Exception handling -

Why is handling exceptions very crucial in any application ?
How do we handle them ?
diff between finally, finalize and final keyword in C# .NET ?

File Handling in C#.NET	File I/O
	Create
	Write
	Append
	Delete
	Copy
Collections in C#.NET: Generic and Non Generic	Collections
	Non generic
	Array list
	Hash table
	Sorted list
	Stack
	Queue
	Generic
	List
	Dictionary
	Sorted list
	Stack
	Queue

Exception Handling in C#	Introduction to Exception Handling Types of Exceptions in C#, The try, catch, finally Blocks Common Exception Classes Creating Custom Exceptions(user define exception) Exception Filters (C# 6.0 and Later) Exception Handling Best Practices
Latest C# 7.0 features	Extension Methods
	Tuples and Deconstruction, Pattern Matching, Local Functions,Out Variables, Ref Locals and Returns, Discards (_) & Expression-bodied Members
C# 8.0 features	Nullable Reference Types, Async Streams, Ranges and Indices,Switch Expressions, Default Interface Methods, Pattern Matching Enhancements,

	Using Declarations & Readonly Members

Case study based on Exception handling:

A **digital banking platform** processes thousands of transactions daily.
To ensure **data integrity, security, and reliability**, the system must:

- Validate inputs
- Handle business rule violations
- Log and communicate meaningful errors
- Prevent system crashes

Invalid Transaction Amount	**Insufficient Balance**	**Daily Transfer Limit Exceeded**
As a customer, I should not be allowed to transfer a negative or zero amount So that invalid transactions are prevented. Handled using built-in exception (`ArgumentException`)	**As a banking system**, I must stop transactions when account balance is insufficient So that overdraft does not occur. Handled using a custom exception (`InsufficientBalanceException`)	**As a compliance team**, I want transactions exceeding the daily limit to be blocked So that regulatory policies are followed. Handled using a custom exception (`DailyLimitExceededException`)

Graceful Error Communication
As an end user, I want clear and friendly error messages So that I understand why a transaction failed. Handled using try-catch-finally blocks

//What are the best practices for Exception Handling in C#.NET?/or in general

//1. Use Specific Exception Types: Catch specific exceptions rather than using a generic catch-all approach to provide more meaningful error handling.

//2. Avoid Swallowing Exceptions: Do not catch exceptions without proper handling or logging, as it can lead to silent failures and make debugging difficult.

//ex : catch(Exception ex) { //do nothing }

//ex: catch(Exception ex) { return null; }

//ex: catch(Exception ex) { Console.WriteLine("Error occurred"); }

//in above cases we are not logging the exception details anywhere, hence it will be difficult to debug the issue.

//instead of this we should log the exception details to a file, database or monitoring system for further analysis.

//so that in future we can refer to the logs to identify and fix the issue. hnece overall objective of exception handling is defeated.

//3. Use Finally Block for Cleanup: Always use finally block to release resources like file handles, database connections etc. to prevent resource leaks.

//4. Log Exception Details: Always log exception details including stack trace, message, and inner exceptions for better debugging and analysis.

//5. Provide Meaningful Error Messages: When throwing exceptions, provide clear and meaningful error messages to help users understand the issue.

//6. Avoid Using Exceptions for Control Flow: Do not use exceptions for regular control flow in your application, as it can lead to performance issues and code complexity.

//ex: using exceptions to handle expected scenarios like validation failures is not recommended.

//ex: if(input is invalid) { throw new Exception("Invalid input"); } //not recommended

//ex: if(input is invalid) { return false; } //recommended

//when we use exceptions for control flow, it can lead to performance overhead due to exception handling mechanisms.

//7. Create Custom Exception Classes: When necessary, create custom exception classes to represent specific error conditions in your application.

//8. Avoid Catching System.Exception: Avoid catching the base System.Exception class unless absolutely necessary, as it can hide critical exceptions that should be handled differently.

//9. Use Exception Filters: Utilize exception filters to catch exceptions based on specific conditions for more granular control over exception handling.

//10. Test Exception Handling Code: Thoroughly test your exception handling code to ensure it behaves as expected under various error conditions.

//We use exception filter to add conditions to catch blocks.
//ex: catch(DivideByZeroException ex) when (some condition) { //handle exception }
//ex in terms of user define exception:
//catch(DailyLimitExceededException ex) when (some condition) { //handle exception }
//condition can be any boolean expression that evaluates to true or false. like checking exception properties or other variables.

//Most common implementatino of exception filter is logging.
//Case study based on logging:
//if there is a need to log only specific exceptions based on certain conditions. like logging only critical exceptions or exceptions occurring in specific modules.
//we can use exception filters to achieve this.
//Step 1: Define a logging method that logs exception details based on certain conditions.
//ex. LogException(Exception ex) to log exception details to a file, database or monitoring system.
//ex. we can add conditions inside this method to log only specific exceptions.
//Step 2: Use exception filters in catch blocks to call the logging method based on certain conditions.
//Step 3: Handle the exception as needed after logging.
//Step 4: Test the implementation to ensure that only the desired exceptions are logged based on the specified conditions.
//Step 5: Refine the conditions and logging logic as needed based on feedback and requirements.

<u>Case study based on implementing filter while handling exception so that logging can be implemented for specific types of exception</u>

Business Scenario

A **large enterprise order-processing system** handles:

- User input validation
- Business rule checks
- External service calls
- Database operations

objective :

The **logging team reports log noise**:

- Too many logs for **expected business errors**
- Critical failures getting buried

The system must **log only critical exceptions**, while **handling known exceptions silently or gracefully**.

Skip Logging for Validation Errors	**Log Only Business-Critical Failures**	**Handle External Service Failures**
As a developer, I do not want validation exceptions logged So that logs are not polluted with expected errors. **Use exception filters to bypass logging**	**As a support engineer**, I want insufficient stock or payment failures logged So that I can troubleshoot production issues. **Log only specific exception types**	**As a system**, I must log API or database failures So that system outages are traceable. **Filter exceptions by type and severity**

Step 1: Creating a custom exception class for

1. Validation Exception(Quantity <=0)
2. BusinessRule Exception(Quantity >100)
3. ExternalService Exception(Payment gateway is unavailable)

Step 2: Create a class Logger (This class simply logs any exception exception that is occurring)

```
static void Log(exception ex)
{
 Display Date and time of and type of exception along with message
}
```

Step 3: Creating a class OrderService with method like PlaceOrder(quatity, paymentserviceDown)
this will raise all custom exception

Step 4: Exception Handling with filters

```
try {
  catch(Validation Exception ex)
when(Logifrequired(ex) == false)
{
        cw("validation error handled without logging");

static bool Logifrequired( exception ex)
{
if ( ex is validationExcpetion)
 return false;

logger.log(ex)
return true ;
```

---

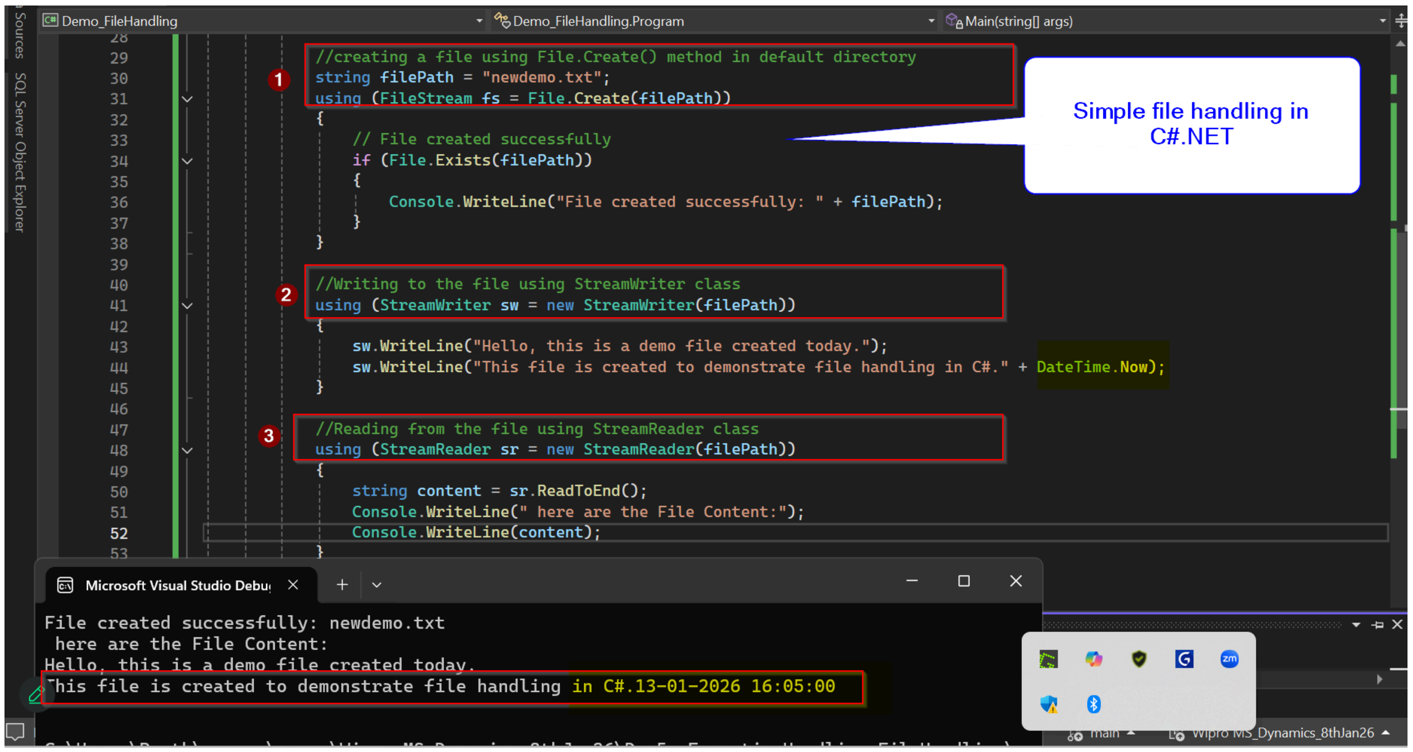
1. Why are exception filters more efficient?
2. When should exception filters be preferred?
3. Can exception filters modify exceptions?
4. Are exception filters supported in async methods?
5. Difference between filtering vs re-throwing?

Serilog : real time Application insight
filters in ASP.NET Core Middleware

File handling in C#.NET :
Why do we need Files when we have DB for Console, web and Cloud based applications ?

1. For creating logs, for caching, storing resources
2. Processing larger dataset ex csv, logs
3. For storing intermediate progress we also use file handling.
4. For storing user preference, user credentials, passwords etc with the help of Cookies on browser

How to implement read/write operation on file in C# ?
Best practices of file handling in C#
Difference between connected and disconnected architecture in Database connectivity ?
five important packages for mastering file and data handling in C# ?

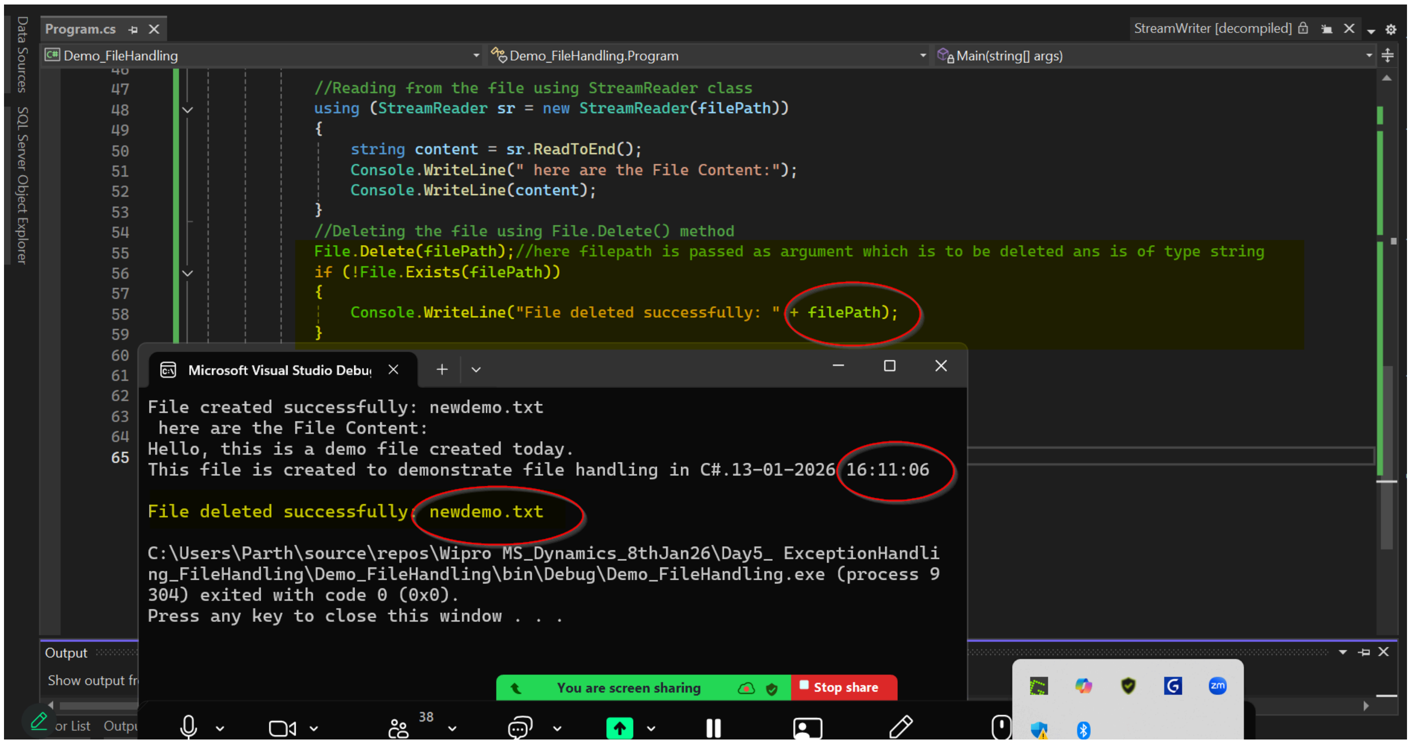

Demo_FileHandling
Demo_FileHandling.Program
Main(string[] args)
Simple file handling in C#.NET
1
2
3
Microsoft Visual Studio Debu

```
            //creating a file using File.Create() method in default directory
            string filePath = "newdemo.txt";
            using (FileStream fs = File.Create(filePath))
            {
                // File created successfully
                if (File.Exists(filePath))
                {
                    Console.WriteLine("File created successfully: " + filePath);
                }
            }

            //Writing to the file using StreamWriter class
            using (StreamWriter sw = new StreamWriter(filePath))
            {
                sw.WriteLine("Hello, this is a demo file created today.");
                sw.WriteLine("This file is created to demonstrate file handling in C#." + DateTime.Now);
            }

            //Reading from the file using StreamReader class
            using (StreamReader sr = new StreamReader(filePath))
            {
                string content = sr.ReadToEnd();
                Console.WriteLine(" here are the File Content:");
                Console.WriteLine(content);
            }
```

```
File created successfully: newdemo.txt
 here are the File Content:
Hello, this is a demo file created today.
This file is created to demonstrate file handling in C#.13-01-2026 16:05:00
```


```
//Reading from the file using StreamReader class
using (StreamReader sr = new StreamReader(filePath))
{
    string content = sr.ReadToEnd();
    Console.WriteLine(" here are the File Content:");
    Console.WriteLine(content);
}
//Deleting the file using File.Delete() method
File.Delete(filePath);//here filepath is passed as argument which is to be deleted ans is of type string
if (!File.Exists(filePath))
{
    Console.WriteLine("File deleted successfully: " + filePath);
}
```

```
File created successfully: newdemo.txt
 here are the File Content:
Hello, this is a demo file created today.
This file is created to demonstrate file handling in C#.13-01-2026 16:11:06

File deleted successfully: newdemo.txt

C:\Users\Parth\source\repos\Wipro MS_Dynamics_8thJan26\Day5_ ExceptionHandli
ng_FileHandling\Demo_FileHandling\bin\Debug\Demo_FileHandling.exe (process 9
304) exited with code 0 (0x0).
Press any key to close this window . . .
```

//in C# we have following types of file handling classes:
//1. StreamReader and StreamWriter: These classes are used for reading and writing text files.
//2. BinaryReader and BinaryWriter: These classes are used for reading and writing binary files.
//3. FileStream: This class is used for reading and writing files as a stream of bytes. example: FileStream fs = new FileStream("file.txt", FileMode.OpenOrCreate);
//4. File: This class provides static methods for creating, copying, deleting, moving, and opening files, and helps in the creation of FileStream objects.
//5. Directory: This class provides static methods for creating, moving, and enumerating through directories and subdirectories.
//6. Path: This class provides methods for working with file and directory path strings.
//These classes provide a comprehensive set of tools for file handling in C#.

//below are the types of modes while working with files in C#:
//read: Opens the file for reading only. An exception is thrown if the file does not exist.
//write: Opens the file for writing only. If the file exists, it is overwritten. If the file does not exist, a new file is created.
//append: Opens the file for writing only. If the file exists, the write operation appends data to the end of the file. If the file does not exist, a new file is created.
//open: Opens the file if it exists. An exception is thrown if the file does not exist.
//openorcreate: Opens the file if it exists; otherwise, a new file is created.
//truncate: Opens the file for writing only and truncates the file to zero bytes. An exception is thrown if the file does

not exist.
//These modes are specified using the FileMode enumeration when creating a FileStream object.