
UNDERSTANDING
JVM FUTURES
DANIEL HINDJOSA
@DHINDJOSA

https://github.com/dhinojosa/
understanding_jvm_futures

WHAT IS THIS ABOUT?

This is a basic introduction into using `Future<V>`, which is required for any concurrent and 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` can only move forwards and once complete it stays in that state forever.

Java Concurrency in Practice -Brian Goetz

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

```
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

CACHED THREAD POOL

```
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.

SINGLE THREAD EXECUTOR

```
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

```
Executors.newScheduledThreadPool()
```

FORK-JOIN THREAD POOL

FORK-JOIN THREAD POOL

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

FORK-JOIN THREAD POOL

By default when a task creates other tasks (`ForkJoinTasks`) they are placed on the same on queue as the main task.

FORK-JOIN THREAD POOL

work-stealing is when a processor runs out of work, it looks at the queues of other processors and "steals" their work items.

https://en.wikipedia.org/wiki/Work_stealing

FORK-JOIN THREAD POOL

Not a member of `Executors`.
Created by instantiation

FORK-JOIN THREAD POOL

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

DEMO: BASIC FUTURES

DEMO: ASYNC THE OLD WAY

DEMO: FUTURES WITH PARAMETERS


FUTURETASKS

FUTURETASKS

Base implementation for `Future<V>`.

DEMO: FUTURE TASKS

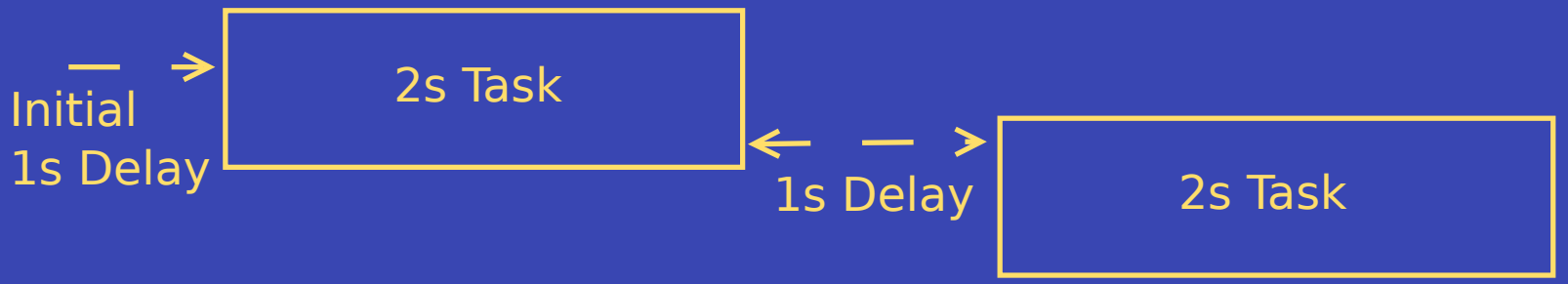

SCHEDULEDFUTURES

DELAY VS. RATE

DELAY VS. RATE

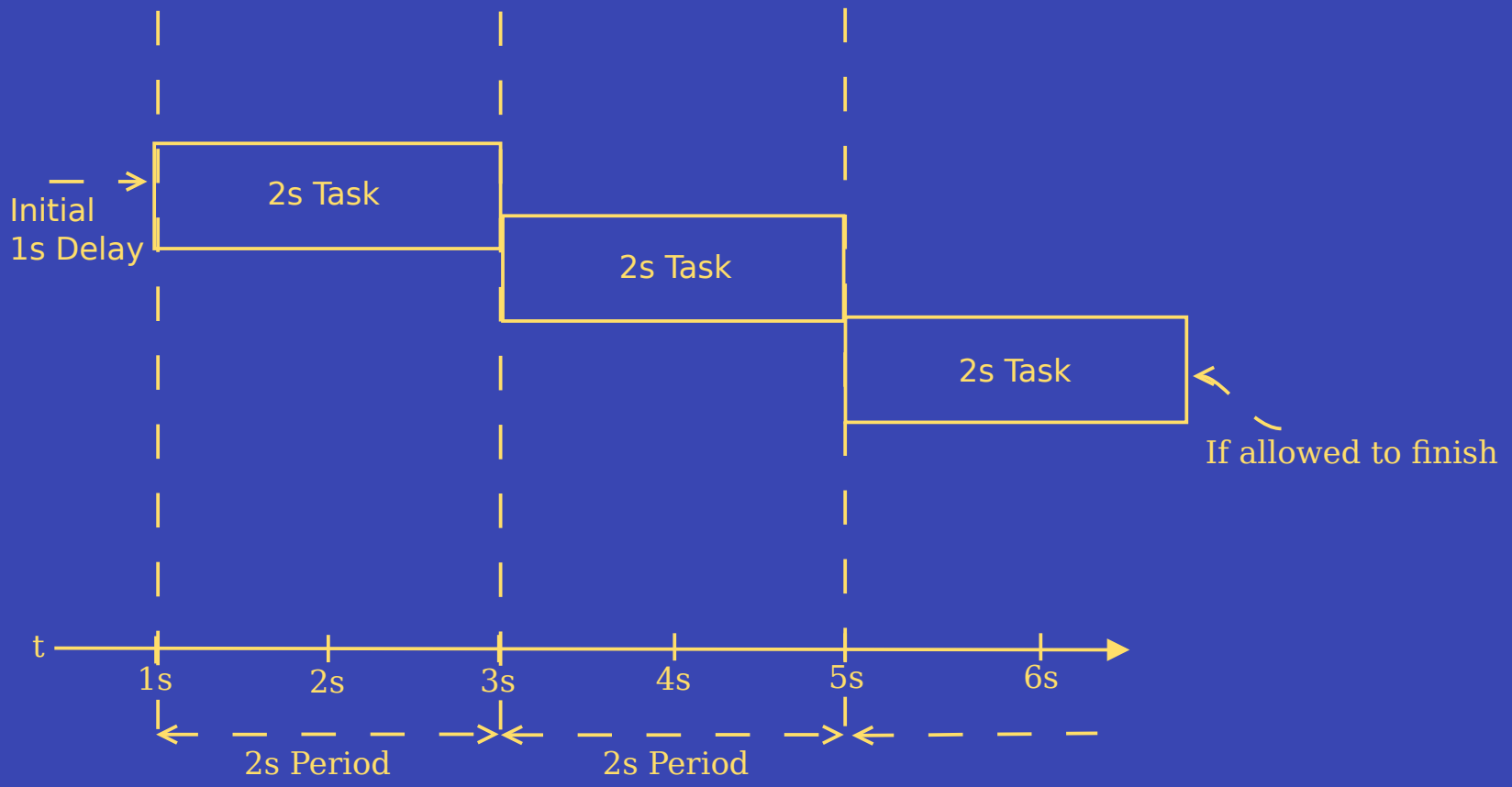
Both only accept
`java.lang.Runnable`
not
`java.util.concurrent.Callable`

DELAY(2S TASK)



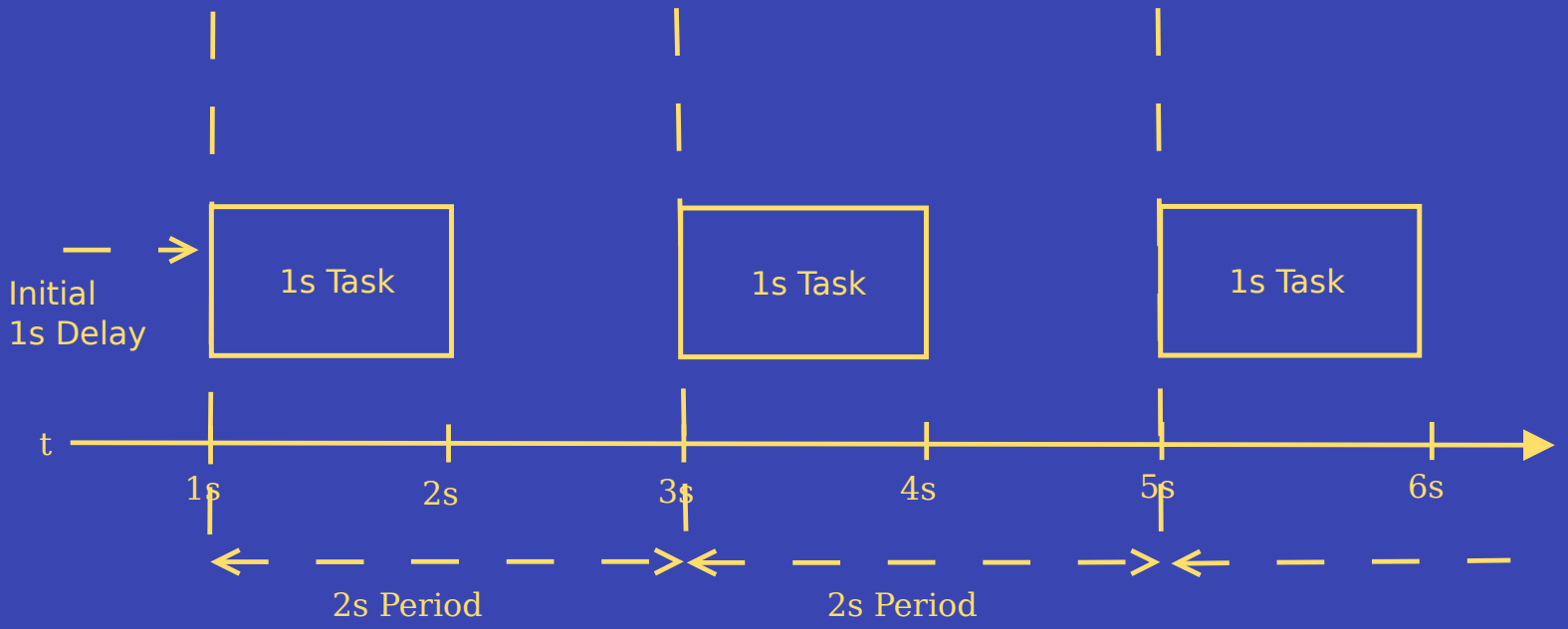
* Warning: Any task can be starved by the previous task.

RATE (2S TASK)



* Warning: If the task exceed the period, it is up to the JVM to decide when the subsequent task will begin.

RATE (1S TASK)


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

DEMO: SCHEDULED FUTURES


COMPLETABLE
FUTURES

COMPLETABLE FUTURES

Staged Completions of Interface

`java.util.concurrent.CompletionStage<T>`

COMPLETABLE FUTURES

Ability to chain functions to `Future<V>`

COMPLETABLE FUTURES

Analogies

```
thenApply(...)    = map

thenCompose(...)  = flatMap

thenCombine(...)  = independent
                     combination

theAccept(...)    = final processing
```

DEMO: COMPLETABLE FUTURES


GUAVA


GUAVA LISTENING
EXECUTORS

LISTENING EXECUTORS

Uses `MoreExecutors` to wrap around an `ExecutorService`

LISTENING EXECUTORS

This in turn returns provide a different `Future` called `ListenableFuture<V>` that extends `Future<V>`

LISTENING EXECUTORS

Extensively uses utility class `Futures` for static utility methods to chain operations.

LISTENING EXECUTORS

`ListenableFuture<V>` contains callback to make asynchrony easier.

LISTENING EXECUTORS

A solid choice if you are using JDK 7.0 or less (although you shouldn't be)

LISTENING EXECUTORS

Analogies in Futures

`transform(...) = map`

`transformAsync(...) = flatMap`

`addCallback = final processing`

LISTENING EXECUTORS

Analogies in
`ListenableFuture<V>`

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

DEMO: GUAVA LISTENING EXECUTORS

ALTERNATE JVM LANGUAGES


SCALA

SCALA PREPARATIONS

Require an `ExecutorService`

SCALA PREPARATIONS

Wrap up `Executor` in an
`java.util.ExecutionContext`

SCALA PREPARATIONS

Implicitly bind your

`ExecutionContext`

SCALA PREPARATIONS

or import:

```
scala.concurrent._

ExecutionContext.Implicits
                .global
```

SCALA PREPARATIONS

Create Future with:

```
Future {...}
```

or `Future.apply {...}`

DEMO: SCALA FUTURES

WHAT IS A PROMISE?

PROMISES

A reference that is given an explicit success or failure answer accept for a `Future` to consume.

PROMISES

Promises officially are available in Scala and Clojure. Promise though can be created in Java with a `CompletableFuture<V>`

DEMO: SCALA PROMISES


CLOJURE

CLOJURE PREPARATIONS

To establish a Future use:

```
(future [&body])
```

CLOJURE PREPARATIONS

Of course usually useful to assign it to var :

```
(def f (future [&body]))
```

CLOJURE PREPARATIONS

In order to get the value of the `Future` use `deref/@`:

`(deref f)` or `@f`

CLOJURE PREPARATIONS

To ask the status of a `Future` use `realized?`:

```
(realized? f)
```

DEMO: CLOJURE FUTURES

CLOJURE PROMISES

To establish a promise use:
`(promise)`

CLOJURE PROMISES

Of course you may need to assign that to a var too:

```
(def p (promise))
```

CLOJURE PROMISES

To fulfill a promise merely call deliver:

```
(deliver promise 40)
```

CLOJURE PROMISES

`deref` works the same:

`(deref p)` or `@p`

CLOJURE PROMISES

`realize` works the same:

```
(realize? p)
```

DEMO: CLOJURE PROMISES


QUESTIONS?

THANK YOU

https://github.com/dhinojosa

https://twitter.com/dhinojosa

http://gplus.to/dhinojosa

http://www.linkedin.com/in/dhevolutionnext