Advanced Java Concurrent Patterns

INTRODUCING THE EXECUTOR PATTERN, FUTURES, AND CALLABLES

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Advanced concurrency: how to improve the Runnable Pattern

First, the Executor Pattern

Then the Java primitives introduced to synchronize tasks

The Compare and Swap, and atomic variables

And browse the Concurrent Collections and Maps from the Collection Framework

Java concurrent primitives

The Java Lock and Semaphore, we will implement the Producer / Consumer pattern with them

The Barrier and Latch, and see the patterns to use them properly

This is a Java course

Fair knowledge of the language and its main API

The Collection framework

Some knowledge about concurrency: the Runnable Pattern, the Producer / Consumer pattern, visibility, happens-before concept

Pluralsight: Applying Concurrency and Multi-Threading to Common Java Patterns

Agenda

Introducing the Executor Pattern, Futures and Callables

Using Locks and Semaphores for the Producer / Consumer Pattern

Controlling Concurrent Applications Using Barriers and Latches

Understanding Compare and Swap (CASing) and Atomic Variables

Leveraging Concurrent Collections to Simplify Application Design

Introducing Executors

```
Runnable task = () -> System.out.println("Hello world!");

Thread thread = new Thread(task);

thread.start();
```

1) A task is an instance of Runnable
2) It is passed to a new instance of Thread
3) The thread is launched

```
Runnable task = () -> System.out.println("Hello world!");

Thread thread = new Thread(task);

thread.start();
```

1) A thread is created on demand, by the user
2) Once the task is done, the thread dies
3) Problem: a thread is an expensive resource…

The Executor Pattern Aims to Fix These Issues

How can we improve the use of threads, as resources?

By creating pools of ready-to-use threads, and using them

Instead of creating a thread with a task as a parameter

We pass a task to a pool of threads, that will execute it

We need (at least) two patterns:

The first one to create a pool of threads

The second one to pass a task to this pool

```java
public interface Executor {
  void execute(Runnable task);
}
```

A pool of thread is an instance of the Executor interface

Implementations are provided in the JDK

```
public interface ExecutorService extends Executor {
   // 11 more methods
}
```

ExecutorService is an extension of Executor

The implementations of both interfaces are the same

The factory class Executors proposes ~20 methods to create executors

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

Let us build a pool with only one thread in it

The thread in this pool will be kept alive as long as this pool is alive

We can shutdown an ExecutorService (more on this later)

```
ExecutorService singleThreadExecutor =
    Executors.newSingleThreadExecutor();
ExecutorService multipleThreadsExecutor =
    Executors.newFixedThreadPoolExecutor(8);
```

The two most used executors:

- Single thread executor
- Fixed thread pool executor

```java
Executor executor =
    Executors.newSingleThreadExecutor();
Runnable task = () -> System.out.println("I run!");
```

Let us build an executor and a task

```
Executor executor =
    Executors.newSingleThreadExecutor();
Runnable task = () -> System.out.println("I run!");
executor.execute(task);
```

Let us build an executor and a task

And pass this task to the executor

```
// Executor pattern

executor.execute(task);


// Runnable pattern

new Thread(task).start();
```

Let us compare the two patterns:

- The Executor pattern does not create a new thread
- The behavior is the same: both calls return immediately, the task is executed in another thread

```java
Executor executor = Executors.newSingleThreadExecutor();

Runnable task1 = () -> someReallyLongProcess();

Runnable task2 = () -> anotherReallyLongProcess();

executor.execute(task1);

executor.execute(task2);
```

Suppose we run this code

Obviously, task2 has to wait for task1 to complete

The single thread executor has a waiting queue to handle that

```java
Executor executor = Executors.newSingleThreadExecutor();

Runnable task1 = () -> someReallyLongProcess();

Runnable task2 = () -> anotherReallyLongProcess();

executor.execute(task1);

executor.execute(task2);
```

This waiting queue is specified:

- A task is added to the waiting queue when no thread is available
- The tasks are executed in the order of their submission

```
Executor executor = Executors.newSingleThreadExecutor();

Runnable task1 = () -> someReallyLongProcess();

Runnable task2 = () -> anotherReallyLongProcess();

executor.execute(task1);

executor.execute(task2);
```

More questions:

Can we know if a task is done or not?

Can we cancel the execution of a task?

```
Executor executor = Executors.newSingleThreadExecutor();

Runnable task1 = () -> someReallyLongProcess();

Runnable task2 = () -> anotherReallyLongProcess();

executor.execute(task1);

executor.execute(task2);
```

More questions:

Can we know if a task is done or not? No...

Can we cancel the execution of a task?

```java
Executor executor = Executors.newSingleThreadExecutor();

Runnable task1 = () -> someReallyLongProcess();

Runnable task2 = () -> anotherReallyLongProcess();

executor.execute(task1);

executor.execute(task2);
```

More questions:

Can we know if a task is done or not?	No...
Can we cancel the execution of a task?	Yes, if the task has not started yet

Building an Executor is more efficient than creating threads on demand

One can pass instances of Runnable to an Executor

The Executor has a waiting queue to handle multiple requests

A task can be removed from the waiting queue

From Runnable to Callable

```
Runnable task = () -> someReallyLongProcess();

Executor executor = ...;

executor.execute(task);
```

A task does not return anything

- No object can be returned
- No exception can be raised

There is no way we can know if a task is done or not

How can a task return a value?

How can we get the exceptions raised by a task?

How can this value or exception go from one thread to another?

We need a new model for our tasks:

With a method that returns a value

And that can throw an Exception

We also need a new object that acts as a bridge between threads

The Runnable Interface

```
@FunctionalInterface
public interface Runnable {

   void run();
}
```

The Callable Interface

```
@FunctionalInterface
public interface Callable<V> {

   V call() throws Exception;
}
```

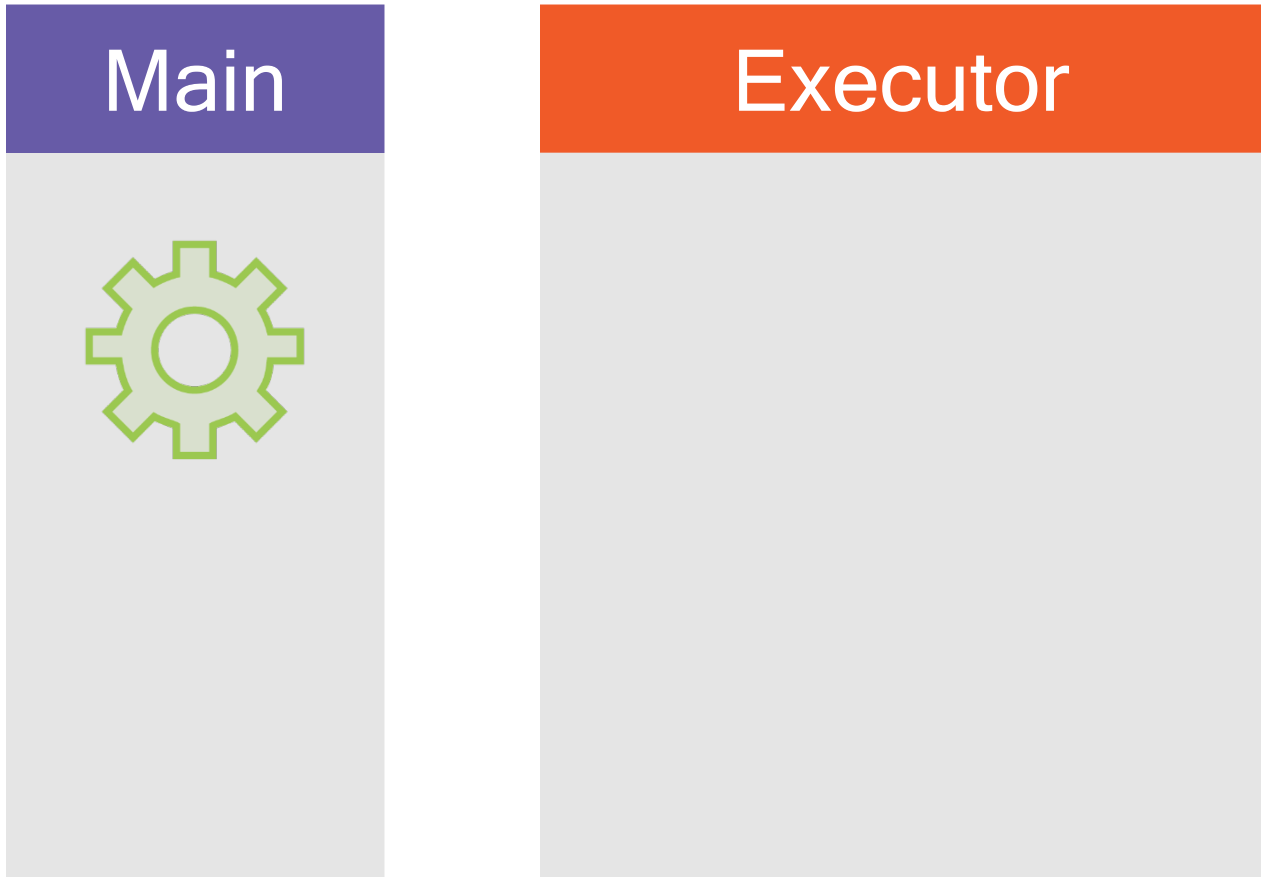
```java
<T> Future<T> submit(Callable<T> task);
```

The Executor interface does not handle callables

The ExecutorService interface has a submit() method

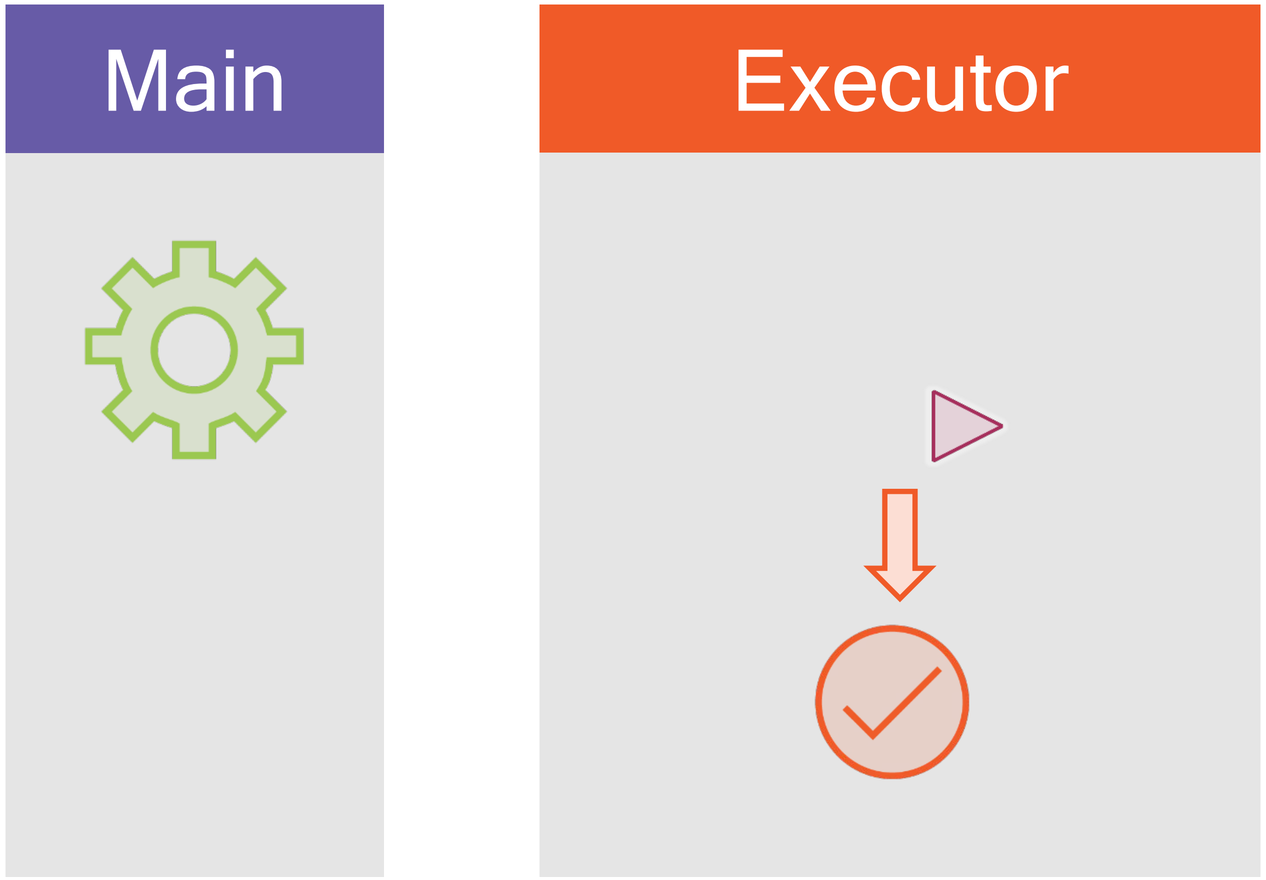
…that returns a Future object

How Does This Future Object Work?

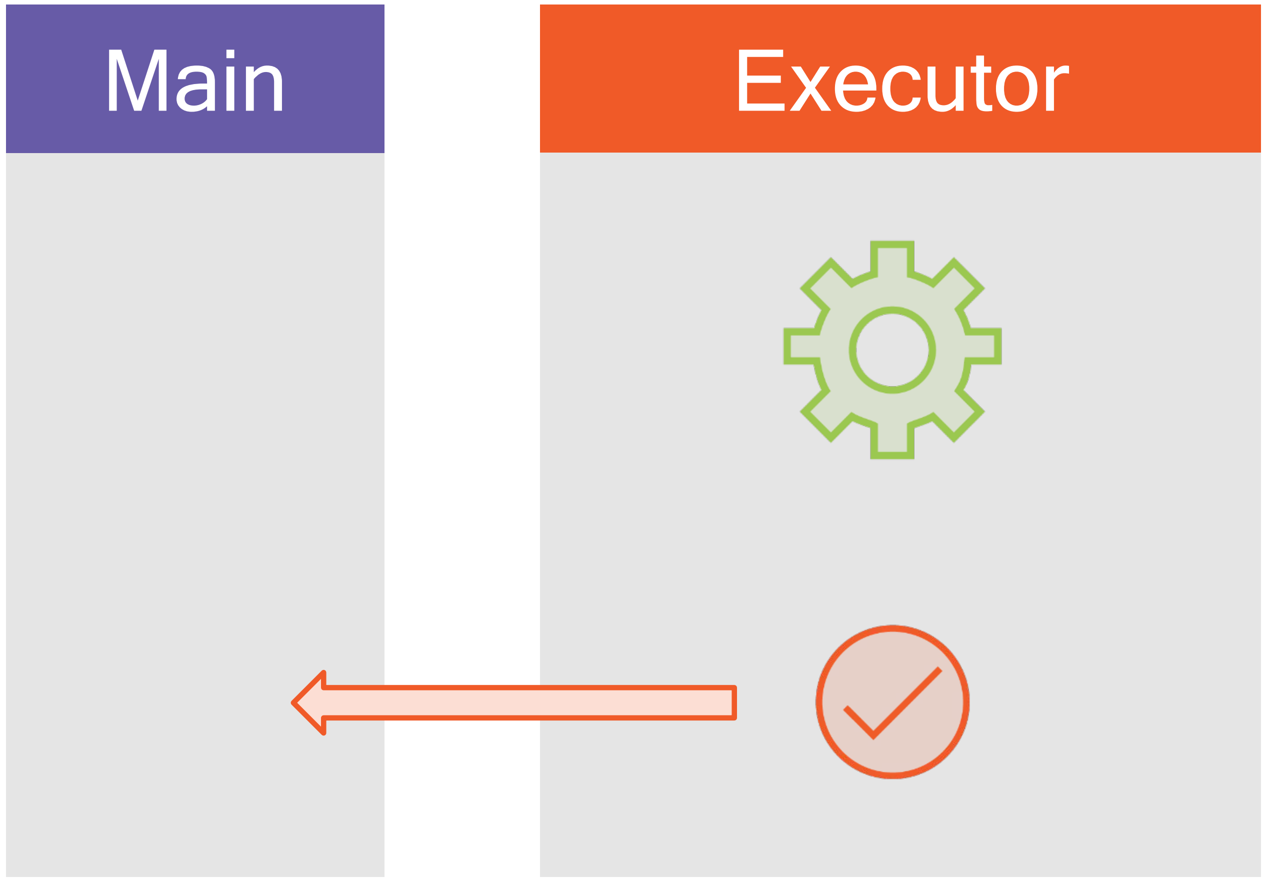


executor.submit()

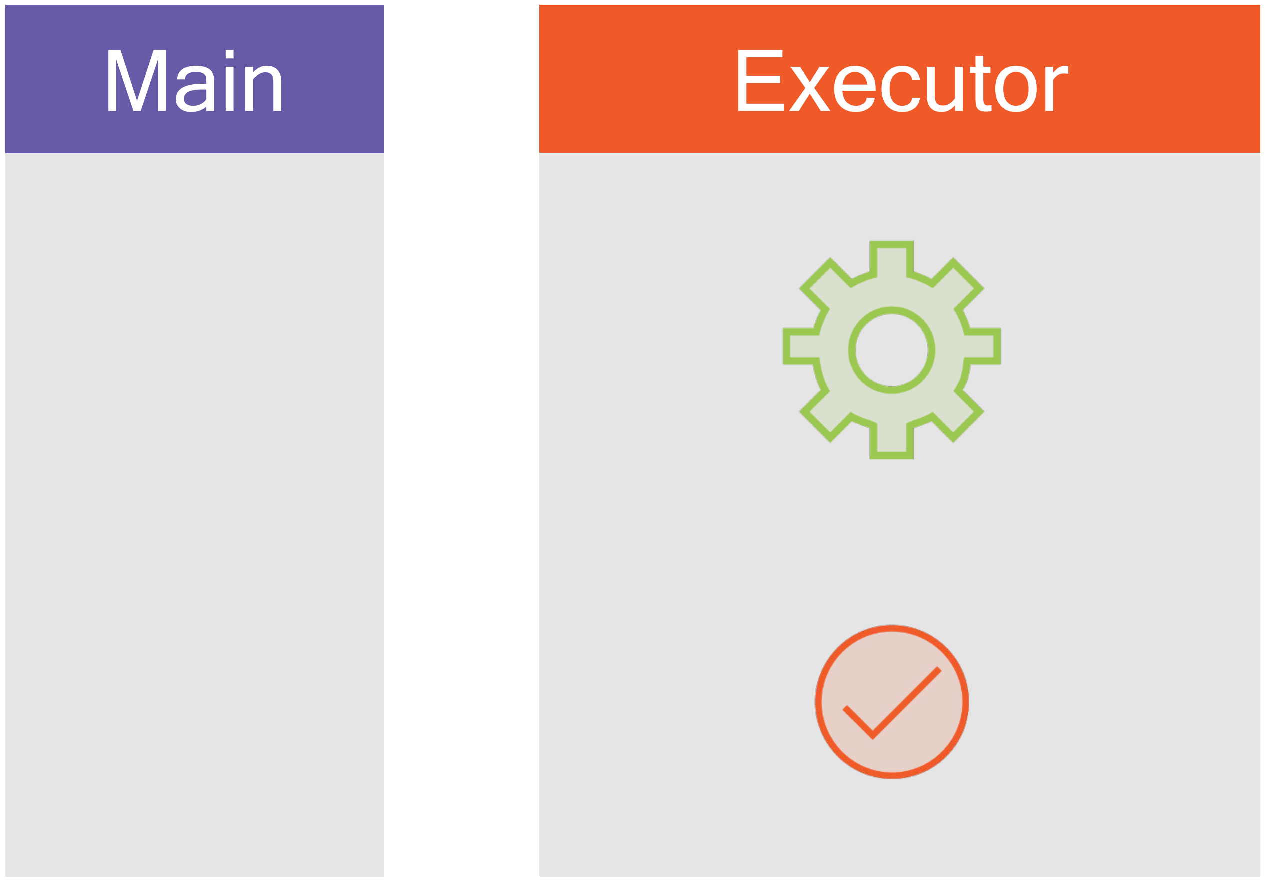
How Does This Future Object Work?



How Does This Future Object Work?



How Does This Future Object Work?



This is the role of the Future object

```java
// In the main thread

Callable<String> task = () -> buildPatientReport();

Future<String> future = executor.submit(task);

String result = future.get();
```

The Future object is returned by the submit() call in the main thread

The get() method of the Future object can be called to return the produced String

The get() call is blocking until the returned String is available

```
// In the main thread
Callable<String> task = () -> buildPatientReport();
Future<String> future = executor.submit(task);
String result = future.get();
```

The Future.get() method can raise two exceptions:

- in case the thread of the executor is interrupted, it throws an InterruptedException
- in case the task throws an exception, it is wrapped in an ExecutionException and re-thrown

Behavior of Future.get()

If the task has completed, then the get() call will return the produced result immediately

If it is not the case, then the get() call blocks until the result is ready

Behavior of Future.get()

If an exception has been thrown, then this exception is also thrown by the get() call, wrapped in an ExecutionException

On can pass a timeout to the get() call, to avoid indefinitely blocking calls

Demo

Let us see some code!

Let us create simple tasks and submit them to an executor

Let us get the results through futures

… and see what happens when things go wrong

Demo Wrapup

What did we see?

How to pass tasks to Executors

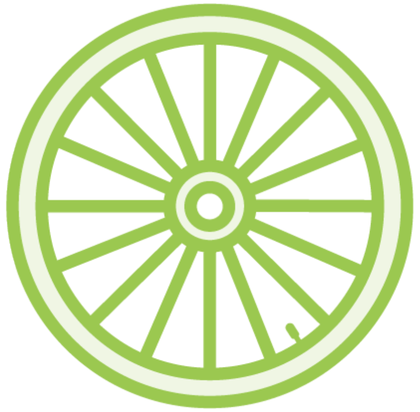
How to properly shutdown an executor

How to pass an object from a callable to the main thread

How to handle timeouts

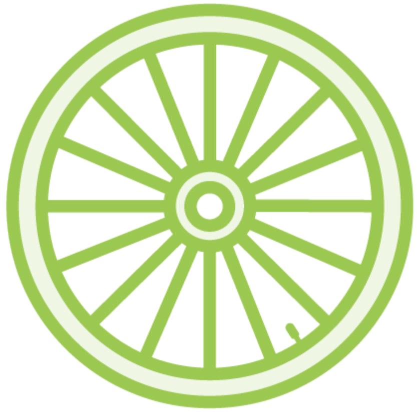
How to handle application exception through futures

Available ExecutorServices



The JDK comes with several ExecutorService implementations

1) newSingleThreadExecutor()
- an executor with only one thread



The JDK comes with several ExecutorService implementations

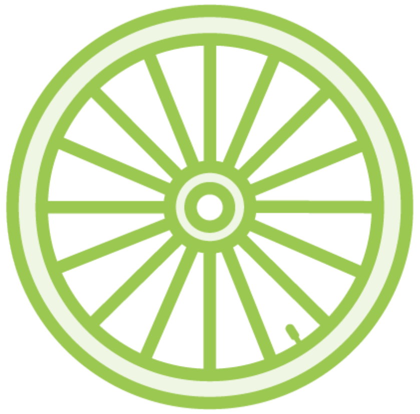
2) newFixedThreadPool(poolSize)
- an executor with poolSize threads

The JDK comes with several ExecutorService implementations

3) newCachedThreadPool()
- creates threads on demand
- keeps unused threads for 60s
- then terminates them

The JDK comes with several ExecutorService implementations

4) newScheduledThreadPool(poolSize)
- creates a pool of threads
- returns a ScheduledExecutorService



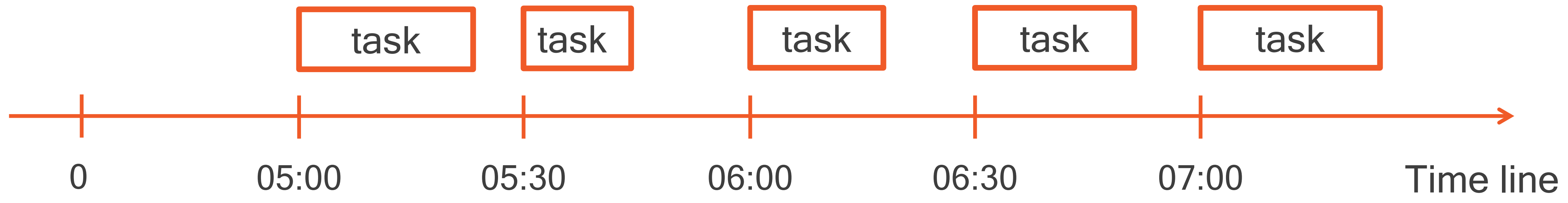
The ScheduledExecutorService:

- schedule(task, delay)

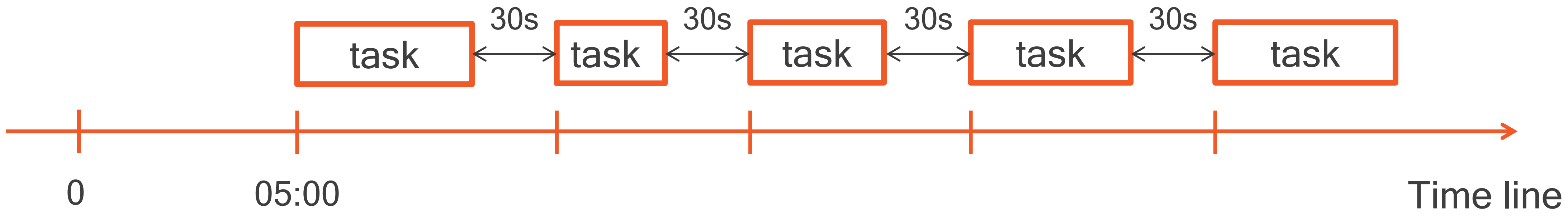
- scheduleAtFixedRate(task, delay, period)

- scheduleWithFixedDelay(
    task, initialDelay, delay)

Schedule at Fixed Rate



Schedule with Fixed Delay



How to Shutdown an ExecutorService

An ExecutorService needs to be properly shutdown

There are 3 methods:

1) shutdown()
- continue to execute all submitted tasks,
- execute waiting tasks,
- do not accept new tasks
- then shutdown

An ExecutorService needs to be properly shutdown

There are 3 methods:

2) shutdownNow()
- halt the running tasks,
- do not execute waiting tasks,
- then shutdown

An ExecutorService needs to be properly shutdown

There are 3 methods:

3) awaitTermination(timeout)
- shutdown()
- wait for the timeout,
- if there are remaining tasks, then halt everything

Module Wrapup

What did we learn?

How to build an executor service

How to create tasks modeled by Callables

How to submit them and get results produced by these tasks

How to properly handle exceptions

How to properly shutdown executors