Asynchronous programming is all about writing non-blocking code by running all the tasks on separate threads instead of the main application thread and keeping the main thread informed about the progress, completion status, or if the task fails.

Asynchronous programming is enabled by the **CompletableFuture** API in Java. It implements the [Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html) and [CompletionStage](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletionStage.html" \t "_blank) interfaces.

* Future has been introduced in java 5
* No manual completion
* Cannot be chained multiple future.
* Cannot combine multiple further together
* No exception handling

Diagram

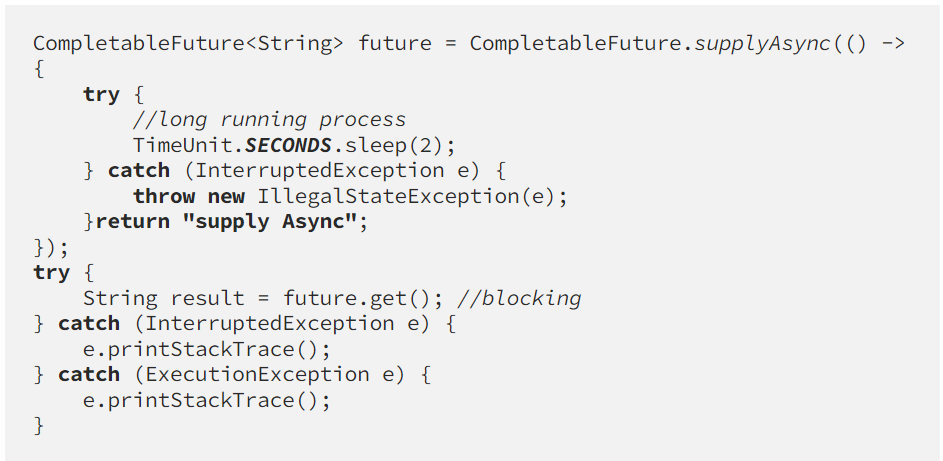
Description automatically generated

Completable Future

* Added in Java 8
* Implements future and completion stage interface
* Provides a huge set of convenience methods for creating, chaining, and combining multiple futures.
* It also has comprehensive exception-handling support.

Methods :

1. runAsync() : it takes runnable object and returns void CompletableFuture.
2. supplyAsync():



The difference between runAsync() and supplyAsync() is that the former returns a Void while supplyAsync() returns a value obtained by the Supplier.

If the custom Executor is not provided, both methods are asynchronously completed by tasks running in the ForkJoinPool (explained above in precursors).

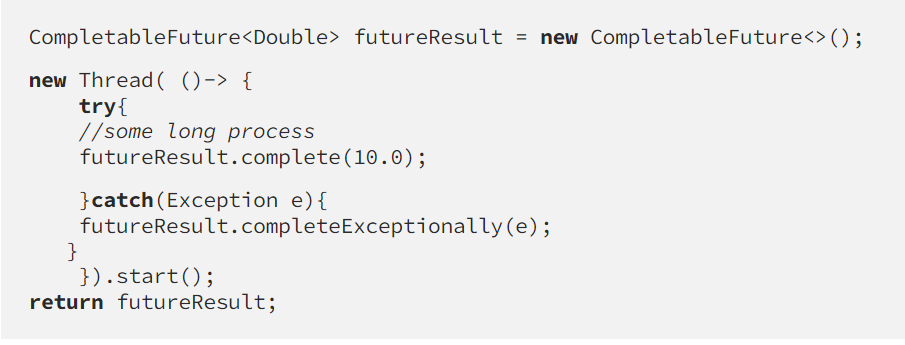
Callback Methods :

1. thenApply()
2. thenAccept()
3. thenReturn()

Completable Future resolves below limitations :

1. Non-Blocking
2. Completing future programmatically
3. Perform error handling
4. Ability to chain several futures
5. Ability to combine results of multiple futures (run in parallel)

Error Handling :



**Advantages of the CompletableFuture API**

1. If a remote API service is down while using it, you can manually complete the future to retrieve the data.
2. The*CompletableFuture* API allows chaining multiple APIs, thereby allowing you to create an asynchronous workflow.
3. It provides an exception handling mechanism.
4. It provides the mechanism to combine multiple futures into a single *CompletableFuture*.
5. It allows a callback function to the API which gets called when the response is available.

**CompletableFuture has over 50 different methods for composing, combining, and executing asynchronous computation steps and handling errors.**

# CompletableFuture for running Asynchronous Tasks

There are mainly two static methods for running asynchronous tasks.

## runAsync()

**If you want to run some background task asynchronously and do not want to return anything from that task, then use the CompletableFuture.runAsync()**.

Since this static method takes a [Runnable](https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html) object and doesn’t return a value, it returns CompletableFuture<Void>. The overloaded version also accepts Executor as the second argument.

1. CompletableFuture.runAsync(Runnable)
2. CompletableFuture.runAsync(Runnable, Executor)

## supplyAsync()

**If you want to** **run some background task asynchronously and want to return anything from that task, then use CompletableFuture.supplyAsync().**

It takes a [Supplier<T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Supplier.html) and returns a CompletableFuture<T> where T is the type of the value obtained by calling the given supplier. It also has the version taking Executor as the second parameter.

1. CompletableFuture.supplyAsync(Supplier<T>)
2. CompletableFuture.supplyAsync(Supplier<T>, Executor)

## thenApply()

**If you want to process and transform the result of a CompletableFuture, then use thenApply().**

It takes a [Function<T,R>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html) as an argument. Function<T,R> is a simple functional interface representing a function that accepts an argument of type T and produces a result of type R.

## thenAccept()

**If you don’t want to return anything from the callback function and just want to execute some code after the completion of the Future, then use** **thenAccept()**.

**CompletableFuture.thenAccept()** accepts a [Consumer<T>](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) and returns a CompletableFuture<Void>. It has access to the result of the future, to which it is attached.

## thenRun()

Similar to **thenAccept(), if you don’t want to return anything from the callback function and just want to execute some code after the completion of the Future, then use** **thenRun()**.

However, **thenAccept()** has access to the result of the CompletableFuture to which it is attached, **thenRun()**, on the other hand, doesn’t even have access to the result of the Future. It takes a Runnable as an argument and returns a CompletableFuture<Void>.

# Combining the results of CompletableFutures

One of the best things that happened with the release of Java 8 is the introduction of functional programming. It follows the monad design pattern.

*“Monad is a software design pattern with a structure that combines program fragments (functions) and wraps their return values in a type with additional computation.” — Wikipedia*

The CompletableFuture API enables you to combine CompletableFuture instances in a chain of computation steps. The result of this chaining is a CompletableFuture that can be chained and combined further.

## thenCompose()

**If you want to fetch data from a remote API service and, from using that data, you want to fetch some other data from another API, you should use thenCompose().**

## **thenCombine()**

**If you want to combine two Futures which run independently and then act on the results when both are completed, where the result of one Future is dependent on the other, then you should use thenCombine().**

## exceptionally()

**If you want to log and return a default value for the exception that occurred in the Future, then use exceptionally()**.