Scaling Java Applications Through Concurrency

Running Processes in the Background

Josh Cummings

@jzheaux | tech.joshcummings.com

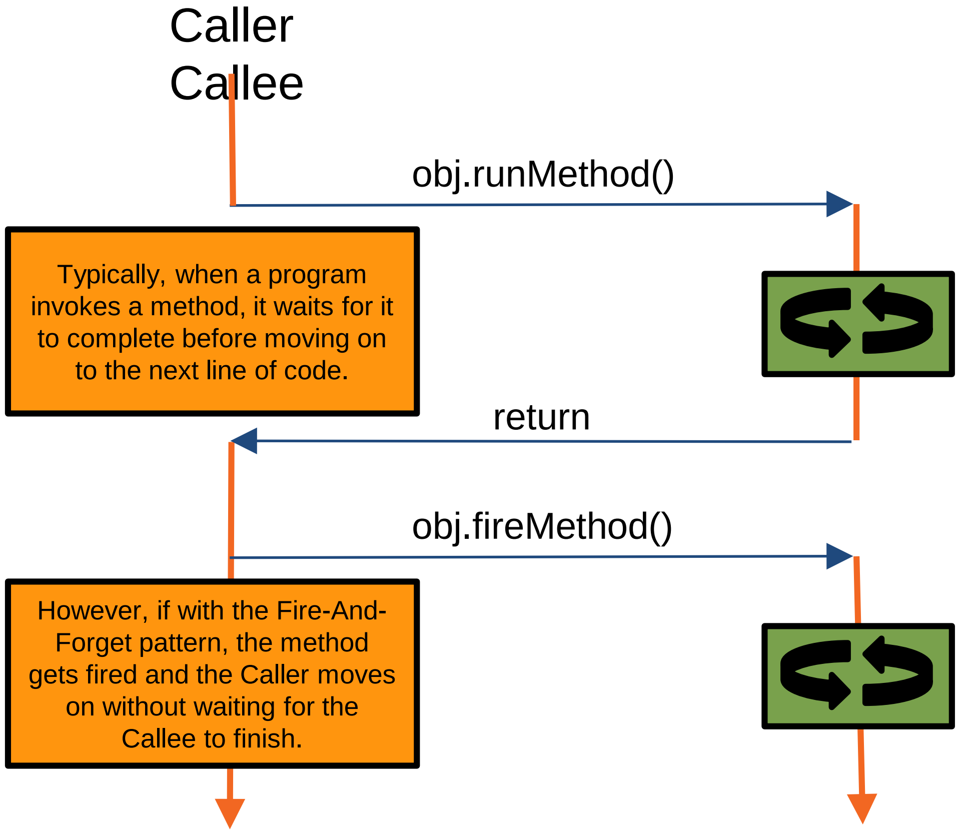
Identity Pipeline: An Integrated Example of Java Concurrency Patterns

- Reads in a stream of identity-related data
- Normalizes and verifies the contents of each identity
- Merges and persists the identities in an in-memory cache
- Records aggregate identity statistics
- Notifies an error queue if any of the above fails

Identity Pipeline: Background Processes

- Reads in a stream of identity-related data
- Normalizes and verifies the contents of each identity
- Merges and persists the identities in an in-memory cache
- Records aggregate identity statistics
- **Notifies an error queue if any of the above fails**

The Fire-And-Forget Pattern

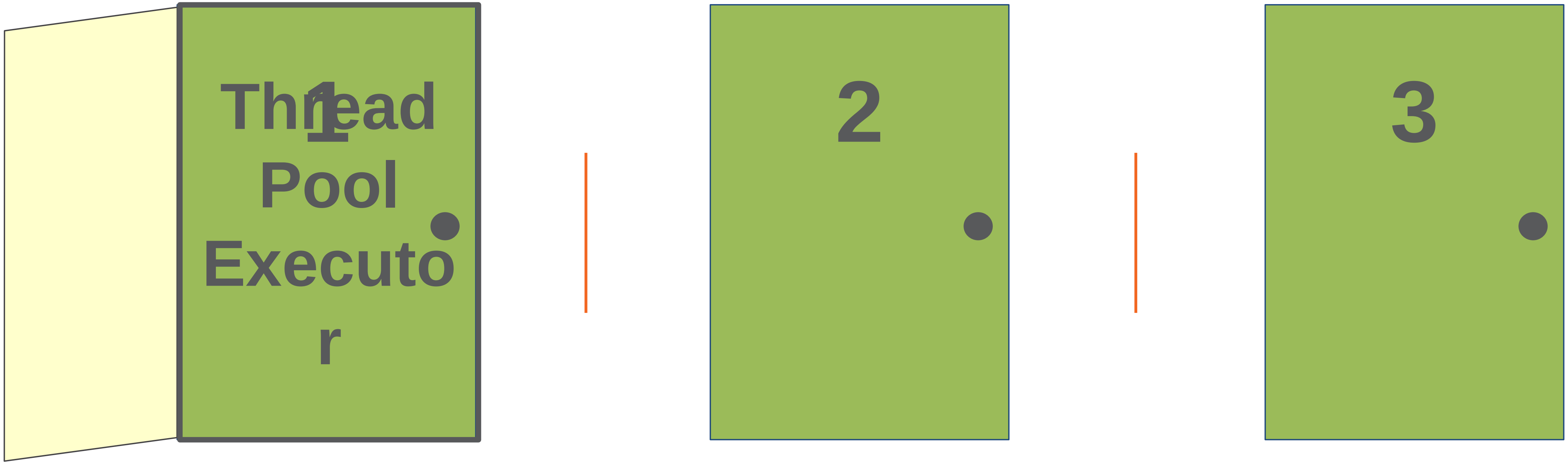


Typically, when a program invokes a method, it waits for it to complete before moving on to the next line of code.

However, if with the Fire-And-Forget pattern, the method gets fired and the Caller moves on without waiting for the Callee to finish.

- Operating on a single-thread bounds the application’s performance to the slowest instructions
- Anticipating a reply doubles the work that needs to be done
- Fire-And-Forget addresses both

Options Found in the Concurrency API



1

Thread
Pool
Executo
r

2

3

ThreadPoolExecutor

- Maintains a pool of threads for rapid reuse
- Abstracts away the use of Thread
- Results may either be blocked on, ignored, or delegated
- Never blocks on task submission, even if the thread pool size is maxxed out

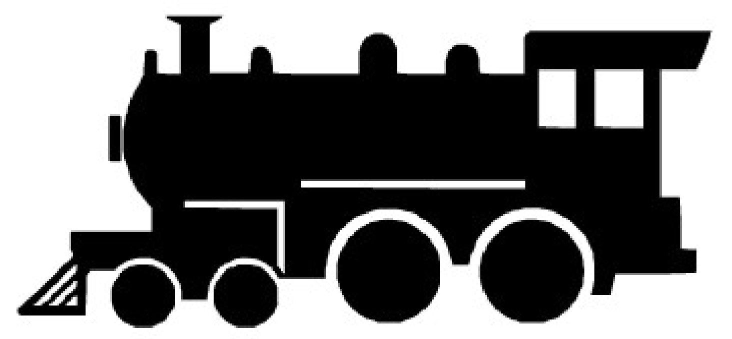
```
// each returns an appropriately
configured // instance of
ThreadPoolExecutor

ExecutorService pool =
Executors.newCachedThreadPool();

ExecutorService pool =
Executors.newFixedThreadPool(10);

ExecutorService single =
Executors.newSingleThreadExecutor();
```

Cached

Light-weight asychronous tasks

Threads are automatically created if no thread is available for a task

Idle threads are re-used and will terminate after 60 seconds of inactivity



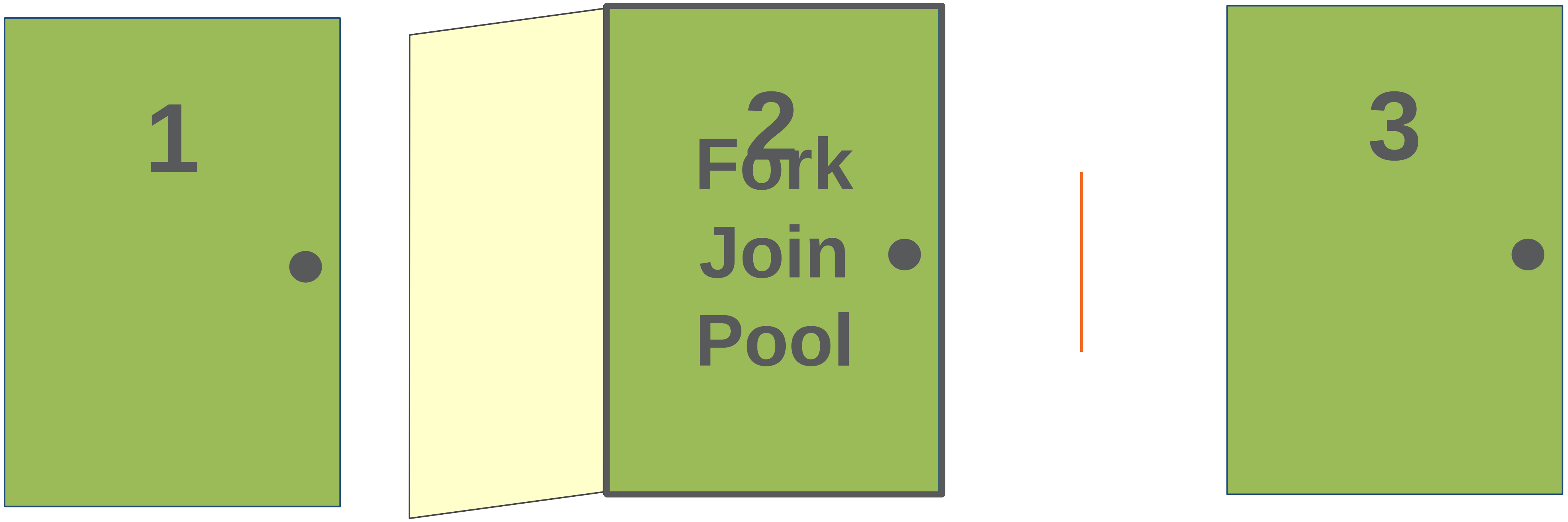
Fixed

Heavy, long-running use

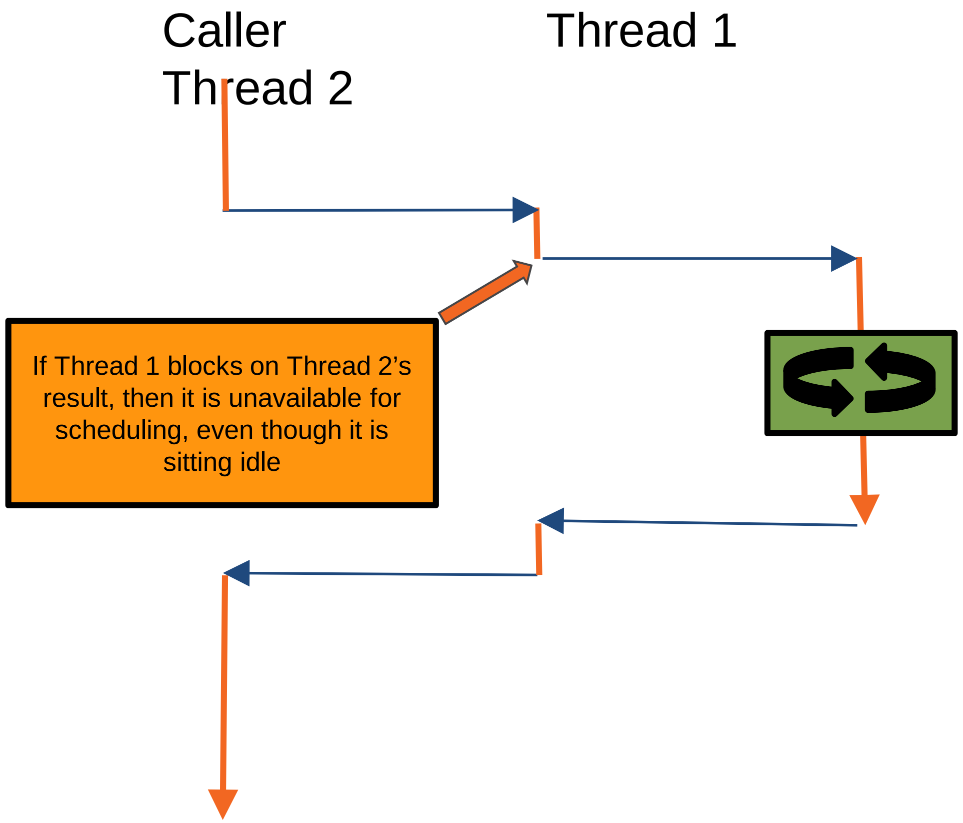
A fixed number of threads is constantly maintained regardless of workload

Threads will be kept active until they are explicitly shut down

Options Found in the Concurrency API



What to do about idling threads?



If Thread 1 blocks on Thread 2's result, then it is unavailable for scheduling, even though it is sitting idle

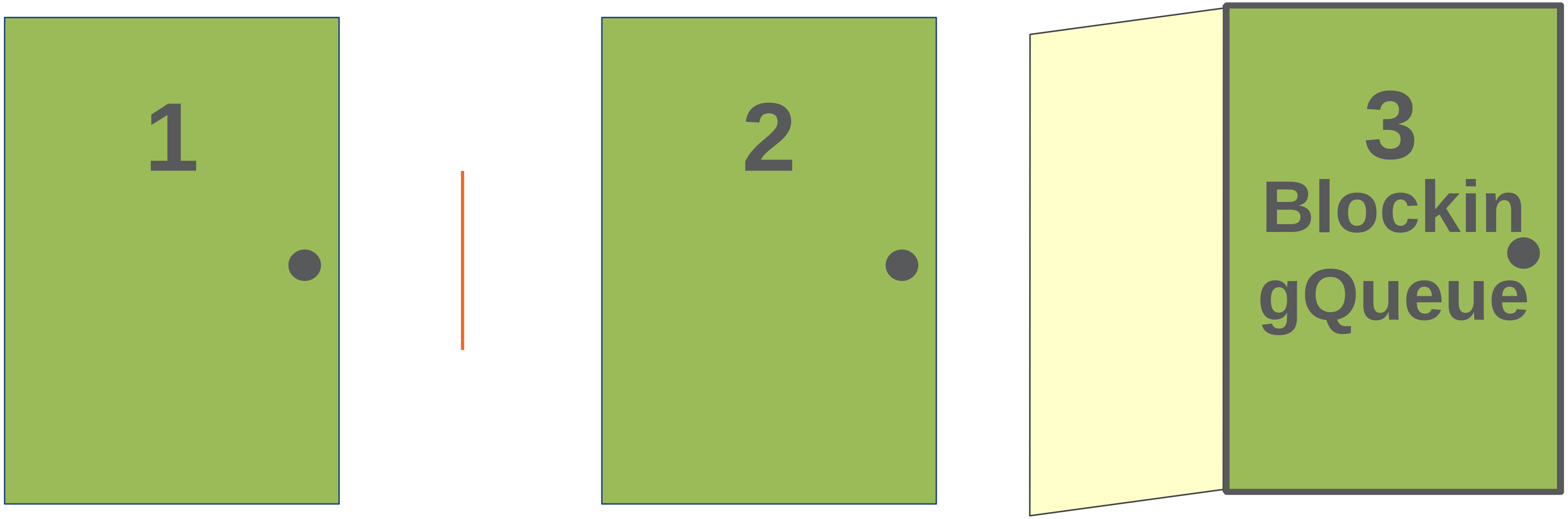
- asynchronous != non-blocking (consider the case of a parallelized sorting algorithm)
- Threads may need to block, waiting for other threads to finish
- In the case of thread pools, this is sub-optimal because a blocking thread cannot be scheduled even though it is not doing work

ForkJoinPool

- Identical benefits to ThreadPoolExecutor
- Processes are daemon threads, meaning that the main execution thread will not wait for them to complete when the JVM shuts down.
- Processes may steal work from one another, which is nice for threaded recursion, e.g. in the case of Divide and Conquer algorithms

```
// each returns an appropriately
configured // instance of ForkJoinPool

ExecutorService pool =
Executors.newWorkStealingPool();

ExecutorService pool =
ForkJoinPool.common();

// # in pool should be a power of two
ExecutorService single = new
ForkJoinPool(16);
```

Options Found in the Concurrency API



A Bit More on ForkJoinPool

If parallelizing...	… then extend ...
recursion with results (e.g. sorting, searching)	RecursiveTask
recursion with no results (e.g. fire-and-forget)	RecursiveAction

BlockingQueue

- Implementations of ExecutorService follow the producer-consumer paradigm, using a queue as the intermediate data structure

- BlockingQueue abstracts away direct use of wait and notify, avoiding common pitfalls

- BlockingQueue has a full complement of produce and consume methods that block, don’t block, or throw exceptions

```
// a list of tasks to be completed at the
// disposal of any number of consumers

BlockingQueue<Task> tasks = new
LinkedBlockingQueue<>();

tasks.offer(task); // adds to the end
without blocking

tasks.take(); // blocks until there is
something in the queue
```

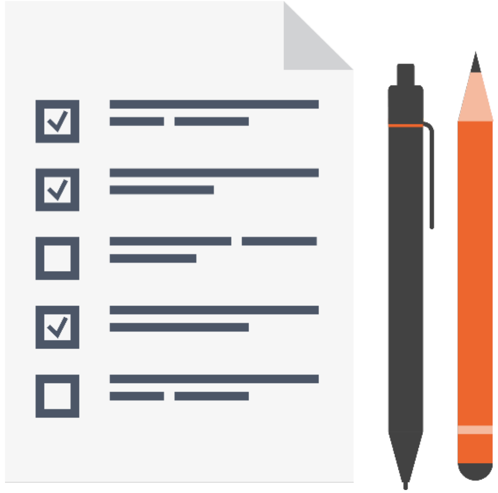
Pro Tip: Debugging Threads

- Name thread pools for easier debugging (use Google Guava)

- Name threads with debug information for more meaningful thread dumps

```
// name your thread pool

ExecutorService pool =
          Executors.newCachedThreadPool(
new
                    ThreadFactoryBuilder()

          .setNameFormat("Malformed-%d")
                    .build());

// provide context in thread name

Thread.currentThread().setName("Malform
ed-identity-" + identity.getId());
```

Review



- Fire-and-forget is a simple pattern that can be implemented by firing an independent process on a separate thread
- Implementations of **ExecutorService** abstract away thread management
- **ForkJoinPool** is useful for threaded recursion
- Java **Queues** can also be used to queue up tasks in a thread-safe fashion independent of a thread pool