Limitations Of Future Interface

The Future interface in Java, while powerful for handling asynchronous operations, does have certain limitations. These constraints can impact how we design and implement concurrency in our Java applications. Here are some of the key limitations:

1. **Blocking Operations:** One of the most significant limitations is that the get() method of the Future interface is blocking. It waits until the task is completed, which can lead to inefficiency, especially in scenarios where you need to wait for the completion of multiple Future tasks. This blocking nature can negate some of the benefits of asynchronous programming.
2. **Lack of Direct Support for Completion Callbacks:** The Future interface does not provide a built-in mechanism to perform a callback function once the future's computation is

complete. We have to manually check if the task is completed, which is not efficient and does not adhere to a reactive programming model.

3. **No Support for Combining Multiple Futures:** Future does not provide native methods to combine multiple futures together or wait for the completion of multiple futures. For example, waiting for the completion of all or any future in a collection of futures requires additional boilerplate code.
4. **Limited Exception Handling:** When using the get() method, it can be challenging to handle exceptions elegantly. The method throws ExecutionException if the computation throws an exception, and InterruptedException if the current thread was interrupted while waiting. This requires extra boilerplate code for handling different types of exceptions that may occur during execution.
5. **Inability to Manually Complete a Future:** There is no direct way to complete a Future manually or set its value.

> Once a Future is created and its computation is underway, you cannot modify or update its state.

CompletableFuture provides a more flexible and powerful way to work with asynchronous computations compared to the traditional Future interface. Here's an example demonstrating both CompletableFuture and Future to illustrate the advantages of CompletableFuture:

java
Copy code

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FutureVsCompletableFutureExample {

    public static void main(String[] args) throws ExecutionException, InterruptedException {
        // Using CompletableFuture
        CompletableFuture<String> completableFuture = CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(2000); // Simulate some computation
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Result of the asynchronous computation";
        });

        completableFuture.thenAccept(result -> System.out.println("CompletableFuture Result: " +
result));

        // Using traditional Future
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<String> future = executor.submit(() -> {
            try {
                Thread.sleep(2000); // Simulate some computation
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Result of the asynchronous computation";
        });
```

```
        System.out.println("Future Result: " + future.get()); // Blocking call to get the result

        executor.shutdown();
    }
}
```

In this example:

CompletableFuture: The CompletableFuture.supplyAsync method is used to perform an asynchronous computation. It takes a Supplier and an optional Executor as parameters. We supply a Supplier that simulates some computation and returns a result asynchronously. Then, we use the thenAccept method to specify a callback that will be invoked with the result once it's available.

Future: We use ExecutorService to submit a Callable to perform an asynchronous computation. This returns a Future object representing the result of the computation. We then call get() on the Future object, which is a blocking call that waits for the computation to complete and retrieves the result.

Advantages of CompletableFuture over Future:

Asynchronous Composition: CompletableFuture provides a rich set of methods for composing asynchronous computations, chaining dependent actions, and handling errors. This enables more complex asynchronous workflows compared to the basic capabilities of Future.

Callback-based Approaches: CompletableFuture supports callback-based approaches like thenApply, thenAccept, thenCombine, etc., making it easier to express the logic to be executed when the computation completes successfully or exceptionally.

Timeouts and Error Handling: CompletableFuture allows specifying timeouts and error handling strategies, providing more control over the asynchronous operations.

Non-blocking Operations: CompletableFuture supports non-blocking operations, allowing you to perform actions without blocking the calling thread.

Overall, CompletableFuture offers more flexibility, composability, and control over asynchronous computations compared to the traditional Future interface.

CompletableFuture.supplyAsync is a method in Java's CompletableFuture class that allows you to asynchronously execute a task and obtain a result. It's similar to ExecutorService.submit, but

it provides additional capabilities for handling the result asynchronously. When you use supplyAsync, you can specify an Executor to control the execution environment of the task.

Here's how you can use CompletableFuture.supplyAsync in detail, including passing an Executor:

java
Copy code

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

public class SupplyAsyncExample {

    public static void main(String[] args) {
        // Create an Executor
        ExecutorService executor = Executors.newFixedThreadPool(4);

        // Asynchronously execute a task and obtain a result using supplyAsync
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            // Task to be executed asynchronously
            // Simulate some computation
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "Result of the asynchronous computation";
        }, executor); // Pass the executor to supplyAsync

        // Attach a callback to handle the result asynchronously
        future.thenAcceptAsync(result -> {
            // This code block will execute when the asynchronous computation completes
            System.out.println("Result: " + result);
        }, executor); // Pass the same executor to thenAcceptAsync

        // Note: The main thread continues executing other tasks concurrently

        // Shutdown the executor when it's no longer needed
        executor.shutdown();
    }
}
```

In this example:

We create an ExecutorService using Executors.newFixedThreadPool(4) to create a fixed-size thread pool with four threads. This ExecutorService will be used to execute the asynchronous task and handle the callback.

We use CompletableFuture.supplyAsync to asynchronously execute a task. Inside the supplyAsync method, we provide a Supplier lambda that represents the task to be executed asynchronously. We also pass the Executor (created in step 1) to specify the execution environment for the task.

We attach a callback using thenAcceptAsync to handle the result of the asynchronous computation. This method takes a Consumer lambda that specifies what to do with the result when it becomes available. We also pass the same Executor to thenAcceptAsync to ensure that the callback is executed asynchronously on the specified executor.

Finally, we continue executing other tasks concurrently in the main thread. When the asynchronous task completes and the result is available, the callback specified in thenAcceptAsync will be executed asynchronously on the provided Executor.

We shut down the executor when it's no longer needed to release its resources.

Using CompletableFuture.supplyAsync with a specified Executor allows you to control the execution environment of the asynchronous task and handle the result asynchronously using a callback. This approach provides more flexibility and control compared to traditional blocking methods.