Manish Jaiswal

Aync Programing Series

MASTERING

Exception Handling in Asynchronous Programming

swipe

Manish Jaiswal

Aync Programing Series

01

The Try-Catch Block

Use a try-catch block to handle exceptions that occur in asynchronous code. Ensure that await is used within the try block to catch exceptions.Use a try-catch block to handle exceptions that occur in asynchronous code. Ensure that await is used within the try block to catch exceptions.

swipe

Manish Jaiswal

Aync Programing Series

02

Example

```
1   try
2   {
3       await SomeAsyncOperation();
4   }
5   catch (Exception ex)
6   {
7       // Handle the exception
8   }
```

swipe

Manish Jaiswal

Aync Programing Series

03

Aggregating Exceptions

When using Task.WhenAll, exceptions from multiple tasks are wrapped in an AggregateException. Handle this by iterating through the **InnerExceptions** collection.

```
try
{
    await Task.WhenAll(task1, task2);
}
catch (AggregateException ex)
{
    foreach (var innerEx in ex.InnerExceptions)
    {
        // Handle each exception
    }
}
```

swipe

04

Examples

1. Handling Timeouts

Use Task.TimeoutAfter or CancellationToken to handle operations that might exceed a time limit.

```
try
{
    await SomeAsyncOperation().TimeoutAfter(TimeSpan.FromSeconds(5));
}
catch (TimeoutException ex)
{
    // Handle the timeout exception
}
```

swipe

05

Examples

2. Retrying Failed Operations

Use a retry pattern to handle transient exceptions, such as network failures.

```
int retryCount = 3;
while (retryCount > 0)
{
    try
    {
        await SomeAsyncOperation();
        break; // Exit loop if successful
    }
    catch (TransientException)
    {
        if (--retryCount == 0) throw; // Rethrow if no retries left
        await Task.Delay(1000); // Wait before retrying
    }
}
```

swipe

Manish Jaiswal

Aync Programing Series

06

Best Practices for Robust Error Handling

1. Always Await Async Methods

Ensure that all asynchronous methods are awaited to prevent unhandled exceptions.

2. Centralize Exception Handling

Centralize your exception handling logic, such as in a middleware or a custom handler, to ensure consistent behavior across your application.

swipe

07

3. Use Exception Filters

C# allows the use of exception filters to handle **specific exceptions** without catching all exceptions.

```
1  try
2  {
3      await SomeAsyncOperation();
4  }
5  catch (Exception ex) when (ex is SpecificException)
6  {
7      // Handle only SpecificException
8  }
```

swipe

08

Use Cases

1. Web APIs

Ensure that asynchronous controller actions properly handle exceptions to avoid exposing sensitive information or crashing the server.

2. Background Services

In long-running background tasks, handle exceptions gracefully to ensure that the service remains responsive and recovers from errors.

3. User Interfaces

In UI applications, ensure that exceptions in asynchronous code do not crash the application and provide user-friendly error messages.

swipe

Manish Jaiswal

Aync Programing Series

09

Interview Questions

- How does exception handling differ between synchronous and asynchronous code in C#?

- What is an AggregateException, and how would you handle it in asynchronous code?

- Explain how you would implement a retry mechanism in asynchronous programming.

- How can you handle a TaskCanceledException in a C# async method?

- What are the benefits of using exception filters in C#?

swipe