In the previous lesson, we looked at `CompletableFuture`. We discussed how to create a `CompletableFuture` object and how to run tasks asynchronously.

In this lesson, we will look at how to process the result of a `CompletableFuture`.

Processing the result of CompletableFuture

Suppose we have a `CompletableFuture` and we need to process the result of its execution. Now, the `get()` method of `CompletableFuture` is blocking. This means we need to wait until we get the result of the first task. After getting the result, we can modify the result.

For our system to be truly asynchronous we should be able to attach a callback to the `CompletableFuture`, which should be automatically executed when the `Future` completes. That way, we won't need to wait for the result, and we can write the logic that needs to be executed after the completion of the `Future` inside our callback function.

There are a few ways in which we can do this. We will look at each of them one by one.

1) `thenApply()`

The `thenApply()` method accepts a `Function<T, R>` instance as parameter. As we have discussed earlier, the `Function<T, R>` interface takes in a parameter of type `T` and returns a result of type `R`.

The `thenApply()` method uses the `Function<T, R>` instance to process the result and returns a `Future` that holds a value returned by the function, i.e., `CompletableFuture<R>`

In the below example, we have a `CompletableFuture` that returns an `Integer`. Then, we call `thenApply()` method to double the result of `CompletableFuture` and return the final result.

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

public class CompletableFutureDemo {

    public static void main(String args[]) {

        // Create a future which returns an integer.
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            return 50;
        });

        // Calling thenApply() which takes a Function as parameter.
        // It takes a number as input and returns double of number.
        CompletableFuture<Integer> resultFuture = future.thenApply(num -> {
            System.out.println(Thread.currentThread().getName());
            return num * 2;
        });

        try {
            System.out.println(resultFuture.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Run Save Reset

2) `thenApplyAsync(Function<T, R> function)`

If you look at the output of the above example closely, you will observe that the same thread executes the code in `supplyAsync()` and `thenApply()`. Moreover, if `supplyAsync()` completes very fast then `thenApply()` executes in the main thread.

To achieve actual asynchronous behavior, all the operations should be executed by a different thread. We can achieve this by using the `thenApplyAsync()` method.

This method executes, the code in a common thread created by ForkJoinPool.

Below is an example of this.

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

public class CompletableFutureDemo {

    public static void main(String args[]) {

        // Create a future which returns an integer.
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            return 50;
        });

        // Calling thenApply() which takes a Function as parameter.
        // It takes a number as input and returns double of number.
        CompletableFuture<Integer> resultFuture = future.thenApplyAsync(num -> {
            System.out.println(Thread.currentThread().getName());
            return num *2;
        });

        try {
            System.out.println(resultFuture.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Run Save Reset

3) `thenApplyAsync(Function<T, R> function, Executor executor)`

There is one overloaded version of `thenApplyAsync()` as well. It takes a `Function<T,R>` and an executor as input. By using this method, we get full control over our asynchronous processing flow.

Below is the example for the same.

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

public class CompletableFutureDemo {

    public static void main(String args[]) {

        ExecutorService executor = Executors.newFixedThreadPool(5);

        // Create a future which returns an integer.
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            return 50;
        });

        // Calling thenApply() which takes a Function as parameter.
        // It takes a number as input and returns double of number.
        CompletableFuture<Integer> resultFuture = future.thenApplyAsync(num -> {
            System.out.println(Thread.currentThread().getName());
            return num *2;
        }, executor);

        try {
            System.out.println(resultFuture.get());
```

Run Save Reset

4) `thenAccept()`

The `thenAccept()` method is used if we don't want to return anything from our callback function.

This method takes a `Consumer<T>` as a parameter and returns a `CompletableFuture<Void>`.

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

public class CompletableFutureDemo {

    public static void main(String args[]) {

        // Create a future which returns an integer.
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            return 50;
        });

        // Calling thenApply() which takes a Function as parameter.
        // It takes a number as input and returns double of number.
        future.thenAccept(num -> {
            System.out.println(Thread.currentThread().getName());
            System.out.println("The value is "+  num);
        });

    }
}
```

Run Save Reset

5) `thenRun()`

The `thenRun()` method is also used if we don't want to return anything from our callback function.

This method takes a `Runnable` as a parameter and returns a CompletableFuture.

The difference between `thenAccept()` and `thenRun()` is that the `thenAccept()` method has access to the result of the `CompletableFuture` on which it is attached. Whereas `thenRun()` doesn't even have access to the `Future`'s result.

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

public class CompletableFutureDemo {

    public static void main(String args[]) {

        // Create a future which returns an integer.
        CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(1);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) {
                throw new IllegalStateException(e);
            }
            return 50;
        });

        // Calling thenApply() which takes a Function as parameter.
        // It takes a number as input and returns double of number.
        future.thenRun(() -> {
            System.out.println(Thread.currentThread().getName());
            System.out.println("Hello");
        });
    }
}
```

Run Save Reset