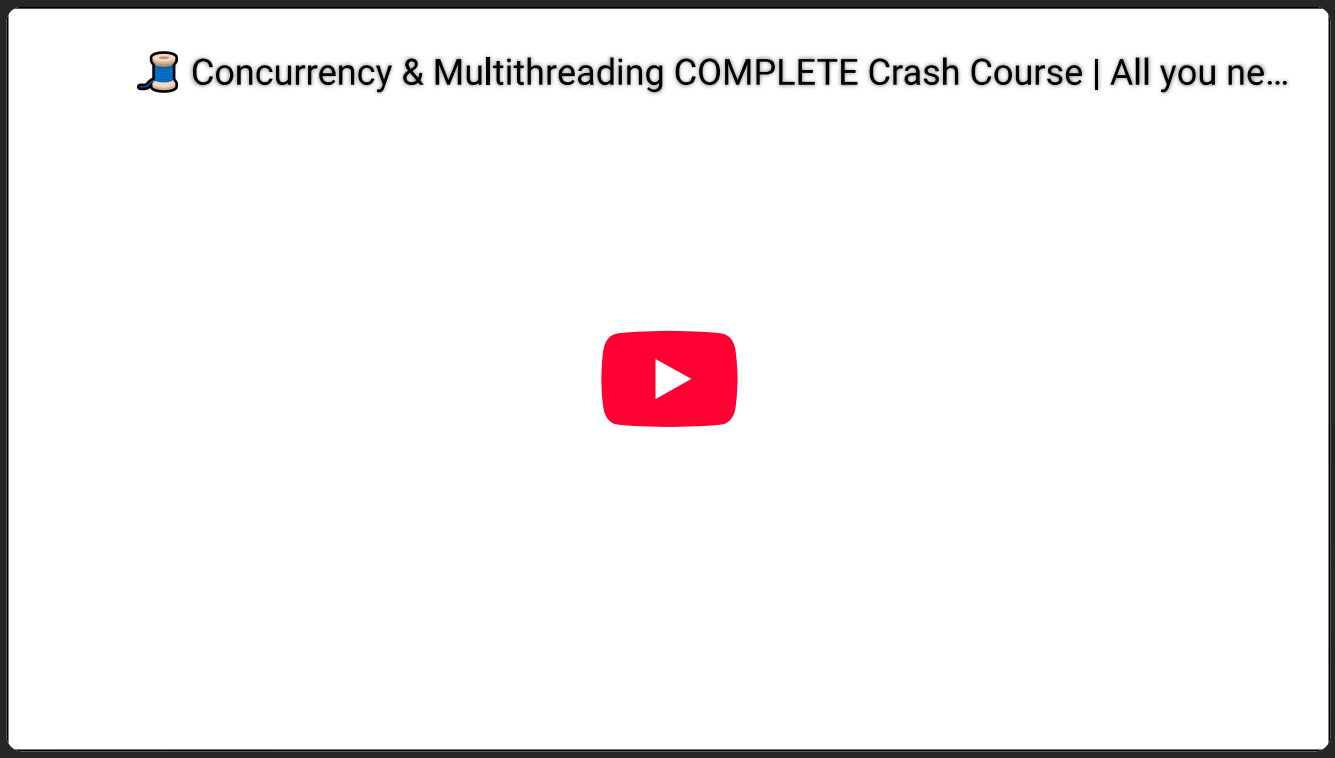
Search articles...

Sign In

Future and CompletableFuture



Topic Tags:

System Design LLD

🐆⬛ **Github Codes Link**: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Future and Completable Future in Java: Asynchronous Programming Essentials 🚀⏱️

In modern Java applications, handling asynchronous operations efficiently is crucial for creating responsive and scalable software. Two powerful tools for managing asynchronous tasks are the Future interface and its enhanced implementation, Completable Future. These constructs allow developers to work with results that may not be immediately available, enabling non-blocking operations and improving application performance. 🔄💻

Future Interface: The Foundation of Asynchronous Results 🏗️

The Future interface, introduced in Java 5, represents the result of an asynchronous computation. It provides a way to check if the computation is complete, wait for its completion, and retrieve the result.

Key Features of Future 🔑

1. Result Retrieval: Allows access to the result of an asynchronous operation once it's available. 📊

Example :

Java

```java
import java.util.concurrent.*;
public class FutureResultExample {
    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        Future<Integer> future = executor.submit(() -> 10 + 20);
        Integer result = future.get();  // blocks until result is ready
        System.out.println("Result: " + result);  // Result: 30
        executor.shutdown();
    }
}
```

Explanation :

○ submit() starts the task asynchronously.

○ future.get() blocks until the task finishes and returns the result.

2. Status Checking: Provides methods to check if a task has completed or been cancelled. ✅

Example :

Java

```java
Future<Integer> future = executor.submit(() -> 42);
// ...some time later
if (future.isDone()) {
    System.out.println("Task completed!"); // Prints -> Task completed!
} else {
```

```
    System.out.println("Still working..."); // Prints -> Still working...
}
```

Explanation : isDone() tells if the task is finished.

3. Cancellation: Supports cancellation of tasks that haven't started or are in progress. ❌

Example :

Java

```java
Future<?> future = executor.submit(() -> {
    while (true) {
        // long-running task
        if (Thread.currentThread().isInterrupted()) break;
    }
});


boolean cancelled = future.cancel(true); // interrupt the thread
System.out.println("Cancelled: " + cancelled);
```

Output :

```
Cancelled: true
```

Explanation :
○ cancel(true) tries to interrupt if the task is running.
○ You should check Thread.interrupted() inside your task to make cancellation responsive because :
▪ When you call future.cancel(true), it sends an interrupt signal to the thread running the task.
▪ But Java doesn't forcefully stop a thread — it just sets the interrupted flag.
▪ So your task must periodically check if it has been interrupted using Thread.interrupted().
▪ If you don't check it, the task will keep running, even if you cancel it.

Java

```java
Future<?> future = executor.submit(() -> {
            while (true) {
```

```
                if (Thread.interrupted()) {
                    System.out.println("Task was interrupted. Stopping...");
                    break;
                }
                // Simulate work
            }
        });
        future.cancel(true);  // sends interrupt
```

Output :

```
Task was interrupted. Stopping...
```

4. Blocking Operations: Primarily uses blocking methods that wait for task completion. ⏳

Example :

Java

```
Future<String> future = executor.submit(() -> {
    Thread.sleep(3000);
    return "Finished after delay";
});
System.out.println("Waiting for result...");
String value = future.get(); // blocks for ~3 seconds
System.out.println(value);
```

Output :

```
Waiting for result...
    Finished after delay
```

Explanation :

○ future.get() blocks until the task completes.

○ Use get(long timeout, TimeUnit unit) if you want to avoid indefinite blocking as :

▪ future.get() blocks the current thread until the task completes — this could take forever if the task hangs.

▪ future.get(timeout, unit) adds a maximum wait time.

▪ If the result isn't ready in that time, it throws a TimeoutException, and your thread can recover gracefully instead of hanging forever.

Basic Usage of Future 🛠️

In Java, a Future is a placeholder for the result of an asynchronous computation. It is commonly used in multithreading to handle tasks that take time to compute, allowing the main thread to continue execution while waiting for the result. A Future is typically obtained from an ExecutorService when submitting tasks.

Java

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

public class FutureExample {
    public static void main(String[] args) {
        // Create a single-threaded executor
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // Submit a task that returns a result in the future
        Future<String> future = executor.submit(() -> {
            Thread.sleep(2000); // Simulate a task that takes 2 seconds
            return "Task completed!";
        });

        try {
            System.out.println("Task submitted, doing other work...");

            // Check if the task is completed (non-blocking)
            System.out.println("Is task done? " + future.isDone());

            // Get the result - this blocks until the task is finished
            String result = future.get(); // Will block until computation
            System.out.println("Result: " + result);

            // Re-check task status after completion
            System.out.println("Is task done? " + future.isDone());
        } catch (InterruptedException | ExecutionException e) {
            // Handle exceptions that might occur during task execution
            System.out.println("Error: " + e.getMessage());
        } finally {
```

```
            // Shut down the executor to release resources
            executor.shutdown();
        }
    }
}
```

Output:

```
Task submitted, doing other work...
Is task done? false
Result: Task completed!
Is task done? true
```

Limitations of Future in Java

1. No Composition 🔗❌
• Future does not support chaining multiple tasks together. You cannot specify a dependent task that should execute once the Future completes, making it difficult to manage sequential asynchronous computations.

2. No Exception Handling ⚠️
• There is no built-in mechanism to handle exceptions in Future. If an exception occurs during execution, you must manually catch it using get(), which can make error handling cumbersome.

3. Blocking Operations 🛑
• Calling get() on a Future blocks the current thread until the result is available, leading to potential performance issues if the task takes too long to complete.

4. No Completion Notification 🔔❌
• Future does not provide an event-driven mechanism to notify when a task completes. You must explicitly poll or call get(), which is inefficient compared to reactive approaches.

This is why CompletableFuture is preferred, as it addresses these limitations with features like chaining, exception handling, and non-blocking callbacks.

When to Use Future ? 🤔

Future is a good choice for simpler asynchronous tasks where blocking is acceptable for result retrieval. However, for more advanced features like chaining, combining tasks, or non-

blocking result handling, alternatives such as CompletableFuture or third-party libraries like RxJava are recommended.

Example 1: Using Future (Blocking)

In this example, we submit a task to an executor and use future.get() to block until the result is ready. Once the result is returned, we print it and then perform an additional operation (printing another message).

Java

```java
import java.util.concurrent.*;
public class FutureExample {
    public static void main(String[] args) {
        // Create an executor service with a single thread.
        ExecutorService executor = Executors.newSingleThreadExecutor();

        // Submit a task that simulates work (sleep for 1 second) and retu
        Future<String> future = executor.submit(() -> {
            Thread.sleep(1000); // Simulate a delay
            return "Result from Future";
        });

        try {
            // Blocking call: waits for the result.
            String result = future.get();
            System.out.println("Future result: " + result);

            // Additional operation after the result is retrieved.
            System.out.println("Processing after Future result");
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            // Shut down the executor.
            executor.shutdown();
        }
    }
}
```

Output :

When you run the Future example, after approximately 1 second the task completes, and then the main thread prints the results sequentially. The output will be:

```
Future result: Result from Future
Processing after Future result
```

Explanation:

• The call to future.get() blocks the main thread until the task completes.

• Only after obtaining the result does it print the result and then execute the subsequent operation.

Example 2: Using CompletableFuture (Non-Blocking) :

Here, we use CompletableFuture to perform the same task. Instead of blocking with .get(), we attach a callback with thenAccept that is invoked when the task is complete.

Java

```java
import java.util.concurrent.CompletableFuture;
public class CompletableFutureExample {
    public static void main(String[] args) {
        // Start an asynchronous task using CompletableFuture.supplyAsync.
        CompletableFuture.supplyAsync(() -> {
            try {
                Thread.sleep(1000); // Simulate a delay
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "Result from CompletableFuture";
        })
        // Register a callback that processes the result once it's ready.
        .thenAccept(result -> {
            System.out.println("CompletableFuture result: " + result);
            // Additional operation after the result is available.
            System.out.println("Processing after CompletableFuture result"
        });
        // Optionally do other work here while the asynchronous task is ru
        System.out.println("Main thread is free to do other tasks while wa
        // To prevent the main thread from exiting immediately,
        // we'll wait for the CompletableFuture to complete.
```

```
        try {
            Thread.sleep(2000); // Wait enough time for the async task to
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

Output :
When you run the CompletableFuture example, the main thread immediately prints its message and then the asynchronous task completes (after approximately 1 second), triggering the callback to print its messages. The output will be:

```
Main thread is free to do other tasks while waiting...
CompletableFuture result: Result from CompletableFuture
Processing after CompletableFuture result
```

Explanation:

• The supplyAsync call initiates the asynchronous task without blocking the main thread.
• The thenAccept callback is registered to execute once the task is complete, printing both the result and the follow-up message.
• Meanwhile, the main thread can perform other work (as shown by the extra print statement) without waiting for the asynchronous result.
• A simple Thread.sleep at the end ensures the application doesn't terminate before the asynchronous work completes.

CompletableFuture: Advanced Asynchronous Programming 🔄

CompletableFuture, introduced in Java 8, extends the Future interface and implements the CompletionStage interface. It addresses the limitations of Future and provides a rich set of methods for composing, combining, and handling asynchronous computations.

Key Features of CompletableFuture 🔑

1. Non-blocking Operations: Supports non-blocking, asynchronous programming. 🏃
2. Composition: Allows chaining of multiple asynchronous operations. 🔗
3. Combination: Provides methods to combine results from multiple futures. 🤝

4. Exception Handling: Robust exception handling mechanisms. 🛡️
5. Completion Callbacks: Supports callbacks when tasks complete. 🔔
6. Explicit Completion: Can be completed explicitly, useful for complex scenarios. ✅

✅ Common CompletableFuture Methods Explained

The CompletableFuture class provides several utility methods to manually complete, query, or block for results. Let's go
through the most important ones.

1. get() – Wait and Retrieve the Result (Throws Checked Exception) :

Java

```java
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hell
try {
    String result = future.get(); // Blocks until result is available
    System.out.println(result);   // Prints "Hello"
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

2. join() – Wait and Retrieve the Result (Throws Unchecked Exception) - Not Recommended (Only when you know Exceptions won't come):

Java

```java
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Worl
String result = future.join(); // Blocks until result is available
System.out.println(result);    // Prints "World"
```

3. complete(value) – Manually Complete a Future :

Java

```java
CompletableFuture<String> future = new CompletableFuture<>();
future.complete("Manual Result");
```

```
System.out.println(future.join()); // Prints "Manual Result"
```

Use When:
• You want to complete a future manually (e.g., timeout fallback, mock).
• If the future is already completed, complete() will return false.

4. isDone() – Checks if the Future is Completed :

Java

```java
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Done
while (!future.isDone()) {
    System.out.println("Waiting...");
}
System.out.println("Result: " + future.join());
```

Use When:
• You want to poll or check status without blocking.
• Returns: true if the computation is complete, otherwise false

Code Snippets :

1. Creating CompletableFuture 🛠️
A CompletableFuture can be created in various ways. A pre-completed future is useful for scenarios where an already-known result needs to be wrapped in a future. For asynchronous tasks, you can use supplyAsync to perform operations in a different thread without blocking the current one.

Java

```java
import java.util.concurrent.*;
public class CompletableFutureCreation {
    public static void main(String[] args) {
        // Create a completed future with a pre-defined result
        CompletableFuture<String> completed = CompletableFuture.completedF
        System.out.println("Completed future result: " + completed.join())

        // Create and run a task asynchronously
        CompletableFuture<String> async = CompletableFuture.supplyAsync(()
```

```
            try {
                Thread.sleep(1000); // Simulate work
                return "Async result";
            } catch (InterruptedException e) {
                return "Interrupted";
            }
        });

        // Non-blocking check and then blocking get
        System.out.println("Is async done? " + async.isDone());
        System.out.println("Async result: " + async.join()); // Blocks unt
    }
}
```

Output :

```
Completed future result: Result
Is async done? false
Async result: Async result
```

2. Transforming Results with CompletableFuture 🔄
With CompletableFuture, you can easily transform results using methods like thenApply. This allows chaining of transformations and can be executed either synchronously or asynchronously based on your requirements.

• thenApplyis good when the transformation is trivial and you're not concerned about the current thread's workload.

• thenApplyAsyncis useful when you want to offload work to prevent a potentially busy or important thread (like an event loop or I/O thread) from doing heavy computation, even though the chain is sequential.

Java

```
import java.util.concurrent.CompletableFuture;
public class CompletableFutureThreadUsageExample {
    public static void main(String[] args) {
        CompletableFuture<String> future = CompletableFuture.supplyAsync((
                System.out.println("supplyAsync running in: " + Thread.cur
```

```
                return "Hello";
            })
            .thenApply(s -> {
                // thenApply runs in the same thread as supplyAsync if ava
                System.out.println("thenApply running in: " + Thread.curre
                return s + " World";
            })
            .thenApplyAsync(s -> {
                // thenApplyAsync uses a different thread from the default
                System.out.println("thenApplyAsync running in: " + Thread.
                return s + "! CompletableFuture is awesome.";
            });

        System.out.println("Final result: " + future.join());
    }
}
```

Output :

```
supplyAsync running in: ForkJoinPool.commonPool-worker-1
thenApply running in: ForkJoinPool.commonPool-worker-1
thenApplyAsync running in: ForkJoinPool.commonPool-worker-2
Final result: Hello World! CompletableFuture is awesome.
```

3. Combining CompletableFutures 🤝

In situations where multiple asynchronous tasks are running in parallel, CompletableFuture makes it easy to combine their results using thenCombine. Additionally, you can wait for all tasks to complete or get the first completed result using allOf and anyOf.

Java

```
import java.util.concurrent.*;
public class CompletableFutureCombination {
    public static void main(String[] args) {
        CompletableFuture<String> future1 = CompletableFuture.supplyAsync(
            try {
                Thread.sleep(1000);
                return "Future 1";
            } catch (InterruptedException e) {
```

```
                return "Interrupted";
            }
        });
        CompletableFuture<String> future2 = CompletableFuture.supplyAsync(
            try {
                Thread.sleep(2000);
                return "Future 2";
            } catch (InterruptedException e) {
                return "Interrupted";
            }
        });

        // Combine results of two futures
        CompletableFuture<String> combined = future1.thenCombine(future2,
            (result1, result2) -> result1 + " + " + result2);
        System.out.println("Combined result: " + combined.join());

        // Wait for all futures to complete
        CompletableFuture<Void> allOf = CompletableFuture.allOf(future1, f
        allOf.thenRun(() -> System.out.println("Both futures completed!"))

        // Wait for any one future to complete
        CompletableFuture<Object> anyOf = CompletableFuture.anyOf(future1,
        System.out.println("First completed: " + anyOf.join());
    }
}
```

Output :

```
Combined result: Future 1 + Future 2
Both futures completed!
First completed: Future 1
```

4. Exception Handling with CompletableFuture ⚠️

CompletableFuture provides robust mechanisms for handling exceptions during asynchronous tasks. Use exceptionally to recover from errors or handle to manage both success and failure scenarios.

Java

```java
import java.util.concurrent.*;
public class CompletableFutureExceptionHandling {
    public static void main(String[] args) {
        CompletableFuture<String> future = CompletableFuture.supplyAsync((
            if (Math.random() > 0.5) {
                throw new RuntimeException("Something went wrong!");
            }
            return "Success";
        })
        // Handle exceptions
        .exceptionally(ex -> {
            System.out.println("Exception caught: " + ex.getMessage());
            return "Recovery value";
        });
        System.out.println("Result: " + future.join());
        // Handle both success and failure
        CompletableFuture<String> handled = CompletableFuture.supplyAsync(
            if (Math.random() > 0.5) {
                throw new RuntimeException("Error occurred");
            }
            return "Success path";
        })
        .handle((result, ex) -> {
            if (ex != null) {
                return "Handled error: " + ex.getMessage();
            }
            return "Handled success: " + result;
        });
        System.out.println(handled.join());
    }
}
```

Output 1 (If Exception Occurs):

```
Exception caught: Something went wrong!
Result: Recovery value
Handled error: Error occurred
```

Output 2 (If No Exception Occurs) :

```
Result: Success
Handled success: Success path
```

5. Timeouts and Cancellation 🕑

Timeouts are essential in cases where an asynchronous task may take longer than expected. CompletableFuture provides methods like orTimeout and completeOnTimeout to handle timeouts gracefully or provide default values.

Java

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

public class CompletableFutureTimeout {
    public static void main(String[] args) {
        CompletableFuture<String> future = CompletableFuture.supplyAsync((
            try {
                Thread.sleep(3000); // Simulate a long-running task
                return "Task completed";
            } catch (InterruptedException e) {
                return "Task interrupted";
            }
        });

        {/* Apply timeout
        - orTimeout(long timeout, TimeUnit unit) - This adds a timeout to
        Arguments:
           - timeout: 2 - The duration of the timeout
           - unit: TimeUnit.SECONDS - The time unit for the timeout value
         What happens after timeout:
           - If the original future doesn't complete within 2 seconds, thi
             will complete exceptionally with a TimeoutException
           - The original future continues running in the background despi
        */}
        CompletableFuture<String> withTimeout = future.orTimeout(2, TimeUn

        try {
```

```
            System.out.println(withTimeout.join());
        } catch (CompletionException e) {
            System.out.println("Timeout occurred: " + e.getCause().getClas
        }

         {/*Provide a default value on timeout
        completeOnTimeout(T value, long timeout, TimeUnit unit) - This add
        Arguments:
           - value: "Default value" - The value to use if the timeout occu
           - timeout: 2 - The duration of the timeout
           - unit: TimeUnit.SECONDS - The time unit for the timeout value
        What happens after timeout:
           - If the original future doesn't complete within 2 seconds, thi
        will complete normally with the provided default value
           - The original future continues running in the background despi
       - If the original future completes before the timeout, its result i
        */}
        CompletableFuture<String> withDefault = future.completeOnTimeout("
        System.out.println("With default: " + withDefault.join());
    }
}
```

Output :

```
Timeout occurred: TimeoutException
With default: Default value
```

CompletableFuture vs Future: A Comparison ⚖️

Feature	Future	CompletableFuture
Introduced in	Java 5	Java 8
Result Retrieval	Blocking	Blocking and non-blocking
Composition	No	Yes
Exception Handling	Limited	Robust
Cancellation	Basic	Advanced
Timeout Support	No built-in	Built-in
Completion Notification	No	Yes
Combining Results	No	Yes
Execution Control	Limited	Extensive

Conclusion 🎯

CompletableFuture represents a significant advancement in Java's asynchronous programming capabilities. While the basic Future interface laid the groundwork for handling asynchronous results, CompletableFuture's composition, combination, and exception handling features make it a far more powerful tool for modern Java applications.