* Callable introduced as interface in java 5
* Changed as functional interface in java 8
* Call() method returns generic value and may throw exception
* It is designed to encapsulate task executed by another thread like runnable interface .
* Callable instance can be executed through ExecutorService
* When call method returns a value , main thread retrieves it to do logic using Futures Object.

Single Task

* Method that executes only one async task.

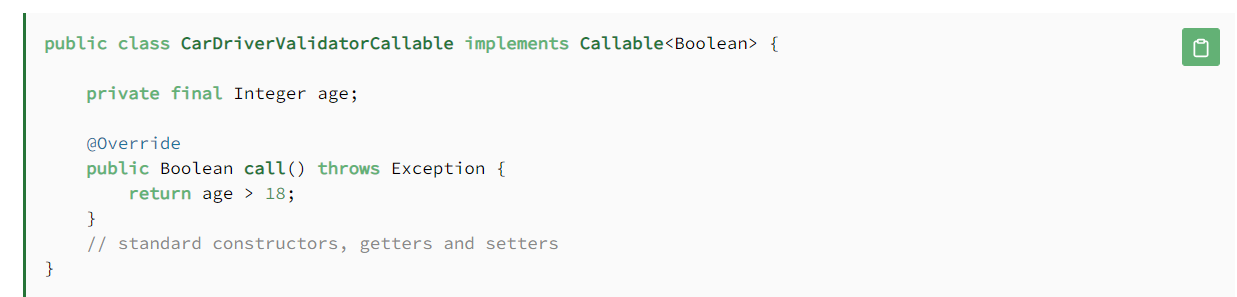


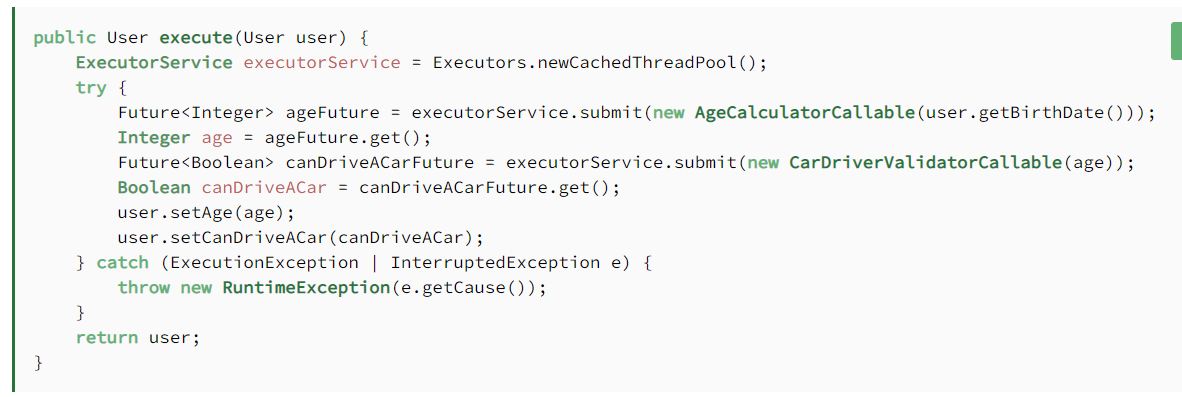
Interuppted Exception : Interuppton occurred when thread is sleeping, active or occupied.

Execution exception : May be wrapper exception .task get aborted duce to some cause .

Chain of Task

* If previous task fails , the current task cant be executed.





Using *Callable* and *Future* in a chain of tasks has some problems:

* Each task in the chain follows the pattern “submit-get”. In a long chain, this produces verbose code.
* When the chain is tolerant to a task failure, we should create a dedicated *try*/*catch* block.
* When invoked, the *get()* method waits until the *Callable* returns a value. So the total execution time of the chain equals the sum of the execution time of all the tasks. But if the next task depends on the correct execution of only one previous task, the chain process is significantly slowed down.

Supplier

*Supplier* is a functional interface whose SAM (Single Abstract Method) is*get()*.

**It doesn't take any argument, returns a value, and throws only unchecked exceptions:**

T **get**();Copy

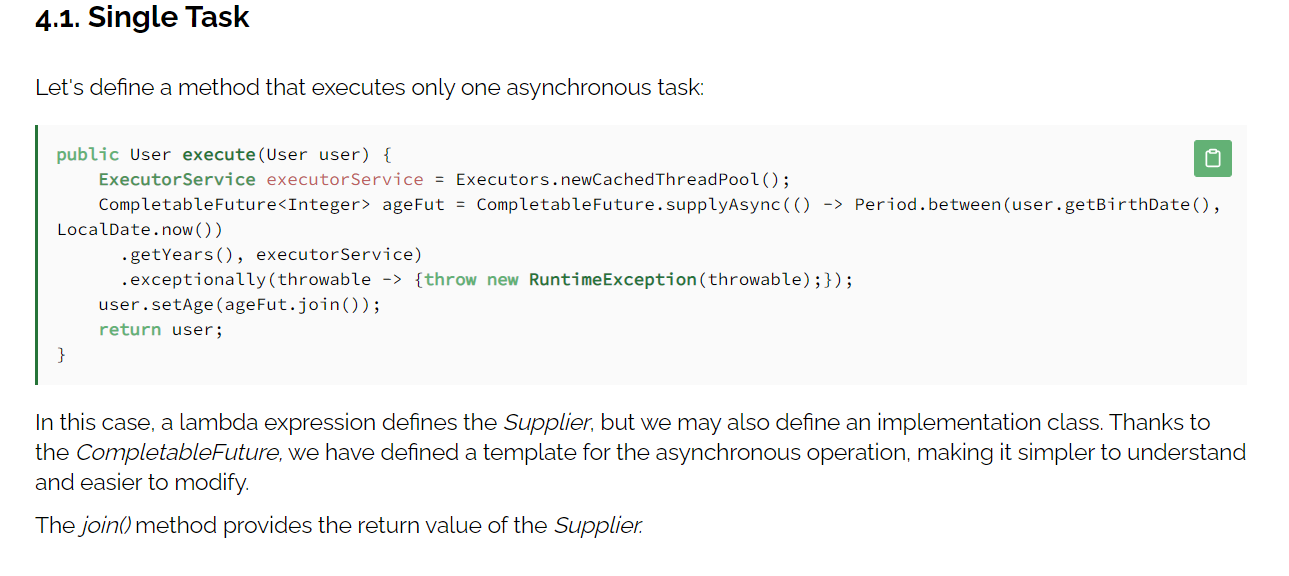
One of the most frequent use cases of this interface is to defer the execution of some code.

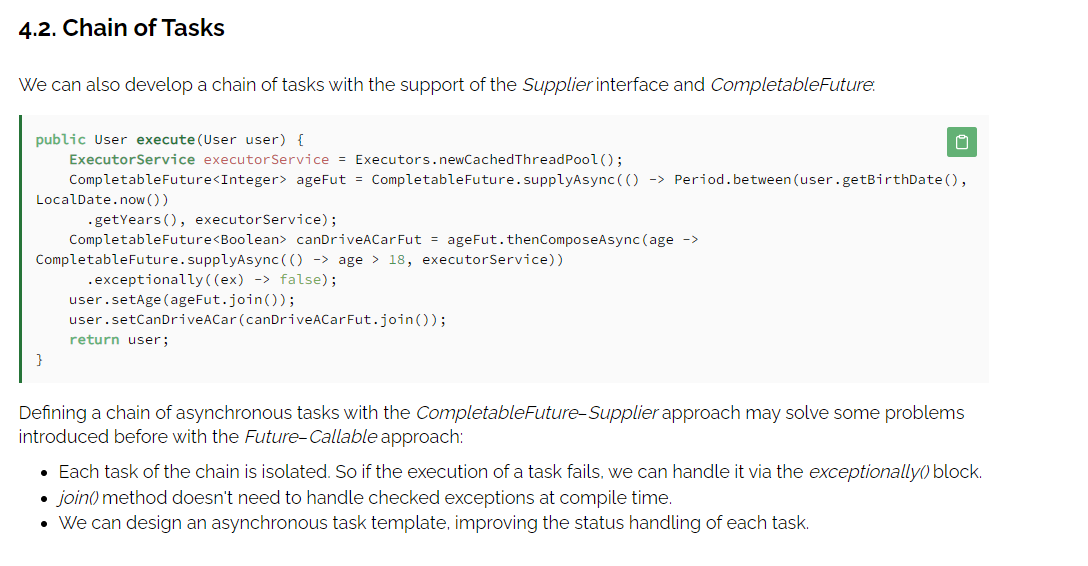
The [*Optional*](https://www.baeldung.com/java-optional) class has a few methods that accept a *Supplier* as a parameter, such as *Optional.or()*, *Optional.orElseGet().*

So the *Supplier* is executed only when the *Optional* is empty.

We can also use it in an asynchronous computation context, specifically in the [CompletableFuture](https://www.baeldung.com/java-completablefuture) API.

Some methods accept a *Supplier* as a parameter, such as the *supplyAsync()*method.





In this article, we discussed the differences between Callable and Supplier interfaces, focusing on the context of asynchronous tasks.

**The main difference at the interface design level is the checked exception thrown by the Callable.**

Callable was not meant for a functional context. It was adapted over time, and functional programming and checked exceptions don't get along.

So any functional API (such as CompletableFuture API) always accepts Supplier rather than Callable.