The Java Executor Interface

(Part 1)

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 simple/single feature provided by the Java Executor interface
Overview of the Java Executor Interface
Overview of the Java Executor Interface

• Provides a method to submit new tasks for execution

<<Java Interface>>

Executor

execute(Runnable): void

Defines a simple API that decouples task submission from the mechanics of how each task will be run
Overview of the Java Executor Interface

- Provides a method to submit new tasks for execution
- Each task implements the Runnable interface

See docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html
Overview of the Java Executor Interface

- Provides a method to submit new tasks for execution
  - Each task implements the Runnable interface
    - Represents a “command” to execute
      - i.e., the Command pattern

See en.wikipedia.org/wiki/Command_pattern
Overview of the Java Executor Interface

• Provides a method to submit new tasks for execution
  • Each task implements the Runnable interface
    • Represents a “command” to execute
    • Provides “one-way” task semantics
      • i.e., does not return a result

<<Java Interface>>
Runnable

run():void

ONE WAY
Overview of the Java Executor Interface

- Provides a method to submit new tasks for execution
  - Each task implements the Runnable interface
    - Represents a “command” to execute
    - Provides “one-way” task semantics
  - Can execute in a background thread or the main thread
    - i.e., depending on the Executor interface’s implementation

See upcoming lesson on “Executor Framework Implementations”
Overview of the Java Executor Interface

- Provides a method to submit new tasks for execution
  - Each task implements the Runnable interface
    - Represents a “command” to execute
    - Provides “one-way” task semantics
    - Can execute in a background thread or the main thread
  - Also implements the Command Processor pattern

Packages a piece of application functionality—as well as its parameterization in an object—to make it usable in another context

See [www.dre.vanderbilt.edu/~schmidt/CommandProcessor.pdf](http://www.dre.vanderbilt.edu/~schmidt/CommandProcessor.pdf)
End of Overview of Java Executor Interface (Part 1)
The Java Executor Interface

(Part 2)

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 simple/single feature provided by the Java Executor interface

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

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

See [github.com/douglascraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutor](https://github.com/douglascraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutor)
PrimeRunnable defines a brute-force means to determine whether number is prime, returning 0 if it is prime or the smallest factor if it’s not.

```java
long isPrime(long n) {
    if (n > 3)
        for (long factor = 2; factor <= n / 2; ++factor)
            if (n / factor * factor == n)
                return factor;
    return 0;
}
```

The goal is to burn non-trivial CPU time!!
Overview of the PrimeChecker App

- This app uses a Java Executor that's implemented with a fixed-size thread pool tuned to the # of processor cores in the computing device

```java
mExecutor = 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)
This app uses a Java Executor that’s implemented with a fixed-size thread pool tuned to the # of processor cores in the computing device.

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

This value is suitable since isPrime() is a “compute-bound” task.

See [docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#availableProcessors](http://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#availableProcessors)
Overview of the PrimeChecker App

- MainActivity creates/executes a PrimeRunnable for each of the "count" random numbers in between 0 & MAX_VALUE

```java
new Random()
    .longs(count, 0, Long.MAX_VALUE)
    .forEach(randomNumber ->
        mExecutor.execute
            (new PrimeRunnable(this, randomNumber)));
```
Overview of the PrimeChecker App

- MainActivity creates/executes a PrimeRunnable for each of the "count" random numbers in between 0 & MAX_VALUE

```java
new Random()
  .longs(count, 0, Long.MAX_VALUE)
  .forEach(randomNumber ->
    mExecutor.execute
    (new PrimeRunnable(this, randomNumber)));
```

Although there may be many PrimeRunnable instances, they will run on a (much) smaller # of threads, which can be tuned transparently.
Evaluating the PrimeChecker App
The main benefit of using the Java Executor interface is that the # & type of threads can be tuned transparently wrt the application logic.

```java
new Random().longs(count, 0, Integer.MAX_VALUE)
    .forEach(randomNumber -> mExecutor.execute
        (new PrimeRunnable(this, randomNumber)));
```

### Evaluating the PrimeChecker App

**Fixed-sized Thread Pool**

**Variable-sized Thread Pool**

**Work-stealing Thread Pool**
Evaluating the PrimeChecker App

• However, there are limitations due to the restrictions of Java Executor
Evaluating the PrimeChecker App

- However, there are limitations due to the restrictions of Java Executor, e.g.
  - One-way semantics of runnables tightly couple PrimeRunnable with MainActivity

```java
public class PrimeRunnable implements Runnable {
    ...
    private final MainActivity mActivity;
    ...
    public PrimeRunnable(MainActivity activity) {
        mActivity = activity;
    }
    public void run() {
        ...
        mActivity.done();
    }
}
```

This tight coupling complicates runtime configuration changes
Evaluating the PrimeChecker App

However, there are limitations due to the restrictions of Java Executor, e.g.

- One-way semantics of runnables tightly couple PrimeRunnable with MainActivity
- The prime checker algorithm is tightly coupled with the PrimeRunnable class

```java
public class PrimeRunnable implements Runnable {
    ...
    long isPrime(long n) {
        if (n > 3)
            for (long factor = 2; factor <= n / 2; ++factor)
                if (n / factor * factor == n)
                    return factor;

        return 0;
    }
    ...
}
```

This tight coupling means the primality check is applied each time even if the results have been computed previously & complicates algorithm improvements
• However, there are limitations due to the restrictions of Java Executor, e.g.
  • One-way semantics of runnables tightly couple PrimeRunnable with MainActivity
  • The prime checker algorithm is tightly coupled with the PrimeRunnable class
  • The lack of lifecycle operations on Java Executor
Evaluating the PrimeChecker App

• However, there are limitations due to the restrictions of Java Executor, e.g.
  • One-way semantics of runnables tightly couple PrimeRunnable with MainActivity
  • The prime checker algorithm is tightly coupled with the PrimeRunnable class
  • The lack of lifecycle operations on Java Executor, e.g.
    • Can’t interrupt/cancel running tasks
Evaluating the PrimeChecker App

- However, there are limitations due to the restrictions of Java Executor, e.g.
  - One-way semantics of runnables tightly couple PrimeRunnable with MainActivity
  - The prime checker algorithm is tightly coupled with the PrimeRunnable class
  - The lack of lifecycle operations on Java Executor, e.g.
    - Can’t interrupt/cancel running tasks
    - Can’t gracefully handle runtime configuration
      - e.g., must restart processing from the beginning
End of Overview of Java Executor Interface (Part 2)