Introduction to the Java ExecutorService

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

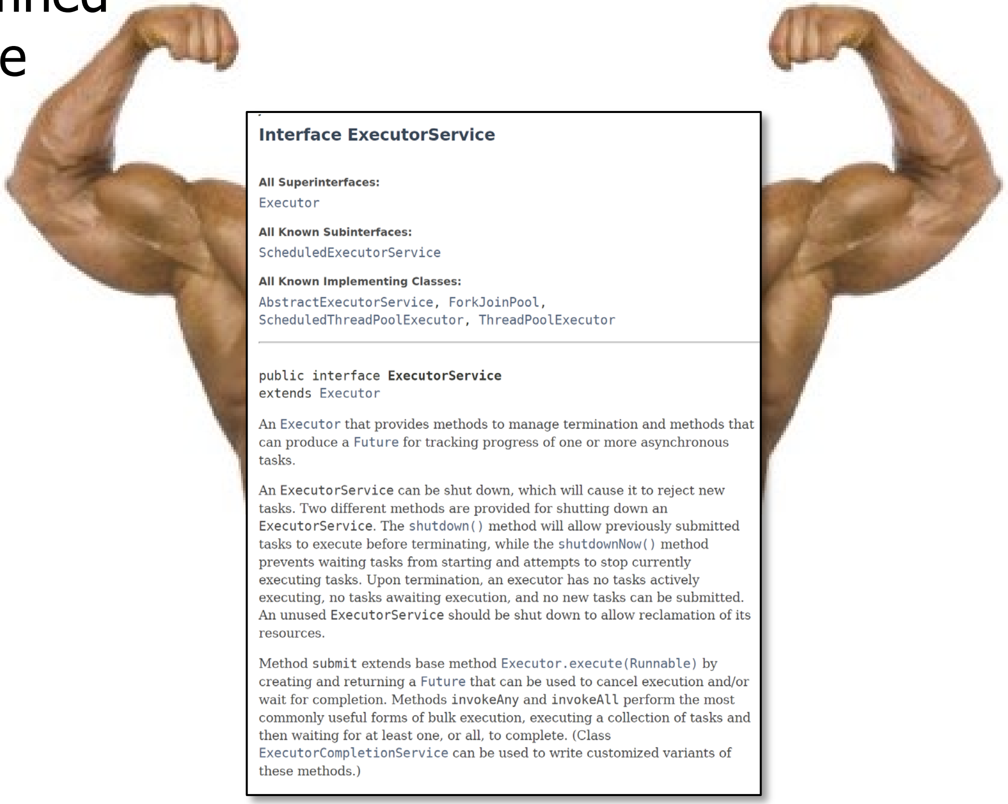
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Recognize the powerful features defined in the Java ExecutorService interface
  - & provided by its associated implementations to manage the lifecycle of concurrent tasks



Interface ExecutorService

All Superinterfaces:
`Executor`

All Known Subinterfaces:
`ScheduledExecutorService`

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

```
public interface ExecutorService
extends Executor
```

An `Executor` that provides methods to manage termination and methods that can produce a `Future` for tracking progress of one or more asynchronous tasks.

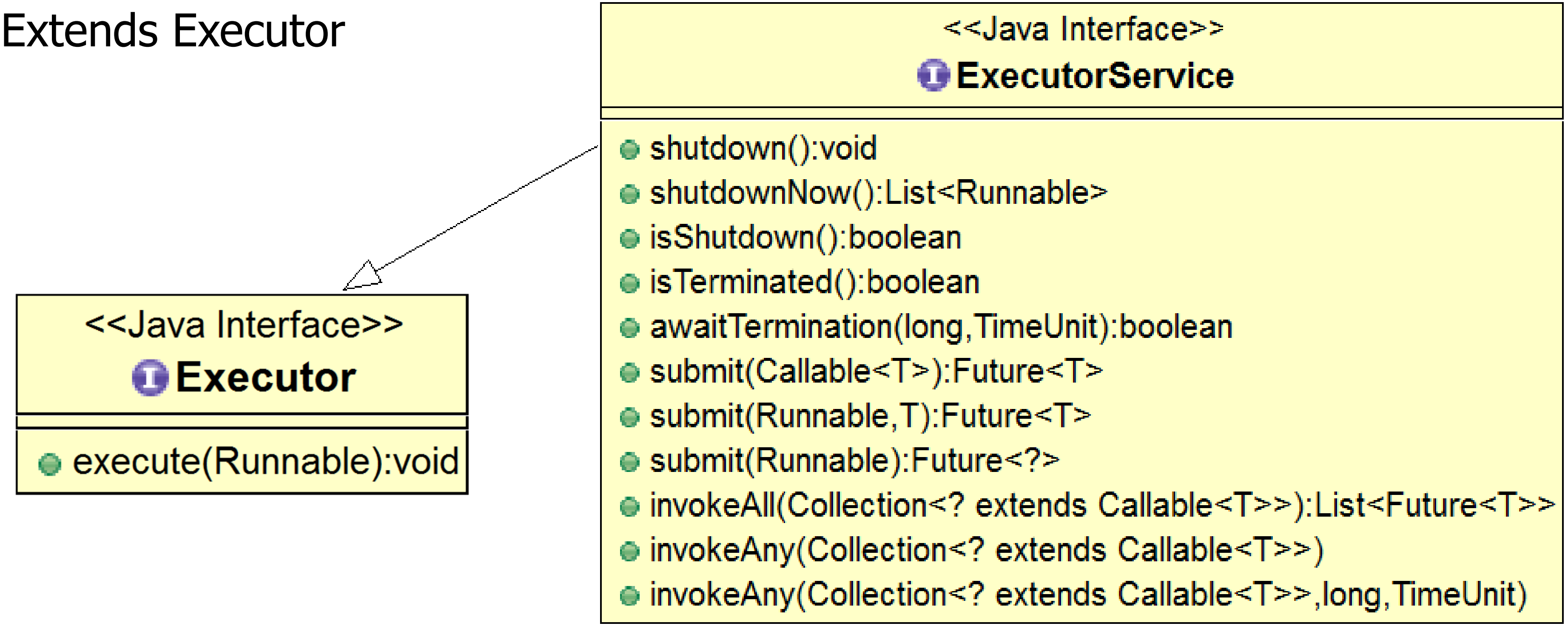
An `ExecutorService` can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an `ExecutorService`. The `shutdown()` method will allow previously submitted tasks to execute before terminating, while the `shutdownNow()` method prevents waiting tasks from starting and attempts to stop currently executing tasks. Upon termination, an executor has no tasks actively executing, no tasks awaiting execution, and no new tasks can be submitted. An unused `ExecutorService` should be shut down to allow reclamation of its resources.

Method `submit` extends base method `Executor.execute(Runnable)` by creating and returning a `Future` that can be used to cancel execution and/or wait for completion. Methods `invokeAny` and `invokeAll` perform the most commonly useful forms of bulk execution, executing a collection of tasks and then waiting for at least one, or all, to complete. (Class `ExecutorCompletionService` can be used to write customized variants of these methods.)

Overview of the ExecutorService Interface

Overview of the ExecutorService Interface

- Extends Executor

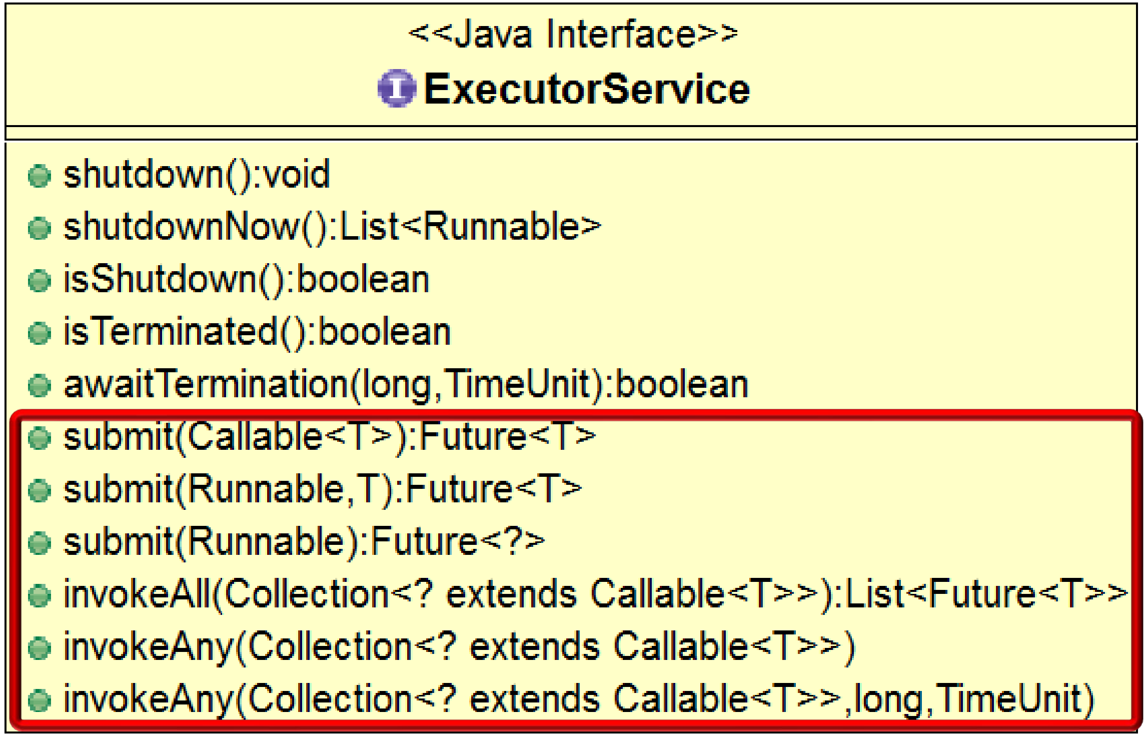


<<Java Interface>>
ExecutorService
shutdown():void
shutdownNow():List<Runnable>
isShutdown():boolean
isTerminated():boolean
awaitTermination(long,TimeUnit):boolean
submit(Callable<T>):Future<T>
submit(Runnable,T):Future<T>
submit(Runnable):Future<?>
invokeAll(Collection<? extends Callable<T>>):List<Future<T>>
invokeAny(Collection<? extends Callable<T>>)
invokeAny(Collection<? extends Callable<T>>,long,TimeUnit)
<<Java Interface>>
Executor
execute(Runnable):void

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

Overview of the ExecutorService Interface

- Extends Executor
  - Submit 1+ tasks & return futures for these tasks



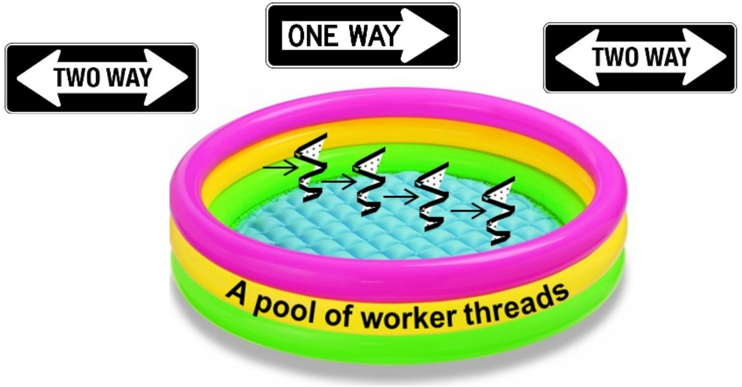
<<Java Interface>>
ExecutorService

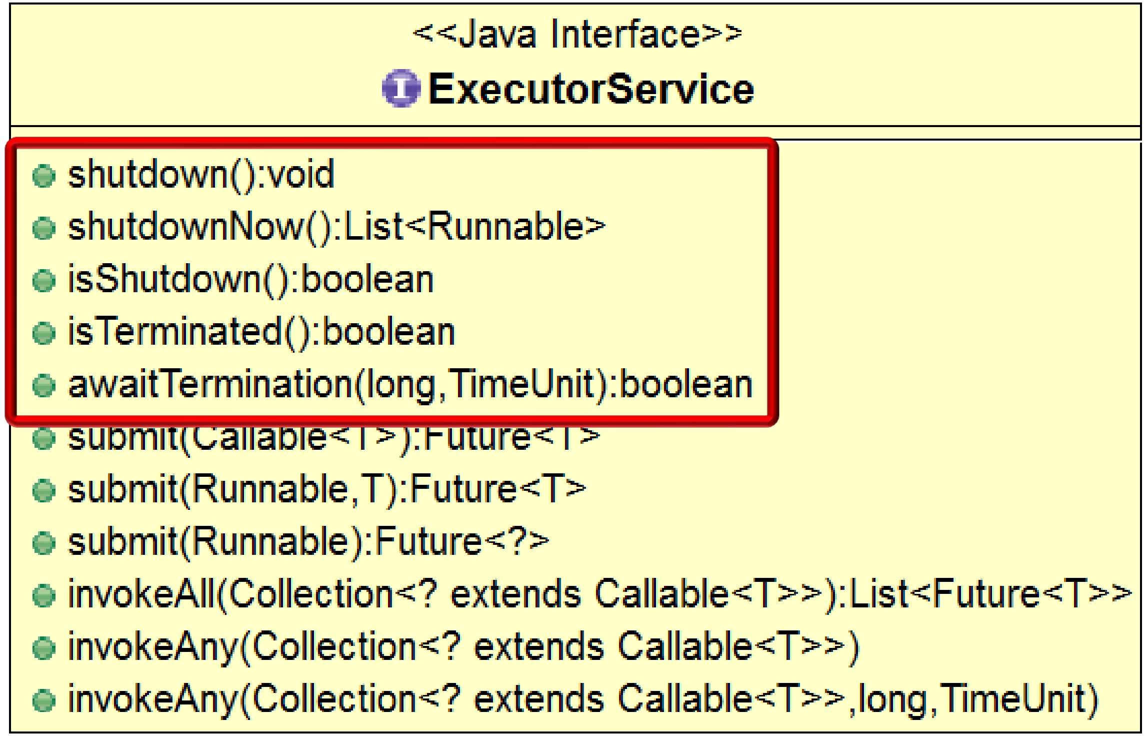
```
shutdown():void
shutdownNow():List<Runnable>
isShutdown():boolean
isTerminated():boolean
awaitTermination(long,TimeUnit):boolean
submit(Callable<T>):Future<T>
submit(Runnable,T):Future<T>
submit(Runnable):Future<?>
invokeAll(Collection<? extends Callable<T>>):List<Future<T>>
invokeAny(Collection<? extends Callable<T>>)
invokeAny(Collection<? extends Callable<T>>,long,TimeUnit)
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

Overview of the ExecutorService Interface

- Extends Executor
  - Submit 1+ tasks & return futures for these tasks
  - Manage lifecycle of tasks & executor service itself
    - e.g., interrupts worker threads in a pool





<<Java Interface>>
ExecutorService

- shutdown():void
- shutdownNow():List<Runnable>
- isShutdown():boolean
- isTerminated():boolean
- awaitTermination(long,TimeUnit):boolean
- submit(Callable<T>):Future<T>
- submit(Runnable,T):Future<T>
- submit(Runnable):Future<?>
- invokeAll(Collection<? extends Callable<T>>):List<Future<T>>
- invokeAny(Collection<? extends Callable<T>>)
- invokeAny(Collection<? extends Callable<T>>,long,TimeUnit)

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

Overview of the ExecutorService Interface

- A task is a unit of computation that (ideally) does not depend on the state, result, or side effects of other tasks

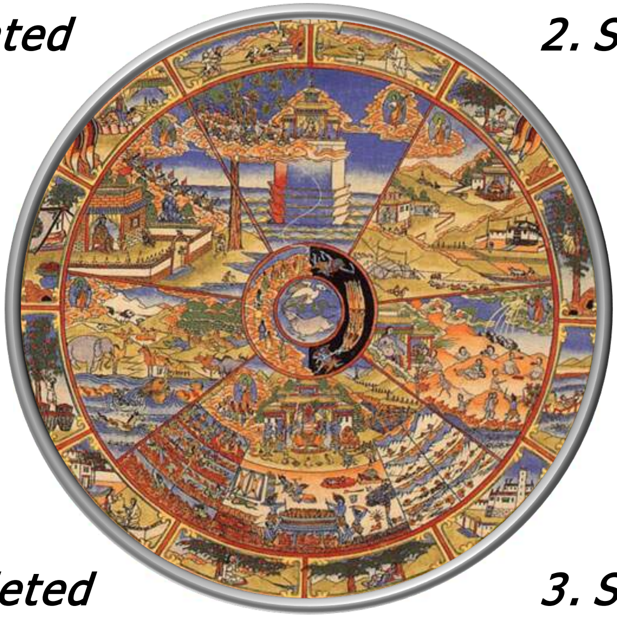
See www.javaworld.com/article/2071822/book-excerpt--executing-tasks-in-threads.html

Overview of the ExecutorService Interface

- A task is a unit of computation that (ideally) does not depend on the state, result, or side effects of other tasks
  - A task lifecycle has four phases

1. Created

2. Submitted



4. Completed

3. Started

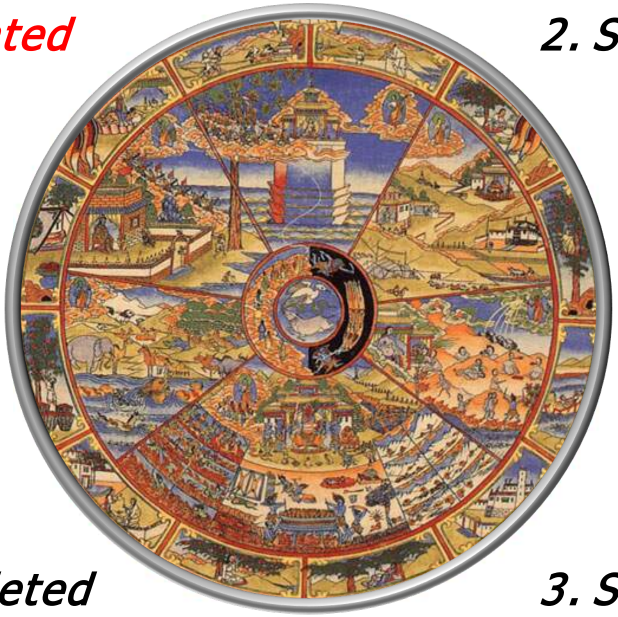
See en.wikipedia.org/wiki/Samsara

Overview of the ExecutorService Interface

- A task is a unit of computation that (ideally) does not depend on the state, result, or side effects of other tasks
  - A task lifecycle has four phases

    1. Created

      - A new task is instantiated



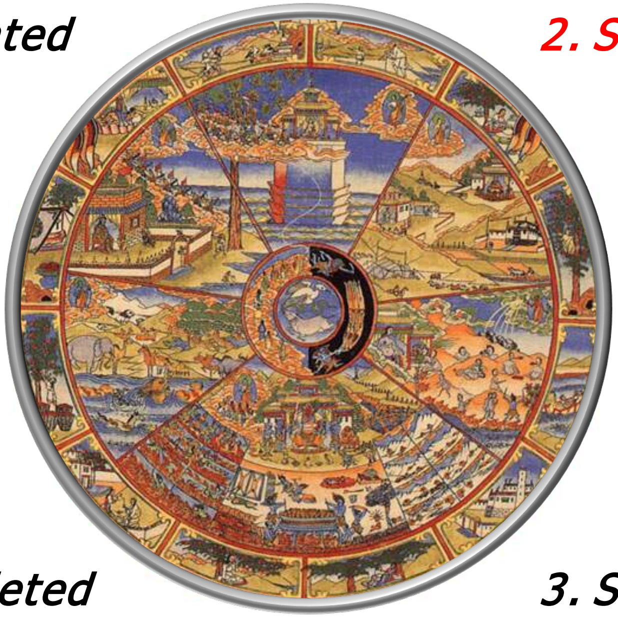
Overview of the ExecutorService Interface

- A task is a unit of computation that (ideally) does not depend on the state, result, or side effects of other tasks
  - A task lifecycle has four phases

    1. Created

    2. Submitted

    - A task is given to an executor service to run
      - e.g., via execute() or submit()

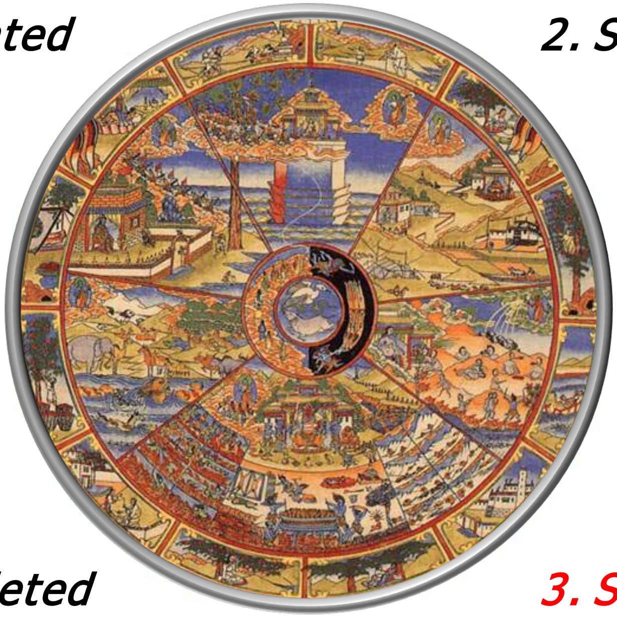


Overview of the ExecutorService Interface

- A task is a unit of computation that (ideally) does not depend on the state, result, or side effects of other tasks
  - A task lifecycle has four phases
    1. *Created*
    2. *Submitted*
    3. ***Started***
       - A task is executed by a worker thread in the executor service
         - e.g., via its call() or run() hook method

1. Created

2. Submitted

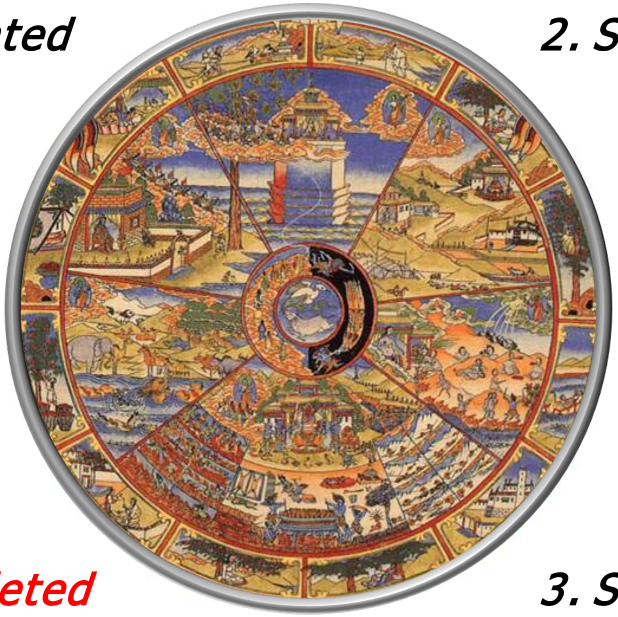


4. Completed

3. Started

Overview of the ExecutorService Interface

- A task is a unit of computation that (ideally) does not depend on the state, result, or side effects of other tasks
  - A task lifecycle has four phases
    1. *Created*
    2. *Submitted*
    3. *Started*
    4. ***Completed***
       - A task finishes (un)successfully or is cancelled
         - e.g., via cancel()



End of Introduction to the Java Executor Service