Java CompletableFuture

ANJU MUNOTH

CompletableFuture

- Class introduced in Java 8 that allows us to write asynchronous, non-blocking code.
- Powerful tool that can help write code that is more efficient and responsive.
- Asynchronous operations facilitate non-blocking I/O, leading to more efficient use of resources.

Key Features of CompletableFuture:

- **Completing Future:** Can manually complete a CompletableFuture with a value or an exception.
- **Combining Futures:** CompletableFuture allows combining multiple futures using methods like thenCombine, thenCompose, etc.
- **Handling Errors:** Can handle errors in asynchronous computations using exceptionally, handle, etc.
- **Running Async Tasks:** Can run tasks asynchronously using runAsync and supplyAsync.
- **Callback Methods:** Can attach callbacks to the future that will execute when the future completes, using methods like thenApply, thenAccept, etc.

Java's future interface

- Future interface in Java represents the result of an asynchronous computation.
- Tasks executed in a separate thread can return a Future object, which can be used to check if the computation is complete, wait for its completion, and retrieve the result.
- **Limitations**: The main limitation of the Future interface is its lack of ability to manually complete the computation, combine multiple futures, or chain actions that rely upon the future's completion.
- These operations either block or require additional mechanisms to handle, making the Future interface less flexible compared to CompletableFuture.

Java's CompletableFuture class

- Introduced in Java 8.
- Part of Java's java.util.concurrent package and provides a way to write asynchronous code by representing a future result that will eventually appear.
- Can perform operations like calculation, transformation, and action on the result without blocking the main thread.
- Helps in writing non-blocking code where the computation can be completed by a different thread at a later time.
- CompletableFuture and the broader Java Concurrency API make use of thread pools (like the ForkJoinPool) for executing asynchronous operations.
- Allows Java applications to handle multiple asynchronous tasks efficiently by leveraging multiple threads.

Java's CompletableFuture class

- when a **CompletableFuture** operation is waiting on a dependent future or an asynchronous computation, it doesn't block the waiting thread.
- Instead, the completion of the operation triggers the execution of dependent stages in the **CompletableFuture** chain, potentially on a different thread from the thread pool.

Example scenario with CompletableFuture

Need to perform a series of dependent and independent asynchronous operations:

- **Fetch user details:** Given a userID, we first retrieve the user's details asynchronously.
- **Fetch credit score:** Once we have the user's details, we fetch their credit score.
- **Calculate account balance**: Independently, we also calculate the user's account balance from a different source.
- **Make a decision**: Finally, we combine the credit score and account balance to make a financial decision.
- Handle potential **errors**

Future vs CompletableFuture

- Future and CompletableFuture are both abstractions for representing a result that will be available in the future,

Step 1:
Fetching user details asynchronously

- Simulating an asynchronous operation to fetch user details using **supplyAsync**.
- Returns a **CompletableFuture** that will complete with the user details:

```java
Java
CompletableFuture<String> getUserDetailsAsync(String userId) {
        return CompletableFuture.supplyAsync(() -> "UserDetails for " + userId);
}
```

Step 2: Transforming and fetching credit score

- Use **thenApply** to transform the result (e.g., formatting user details) and **thenCompose** to fetch the credit score, demonstrating the chaining of asynchronous operations:
- **thenApply** is for synchronous transformations, while **thenCompose** allows for chaining another asynchronous operation that returns a **CompletableFuture**.

```java
Java
CompletableFuture<String> userDetailsFuture = getUserDetailsAsync("userId123")
        .thenApply(userDetails -> "Transformed " + userDetails);

CompletableFuture<Integer> creditScoreFuture = userDetailsFuture
        .thenCompose(userDetails -> getCreditScoreAsync(userDetails));
```

Step 3: Calculating account balance in parallel

- Calculate the account balance using another asynchronous operation, showcasing how independent futures can run in parallel:

Java

```java
CompletableFuture<Double> accountBalanceFuture =
calculateAccountBalanceAsync("userId123");
```

Step 4: Combining results and making a decision.

- **thenCombine** we merge the results of two independent CompletableFuture - credit score and account balance - to make a decision:

```java
Java
CompletableFuture<Void> decisionFuture = creditScoreFuture
        .thenCombine(accountBalanceFuture, (creditScore, accountBalance) ->
        makeDecisionBasedOnCreditAndBalance(creditScore, accountBalance))
        .thenAccept(decision -> System.out.println("Decision: " + decision));
```

Step 5: Error handling

- Error handling is crucial in asynchronous programming.
- Use **exceptionally** to handle any exceptions that may occur during the asynchronous computations, providing a way to recover or log errors:

```java
.exceptionally(ex -> {
        System.err.println("An error occurred: " + ex.getMessage());
        return null;
});
```

Async Methods of CompletableFuture

- CompletableFuture provides a set of asynchronous methods to execute multiple tasks concurrently and process the results as soon as they become available.
- Can create a chain of dependent tasks and execute them in parallel, improving the performance of your application.

supplyAsync

- Run a piece of code asynchronously and return a **CompletableFuture** that will be completed with the value obtained from that code.
- Execute a **Supplier<T>** asynchronously, where **T** is the type of value returned by the Supplier.

```Java
static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
```

supplyAsync

Java

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    // Simulate a long-running operation
    try {
        TimeUnit.SECONDS.sleep(2);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
    return "Result of the asynchronous operation";
});
```

supplyAsync

- When you invoke supplyAsync, it executes the given Supplier **asynchronously (usually in a different thread).**
- Method immediately returns a CompletableFuture object.
- CompletableFuture will be completed in the future when the Supplier finishes its execution, with the result being the value provided by the Supplier.
- Allows the main thread to continue its operations without waiting for the task to be completed.
- Particularly useful in web applications or any I/O-bound applications where you don't want to block the current thread.
- By default, tasks submitted via supplyAsync without specifying an executor are executed in the common **fork-join pool** (ForkJoinPool.commonPool()).

supplyAsync

- can also specify a custom **Executor** if you need more control over the execution environment:

```java
Executor executor = Executors.newCachedThreadPool();
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
        // Task here
        return "Result";
}, executor);
```

runAsync

- **CompletableFuture.runAsync** is akin to **CompletableFuture.supplyAsync** but for scenarios where you don't need to return a value from the asynchronous operation.

- Both methods are intended for executing tasks asynchronously, but they differ in their return types and the type of tasks they're suited for.

- runAsync is used to execute a Runnable task asynchronously, which does not return a result.

- Since Runnable does not produce a return value, runAsync returns a CompletableFuture<Void>.

```java
static CompletableFuture<Void> runAsync(Runnable runnable)
static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
```

runAsync

- **Asynchronous execution**: Executes the given **Runnable** task in a separate thread, allowing the calling thread to proceed without waiting for the task to complete.
- **No return value**: Suitable for asynchronous tasks that perform actions without needing to return a result, such as logging, sending notifications, or other side effects.
- **Custom executor support**: Allows specifying a custom **Executor** for more control over task execution, such as using a dedicated thread pool.

runAsync

Java

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
        // Simulate a task that takes time but doesn't return a result
        try {
                TimeUnit.SECONDS.sleep(1);
                System.out.println("Task completed");
        } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
        }
});

// Do something else while the task executes
future.join(); // Wait for the task to complete if necessary
```

join()

- **join** method on a **CompletableFuture** is a blocking call that causes the current thread to wait until the **CompletableFuture** is completed.

- During this waiting period, the current thread is inactive, essentially "joining" the completion of the task represented by the **CompletableFuture**.

join()

- **Blocking behaviour**: **join()** blocks until the **CompletableFuture** upon which it is called completes, either normally or exceptionally.
  - Makes the asynchronous operation synchronous for the calling thread, as the thread will not proceed until the future is completed.
- **Exception handling**: Unlike **get()**, which throws checked exceptions (such as **InterruptedException** and **ExecutionException**), **join()** is designed to throw an unchecked exception (**CompletionException**) if the **CompletableFuture** completes exceptionally.
  - This can simplify error handling in certain contexts where checked exceptions are undesirable.
- **Usage**: Used when you need to **synchronise asynchronous computation at some point,** for example, when the result of the asynchronous computation is required for subsequent operations, or at the end of a program to ensure that all asynchronous tasks have completed.

Example scenario

- If you have a main application thread that kicks off an asynchronous task using **CompletableFuture.runAsync()** or **CompletableFuture.supplyAsync()**, and later in the program you need the result of that task or need to ensure that the task has completed before proceeding, you might call **join()**:

```java
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
        // Long-running task
        try {
                TimeUnit.SECONDS.sleep(1); // Simulates a delay
        } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
        }
        System.out.println("Async task finished");
});

System.out.println("Waiting for the async task to complete...");
future.join(); // Blocks here until the above task completes
System.out.println("Main thread can now proceed");
```

get()

- Blocks the current thread until the **CompletableFuture** completes, either normally or exceptionally.
- Once the future completes, **get()** returns the result of the computation if it completed normally, or throws an exception if the computation completed exceptionally.

get()

- **Blocking behaviour:** Like join() ,get() is a blocking call.
  - Makes the caller thread wait until the CompletableFuture's task is completed.
- **Checked** exceptions: get() can throw checked exceptions, including:
  - InterruptedException: If the current thread was interrupted while waiting.
  - ExecutionException: If the computation threw an exception. This exception wraps the actual exception thrown by the computation, which can be obtained by calling getCause() on the ExecutionException.
- **Timeout**: The overloaded version of get(long timeout, TimeUnit unit) allows specifying a maximum wait time. If the timeout is reached before the future completes, it throws a TimeoutException, providing a mechanism to avoid indefinite blocking.
- **Use case:** Use get() when you need to handle checked exceptions explicitly, or when you need to retrieve the result of the computation within a certain timeframe.

Java

```java
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
        // Simulate a long-running operation
        try {
                TimeUnit.SECONDS.sleep(2); // 2-second delay
        } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
        }
        return "Result of the asynchronous operation";
});

try {
        // Attempt to retrieve the result, waiting up to 3 seconds
        String result = future.get(3, TimeUnit.SECONDS);
        System.out.println(result);
} catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        System.out.println("The current thread was interrupted while
waiting.");
} catch (ExecutionException e) {
        System.out.println("The computation threw an exception: " +
e.getCause());
} catch (TimeoutException e) {
        System.out.println("Timeout reached before the future completed.");
}
```

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

- **Purpose**: Applies a synchronous transformation function to the result of the **CompletableFuture** when it completes.
- **Behaviour**: Executes on the same thread that completed the previous stage, or in the thread that calls **get()** or **join()** if the future has already completed.
- **Return type**: **CompletableFuture<U>** where U is the type returned by the function

thenApplyAsync():

- Method is used to process the result of a task asynchronously and return a new CompletableFuture with the transformed result.
- Processing is done by a separate thread in the ForkJoinPool.commonPool()

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

- **Purpose**: Similar to **thenApply**, but the transformation function is executed asynchronously, typically using the default executor.

- **Behaviour**: Can execute the function in a different thread, providing better responsiveness and throughput for tasks that are CPU-intensive or involve blocking.

- **Return type**: **CompletableFuture<U>**

thenApplyAsync():

Java

```
CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> 42);
CompletableFuture<String> applied = future.thenApply(result -> "Result: " +
result);
applied.thenAccept(System.out::println); // Prints: Result: 42

CompletableFuture<String> appliedAsync = future.thenApplyAsync(result ->
"Async Result: " + result);
appliedAsync.thenAccept(System.out::println); // Prints: Async Result: 42
```

thenApplyAsync():

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<Integer> transformedFuture = future.thenApplyAsync(s -> {
System.out.println("Thread: " + Thread.currentThread().getName());
return s.length();
});

transformedFuture.thenAccept(length -> {
System.out.println("Thread: " + Thread.currentThread().getName());
System.out.println("Length of Hello: " + length);
});
```

thenAccept and thenAcceptAsync

- **Purpose**: Consumes the result of the **CompletableFuture** without returning a result.
- **thenAccept** is synchronous, while **thenAcceptAsync** is asynchronous.
- **Use case**: Useful for executing side-effects, such as logging or updating a user interface, with the result of the **CompletableFuture**.

thenAcceptAsync():

- Method is used to consume the result of a task asynchronously, without returning a value.
- Processing is done by a separate thread in the ForkJoinPool.commonPool()

thenAccept and thenAcceptAsync

```
Java
CompletableFuture<Void> accepted = CompletableFuture.supplyAsync(() -> "Hello")
        .thenAccept(result -> System.out.println(result + ", World!")); //
Prints: Hello, World!

CompletableFuture<Void> acceptedAsync = CompletableFuture.supplyAsync(() ->
"Async Hello")
        .thenAcceptAsync(result -> System.out.println(result + ", World!")); //
Prints: Async Hello, World!
```

thenCombine

- thenCombine(CompletionStage<? extends V> other, BiFunction<? super T,? super V,? extends U> fn)

- **Purpose**: Combines the result of this **CompletableFuture** with another asynchronously computed value. The combination is done when both futures complete.
- **Behaviour**: The **BiFunction** provided is executed synchronously, using the thread that completes the second future.
- **Return type: CompletableFuture<U>**

thenCombineAsync

- thenCombineAsync(CompletionStage<? extends V> other, BiFunction<? super T,? super V,? extends U> fn)

- Purpose: Similar to thenCombine, but the BiFunction is executed asynchronously.
- **Behaviour**: Useful when the combination function is computationally expensive or involves blocking.
- **Return type: CompletableFuture<U>**

thenCombineAsync

Java

```java
CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() ->
40);
CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> 2);
CompletableFuture<Integer> combined = future1.thenCombine(future2,
Integer::sum);
combined.thenAccept(result -> System.out.println("Sum: " + result)); //
Prints: Sum: 42

CompletableFuture<Integer> combinedAsync = future1.thenCombineAsync(future2,
Integer::sum);
combinedAsync.thenAccept(result -> System.out.println("Async Sum: " +
result)); // Prints: Async Sum: 42
```

thenComposeAsync()

- Method in CompletableFuture that allows to chain multiple asynchronous tasks together in a non-blocking way.
- Used when you have one CompletableFuture object that returns another CompletableFuture object as its result, and you want to execute the second task after the first one has completed.
- Method takes a Function object as its argument, which takes the result of the first CompletableFuture object as its input and returns another CompletableFuture object as its result.
- Second task is executed when the first one completes, and its result is passed to the next stage of the pipeline.

thenComposeAsync()

```
CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> future2 = future1.thenComposeAsync(s -> CompletableFuture.supplyAsync(() -> s
+ " World"));

future2.thenAccept(result -> System.out.println(result));
```

exceptionally(Function<Throwable,? extends T> fn)

- **Purpose**: Handles exceptions arising from the **CompletableFuture** computation, allowing for a fallback value to be provided or a new exception to be thrown.
- **Use case**: Essential for robust error handling in asynchronous programming, allowing for recovery or logging of failures.

```java
Java
CompletableFuture<String> exceptionFuture = CompletableFuture.supplyAsync(() ->
{
        if (true) throw new RuntimeException("Exception!");
        return "No Exception";
}).exceptionally(ex -> "Exception Handled: " + ex.getMessage());
exceptionFuture.thenAccept(System.out::println); // Prints: Exception
Handled: java.lang.RuntimeException: Exception!
```

Misuse of CompletableFuture leading to subtle bugs and performance issues

- **Blocking calls inside CompletableFuture:** Using get() or join() within a CompletableFuture's chain can block the asynchronous execution, negating the benefits of non-blocking code.
  - Solution: Replace blocking calls with non-blocking constructs like thenCompose for chaining futures or thenAccept for handling results.
- **Ignoring returned futures:** Not handling the CompletableFuture returned by methods like thenApplyAsync can lead to unobserved exceptions and behaviour that does not execute as expected.
  - Solution: Always chain subsequent operations or attach error handling (e.g., exceptionally or handle) to every CompletableFuture.

Debugging Challenges in Asynchronous Code

- **Stack traces lack context**: Exceptions in asynchronous code can have stack traces that don't easily lead back to the point where the async operation was initiated.
- **Strategies**:
  - Use **handle** or **exceptionally** to catch exceptions within the future chain and add logging or breakpoints.
  - Consider wrapping asynchronous operations in higher-level methods that catch and log exceptions, providing more context.

Strategies to identify and fix common issues

- **Consistent error handling**: Attach an **exceptionally** or **handle** stage to each **CompletableFuture** to manage exceptions explicitly.
- **Avoid common pitfalls**: For example - executing long-running or blocking operations in **supplyAsync** without specifying a custom executor. This can lead to saturation of the common fork-join pool.
  Solution: Use a custom executor for CPU-bound tasks to prevent interference with the global common fork-join pool.
- **Debugging Asynchronous Chains**: Break down complex chains of **CompletableFuture** operations into smaller parts. Test each part separately to isolate issues.

Tools and techniques for debugging CompletableFuture chains

- **Logging**: Insert logging statements within completion stages (e.g., after **thenApply**, **thenAccept**) to trace execution flow and data transformation.
- **Visual debugging tools**: Some IDEs and tools offer visual representations of **CompletableFuture** chains, which can help in understanding the flow and identifying where the execution might be hanging or failing.
- **Custom executors for monitoring**: Use custom executors wrapped with logging or monitoring to track task execution and thread usage. This is particularly useful for identifying tasks that run longer than expected.
- **Async profiling**: Tools like async-profiler can help identify hotspots and thread activity specific to asynchronous operations.

Fetching Data from Multiple APIs

- Suppose you have an application that needs to fetch data from multiple web services simultaneously and then combine the results.
- Using CompletableFuture, you can perform these API calls in parallel:

```java
public static void main(String[] args) {
        CompletableFuture<String> api1 = CompletableFuture.supplyAsync(() ->
fetchFromApi1());
        CompletableFuture<String> api2 = CompletableFuture.supplyAsync(() ->
fetchFromApi2());

        CompletableFuture<Void> combinedFuture = CompletableFuture.allOf(api1,
api2);

        combinedFuture.thenRun(() -> {
            try {
                String result1 = api1.get();
                String result2 = api2.get();
                System.out.println("Combined result: " + result1 + " + " + result2);
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }
}
```

```java
private static String fetchFromApi1() {
        // Simulate API call
        return "Data from API 1";
    }

private static String fetchFromApi2() {
        // Simulate API call
        return "Data from API 2";
    }
```

Asynchronous File Processing

- Tasks like reading or writing large files that you want to perform asynchronously to avoid blocking the main thread.

```java
import java.util.concurrent.CompletableFuture;
import java.nio.file.*;

public class FileProcessingExample {
    public static void main(String[] args) {
        CompletableFuture<Void> future = CompletableFuture.runAsync(() ->
processFile("example.txt"));

        future.thenRun(() -> System.out.println("File processing completed!"));

        // Wait for the asynchronous task to complete
        future.join();
    }

    private static void processFile(String fileName) {
        try {
            // Simulate file processing
            Thread.sleep(2000);
            System.out.println("Processing file: " + fileName);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Web Scraping

- If you're scraping data from multiple web pages, you can use CompletableFuture to perform these operations in parallel and then aggregate the results.

```java
public static void main(String[] args) {
    List<String> urls = Arrays.asList("https://example.com/page1",
"https://example.com/page2");

    CompletableFuture<?>[] futures = urls.stream()
        .map(url -> CompletableFuture.supplyAsync(() -> scrapePage(url)))
        .toArray(CompletableFuture[]::new);

    CompletableFuture<Void> allOf = CompletableFuture.allOf(futures);

    allOf.thenRun(() -> {
        try {
            Arrays.stream(futures).forEach(future -> {
                try {
                    System.out.println(future.get());
                } catch (Exception e) {
                    e.printStackTrace();
                }
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }).join();
}
```

```java
private static String scrapePage(String url) {
    // Simulate web scraping
    return "Content from " + url;
}
```

Background Data Processing

- In data-driven applications, you might have background tasks such as updating statistics or performing batch processing.

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

public class BackgroundProcessingExample {
    public static void main(String[] args) {
        CompletableFuture<Void> future = CompletableFuture.runAsync(() ->
updateStatistics());

        future.thenRun(() -> System.out.println("Statistics updated!"));

        // Continue with other tasks
        // ...
    }

    private static void updateStatistics() {
        try {
            // Simulate long-running task
            Thread.sleep(3000);
            System.out.println("Statistics processing completed.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

User Interface Responsiveness

- In GUI applications, you can use CompletableFuture to perform time-consuming tasks asynchronously, ensuring the UI remains responsive.

```java
public static void main(String[] args) {
      JFrame frame = new JFrame("Async UI Example");
      JButton button = new JButton("Fetch Data");
      button.addActionListener(e -> fetchDataAsync());
      frame.add(button);
      frame.setSize(200, 200);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
  }
```

```java
private static void fetchDataAsync() {
      CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
         try {
            // Simulate long-running task
            Thread.sleep(2000);
            System.out.println("Data fetched!");
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      });

      future.thenRun(() -> System.out.println("Update UI with fetched data"));
  }
```

Reading file synchronously

```
public static String readFileContentsSync(String filePath) throws IOException {
        Path path = Path.of(filePath);
        return Files.readString(path);
    }
```

```
try {
            String contents = readFileContentsSync("sampleFile1");
            System.out.println(contents);
    } catch (IOException e) {
            e.printStackTrace();
    }
```

```java
public static CompletableFuture<String> readFileContentsAsync(String filePath) {
        return CompletableFuture.supplyAsync(() -> {
            try {
                Path path = Path.of(filePath);
                return Files.readString(path);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        });
    }
```

```java
CompletableFuture<String> future = readFileContentsAsync("sampleFile1");

future.thenAccept(contents -> System.out.println("File Contents:\n" + contents))
.exceptionally(ex -> {
            System.err.println("An error occurred: " + ex.getMessage());
            return null;
        });

// Do other tasks here while the file is being read
System.out.println("reading file");
// Wait for the asynchronous task to complete
future.join();
```

- CompletableFuture.supplyAsync(() -> { ... }): This method runs the provided lambda expression in a separate thread, making the file reading operation asynchronous.
- Exception Handling: If an exception occurs during file reading, it is caught and rethrown as a RuntimeException, which can be handled using exceptionally.
- thenAccept: This method attaches a callback that will execute when the future completes successfully, printing the file contents.
- exceptionally: This method attaches a callback that will handle any exceptions thrown during the asynchronous operation.

```java
public static CompletableFuture<Void> writeFileContentsAsync(String filePath, String content) {
        return CompletableFuture.runAsync(() -> {
            try {
                Path path = Path.of(filePath);
                Files.writeString(path, content);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        });
    }
```

```java
String filePath = "example.txt";
        String content = "This is the content to be written into the file.";
        CompletableFuture<Void> future = writeFileContentsAsync(filePath, content);
        future.thenRun(() -> System.out.println("File writing completed!"))
              .exceptionally(ex -> {
                  System.err.println("An error occurred: " + ex.getMessage());
                  return null;
              });
        // Do other tasks here while the file is being written
// Wait for the asynchronous task to complete
        future.join();
```