Until now, we have looked at how to create a `CompletableFuture` and how to add callbacks.

One more interesting thing that we can do is combine CompletableFuture instances in a chain of computation steps. Suppose we want to get some data from a service and save it to the database. We can write two CompletableFuture instances and chain them together. The first instance will complete and share its result to the second instance.

There are two methods which help us achieve this. The first one is `thenCompose()`, and the second one is `thenCombine()`. We will look at each one of them below.

## 1) `thenCompose()`

The `thenCompose()` method takes a `Function<? super T,? extends CompletionStage<U>>` as input, i.e., it takes a function that has the result of previous computation as input and returns a `CompletableFuture` as the output.

Below is an example of `thenCompose()`. This is the same example that we used for `thenApply()`. Now, you might be wondering what the difference between `thenCompose()` and `thenApply()` is. Why do we need to use `thenCompose()` when we can use `thenApply()`?

The problem in using `thenApply()` is that, if we use `theApply()` in the below example, the result will be nested in CompletableFuture, i.e., `CompletableFuture<CompletableFuture<Integer>>`.

We should always use `thenCompose()` if you need a flat 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 thenCompose() which takes a Function as parameter.
        // It takes a number as input and returns a CompletableFuture of double of number.
        CompletableFuture<Integer> resultFuture = future.thenCompose(num ->
        CompletableFuture.supplyAsync(() -> num * 2));

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

Run Save Reset

## 2) `thenCombine()`

In the previous example, we used `thenCompose()` which takes the input of the previous input as a parameter. However, if we want to execute two independent Futures and do something with their results, we can use the `thenCombine()` method.

It accepts a `Future` and a `BiFunction` to process both results.

We will look at the same example, that we looked for `thenCompose()` but this time, we will use `thenCombine()`.

The callback function passed to `thenCombine()` will be called when both the futures are executed.

```
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 thenCompose() which takes a Function as parameter.
        // It takes a number as input and returns a CompletableFuture of double of number.
        CompletableFuture<Integer> resultFuture = future.thenCombine(
                CompletableFuture.supplyAsync(() -> 20) , (num1, num2) -> num1 + num2);

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

Run Save Reset