Setting up Asynchronous Operations with CompletionStage

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

Agenda

- Concept of asynchronous operation
- What is a task?
- What does it mean to launch it asynchronously?
- Patterns

What Is a Task?

A task is something a computer has to execute

It may take an input

It may produce an output

It may have side-effects

It has to be an object!

There is a confusion

- the implementation of the task

- the object that wraps this task

This wrapper carries information on the state of this task

The 1st information is its completion!

Defining and Writing a Task

What are the available models to launch tasks in another thread?

- Runnable

- Callable

```
public interface Runnable {
    void run();
}


Runnable runnable = () -> {};
Runnable loggingAMessage = () -> logger.info("Closing connection");
Runnable sendingAMessage = () -> sendMessage("Data has been read");
```

The Runnable interface models a function that does not take any parameter and does not return anything

```
public interface Callable<V> {
    V call() throws Exception;
}


Callable<User> readUser = () -> connection.readUser(1L);

Callable<String> readPage = () -> readPage("http://somesite.com");
```

The Callable interface models a task that can produce a result and that can fail with an exception

It is always possible to launch a task in the current thread

The Java.util.concurrent API brings patterns to launch them in another thread

Launching Tasks in a Thread

There are two patterns:

- the Runnable pattern

- the Executor pattern

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

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

The Runnable pattern is the first pattern available, since Java 1

It is an obsolete pattern that should not be used anymore!

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

ExecutorService service = Executors.newSingleThreadExecutor();


   service.submit(task);
```

The executor service pattern has been introduced in Java 5

```
Callable<String> task = () -> readPage("http://mysite.com");

ExecutorService service = Executors.newSingleThreadExecutor();


   service.submit(task);
```

The executor service pattern has been introduced in Java 5

```
Callable<String> task = () -> readPage("http://mysite.com");

ExecutorService service = Executors.newSingleThreadExecutor();

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

The executor service pattern has been introduced in Java 5

One can get the result of the reading of the web page through a future object

What can be done with this future object?

- it can be queried for the returned object

- it can be cancelled

And that's it...

From Future to CompletableFuture

A CompletableFuture object is (almost) the same as a Future object

With more methods

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

ExecutorService service = Executors.newSingleThreadExecutor();

Future<?> future = service.submit(task);
```

This is the classical executor pattern

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

ExecutorService service = Executors.newSingleThreadExecutor();

Future<?> future = service.submit(task);

CompletableFuture<Void> completableFuture =
   CompletableFuture.runAsync(task);
```

This is the classical executor pattern

This is the completable future pattern

```
Callable<String> task = () -> readPage("http://mysite.com");

ExecutorService service = Executors.newSingleThreadExecutor();

service.submit(task);
```

The completable future pattern does not work with callables...

```
Supplier<String> task = () -> readPage("http://mysite.com");

ExecutorService service = Executors.newSingleThreadExecutor();

// service.submit(task); // does not compile anymore

CompletableFuture<String> completableFuture =
   CompletableFuture.supplyAsync(task);
```

The completable future pattern does not work with callables...

But it does with Suppliers!

Be careful: a supplier cannot throw any checked exceptions

So far we have two patterns:

- runAsync() that takes a Runnable

- supplyAsync() that takes a Supplier

Running in Another Thread

By default, the async tasks are run in the Common Fork / Join Pool

You can also pass an executor as a parameter

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

ExecutorService service = Executors.newSingleThreadExecutor();

Future<?> future = service.submit(task);

CompletableFuture<Void> completableFuture =
   CompletableFuture.runAsync(task, service);
```

In this case, the task will be executed in this executor

```
Supplier<String> task = () -> readPage("http://mysite.com");

ExecutorService service = Executors.newSingleThreadExecutor();

// service.submit(task); // does not compile anymore

CompletableFuture<String> completableFuture =
   CompletableFuture.supplyAsync(task, service);
```

In this case, the task will be executed in this executor

```
Future<?> future =
   executorService.submit(() -> System.out.println("Hello world!"));

CompletableFuture<?> completableFuture =
   CompletableFuture.runAsync(
                           () -> System.out.println("Hello world!"),
                           executorService);
```

Going from the executor pattern to the completable future pattern is in fact very easy

These two lines of code do the same

```
Future<String> future =
   executorService.submit(() -> readPage("http://mysite.com"));

CompletableFuture<String> completableFuture =
   CompletableFuture.supplyAsync(
                           () -> readPage("http://mysite.com"),
                           executorService);
```

Going from the executor pattern to the completable future pattern is in fact very easy

These two lines of code do the same

A Closer Look at CompletableFuture

CompletableFuture is a class that implements:

- Future

- and CompletionStage

This is the reason why both are compatible

CompletionStage adds methods to chain tasks

CompletableFuture adds more methods

A task has a state

It may be:

- running

- completed normally

- completed exceptionally

```
T get();
T get(long timeOut, TimeUnit unit):

void cancel();

boolean isDone();
boolean isCancelled();
```

There are 5 methods on Future

- two methods to get the result

- one to cancel the execution of the task

- and two to check if the task is done or has been cancelled

```
T join(); // may throw an unchecked exception
T getNow(T valueIfAbsent):

boolean complete(V value);
void obtrudeValue(V value);

boolean completeExceptionally(Throwable t);
void obtrudeException(Throwable t);
```

CompletableFuture brings 5 future-like methods

- two methods to get the result in a different way

- two methods to force the returned value

- and two to force an exception

complete(value):

Checks if the task is done

- if it is done: then it does nothing

- if it is not, then it completes it and sets the returned value to value

obtrudeValue(value):

Checks if the task is done

- if it is done: then forces the returned value to value

- if it is not, then it completes it and sets the returned value to value

This should be used in error recovery operations

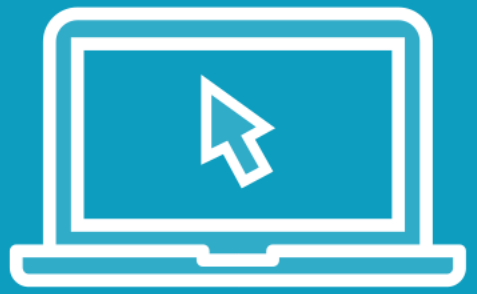
completeExceptionally(throwable):

- forces the completion if the task is not done

obtrudeException(throwable):

- forces the completion even if the task is done

Demo



Let us see some code!

Let us launch tasks asynchronously

And see how we can get the results

Module Wrap Up

What did you learn?

What is a task in this API

The difference between Future and CompletableFuture

How to launch tasks using the CompletableFuture API