Thread Pools

David Flynn

SOFTWARE ENGINEER FLYNN IT LTD

Introduction

We could implement our own pool of threads, but…

- What if an exceptional workload needs handling?
- If threads throw exceptions and die?
- We might need to manage hundreds of threads

Introduction

Thread pools do this for us, so we don't need to reinvent the wheel

- It's more of an engine, but we can extend it if needed

Passing Tasks to Thread Pools

Task is passed to a ThreadPoolExecutor

- Task is a Runnable [starts at run() method]

If task is accepted – either:

- Create a thread
- Assign existing thread

Handling Task Completion

With standalone threads:

- Use join
- Save results into shared memory
- Catch / report any unchecked exceptions
- Using uncaught exception handlers

With thread pools:

- Threads are long-lived so join is not useful
- Need another mechanism

Future

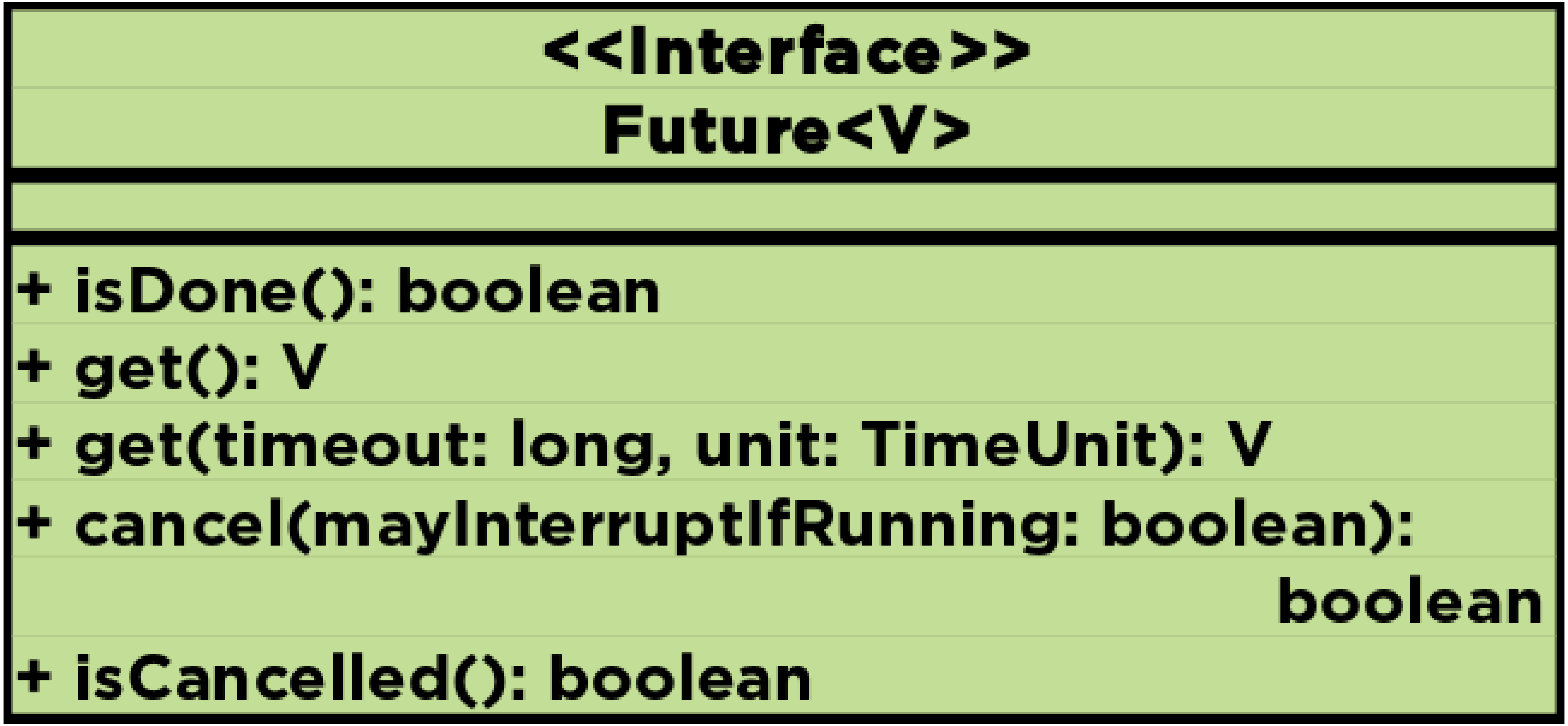
May only need results at point of use

Avoid monitoring

Submitting tasks returns a Future

Manages tasks which run “in the Future”

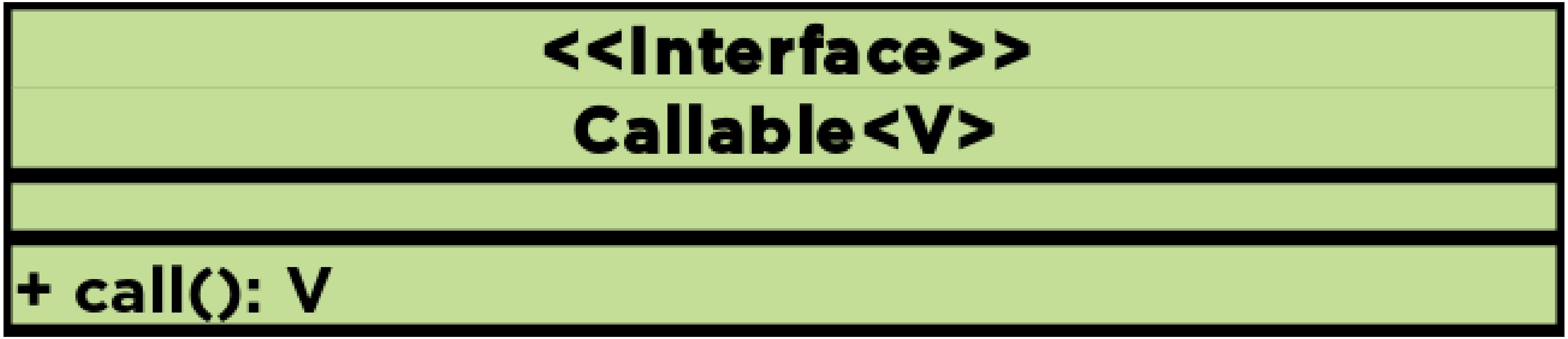
Future



<<Interface>>
Future<V>

+ isDone(): boolean
+ get(): V
+ get(timeout: long, unit: TimeUnit): V
+ cancel(mayInterruptIfRunning: boolean): boolean
+ isCancelled(): boolean

Callable



<<Interface>>

Callable<V>

+ call(): V

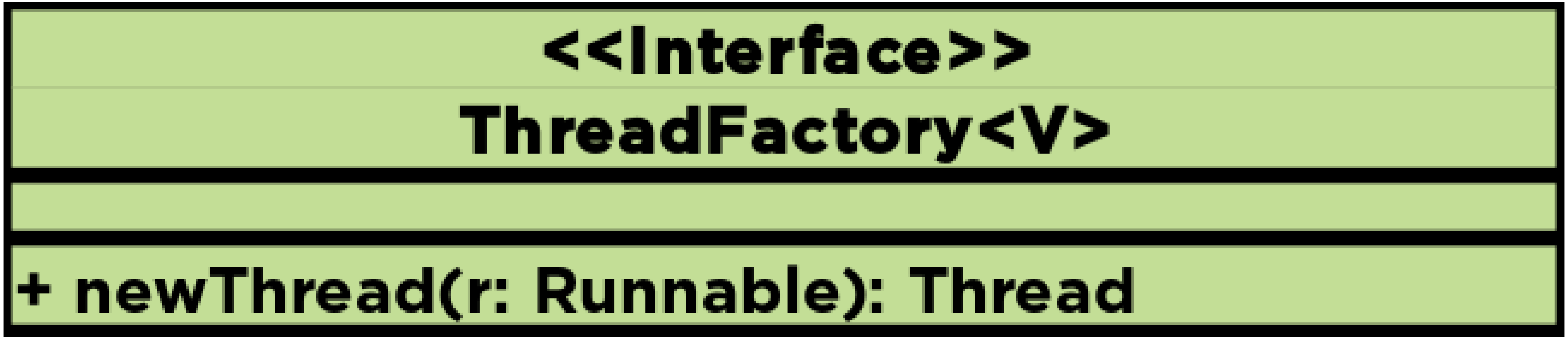
Adapting Runnable to Callable

Use Executors' static method callable

Not needed when passing to thread pools

- Useful if we have a Runnable but need Callable in other cases

ThreadFactory

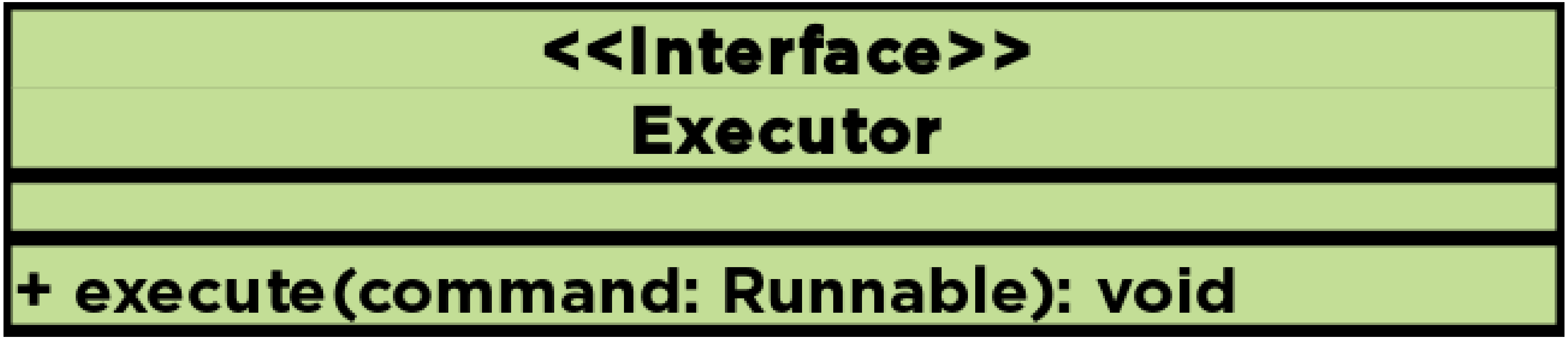


<<Interface>>

ThreadFactory<V>

+ newThread(r: Runnable): Thread

Executor

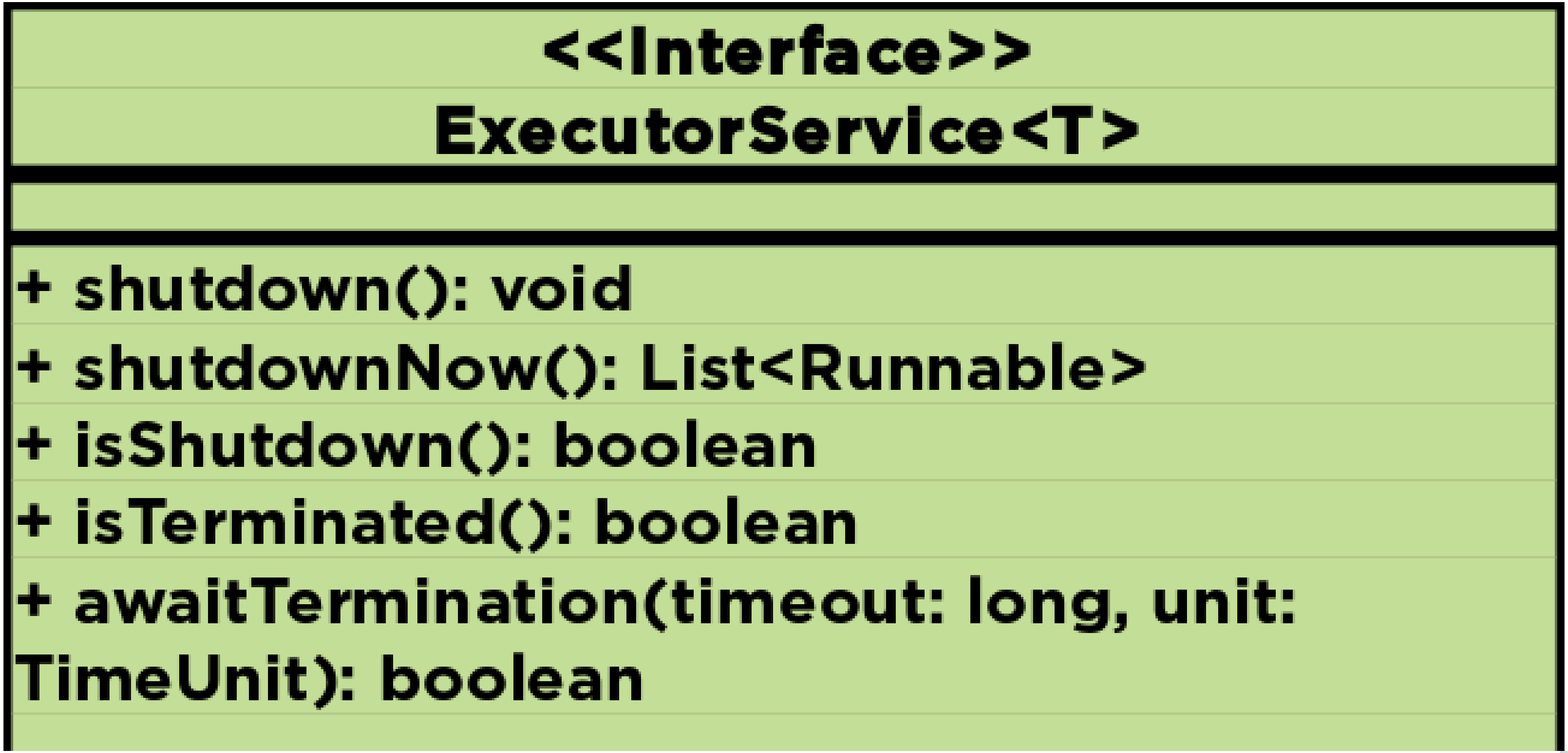


<<Interface>>

Executor

+ execute(command: Runnable): void

ExecutorService



<<Interface>>

ExecutorService<T>

+ shutdown(): void

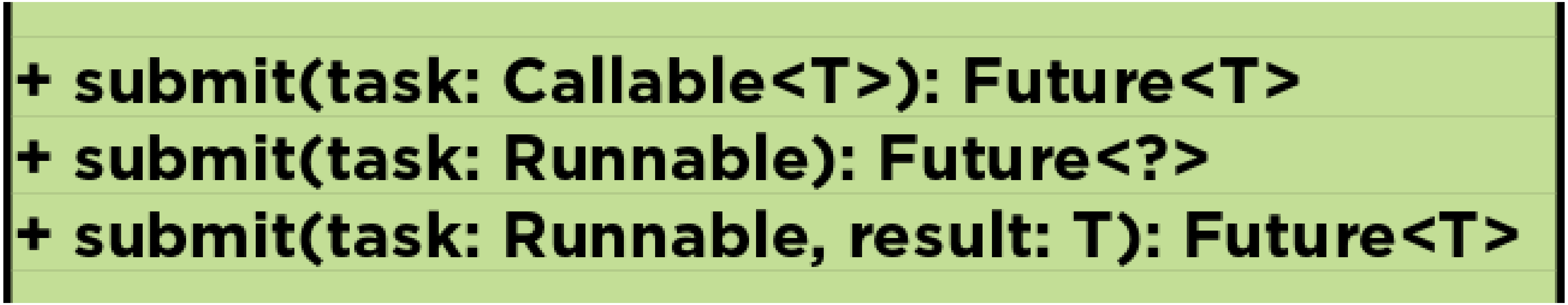
+ shutdownNow(): List<Runnable>

+ isShutdown(): boolean

+ isTerminated(): boolean

+ awaitTermination(timeout: long, unit: TimeUnit): boolean

ExecutorService

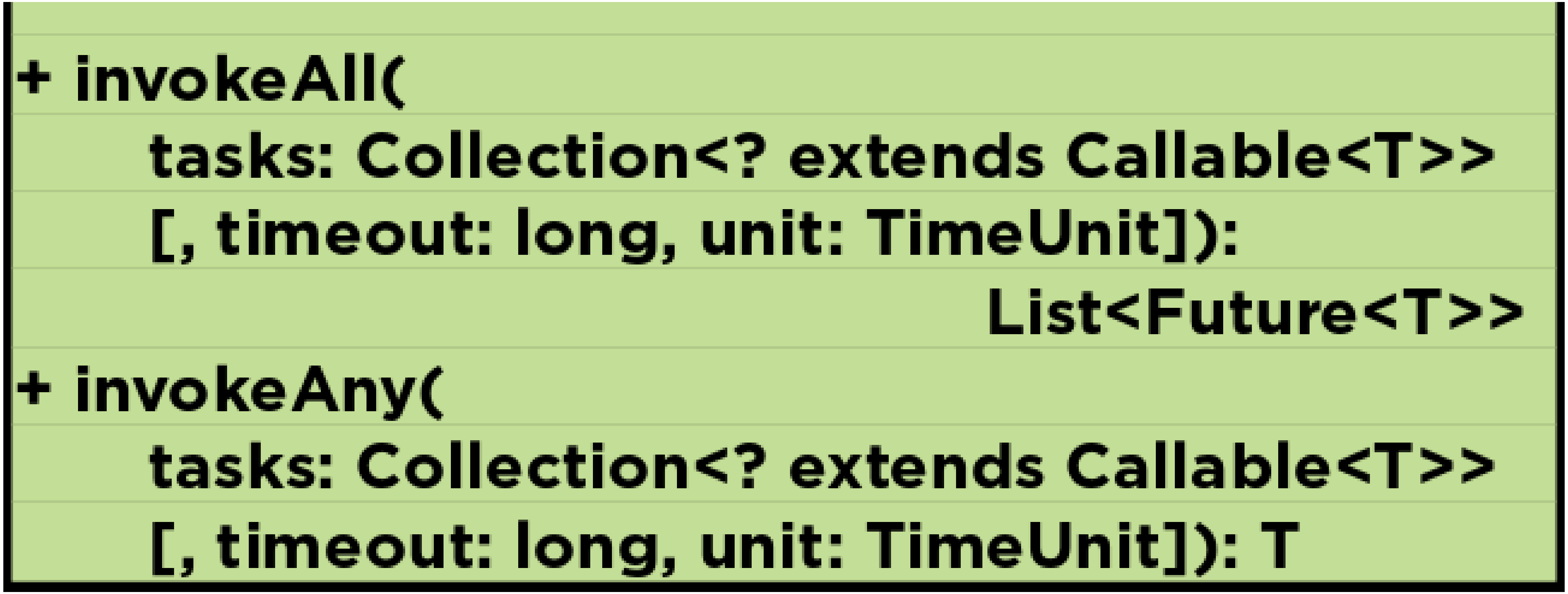


+ submit(task: Callable<T>): Future<T>

+ submit(task: Runnable): Future<?>

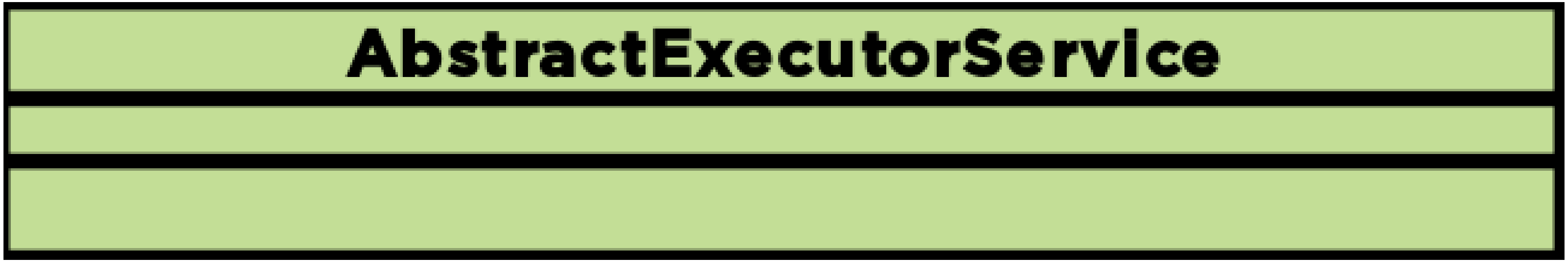
+ submit(task: Runnable, result: T): Future<T>

ExecutorService

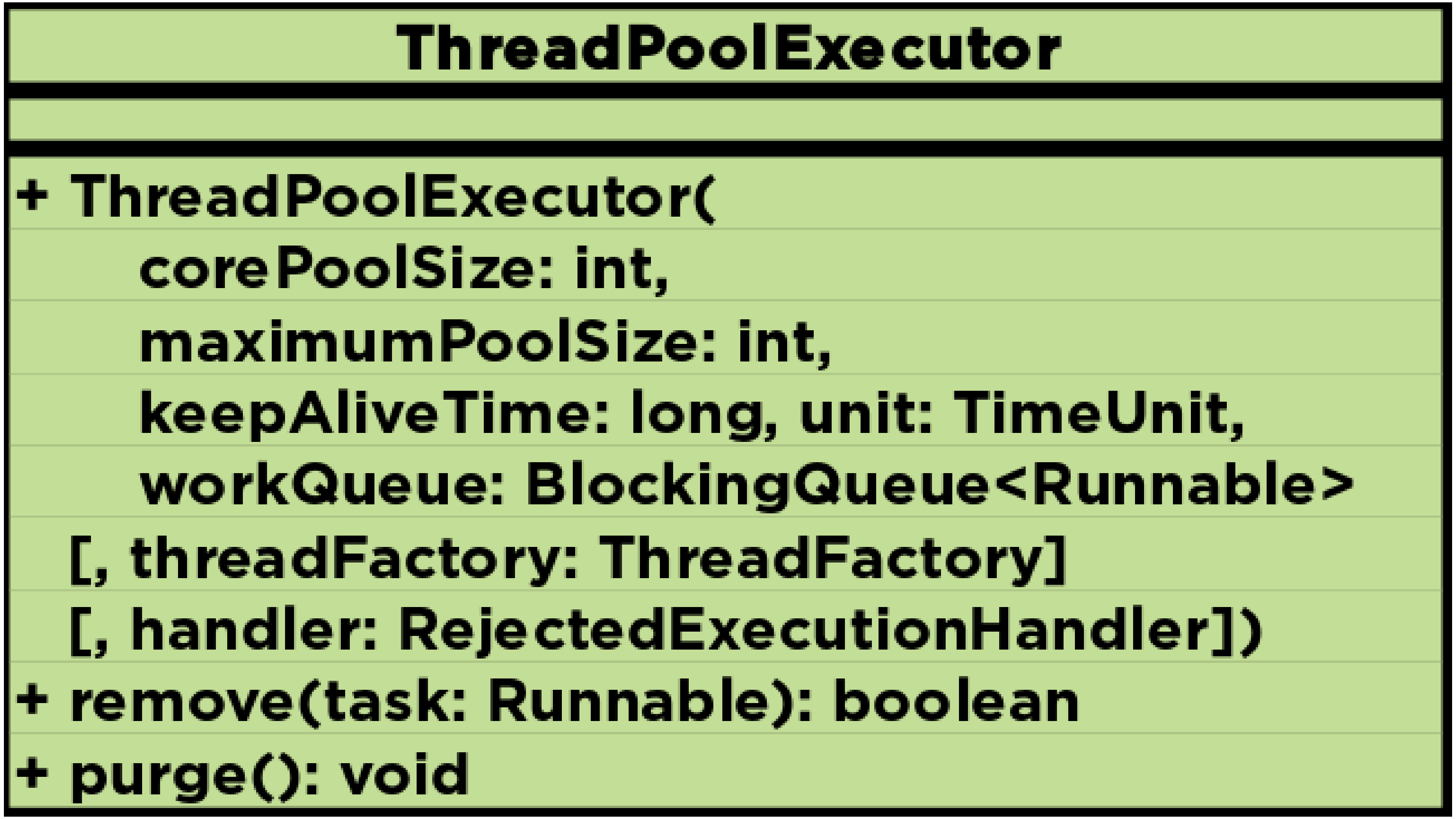


```
+ invokeAll(
      tasks: Collection<? extends Callable<T>>
      [, timeout: long, unit: TimeUnit]):
                                List<Future<T>>
+ invokeAny(
      tasks: Collection<? extends Callable<T>>
      [, timeout: long, unit: TimeUnit]): T
```

AbstractExecutorService



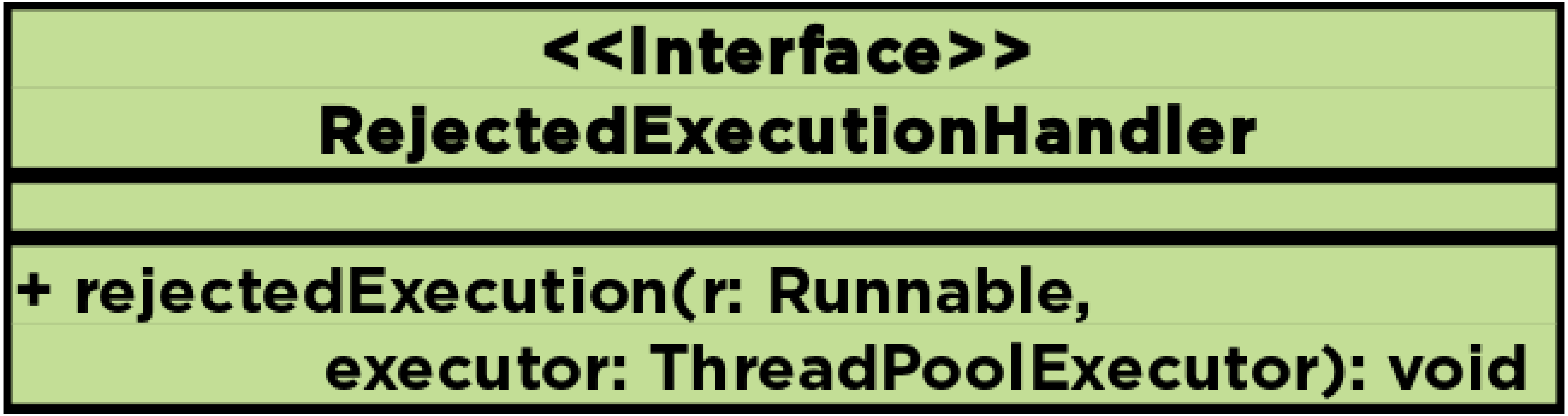
ThreadPoolExecutor



```
ThreadPoolExecutor

+ ThreadPoolExecutor(
      corePoolSize: int,
      maximumPoolSize: int,
      keepAliveTime: long, unit: TimeUnit,
      workQueue: BlockingQueue<Runnable>
  [, threadFactory: ThreadFactory]
  [, handler: RejectedExecutionHandler])
+ remove(task: Runnable): boolean
+ purge(): void
```

RejectedExecutionHandler



<<Interface>>

RejectedExecutionHandler

+ rejectedExecution(r: Runnable,
executor: ThreadPoolExecutor): void

RejectedExecutionHandler

CallerRunsPolicy

- If not shutdown, caller thread runs instead

AbortPolicy (default)

- Throws RejectedExecutionException

DiscardPolicy

- Silently discards the task

DiscardOldestPolicy

- Runs instead of oldest waiting task

Executors

Helper class for creating thread pools

Types include

- Fixed size
- Single thread
- Scheduled

Monte Carlo Simulations

Problem of imprecise data or model

- Introduce randomness due to uncertainty [e.g. draw from a probability distribution]

Average taken over many simulations

Report likely output

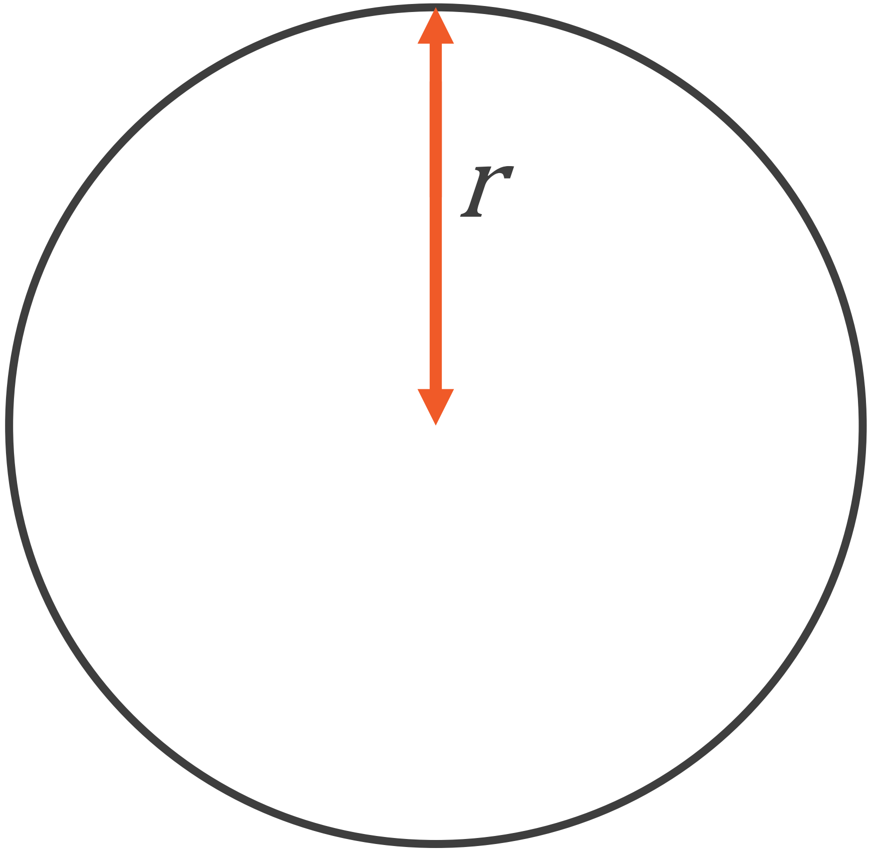
Applications of Monte Carlo Simulations

Weather forecasting

Financial [and economic] predictions

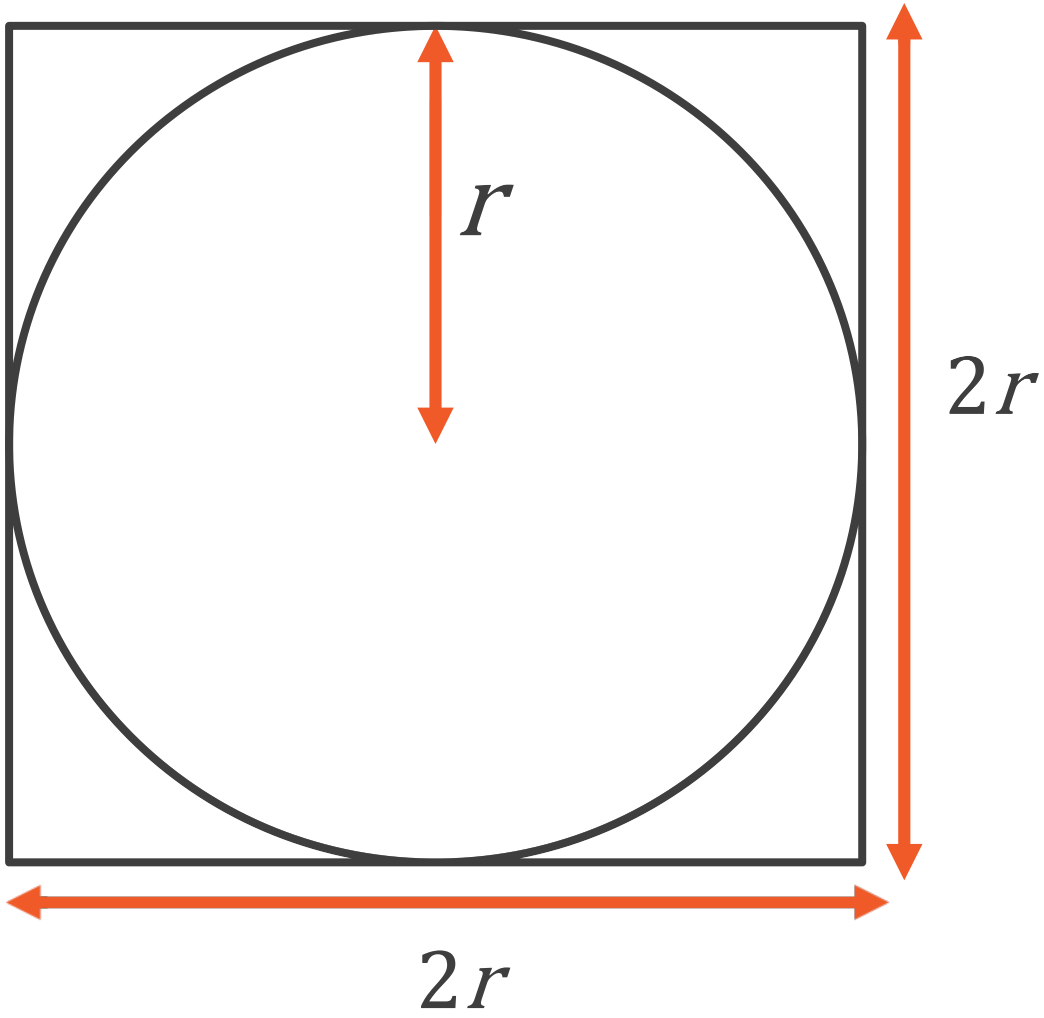
Financial pricing

Monte Carlo Method to Calculate Pi



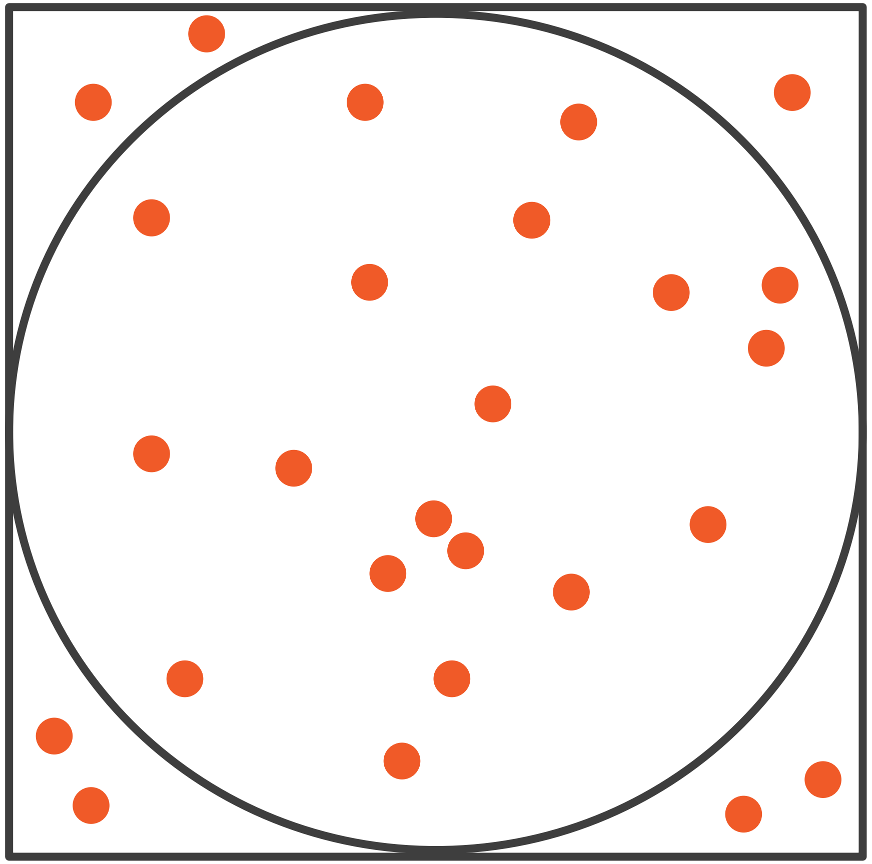
$$A = \pi r^2$$

Monte Carlo Method to Calculate Pi



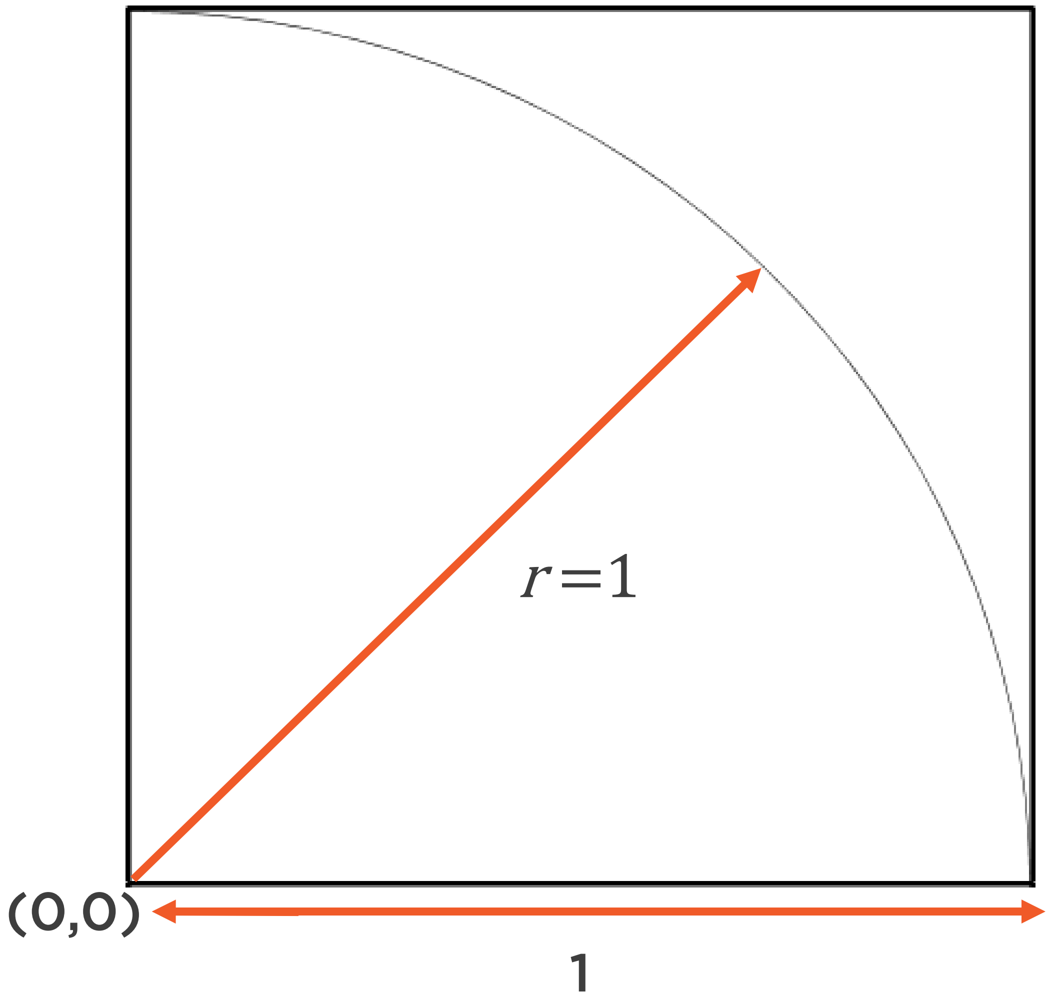
$$A = 4r^2$$

Monte Carlo Method to Calculate Pi

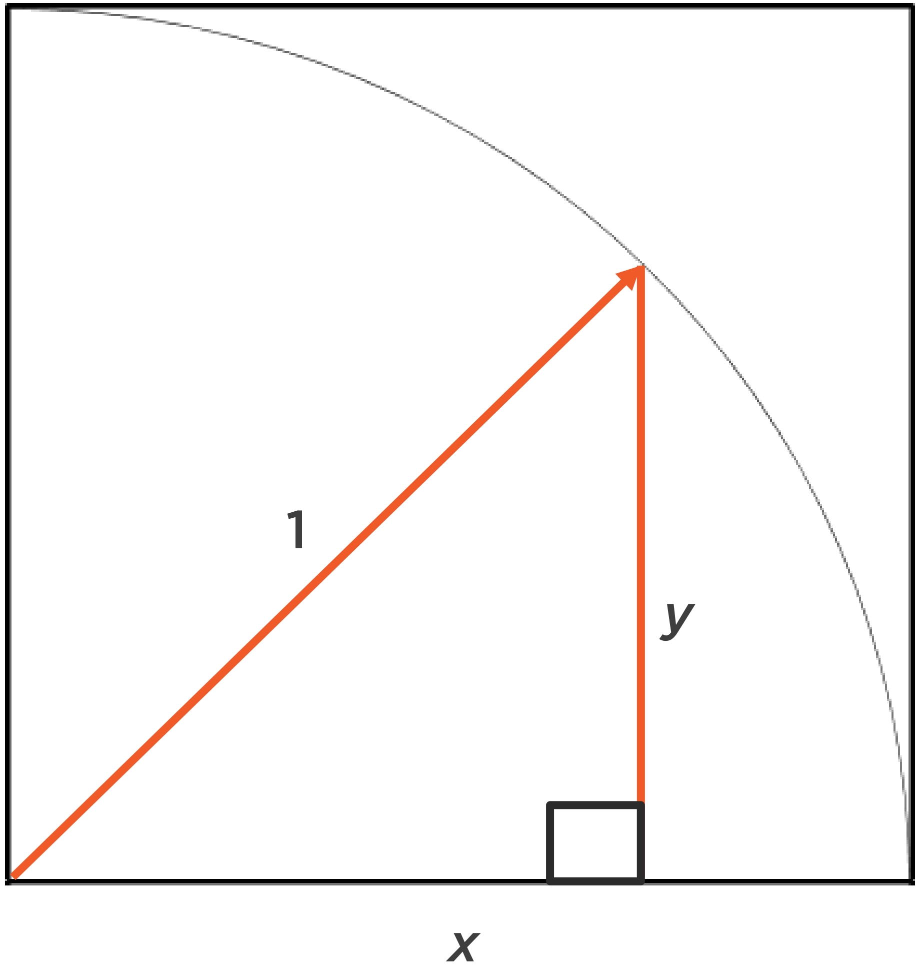


Total points : total in circle = Area of square : area of circle

Simplification



Simplification



If $x^2 + y^2 \leq 1$, point is inside circle

Otherwise, outside

Multithreaded Version

This an intermediate stage – multithreaded without a thread pool

- I’m leaving this as an exercise
- The only tricky bit is knowing if all the threads are completed
- Hint: Join can be used

Results

	PI estimate	Time taken (ms, 2sf)
Single threaded	3.141432	2800
Single threaded (ThreadLocalRandom)	3.14135472	400
Thread pool (4 workers, 4 tasks)	3.14132512	150

Timed Tasks

E.g. Heartbeat tasks, cancel remaining tasks after a period

Old way via Timer class and submitting TimerTasks

Problems with TimerTasks

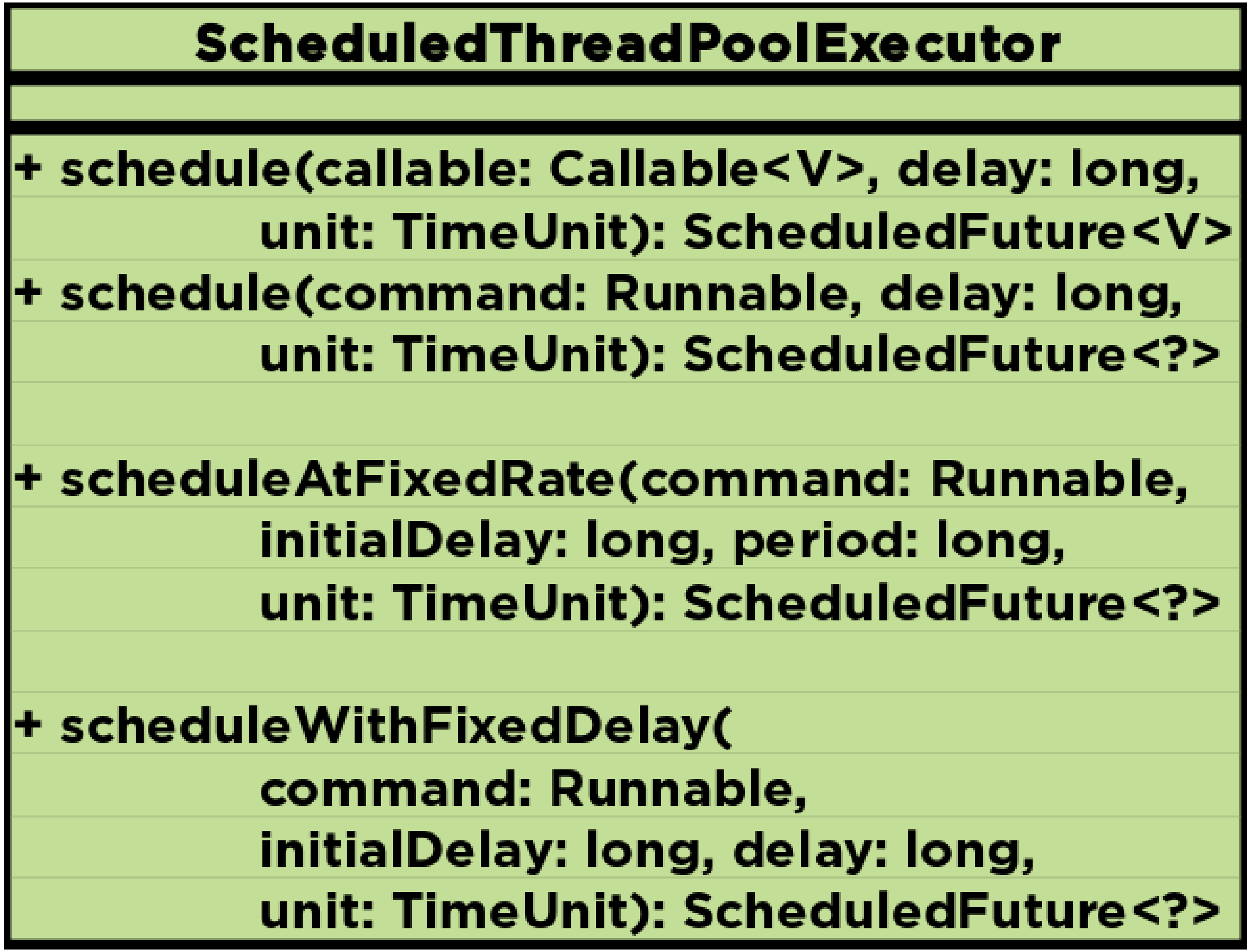
Only a single thread to run the tasks – what if tasks overrun?

May run late tasks in succession or not at all

Doesn't handle unchecked exceptions causing the timer thread to exit

Treat as deprecated

ScheduledThreadPoolExecutor



ScheduledThreadPoolExecutor

```
+ schedule(callable: Callable<V>, delay: long,
          unit: TimeUnit): ScheduledFuture<V>
+ schedule(command: Runnable, delay: long,
          unit: TimeUnit): ScheduledFuture<?>

+ scheduleAtFixedRate(command: Runnable,
          initialDelay: long, period: long,
          unit: TimeUnit): ScheduledFuture<?>

+ scheduleWithFixedDelay(
          command: Runnable,
          initialDelay: long, delay: long,
          unit: TimeUnit): ScheduledFuture<?>
```

Thread Pools Summary

Handle creating and managing threads on our behalf

Maintain core number of threads

Create extra threads if core threads are occupied

Submitting Tasks Summary

Pass task via submit as:

- Runnable
- Runnable and result object
- Callable

Receive Future back which can:

- Check for result
- Block for result (optional timer)
- Cancel task

Periodic Tasks

Timer and TimerTask are not reliable

Use ScheduledThreadPoolExecutor