

# Java Fundamentals: Asynchronous Programming using CompletionStage

---

INTRODUCING ASYNCHRONOUS VS  
CONCURRENT TASKS



**José Paumard**

PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard <https://github.com/JosePaumard>





## CompletionStage / CompletableFuture

- API introduced in Java 8
- create pipelines of tasks
- in an asynchronous way
- error handling
- performance



This is a Java course

Fair knowledge of the language and its main API

Concurrent Programming

Lambda expressions

The Stream API

This is a fundamental course



Applying Concurrency and Multi-threading  
to Common Java Patterns

From Collections to Streams in Java 8  
Using Lambda Expressions

Streams, Collectors, and Optionals for Data  
Processing in Java 8



# Agenda of the Course



Introducing the big picture: defining the vocabulary, what can you do?

Launching a first task, technical details

Launching tasks on the completion of other tasks

What can you do when things go wrong?

Let us talk about performance



# Agenda



Let us define the technical terms!

Launching a concurrent task

Synchronous vs. asynchronous

Blocking vs. non-blocking

Message driven vs. event driven



# Setting up the Problem

---



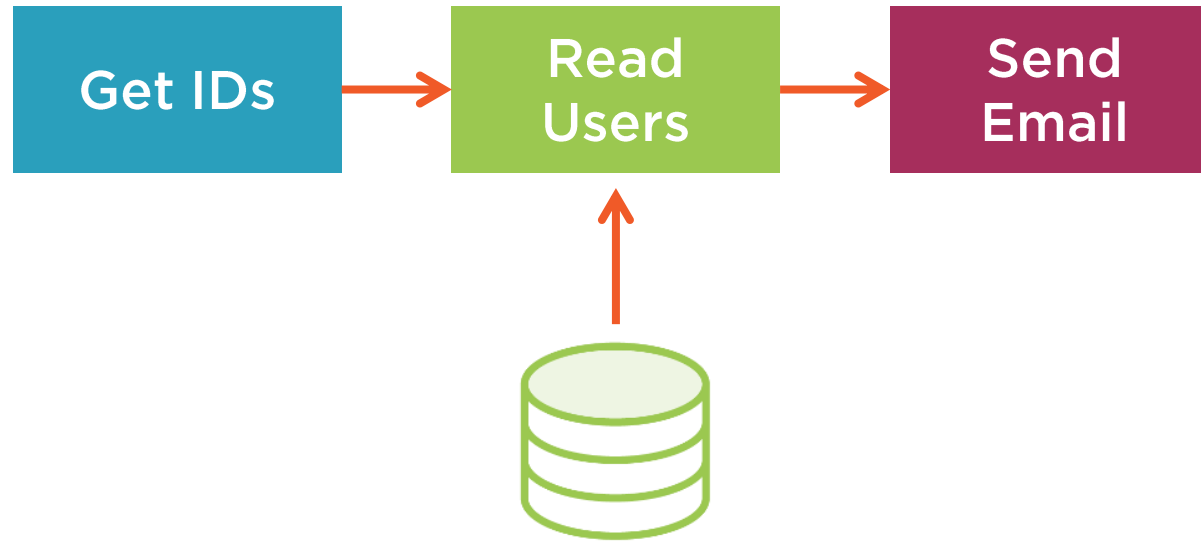


The problem is the following:

- we want to launch a task
- get the result
- use this result to launch another task



# A Classical Use Case





Reading users from a DB takes time  
Without blocking our main thread  
Launch this task in another thread?





Then we need to get the list of users

And launch the sending of emails

What thread will get this result?

And launch the other task?





Will the solution come from concurrent programming?

What does synchronous mean?

What does blocking a thread mean?

What about an event-driven system?

# Concurrency

---





Concurrent means several threads

A task is created in a thread

And can be executed in another thread

The result is passed to the first thread

```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};
```



ExecutorService



```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things
```

future



ExecutorService





```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things  
  
... = future.get(); // block until the task  
                    // is done
```

future



ExecutorService



```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things  
  
... = future.get(); // block until the task  
                    // is done
```



**ExecutorService**



```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();
```

```
Runnable task = () -> {...};
```

```
Future future = service.submit(task);
```

```
// more things
```

```
... = future.cancel();
```



ExecutorService





The executor pattern enables the launching of tasks in other threads

But it does not offer the non-blocking behavior

Since getting the result can block the launching thread

# Synchronous vs. Asynchronous

---



# Synchronous

The thread that launched the task needs to wait for the task to complete to continue to work



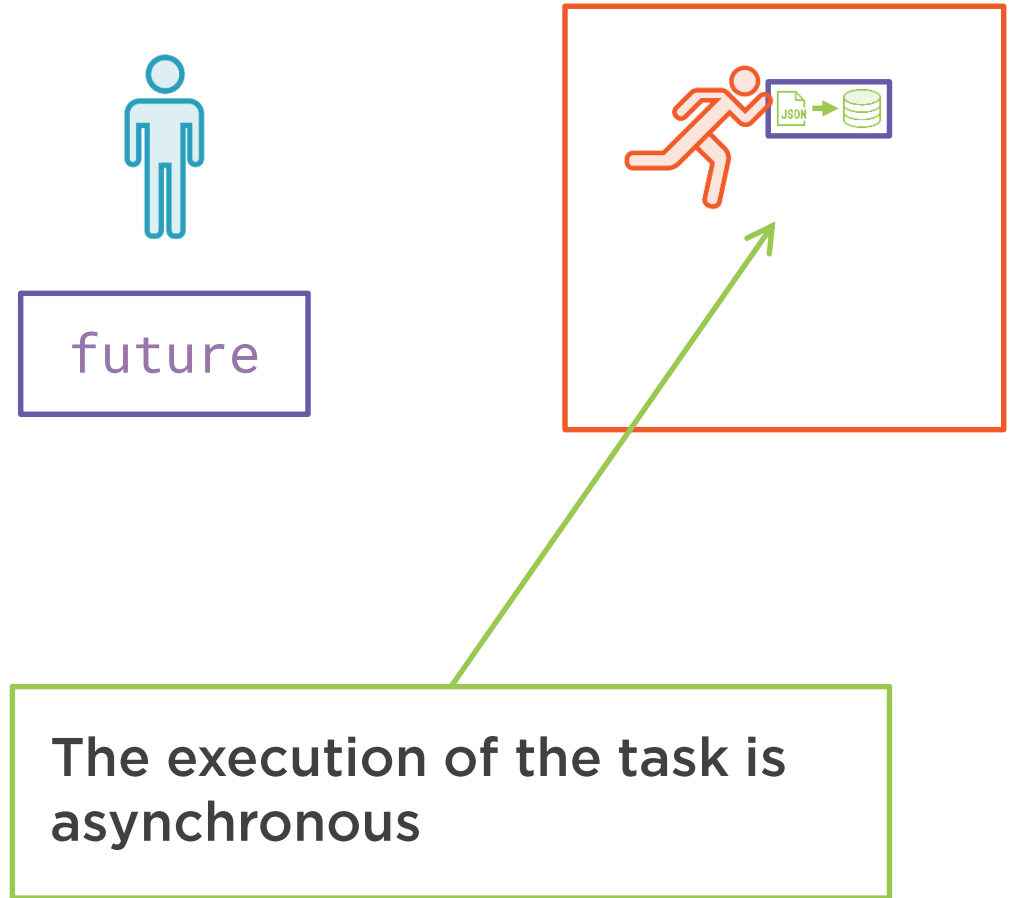
# Synchronous vs. Asynchronous

The thread that launched the task needs to wait for the task to complete to continue to work

The task is executed at some point in the future



```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things  
  
... = future.get(); // block until the task  
                  // is done
```





```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things  
  
... = future.get(); // block until the task  
                  // is done
```



future



But getting the result is  
synchronous!





Does asynchronous mean “in another thread”?



```
List<String> strings = ...;  
strings.sort(Comparator.naturalOrder());
```

When does this comparator is executed?

Some time in the future

In what thread is it executed?

The main thread!





Asynchronous **and** concurrent **are** different notions

A task can be asynchronous **and** in the same thread



# Blocking vs. Non-Blocking

---





Blocking means that a thread has to wait to execute a task or access a resource

Synchronization is a blocking way to prevent race conditions

```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things  
  
... = future.get(); // block until the task  
                  // is done
```



future



The future.get() call is  
blocking





Suppose that, once the JSON is recorded in the database...

... we need to notify the user by a green light on the interface





```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things  
  
... = future.get();  
  
lightIcon.setColor(Color.GREEN);
```



future



This code is synchronous and blocking



```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things  
  
... = future.get();  
  
lightIcon.setColor(Color.GREEN);
```



future



This code is synchronous and blocking  
And should be executed in the right thread!



```
ExecutorService service =  
    Executors.newSingleThreadExecutorService();  
  
Runnable task = () -> {...};  
  
Future future = service.submit(task);  
  
// more things  
  
... = future.get();  
  
SwingUtilities.invokeLater(  
    () -> lightIcon.setColor(Color.GREEN));
```



future



This code is synchronous and blocking  
And should be executed in the right thread!





Is there a non-blocking way of chaining a task (lighting the green light)

On the completion of another task (writing the JSON)?





The answer is no...  
At least with this API





What we need here is the following:

- being able to trigger a task on the completion of another one
- being able to specify in which thread a task is executed

# Message Driven vs. Event Driven

---





One solution could be to fire an event  
And create a listener for this event  
How does it work?



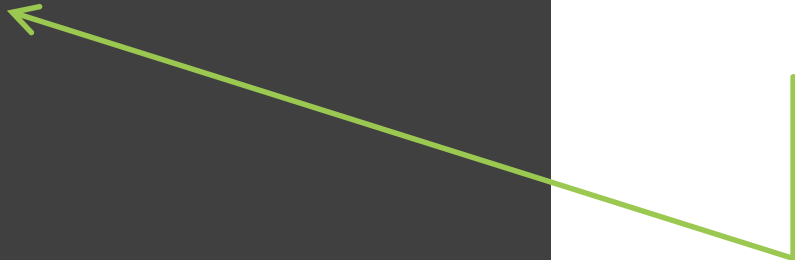


```
@Inject  
Event<Signal> event;
```

```
Signal signal = new Signal(...);  
event.fire(signal);
```



```
for (Listener listener: listeners) {  
    listener.notify(signal);  
}
```



Notifying the listeners is  
synchronous  
It occurs in the same thread  
And it is blocking



# Event

An event is triggered to notify the change of a state

The recipient has to declare itself to the source

It is (usually) synchronous and blocking





Events are not the right tool, what about messages?

A message is sent to a broker

That will trigger the subscribers of this kind of message



# Event vs. Message

An event is a signal

A message is a piece of data that recipients will get and react upon



# Module Wrap Up



What did you learn?

Notions you are going to need

Execution of tasks in a specific thread

Non blocking calls

Asynchronous execution

The problem is now set up, we can call the `CompletionStage` API to solve it

