Learning Objectives in this Part of the Lesson

- Recognize the simple/single feature provided by the Java Executor interface

### Interface Executor

**All Known Subinterfaces:**
ExecutorService, ScheduledExecutorService

**All Known Implementing Classes:**
AbstractExecutorService, ForkJoinPool, ScheduledThreadPoolExecutor, ThreadPoolExecutor

```java
public interface Executor

An object that executes submitted Runnable tasks. This interface provides a way of decoupling task submission from the mechanics of how each task will be run, including details of thread use, scheduling, etc. An Executor is normally used instead of explicitly creating threads. For example, rather than invoking new Thread(new RunnableTask()).start() for each of a set of tasks, you might use:

```java
erector executor = anExecutor;
erector.execute(new RunnableTask1());
erector.execute(new RunnableTask2());
```

```
...
```

However, the Executor interface does not strictly require that execution be asynchronous. In the simplest case, an executor can run the submitted task immediately in the caller's thread:
Overview of the Executor Interface
Overview of the 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 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 Executor Interface

- Provides a method to submit new tasks for execution

- Each task implements the Runnable interface

- Represents a “command” to execute
  - A command is an object used to encapsulate all information needed to perform an action or trigger an event at a later time

See [en.wikipedia.org/wiki/Command_pattern](en.wikipedia.org/wiki/Command_pattern)
Overview of the 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
Overview of the 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 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
• 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
End of Overview of Java Executor (Part 1)
The Java Executor
(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

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

- This app shows how to use the Java Executor framework to determine if $N$ random numbers are prime or not.

See [github.com/douglasraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutor](https://github.com/douglasraigschmidt/POSA/tree/master/ex/M4/Primes/PrimeExecutor)
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());
```
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 prime.

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

The goal is to burn non-trivial CPU time!!
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, Integer.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
• However, there are limitations due to the restrictions of Java Executor, e.g.
  • One-way semantics of runnables
  • Tightly couple the 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
• However, there are limitations due to the restrictions of Java Executor, e.g.
  • The one-way semantics of runnables
  • 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.
  • The one-way semantics of runnables
  • 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.
  - The one-way semantics of runnables
  - The lack of lifecycle operations on Java Executor, e.g.
    - Can’t interrupt/cancel running tasks
    - Can’t gracefully handle runtime configuration
End of Overview of Java Executor (Part 2)