CompletableFuture: Introduction

This lesson introduces the newly added CompletableFuture interface.

We'll cover the following

- Introduction to CompletableFuture interface
- Limitations of Future interface
- Creating a CompletableFuture.
- Asynchronous computation using runAsync()
- Asynchronous computation using supplyAsync()

Introduction to `CompletableFuture` interface

A `CompletableFuture` is a class in Java that belongs to the `java.util.concurrent` package.

It is used for asynchronous computation. The code is executed as a non-blocking call in a separate thread, and the result is made available when it is ready.

By doing this, the main thread does not block/wait for the completion of the task, and it can execute other tasks in parallel.

The `CompletableFuture` class implements the `CompletionStage` and `Future` interface. The `CompletionStage` is a promise. It promises that the computation eventually will be done.

Before Java 8, `Future` interface, which was added in Java 1.5, was available for asynchronous computation. The limitation of `Future` interface is that it does not have any methods to combine these computations or handle errors. We will address more limitations of Future interface in the next section.

`CompletableFuture` has lots of different methods for composing, combining, executing asynchronous computation steps, and handling errors.

Limitations of `Future` interface

The `Future` interface provides an `isDone()` method to check if computation is done, the `get()` method to get the result of computation, and the `cancel()` method to cancel the computation.

However, there are some limitations of the `Future` interface, which we will discuss here:

1. We cannot perform further action on a `Future`'s result without blocking. We have a `get()` method, which blocks until the computation is complete.

2. Future chaining is not possible. If you want to execute one `Future` and then trigger another future once the first one is complete, this is not possible.

3. We cannot combine multiple `Future` together. If we want to run five different futures in parallel and then combine their result then this is not possible.

4. `Future` does not have any exception handling mechanism.

Looking at all these limitations, Java 8 introduced the `CompletableFuture`.

Creating a `CompletableFuture`.

We can easily create a CompletableFuture using the no-arg constructor and provide it to some `Thread`. The problem is that if that `Thread` calls the `get()` method on our `CompletableFuture` object, it blocks until the computation is complete. We can complete the `CompletableFuture` using the `complete()` method.

Here is an example. In the below example, we have a method that returns a `CompletableFuture` of the square of a number.

```
public Future<String> getSquareAsynchronously(int num) throws InterruptedException {
    CompletableFuture<Integer> completableFuture
            = new CompletableFuture<>();

    Executors.newCachedThreadPool().submit(() -> {
        Thread.sleep(500);
        // The complete() call will complete this CompetableFuture.
        completableFuture.complete(num * num);
        return null;
    });

    return completableFuture;
}
```

If we are sure about the result of computation, we can use the static `completedFuture()` method with an argument that represents a result of this computation.

The `get()` method of the `Future` will never block.

```
import java.util.concurrent.CompletableFuture;

public class CompletableFutureDemo {

    public static void main(String args[]) {
        CompletableFuture<String> completableFuture = CompletableFuture.completedFuture("Hello Wor
        try {
            System.out.println(completableFuture.get());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Asynchronous computation using runAsync()

The `runAsync()` is a static method that runs some background tasks asynchronously and returns a `CompletableFuture<Void>`. This method takes a `Runnable` as a parameter.

This method is particularly useful if we just need to run some code in parallel but do not want any result in return.

In the below example, we will run running a task using `runAsync()`. This will start running the code in a parallel thread.

Then, we print a statement, that will print immediately.

After that, we will call the `get()` method on our future object. This will block the main thread.

Once our parallel thread completes its execution, the main thread will continue.

```
import java.util.concurrent.*;

public class CompletableFutureDemo {
    public static void main(String args[]) {
        // Passing a runnable to runAsync() method.

        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            System.out.println("Doing some processing " + Thread.currentThread().getName());
        });

        System.out.println("This will print immediately " + Thread.currentThread().getName());

        try {
            future.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("This will print after 5 seconds " + Thread.currentThread().getName());

    }
}
```

In the previous example, we are providing only the runnable object to the `runAsync()` method.

By default, asynchronous execution uses `ForkJoinPool.commonPool()`, which uses daemon threads to execute the `Runnable` task.

However, if we want, we can provide our own `Executor` to the `runAsync()` method as well. Here is the code for it.

```
import java.util.concurrent.*;

public class CompletableFutureDemo {

    public static void main(String args[]) {

        Executor executor = Executors.newFixedThreadPool(5);

        // Passing a runnable and executor as parameter to runAsync() method.
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            System.out.println("Doing some processing");
        }, executor);
        }, executor);

        System.out.println("This will print immediately");

        try {
            future.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("This will print after 5 seconds");

    }
}
```

Asynchronous computation using `supplyAsync()`

If we need to get the result of the computation, we should use `supplyAsync()`. It takes a `Supplier<T>` as input and returns `CompletableFuture<T>` where `T` is the type of the value obtained by calling the given supplier

```
import java.util.concurrent.*;

public class CompletableFutureDemo {

    public static void main(String args[]) {

        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            return "Hello World";
        });

        System.out.println("This will print immediately");

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("This will print after 5 seconds");

    }
}
```

There is an overloaded version of `supplyAsync()` method as well. It takes a `Supplie<T>` and an executor as input.

Below is an example.

```
import java.util.concurrent.*;

public class CompletableFutureDemo {

    public static void main(String args[]) {

        Executor executor = Executors.newFixedThreadPool(5);
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            return "Hello World";
        }, executor);

        System.out.println("This will print immediately");

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("This will print after 5 seconds");

    }
}
```

Here is a short quiz to recap what you learned in this lesson!

1 Which of the following methods should be used if we need to get the result of the computation?

2 Which of the following parameters can be passed to the supplyAsync() method? Select all that apply.

Retake Quiz

In the next lesson, we will look at some more features of CompletableFuture.