C# Exception Handling - Complete Deep Dive

1. What is an Exception?

An **exception** is an unexpected event or error that occurs during program execution, disrupting the normal flow of the program. When an exception occurs, the program creates an **exception object** containing information about the error and "throws" it.

Key Concepts:

- **Exception**: An object that represents an error condition
- **Throwing**: Creating and raising an exception
- **Catching**: Handling an exception to prevent program crash
- **Exception Propagation**: How exceptions travel up the call stack

Without Exception Handling:

```
int n = 0;
int r = 100 / n;  // Program crashes with DivideByZeroException
Console.WriteLine(r);  // This line never executes
```

What happens:

1. Division by zero occurs
2. CLR (Common Language Runtime) creates a `DivideByZeroException` object
3. Program terminates abruptly
4. User sees an error message
5. Remaining code doesn't execute

---

2. The Exception Hierarchy

All exceptions in C# inherit from `System.Exception` class. Understanding this hierarchy is crucial:

```
System.Object
    └── System.Exception
            ├── SystemException
            │       ├── ArithmeticException
            │       │       ├── DivideByZeroException
            │       │       ├── OverflowException
            │       │       └── NotFiniteNumberException
            │       ├── FormatException
            │       ├── IndexOutOfRangeException
            │       ├── NullReferenceException
            │       ├── InvalidOperationException
            │       └── ArgumentException
            │               ├── ArgumentNullException
            │               └── ArgumentOutOfRangeException
            └── ApplicationException (for custom exceptions)
```

Common Built-in Exceptions:

Exception Type	Description	Example
`DivideByZeroException`	Division by zero	`x / 0`
`FormatException`	Invalid format conversion	`int.Parse("abc")`
`NullReferenceException`	Accessing null object	`string s = null; s.Length`
`IndexOutOfRangeException`	Array index out of bounds	`arr[100]` for small array
`InvalidOperationException`	Invalid operation for current state	Modifying collection while iterating
`ArgumentException`	Invalid argument passed	Negative value where positive expected
`FileNotFoundException`	File doesn't exist	Opening non-existent file
`OverflowException`	Arithmetic overflow	`checked { int.MaxValue + 1 }`

---

3. Try-Catch-Finally Block

The `try-catch-finally` construct is the fundamental mechanism for handling exceptions in C#.

Basic Structure:

```
try
{
    // Code that might throw an exception
}
catch (SpecificException ex)
{
    // Handle specific exception
}
catch (Exception ex)
{
    // Handle any other exception
}
finally
{
    // Always executes (optional)
}
```

The Try Block:

- Contains code that **might** throw an exception
- Must be followed by at least one `catch` or a `finally` block
- Can contain multiple statements
- If an exception occurs, execution immediately jumps to the appropriate catch block

Example from code:

```
try
{
    Console.WriteLine("enter number");
    int n = int.Parse(Console.ReadLine());  // May throw FormatException
    int r = 100 / n;                        // May throw DivideByZeroException
    Console.WriteLine(r);
}
```

---

4. Catch Blocks - Multiple Exception Handling

You can have **multiple catch blocks** to handle different exception types differently. The order matters!

Rules for Multiple Catch Blocks:

1. **Most specific exceptions first**, most general last
2. Only **one catch block** executes per exception
3. If no matching catch is found, exception propagates up the call stack

4. Compiler enforces proper ordering (specific before general)

Example from code:

```csharp
try
{
    Console.WriteLine("enter number");
    int n = int.Parse(Console.ReadLine());
    int r = 100 / n;
    Console.WriteLine(r);
}
catch(FormatException ex)
{
    Console.WriteLine("invalid value");
}
catch(DivideByZeroException ex)
{
    Console.WriteLine("invalid operation: divide by zero");
}
catch(Exception ex)
{
    // Catches all other exceptions
    Console.WriteLine(ex.Message);
}
```

Execution Flow:

1. User enters "abc" → `FormatException` → First catch executes → "invalid value"
2. User enters "0" → `DivideByZeroException` → Second catch executes → "invalid operation: divide by zero"
3. Some other error → Third catch executes (general exception handler)

Wrong Order (Compile Error):

```csharp
catch(Exception ex)        // ❌ This catches everything
{
    // ...
}
catch(FormatException ex)  // ❌ Unreachable code! Compile error
{
    // ...
}
```

5. The Exception Object

When you catch an exception, you receive an **exception object** with valuable information for debugging and logging.

Key Properties:

5.1 Message Property

- Human-readable description of the error
- Most commonly used property
- Set when exception is created

```
catch(Exception ex)
{
    Console.WriteLine(ex.Message);
    // Output: "Input string was not in a correct format."
}
```

5.2 Source Property

- Name of the application or assembly that threw the exception
- Useful in large applications with multiple assemblies
- Can be set manually

```
Console.WriteLine(ex.Source);
// Output: "mscorlib" or your application name
```

5.3 StackTrace Property

- String representation of the call stack when exception occurred
- Shows the sequence of method calls leading to the error
- Critical for debugging
- Performance cost to generate, so only captured when needed

```
Console.WriteLine(ex.StackTrace);
/* Output:
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo
info)
   at Day5PII.Program.Main(String[] args) in C:\...\Program.cs:line 25
*/
```

5.4 InnerException Property

- References the exception that caused this exception
- Creates a chain of exceptions
- Useful when one exception causes another
- Can be null if there's no underlying exception

```
try
{
    try
    {
        int.Parse("abc");
    }
    catch(Exception innerEx)
    {
        throw new InvalidOperationException("Failed to process input",
innerEx);
    }
}
catch(Exception ex)
{
    Console.WriteLine(ex.Message);          // "Failed to process input"
    Console.WriteLine(ex.InnerException);  // FormatException details
}
```

5.5 TargetSite Property

- Returns MethodBase object representing the method that threw exception
- Contains method name, parameters, return type

```
Console.WriteLine(ex.TargetSite.Name);  // Method name like "ParseInt32"
```

5.6 Data Property

- Dictionary for storing additional custom key-value pairs
- Useful for adding context-specific information

```
try
{
    throw new Exception("Error");
}
catch(Exception ex)
{
    ex.Data.Add("UserId", 123);
    ex.Data.Add("Timestamp", DateTime.Now);
    Console.WriteLine(ex.Data["UserId"]);
}
```

Complete Logging Example from Code:

```
catch(Exception ex)
{
    // Comprehensive error logging
    Console.WriteLine(ex.InnerException);
    Console.WriteLine($"{DateTime.Now} \t {ex.Message} \t{ex.Source} \t
{ex.StackTrace}");
}
```

Output Example:

```
1/3/2026 2:30:45 PM    Input string was not in a correct format.    mscorlib
at System.Number.ParseInt32...
```

---

6. The Finally Block

The `finally` block **always executes**, regardless of whether an exception occurred or was handled.

Characteristics:

- Executes after `try` and `catch` blocks
- Runs even if exception is thrown and not caught
- Runs even if `return` statement is in try or catch
- Used for **cleanup operations**
- Optional but highly recommended for resource management

Example from code:

```
try
{
    int n = int.Parse(Console.ReadLine());
    int r = 100 / n;
}
```

```csharp
catch(Exception ex)
{
    Console.WriteLine("Error occurred");
}
finally
{
    Console.WriteLine("finally");  // Always executes
}
```

Execution Scenarios:

Scenario 1: No Exception

```csharp
try
{
    Console.WriteLine("Try");      // ✓ Executes
}
catch
{
    Console.WriteLine("Catch");    // ✗ Skipped
}
finally
{
    Console.WriteLine("Finally");  // ✓ Executes
}
// Output: Try, Finally
```

Scenario 2: Exception Caught

```csharp
try
{
    throw new Exception();
    Console.WriteLine("Try");      // ✗ Not reached
}
catch
{
    Console.WriteLine("Catch");    // ✓ Executes
}
finally
{
    Console.WriteLine("Finally");  // ✓ Executes
}
// Output: Catch, Finally
```

Scenario 3: Exception Not Caught

```
try
{
    throw new InvalidOperationException();
}
catch(FormatException ex)
{
    // Wrong exception type
}
finally
{
    Console.WriteLine("Finally");  // ✓ Still executes!
}
// Output: Finally, then program terminates
```

Scenario 4: Return in Try Block

```
static int TestFinally()
{
    try
    {
        return 1;                    // ✓ Executes
    }
    finally
    {
        Console.WriteLine("Finally");  // ✓ Executes BEFORE return!
    }
}
// Output: Finally, then returns 1
```

Common Use Cases for Finally:

1. Closing Resources

```
FileStream fs = null;
try
{
    fs = new FileStream("file.txt", FileMode.Open);
    // Read file
}
catch(Exception ex)
{
    Console.WriteLine("Error reading file");
}
finally
{
    if(fs != null)
        fs.Close();  // Ensures file is always closed
}
```

2. Releasing Database Connections

```
SqlConnection connection = null;
try
{
    connection = new SqlConnection(connectionString);
    connection.Open();
    // Database operations
}
finally
{
    if(connection != null && connection.State == ConnectionState.Open)
        connection.Close();
}
```

3. Unlocking Resources

```
lock(lockObject)
{
    try
    {
        // Critical section
    }
    finally
    {
        Monitor.Exit(lockObject);  // Ensure lock is released
    }
}
```

4. Resetting State

```
Cursor.Current = Cursors.WaitCursor;
try
{
    // Long operation
}
finally
```

```
{
    Cursor.Current = Cursors.Default;  // Restore cursor
}
```

---

7. Custom Exceptions

C# allows you to create **custom exception classes** for application-specific errors.

Why Create Custom Exceptions?

- Provide **meaningful error information** specific to your domain
- Allow **precise exception handling** by type
- Include **additional properties** relevant to the error
- Improve code **readability and maintainability**
- Follow the **Single Responsibility Principle**

Creating Custom Exceptions:

Basic Structure:

```
class CustomException : Exception
{
    // Custom properties

    // Constructors
    public CustomException() { }

    public CustomException(string message) : base(message) { }

    public CustomException(string message, Exception inner)
        : base(message, inner) { }
}
```

Example from Code: InvalidAgeException

```
class InvalidAgeException : Exception
{
    public int agevalue { get; set; }

    public InvalidAgeException(int age)
        : base("error: invalid age, age must between 20 and 60")
    {
        agevalue = age;
    }
}
```

Breaking Down the Custom Exception:

1. Inheritance

```
class InvalidAgeException : Exception
```

- Inherits from `System.Exception`
- Gets all standard exception functionality
- Can be caught as `Exception` or `InvalidAgeException`

2. Custom Property

```
public int agevalue { get; set; }
```

- Stores the **actual invalid age** that caused the exception
- Provides context for error handling
- Allows caller to retrieve the problematic value

3. Constructor with Base Call

```
public InvalidAgeException(int age)
    : base("error: invalid age, age must between 20 and 60")
{
    agevalue = age;
}
```

- `: base(message)` calls the parent `Exception` constructor
- Sets the `Message` property
- Initializes custom property `agevalue`

---

8. Throwing Exceptions

You can manually throw exceptions using the `throw` keyword.

Basic Throw Syntax:

```
throw new ExceptionType("Error message");
```

Example from Code:

```
class employee
{
    int age;
    public int Age
    {
        set
        {
            if (value > 20 && value < 60)
                age = value;
            else
                throw new InvalidAgeException(value);  // Throwing custom
exception
        }
        get
        {
            return age;
        }
    }
}
```

How It Works:

1. Validation logic checks if age is between 20 and 60
2. If invalid, creates new `InvalidAgeException` object with the invalid age
3. `throw` keyword raises the exception
4. Execution immediately jumps to nearest catch block
5. If no catch block, exception propagates up the call stack

Catching Custom Exception:

```
try
{
    employee em = new employee() { id = 1, name = "ali", Age = 18 };
    // Age is 18, which is < 20, so exception is thrown
}
catch(InvalidAgeException ex)
{
    Console.WriteLine(ex.agevalue);  // Output: 18
    Console.WriteLine(ex.Message);    // Output: "error: invalid age..."
}
```

Different Ways to Throw:

1. Throw New Exception

```
throw new InvalidOperationException("Cannot perform this operation");
```

2. Re-throw Caught Exception

```
try
{
    // Some code
}
catch(Exception ex)
{
    // Log the error
    Console.WriteLine(ex.Message);

    throw;  // Re-throws the same exception, preserving stack trace
}
```

3. Wrap Exception

```
try
{
    // Some code
}
catch(Exception ex)
{
    throw new CustomException("Higher level error",ex);// ex becomes
InnerException
}
```

4. Conditional Throw

```
if (amount < 0)
    throw new ArgumentException("Amount cannot be negative", nameof(amount));
```

9. Property Validation with Exceptions

The code demonstrates a common pattern: **property validation with custom exceptions**.

Full Property Implementation:

```
int age;  // Private backing field

public int Age
{
    set
    {
        if (value > 20 && value < 60)
            age = value;  // Valid: set the backing field
        else
            throw new InvalidAgeException(value);  // Invalid: throw exception
    }
    get
    {
        return age;  // Return the backing field
    }
}
```

Why This Pattern?

1. Encapsulation

- Private backing field protects data
- Public property provides controlled access
- Validation logic is centralized in one place

2. Data Integrity

- Ensures invalid data never enters the object
- Maintains business rules automatically
- No need to validate elsewhere in code

3. Clear Error Reporting

- Exception clearly indicates what went wrong
- Provides the problematic value for debugging
- Caller can handle or log appropriately

Alternative Approaches:

Using Auto-Property with Validation Method:

```
private int age;
public int Age
{
    get => age;
    set => age = ValidateAge(value);
}

private int ValidateAge(int value)
{
    if (value <= 20 || value >= 60)
        throw new InvalidAgeException(value);
    return value;
}
```

Using Expression-Bodied Members (C# 7.0+):

```
private int age;
public int Age
{
    get => age;
    set => age = value > 20 && value < 60
        ? value
        : throw new InvalidAgeException(value);
}
```

Using Guard Clauses:

```
public int Age
{
    set
    {
        if (value <= 20)
            throw new InvalidAgeException(value);
        if (value >= 60)
            throw new InvalidAgeException(value);

        age = value;
    }
    get => age;
}
```

10. Exception Handling Best Practices

10.1 Catch Specific Exceptions

❌ **Bad:**

```
try
{
    // Code
}
catch(Exception ex)  // Too broad
{
    Console.WriteLine("Error");
}
```

✅ **Good:**

```
try
{
    // Code
}
catch(FormatException ex)
{
    Console.WriteLine("Invalid format");
}
catch(DivideByZeroException ex)
{
    Console.WriteLine("Division by zero");
}
catch(Exception ex)  // General catch as fallback
{
    Console.WriteLine("Unexpected error");
}
```

10.2 Don't Swallow Exceptions

❌ **Bad:**

```
try
{
    // Code
}
catch(Exception ex)
{
    // Silent failure - very dangerous!
}
```

✅ **Good:**

```
try
{
    // Code
}
catch(Exception ex)
{
    // At minimum: log the error
    Logger.LogError(ex);

    // Optionally: re-throw or handle
    throw;
}
```

10.3 Use Finally for Cleanup

❌ **Bad:**

```
FileStream fs = new FileStream("file.txt", FileMode.Open);
try
{
    // Use file
}
catch(Exception ex)
{
    fs.Close();  // Only closes on exception
}
fs.Close();  // What if exception occurs? This won't run!
```

✅ **Good:**

```
FileStream fs = new FileStream("file.txt", FileMode.Open);
try
{
    // Use file
}
finally
{
    fs.Close();  // Always closes
}
```

✅ **Better: Use 'using' statement:**

```
using(FileStream fs = new FileStream("file.txt", FileMode.Open))
{
    // Use file
}  // Automatically calls Dispose() which closes the file
```

10.4 Provide Context in Custom Exceptions

❌ **Bad:**

```
throw new InvalidAgeException();  // No information
```

✅ **Good:**

```
throw new InvalidAgeException(age);  // Provides the invalid value
```

10.5 Don't Use Exceptions for Flow Control

❌ **Bad:**

```
try
{
    int value = int.Parse(input);
}
catch(FormatException)
{
    value = 0;  // Using exception as normal flow
}
```

✅ **Good:**

```
if(int.TryParse(input, out int value))
{
    // Use value
}
else
{
    value = 0;  // Handle invalid input normally
}
```

10.6 Document Exceptions in XML Comments

```
/// <summary>
/// Sets the employee's age
/// </summary>
/// <exception cref="InvalidAgeException">
/// Thrown when age is not between 20 and 60
/// </exception>
public int Age { get; set; }
```

10.7 Performance Considerations

- Exceptions are **expensive** (stack trace generation, unwinding)
- Use for **exceptional conditions**, not expected failures
- Prefer validation methods like `TryParse()` for expected failures
- Consider the 80/20 rule: if error happens > 20% of time, don't use exceptions

---

11. Exception Propagation

When an exception is thrown, it travels up the **call stack** until it finds a matching catch block.

Example:

```
void MethodA()
{
    try
    {
        MethodB();
    }
    catch(Exception ex)
    {
        Console.WriteLine("Caught in MethodA");
    }
}

void MethodB()
{
    MethodC();  // No try-catch, exception propagates up
}

void MethodC()
{
    throw new Exception("Error in MethodC");
}

// Call flow:
// MethodA() → MethodB() → MethodC() → Exception!
// Exception travels: MethodC → MethodB → caught in MethodA
```

Unhandled Exceptions:

If no catch block is found in the entire call stack:

1. Exception reaches the top level
2. Application terminates
3. Error details are displayed/logged
4. In ASP.NET, error page is shown

---

12. Advanced Exception Handling Patterns

12.1 Exception Filters (C# 6.0+)

```
try
{
    // Code
}
catch(Exception ex) when (ex.Message.Contains("timeout"))
{
    // Only catches exceptions with "timeout" in message
}
catch(Exception ex) when (logLevel > 0)
{
    // Conditional catching based on state
}
```

12.2 Throw Expressions (C# 7.0+)

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value));
}

// In null-coalescing
string result = input ?? throw new ArgumentNullException(nameof(input));

// In ternary operator
int result = value >= 0 ? value : throw new ArgumentException("Must be
positive");
```

12.3 Multiple Exception Types (C# 7.1+)

```
try
{
    // Code
}
catch(FormatException)
catch(OverflowException)
{
    // Same handling for both exceptions
    Console.WriteLine("Invalid number format");
}
```

---

13. When NOT to Use Exceptions

Use Validation Instead:

```
// ❌ Using exceptions
public void ProcessOrder(Order order)
{
    if(order == null)
        throw new ArgumentNullException(nameof(order));
}

// ✅ Better: defensive programming
public void ProcessOrder(Order order)
{
    if(order == null)
        return;  // Or handle gracefully
}
```

Use TryParse Pattern:

```
// ❌ Exception-based
try
{
    int number = int.Parse(input);
}
catch(FormatException)
{
    // Handle
}

// ✅ Better: TryParse
if(int.TryParse(input, out int number))
{
    // Use number
}
else
{
    // Handle invalid input
}
```

Key Takeaways Summary

1. **Exceptions** represent runtime errors that disrupt normal program flow
2. **Try-Catch-Finally**: Fundamental error handling structure
3. **Multiple Catch Blocks**: Handle different exceptions differently (specific first!)
4. **Exception Object**: Contains Message, StackTrace, Source, InnerException
5. **Finally Block**: Always executes, perfect for cleanup
6. **Custom Exceptions**: Create domain-specific exceptions with additional context
7. **Throwing Exceptions**: Use `throw` keyword for validation and error conditions
8. **Property Validation**: Common pattern for enforcing business rules
9. **Best Practices**: Catch specific exceptions, don't swallow, use finally, document
10. **Performance**: Exceptions are expensive, use for exceptional cases only

Understanding exception handling is crucial for building **robust**, **maintainable**, and **production-ready** applications!

By Abdullah Ali

Contact : +201012613453