThread.start();
```

FutureRunnable is stored in a field so its state can be updated during a runtime configuration change.

See developer.android.com/guide/topics/resources/runtime-changes.html
• FutureRunnable provides a means to get the results as futures complete

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

    public void run() {
        for (Future<PrimeResult> f : mFutures) {
            ...
            PrimeResult pr = f.get();
            if (pr.mSmallestFactor != 0)
                ...
            else ...
            ...
            mActivity.done();
        ...
    }
}
```
Overview of the PrimeChecker App

- FutureRunnable provides a means to get the results as futures complete

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

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

f.get() blocks synchronously until the async processing associated with that future in the thread pool completes
Overview of the PrimeChecker App

- RetainedState maintains key concurrency state across runtime configuration changes

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

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

```java
void onCreate(...) {
    mRetainedState = (RetainedState) getLastNonConfigurationInstance
    if (mRetainedState != null) {
        ...
    }
}
```
Overview of the PrimeChecker App

- RetainedState maintains key concurrency state across runtime configuration changes

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

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

... Retained state is loaded/stored via Android hook methods ...

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

See 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
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
  - Lifecycle operations enable interruption of running tasks

```java
void interruptComputations() {
    mRetainedState.mExecutorService.shutdownNow();
    mRetainedState.mThread.interrupt();
    ...
    mRetainedState.mExecutorService.awaitTermination(500,
        TimeUnit.MILLISECONDS);
```

Shutting down an executor service interrupts all threads running tasks
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
  - Lifecycle operations enable interruption of running tasks
  - 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;
    else return 0;
}
```

Running tasks continue to execute & update the GUI until they finish or are interrupted
Evaluating the PrimeChecker App

• 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

```java
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”
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
  - 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 4)