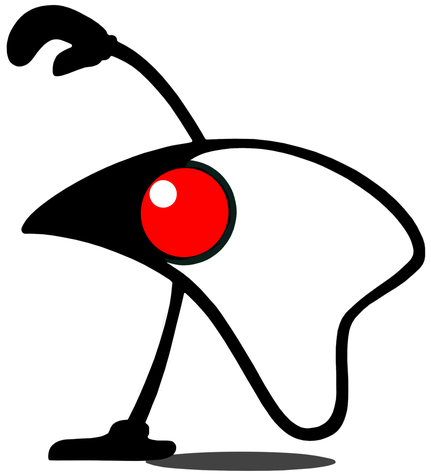
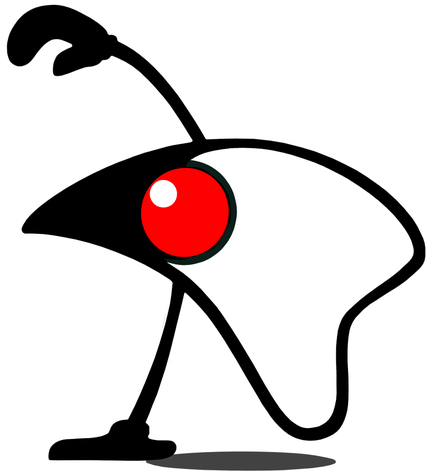


The Java Sessions

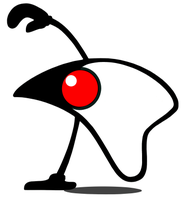
Futures

Daniel Hinojosa
@dhinojosa



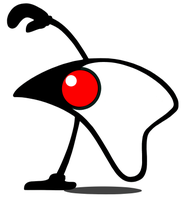
Slides and Code Available

https://github.com/dhinojosa/
java-sessions-futures



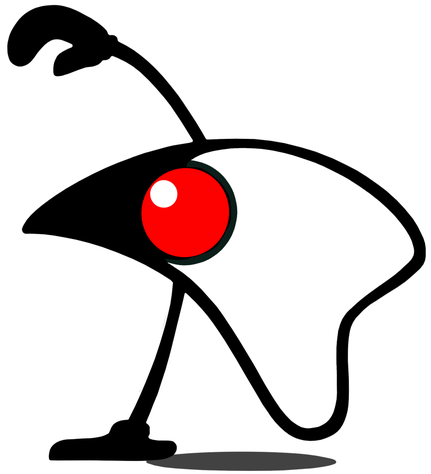
What is this about?

This is a basic introduction in using a `Future<V>`, which is a requirement for any concurrent or asynchronous programming

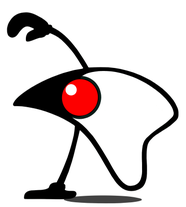


Where are Futures used?

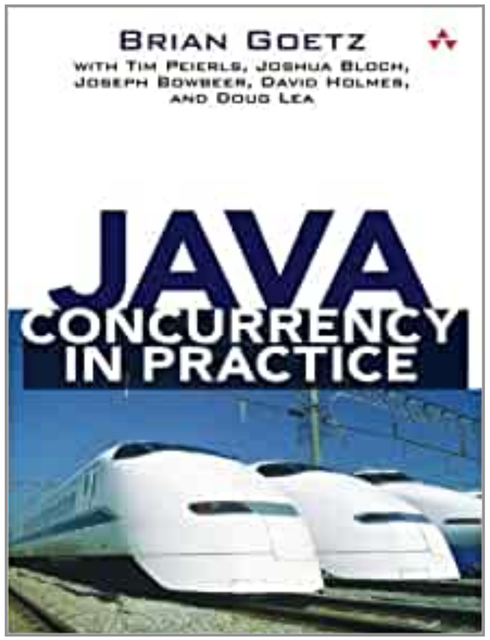
Everywhere... `Future<V>` is an "essential currency" for anything asynchronous



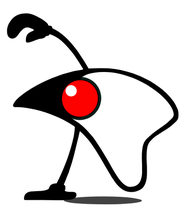
About Futures



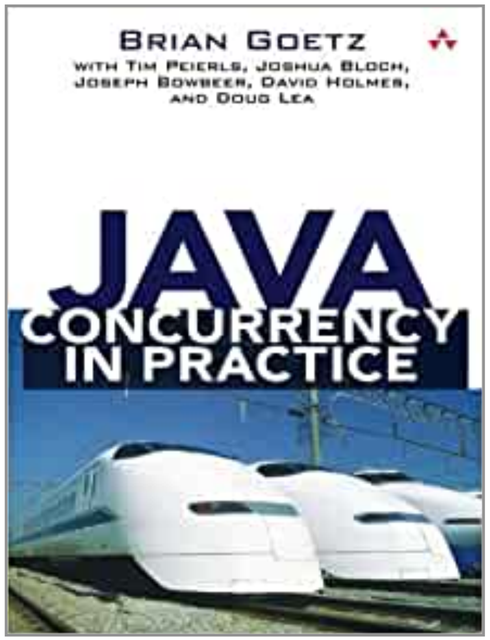
Future def.



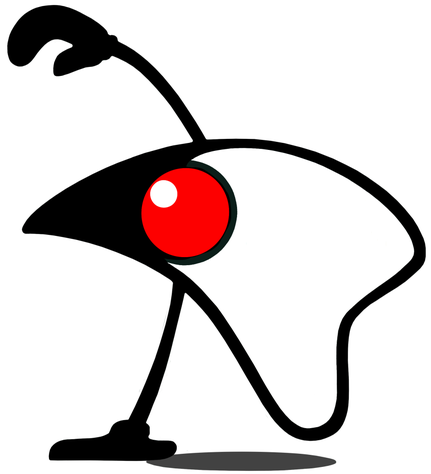
"Future represents the lifecycle of a task and provides methods to test whether the task has completed or has been cancelled"



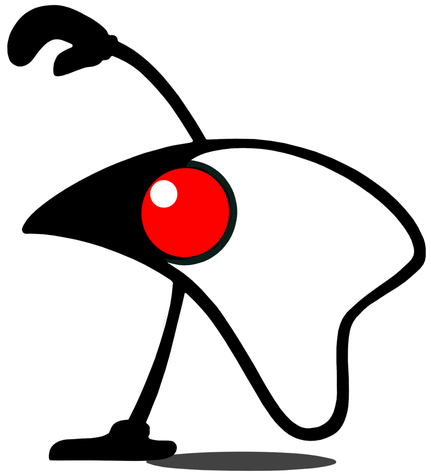
Future def.



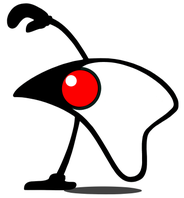
“Future can only move forwards and once complete it stays in that state forever. ”



Thread Pool Varieties

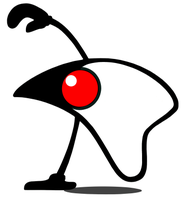


Fixed Thread Pool



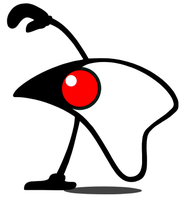
Fixed Thread Pool

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.



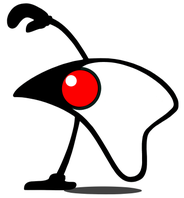
Fixed Thread Pool

Keeps threads constant and uses the queue to manage tasks waiting to be run



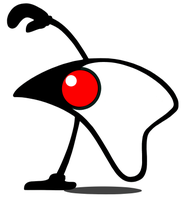
Fixed Thread Pool

If a thread fails, a new one is created in its stead



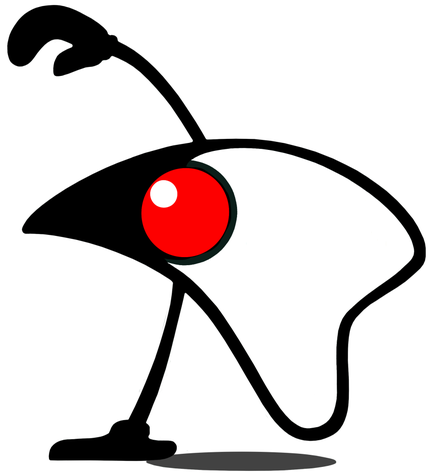
Fixed Thread Pool

If all threads are taken up, it will wait on an unbounded queue for the next available thread

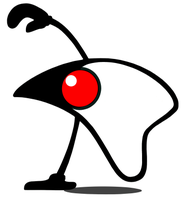


Fixed Thread Pool

```
ExecutorService executorService =
  Executors.newFixedThreadPool();
```

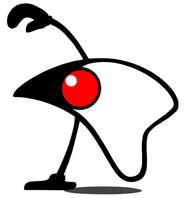


Cached Thread Pool



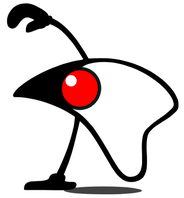
Cached Thread Pool

Flexible thread pool implementation that will reuse previously constructed threads if they are available



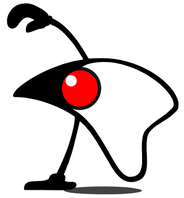
Cached Thread Pool

If no existing thread is available, a new thread is created and added to the pool



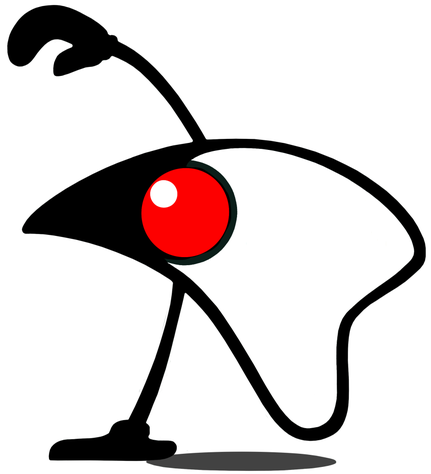
Cached Thread Pool

Threads that have not been used for sixty seconds are terminated and removed from the cache

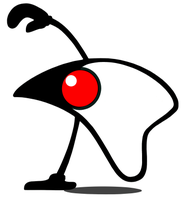


Fixed Thread Pool

```
ExecutorService executorService =
  Executors.newCachedThreadPool();
```

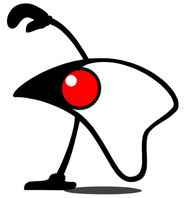


Single Thread Executor



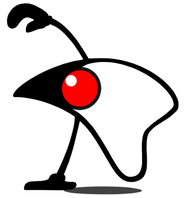
Single Thread Executor

Creates an `Executor` that uses a single worker thread operating off an unbounded queue



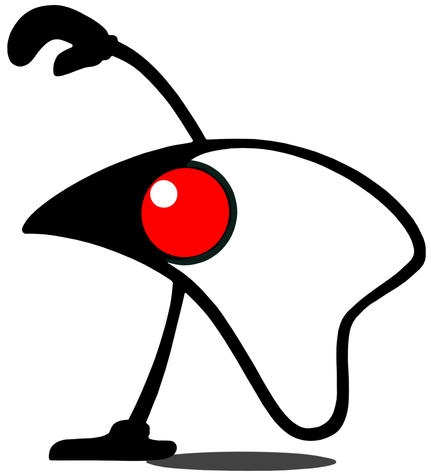
Single Thread Executor

If a thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.

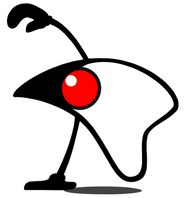


Fixed Thread Pool

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

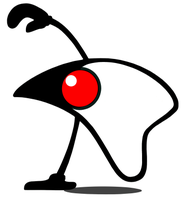


Scheduled Thread Pool



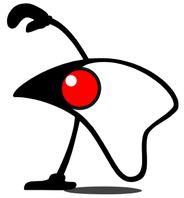
Scheduled Thread Pool

Can run your tasks after a delay or periodically



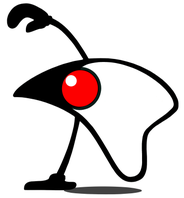
Scheduled Thread Pool

This method does not return an `ExecutorService`, but a `ScheduledExecutorService`



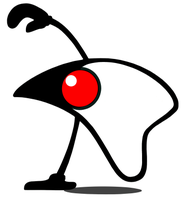
Scheduled Thread Pool

Runs periodically until `canceled()` is called.



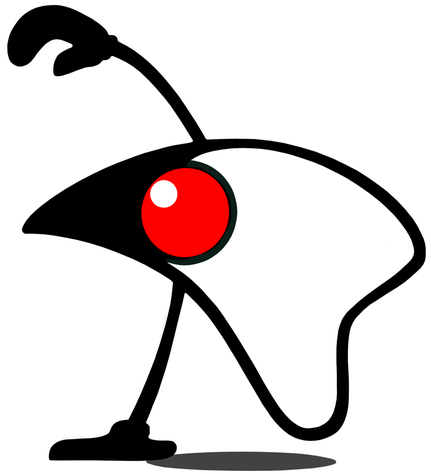
Scheduled Thread Pool

Returns a `ScheduledFuture<V>`

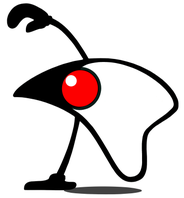


Scheduled Thread Pool

```
ScheduledExecutorService executorService =
  Executors.newScheduledThreadPool();
```

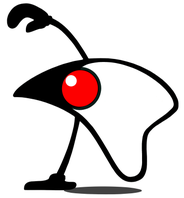


Fork-Join Thread Pool



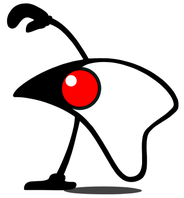
Fork Join Pool

An `ExecutorService`, that participates in *work-stealing*.



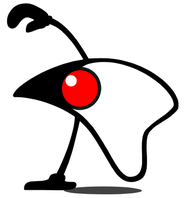
Fork Join Pool

- By default when a task creates other tasks (`ForkJoinTasks`) they are placed on the same on queue as the main task.
- work-stealing is when a processor runs out of work, it looks at the queues of other processors and "steals" their work items.



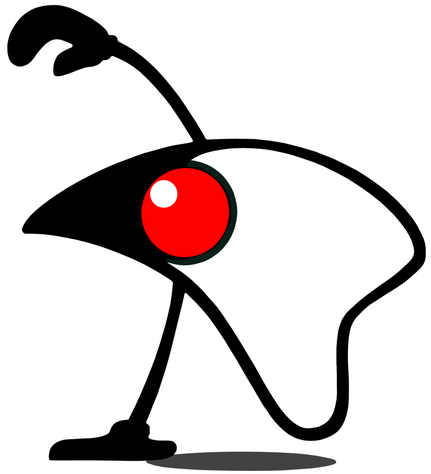
Fork Join Pool

- Not a member of `Executors`
- Created by instantiation

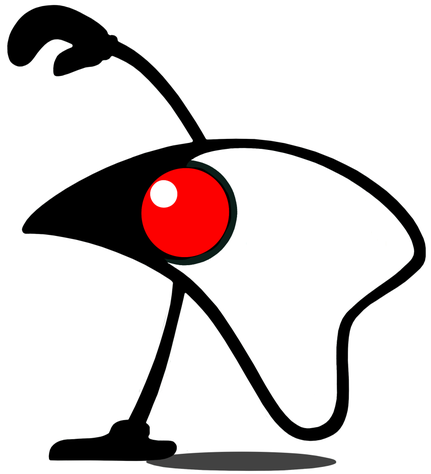


Fork Join Pool

Brought up since this will be in many cases the "default" thread pool on the JVM

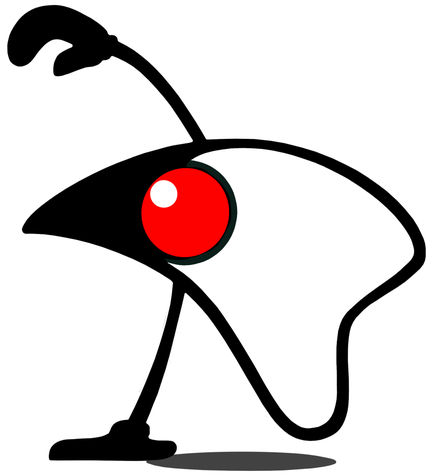


Futures Circa JDK 5

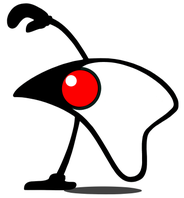


Demo & Challenges

Basic Futures

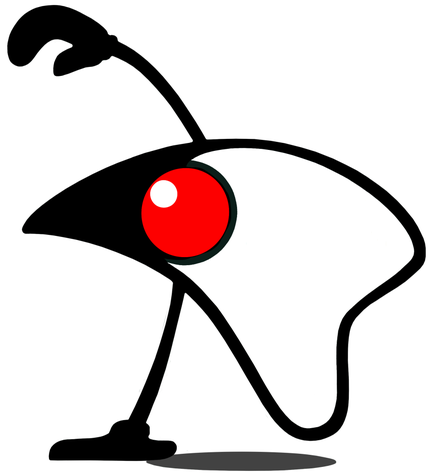


Scheduled Delay

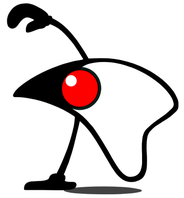


Scheduled Delay

Creates a `ScheduledFuture<T>` that will be enabled after a given delay.

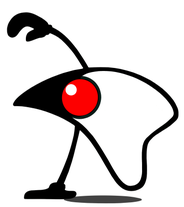


Delay vs. Rate

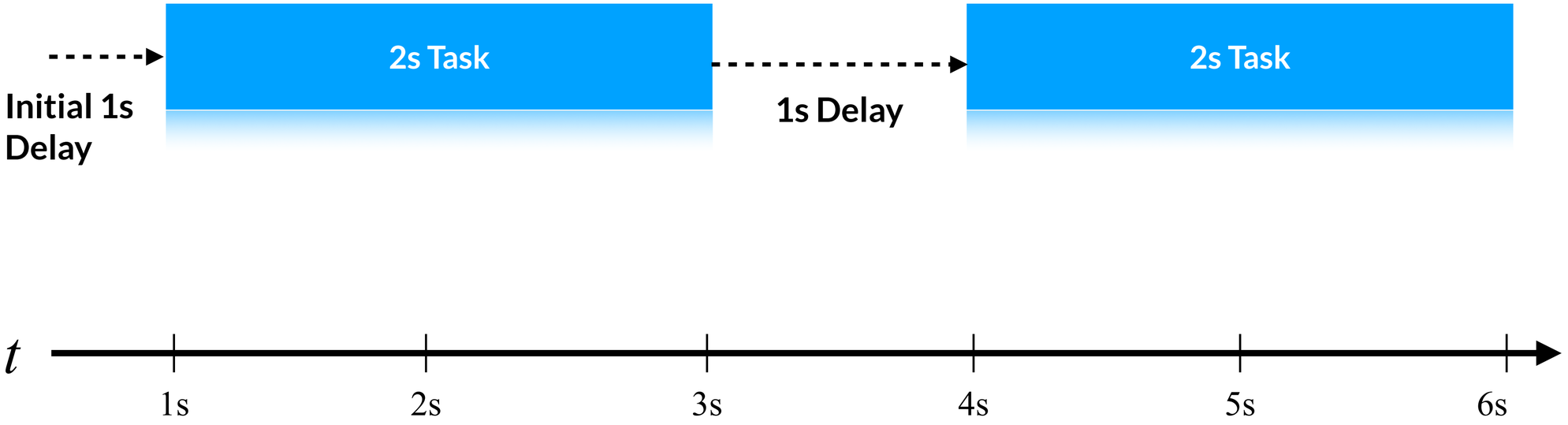


Delay vs. Rate

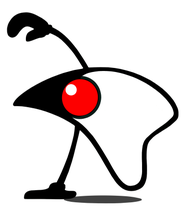
Both only accept `java.lang.Runnable` not `java.util.concurrent.Callable`



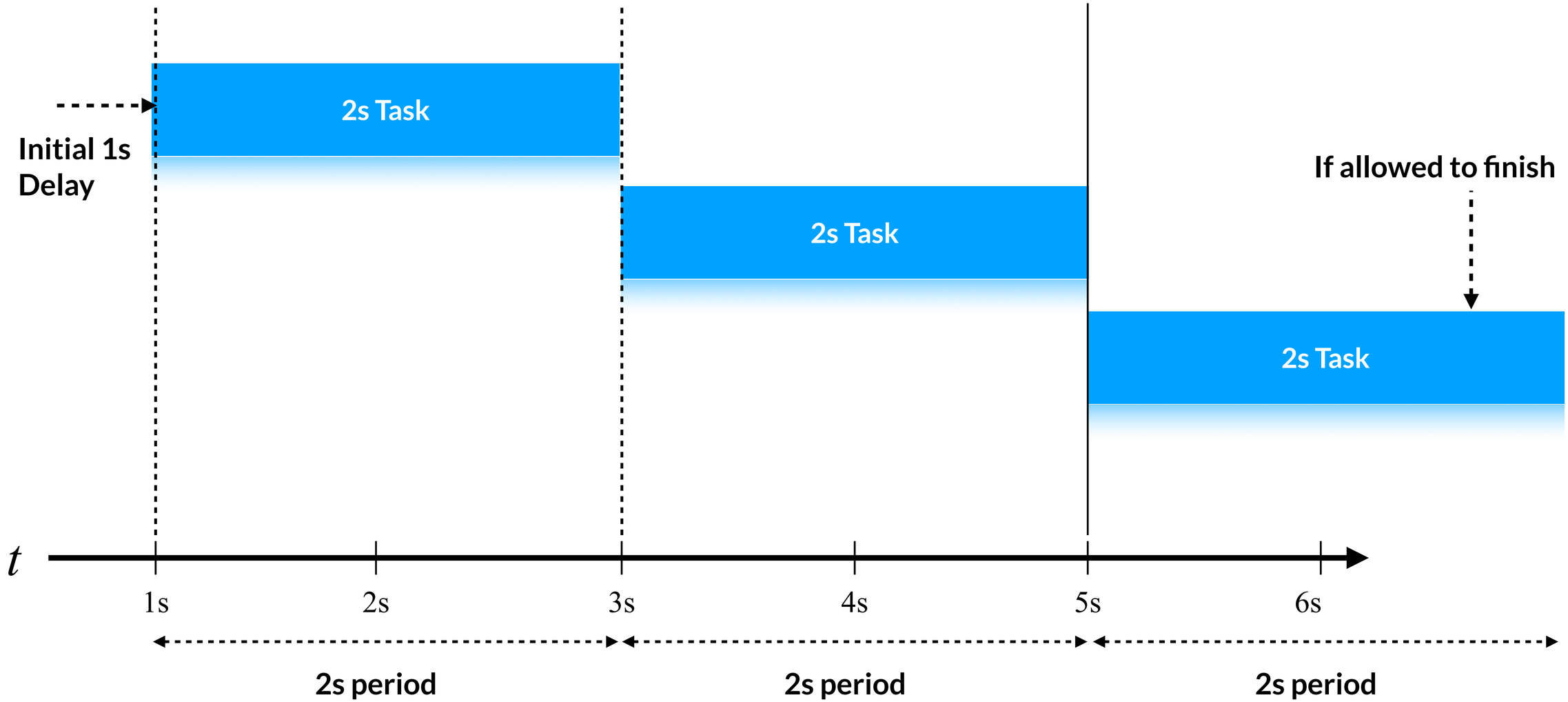
Delay [2s Task]

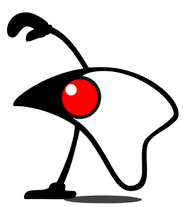


Warning: Any task can be starved by the previous task.

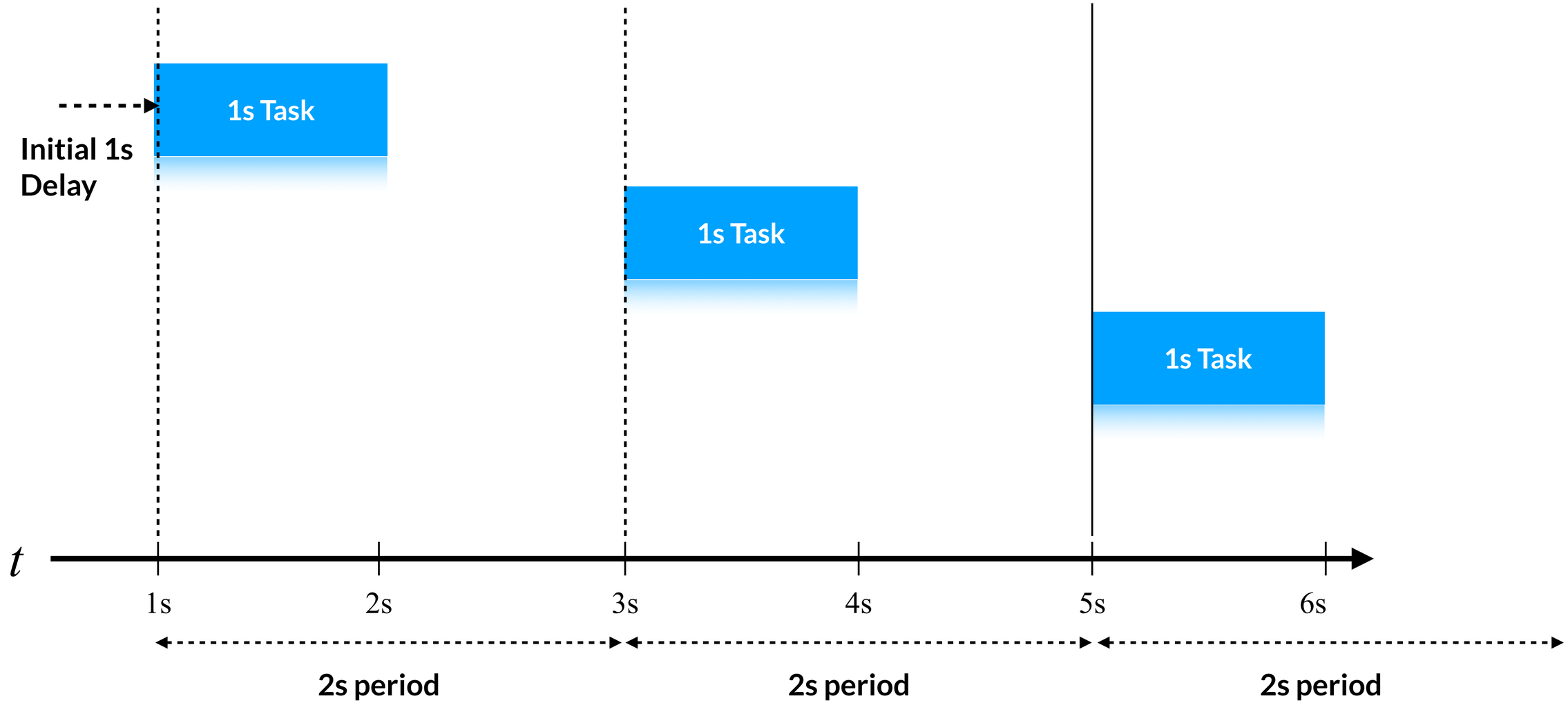


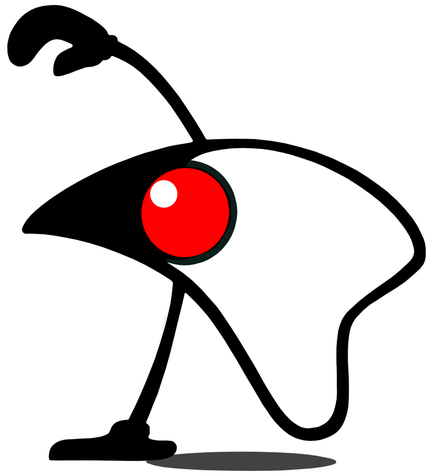
Rate [2s Task]


Initial 1s
Delay
2s Task
2s Task
2s Task
If allowed to finish
t
1s
2s
3s
4s
5s
6s
2s period
2s period
2s period



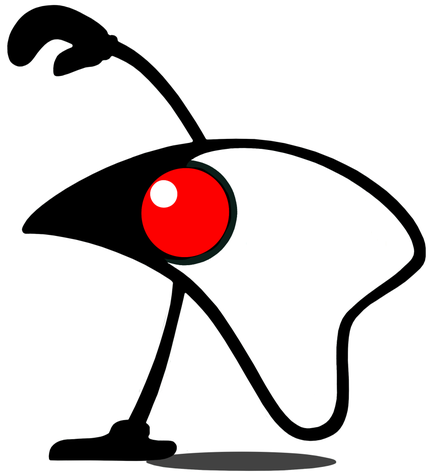
Rate [1s Task]


1s Task
Initial 1s
Delay
1s Task
1s Task
t
1s
2s
3s
4s
5s
6s
2s period
2s period
2s period

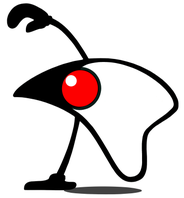


Demo

Scheduled Futures

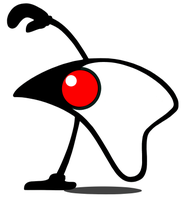


Guava 2009



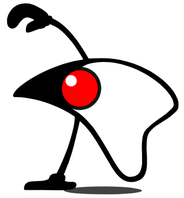
Listening Executors

- Uses `MoreExecutors` to wrap around an `ExecutorService`
- This in turn returns provide a different `Future` called `ListenableFuture<V>` that extends `Future<V>`
- Extensively uses utility class
- Futures for static utility methods to chain operations.



Listening Executors

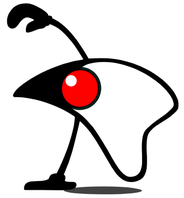
- `ListenableFuture<V>` contains callbacks to make asynchrony easier.
- This was a solid choice if you were stuck in JDK 7 or less, although you shouldn’t be



Listening Executors

Analogies in Guava Futures

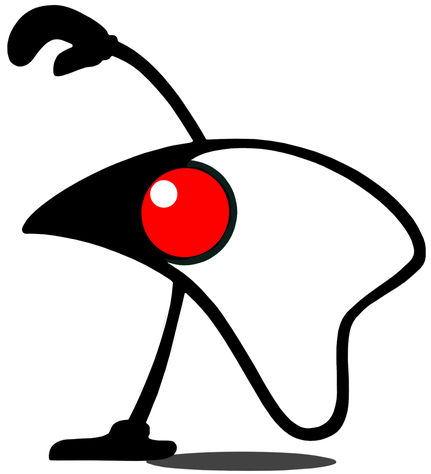
- `transform(...)` = map
- `transformAsync(...)` = flatMap
- `addCallback(...)` = final processing



Listening Executors

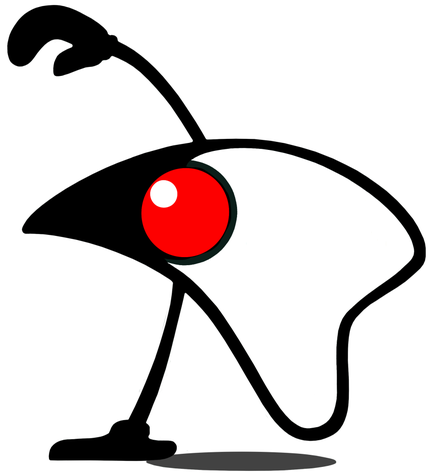
Analogies in Guava `ListenableFuture<V>`

- `addListener(...)` = final processing

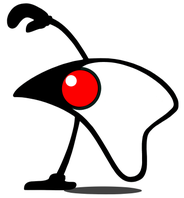


Demo

Guava ListenableFuture

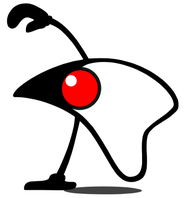


Completable Futures



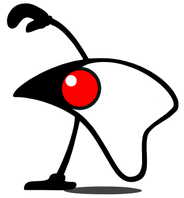
Completable Futures

Staged Completions of Interface
`java.util.concurrent.CompletionStage<T>`



Completable Futures

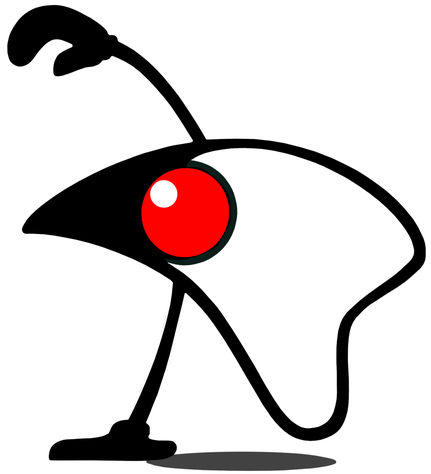
Ability to chain with `CompletableFuture<V>`



Completable Futures

Analogies in Completable Futures

- `thenApply(...)` = map
- `thenCompose(...)` = flatMap
- `thenCombine(...)` = Independent Combination
- `thenAccept(...)` = final processing



Demo & Challenges

Completable Futures