Optional in Java

Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value. This class has various utility methods to facilitate code to handle values as 'available' or 'not available' instead of checking null values. It is introduced in Java 8 and is similar to what Optional is in Guava.

static <T> Optional<T> empty()

Returns an empty Optional instance.

static <T> Optional<T> of(T value)

Returns an Optional with the specified present non-null value.If we enter null value it will throw error.

static <T> Optional<T> ofNullable(T value)

Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional if entered value is null.

T get()

If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.

void ifPresent(Consumer<? super T> consumer)

If a value is present, it invokes the specified consumer with the value, otherwise does nothing.

boolean isPresent()
Returns true if there is a value present, otherwise false.

```
Optional<String>
emailOptional=Optional.ofNullable(customer.getEmail());
if(emailOptional.isPresent()){
        System.out. println(emailOptional.get());
}
```

T orElse(T other)
Returns the value if present, otherwise returns other custom.

```
System.out. println(emailOptional.orElseThrow(()->new
IllegalArgumentException("email not present"));

System.out.println(emailOptional.map(String::toUpperCase)
.orElseGet(()->" default email....."));
```

findAny() & findFirst() return Optional.

Instead of using primitive data type in get method we can use Optional, which is more secure,

Java -8 Completable future

What is a complete future?

CompletableFuture is used for asynchronous programming in java. Asynchronous programming means you can write non blocking code where concurrently you can run N no of tasks in a separate thread without blocking the main thread.

In this way we can improve the performance of the program because multiple tasks are splitted and executed in separate threads rather than depending on the main thread.

When task completes, it notify to main thread(weather task are completed or failed)

There are different ways to implement asynchronous programming in java below mechanism for example:-

Futures
ExecutorService
Callback Interfaces
Thread Pools
Etc.....

Then why do we need a complete future?
CompletableFuture have advantage over Futures

It cannot be manually completed.
Multiple futures cannot be chained together.
We cannot combine multiple futures together.
No proper exception handling mechanism.

```
CompletableFuture<String> future=new
CompletableFuture<>();
//It will wait for result;
future.get();

//return a dummy result while the task is not completed.
future.complete("waiting..");
```

runAsync() vs supplyAsync()

If we run some background task asynchronously and do not want to return anything from that task, then use completableFuture.runAsync() method. It takes a runnable object and return CompletableFuture<Void>

1.completableFuture.runAsync(Runnable) ;
2.completableFuture.runAsync(Runnable, Executor) ;

If we run some background task asynchronously and want to return anything from that task, then use completableFuture.supplyAsync() method. It takes a supplier<T> and returns CompletableFuture<T> Where T is the type of value obtained by calling the given supplier.

1.completableFuture.supplyAsync(supplier<T>) ;
2.completableFuture.supplyAsync(supplier<T>, Executor) ;

The `CompletableFuture.get()` method is blocking. It waits until the Future is completed and returns the result after its completion.

But, that's not what we want right? For building asynchronous systems we should be able to attach a callback to the CompletableFuture which should automatically get called 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.

You can attach a callback to the CompletableFuture using `thenApply()`, `thenAccept()` and `thenRun()` methods -

1. thenApply()

You can use `thenApply()` method to process and transform the result of a CompletableFuture when it arrives. It takes a `Function<T,R>` 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.

2. thenAccept() and thenRun()

If you don't want to return anything from your callback function and just want to run some piece of code after the completion of the Future, then you can use `thenAccept()` and `thenRun()` methods. These methods are consumers and are often used as the last callback in the callback chain.

`CompletableFuture.thenAccept()` takes a `Consumer<T>` and returns `CompletableFuture<Void>`. It has access to the result of the CompletableFuture on which it is attached.

```
// thenAccept() example
CompletableFuture.supplyAsync(() -> {
        return ProductService.getProductDetail(productId);
}).thenAccept(product -> {
        System.out.println("Got product detail from remote service " +
product.getName())

});
```

While `thenAccept()` has access to the result of the CompletableFuture on which it is attached, `thenRun()` doesn't even have access to the Future's result. It takes a `Runnable` and returns `CompletableFuture<Void>` -

```
// thenRun() example
CompletableFuture.supplyAsync(() -> {
    // Run some computation
}).thenRun(() -> {
    // Computation Finished.

});
```

A note about async callback methods -

All the callback methods provided by CompletableFuture have two async variants -

```
// thenApply() variants
<U> CompletableFuture<U>
thenApply(Function<? super T,? extends U> fn)

<U> CompletableFuture<U>
thenApplyAsync(Function<? super T,? extends   U> fn)

<U> CompletableFuture<U>
thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)
```

These async callback variations help you further parallelize your computations by executing the callback tasks in a separate thread.

Combining two CompletableFutures together

1. Combine two dependent futures using thenCompose() -

If you want the final result to be a top-level Future, use `thenCompose()` method instead -

```
CompletableFuture<Double> result = getUserDetail(userId)
.thenCompose(user -> getCreditRating(user));
```

So, Rule of thumb here - If your callback function returns a CompletableFuture, and you want a flattened result from the CompletableFuture chain (which in most cases you would), then use thenCompose().

2. Combine two independent futures using thenCombine() -

While `thenCompose()` is used to combine two Futures where one future is dependent on the other, `thenCombine()` is used when you want two Futures to run independently and do something after both are complete.

```
System.out.println("Retrieving weight.");

CompletableFuture<Double> weightInKgFuture =
CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return 65.0;
});

System.out.println("Retrieving height.");

CompletableFuture<Double> heightInCmFuture =
CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return 177.8;
});

System.out.println("Calculating BMI.");

CompletableFuture<Double> combinedFuture = weightInKgFuture
        .thenCombine(heightInCmFuture, (weightInKg, heightInCm) -> {
    Double heightInMeter = heightInCm/100;
    return weightInKg/(heightInMeter*heightInMeter);
});


System.out.println("Your BMI is - " + combinedFuture.get());
```

The callback function passed to thenCombine() will be called when both the Futures are complete.

Combining multiple CompletableFutures together

We used thenCompose() and thenCombine() to combine two CompletableFutures together. Now, what if you want to combine an arbitrary number of CompletableFutures? Well, you can use the following methods to combine any number of CompletableFutures -

```
static CompletableFuture<Void>      allOf(CompletableFuture<?>... cfs)

static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)
```

1.CompletableFuture.allOf()

CompletableFuture.allOf() is used in scenarios when you have a List of independent futures that you want to run in parallel and do something after all of them are complete.

2. CompletableFuture.anyOf()

CompletableFuture.anyOf() as the name suggests, returns a new CompletableFuture which is completed when any of the given CompletableFutures complete, with the same result.

CompletableFuture Exception Handling

We explored How to create CompletableFuture, transform them, and combine multiple CompletableFutures. Now let's understand what to do when anything goes wrong.

Let's first understand how errors are propagated in a callback chain. Consider the following CompletableFuture callback chain -

```
CompletableFuture.supplyAsync(() -> {
        // Code which might throw an exception
        return "Some result";
}).thenApply(result -> {
        return "processed result";
}).thenApply(result -> {
        return "result after further processing";
}).thenAccept(result -> {
        // do something with the final result
});
```

If an error occurs in the original `supplyAsync()` task, then none of the `thenApply()` callbacks will be called and future will be resolved with the exception occurred. If an error occurs in first `thenApply()` callback then 2nd and 3rd callbacks won't be called and the future will be resolved with the exception occurred, and so on.

1. Handle exceptions using exceptionally() callback

The `exceptionally()` callback gives you a chance to recover from errors generated from the original Future. You can log the exception here and return a default value.

```
Integer age = -1;

CompletableFuture<String> maturityFuture =
CompletableFuture.supplyAsync(() -> {

    if(age < 0) {

        throw new IllegalArgumentException("Age can not be negative");

    }

    if(age > 18) {

        return "Adult";

    } else {

        return "Child";

    }

}).exceptionally(ex -> {

    System.out.println("Oops! We have an exception - " + ex.getMessage());

    return "Unknown!";

});


System.out.println("Maturity : " + maturityFuture.get());
```

Note that the error will not be propagated further in the callback chain if you handle it once.

2. Handle exceptions using the generic handle() method

The API also provides a more generic method - `handle()` to recover from exceptions. It is called whether or not an exception occurs.

```
Integer age = -1;

CompletableFuture<String> maturityFuture =
CompletableFuture.supplyAsync(() -> {
    if(age < 0) {
        throw new IllegalArgumentException("Age can not be negative");
    }
    if(age > 18) {
        return "Adult";
    } else {
        return "Child";
    }
}).handle((res, ex) -> {
    if(ex != null) {
        System.out.println("Oops! We have an exception - " + ex.getMessage());
        return "Unknown!";
    }
    return res;
```

```
});
```

```
System.out.println("Maturity : " + maturityFuture.get());
```

If an exception occurs, then the `res` argument will be null, otherwise, the `ex` argument will be null.