.NET

Exception Handling

1. Introduction

- Exception handling has been a fundamental part of C#, providing a structured way to handle runtime errors.
- The .NET exception model is based on a hierarchy of exception classes, all deriving from System.Exception.

2. Core Exception Handling Constructs

2.1 try-catch Blocks

- The fundamental structure for catching exceptions:

```
try {
    // Code that might throw exceptions
    File.ReadAllText("nonexistent.txt");
}
catch (FileNotFoundException ex) {
    Console.WriteLine($"File not found: {ex.FileName}");
}
catch (Exception ex) {
    Console.WriteLine($"General error: {ex.Message}");
}
```

2.2 finally Block

- Guarantees execution for cleanup:

```
FileStream file = null;
try {
    file = File.Open("data.txt", FileMode.Open);
    // Process file
}
finally {
    file?.Dispose(); // Always executes
}
```

2.3 using Statement

- Simplified resource cleanup (implements IDisposable):

```
using (var file = File.Open("data.txt", FileMode.Open)) {
    // Automatically disposed when block exits
}
```

3. Exception Types Hierarchy

3.1 Common Exception Types

```
// System Exceptions
catch (ArgumentNullException ex) { ... }
catch (ArgumentException ex) { ... }
catch (InvalidOperationException ex) { ... }

// IO Exceptions
catch (FileNotFoundException ex) { ... }
catch (DirectoryNotFoundException ex) { ... }
```

```
// Custom Exceptions
catch (MyCustomException ex) { ... }
```

```
System.Object
  └── System.Exception
      ├── System.SystemException
      │   ├── System.NullReferenceException
      │   ├── System.ArithmeticException
      │   │   └── System.DivideByZeroException
      │   ├── System.ArgumentException
      │   │   ├── System.ArgumentNullException
      │   │   └── System.ArgumentOutOfRangeException
      │   ├── System.InvalidOperationException
      │   ├── System.IndexOutOfRangeException
      │   ├── System.IO.IOException
      │   │   └── System.IO.FileNotFoundException
      │   └── System.Threading.ThreadAbortException
      └── System.ApplicationException
          └── MyCustomException
```

3.2 Creating Custom Exceptions

```
public class InventoryException : ApplicationException {
    public int ItemId { get; }
    public InventoryException(int itemId, string message) : base(message) {
        ItemId = itemId;
    }
}
```

```
public class InventoryManagement {
  // fields
  public InventoryItem GetItemDetails(int itemId) {
    // ...
    if(notFound)
      throw new InventoryException(itemId, "Item is not in stock");
    // ...
  }
}
```

```
// Usage
try {
  InventoryManagement im = new InventoryManagement();
  // ...
  InventoryItem item = im.GetItemDetails(id);
  // ...
}
catch(InventoryException e) {
  Console.WriteLine(e.ToString());
}
```

4. Advanced Exception Handling

4.1 Exception Filters (C# 6)

- Conditional catch blocks:

```
try { ... }
catch (HttpRequestException ex) when (ex.StatusCode == 404) {
```

```
    Console.WriteLine("Resource not found");
}
```

4.2 AggregateException

- Handling multiple exceptions:

```
try {
    Parallel.ForEach(items, ProcessItem);
}
catch (AggregateException ae) {
    foreach (var ex in ae.InnerExceptions) {
        Console.WriteLine(ex.Message);
    }
}
```

5. Best Practices

5.1 Do's

```
// Catch specific exceptions
catch (SqlException ex) { ... }

// Provide meaningful messages
throw new InvalidOperationException("Connection pool exhausted");

// Log complete exception details
logger.LogError(ex, "Processing failed for {ItemId}", itemId);
```

5.2 Don'ts

```
// Avoid empty catch blocks
catch (Exception) { }

// Don't catch without action
catch (Exception ex) {
    throw; // Just rethrowing is usually pointless
}

// Avoid exception-based flow control -- This is not good practice
try {
    using (StreamReader reader = new StreamReader("your_file.txt")) {
        while (true) {
            string line = reader.ReadLine();
            if (line == null) // Alternative check for end of stream
                break;
            // Process the line
        }
    }
}
catch (EndOfStreamException ex) {
    // Handle the end of stream condition
    Console.WriteLine("End of stream reached: " + ex.Message);
}
```

6. Performance: Exception Costs

- Stack trace generation is expensive
- Avoid exceptions in normal flow
- Use Try-pattern for expected cases:

```
if (int.TryParse(input, out var number)) {
    // Success case
}
else {
    // Handle invalid input
}
```

MSDN References

- Exception Handling
- Best Practices
- Creating Custom Exceptions

Resource Management & Garbage Collection

1. Introduction

- The .NET runtime provides automatic memory management through its garbage collector (GC). The CLR memory management system handles:
  - Allocation of objects on the managed heap
  - Reclamation of unused memory
  - Compaction of surviving objects

2. Garbage Collection Fundamentals

2.1 Generational Collection

- The heap is divided into generations:
  - **Generation 0**: Short-lived objects (most collections occur here)
  - **Generation 1**: Buffer between short-lived and long-lived objects
  - **Generation 2**: Long-lived objects
  - **Large Object Heap (LOH)**: Objects > 85KB (collected less frequently)

2.2 Collection Triggers

- GC runs when:
    1. Generation 0 reaches its budget/threshold
    2. System memory is low
    3. AppDomain is unloading
    4. Explicitly called via `GC.Collect()`

2.3 Garbage Collection Process

1. Marking Phase: The GC identifies live objects by traversing the object graph starting from roots (like global variables and stack references). It marks reachable objects as "live" and identifies unreachable objects as candidates for collection.
2. Relocating Phase (optional): If the GC determines there's significant fragmentation in the heap, it updates references of live objects to new compacted locations before the compacting phase.
3. Compacting Phase: The GC reclaims the memory occupied by dead objects and compacts the remaining live objects into a contiguous block of memory. This reduces fragmentation and improves memory locality.

2.4 GC Flavors

- **Workstation GC**: Optimized for UI apps (lower latency)
- **Server GC**: Optimized for throughput (multiple CPU cores)

3. Deterministic Resource Cleanup

3.1 IDisposable Interface

- It's an interface in the System namespace.

```
public interface IDisposable {
    void Dispose();
}
```

- It contains a single method: Dispose().
- Implementing IDisposable signals that a class holds resources that need explicit cleanup.

3.2 Standard Disposable Pattern

```
public class Resource : IDisposable {
    private bool _disposed = false;
    public void Dispose()
    {
        // Dispose managed resources
        _disposed = true;
        GC.SuppressFinalize(this);
    }
    ~Resource() {
      if(!_disposed)
        Dispose();
    }
}
```

3.3 Usage Patterns

```
// Explicit disposal
var resource = new Resource();
try { /* use resource */ }
finally { resource.Dispose(); }
```

```
// Using statement (preferred)
using (var resource = new Resource()) {
    // Automatic disposal
}
```

4. Finalization (Destructors)

4.1 Finalizer Syntax

```
public class Resource {
    ~Resource() { // Finalizer
        // Cleanup unmanaged resources
    }
}
```

4.2 Finalization Process

1. Object becomes unreachable
2. GC queues object for finalization
3. Finalizer thread calls finalizer
4. Memory reclaimed in next GC cycle

4.3 Performance Impact

- Finalizable objects survive first collection
- Require additional GC work
- Add latency to cleanup process

5. Best Practices

5.1 Resource Management

1. Implement `IDisposable` for types holding:
    - File handles

    - Database connections
    - Native resources
2. Prefer `using` over manual `try-finally`
3. Avoid finalizers unless absolutely necessary

5.2 GC Optimization

1. Minimize large object allocations
2. Consider object pooling for frequent allocations
3. Use `struct`s for small, short-lived data
4. Avoid unnecessary references

5.3 Anti-Patterns

1. Calling `GC.Collect()` explicitly
2. Empty finalizers
3. Resurrecting (reuse/reassign) objects in finalizers
4. Overusing object pinning (`fixed` keyword)

MSDN References

- Garbage Collection
- IDisposable Pattern
- Memory Management

Indexers and Iterators

1. Indexers

Concept

- Indexers allow objects to be indexed like arrays, providing a way to access elements using the `[]` notation. They are essentially properties named "this" that take parameters.

Example

```
// consider a user defined stack of strings
public class DemoStack {
    // ...
    public string this[int index] {
        get {
            if (index < 0 || index > _top)
                throw new IndexOutOfRangeException();
            return _items[index];
        }
        set {
            if (index < 0 || index > _top)
                throw new IndexOutOfRangeException();
            _items[index] = value;
        }
    }
}
```

Usage

```
var stack = new DemoStack<string>(5);
stack.Push("First");
stack.Push("Second");
Console.WriteLine(stack[0]); // Accesses "First"
stack[1] = "Updated";        // Modifies "Second"
```

2. Iterators

Concept

- Iterators provide a way to traverse collections using `foreach` by implementing `IEnumerable` or `IEnumerable<T>`. The `yield` keyword simplifies iterator implementation.

Implementation Example

```
public class DemoStack : IEnumerable {
    // ...
    public IEnumerator GetEnumerator() {
        for (int i = _top; i >= 0; i--) {
            yield return _items[i]; // Lazy evaluation
        }
    }
}
```

Usage

```
foreach (var item in stack) {
    Console.WriteLine(item); // Prints from top to bottom
}
```

3. Key Features

Indexer Characteristics

1. Can be overloaded with different parameter types
2. Can have multiple dimensions (`[x,y]`)
3. Can be defined in interfaces
4. Support get-only or set-only accessors

Iterator Characteristics

1. `yield return` provides lazy evaluation
2. Maintains state between iterations
3. Can implement complex traversal logic
4. Compiler generates state machine

4. Best Practices

1. **Indexers**:
    - Validate index parameters
    - Consider performance for large collections
    - Document expected behavior
2. **Iterators**:
    - Prefer `IEnumerable<T>` over `IEnumerable`
    - Avoid modifying collections during iteration
    - Consider thread safety

MSDN References

- Indexers (C#)
- Iterators (C#)
- yield (C# Reference)

Interfaces Advanced Features

1. Default Interface Methods (C# 8.0+)

Introduction

- Traditionally purely abstract, interfaces gained implementation capability in C# 8.0 (.NET Core 3.0, 2019) to support API evolution without breaking changes.

Key Characteristics

1. **Backward Compatibility**: Add functionality without forcing implementation
2. **Multiple Inheritance**: Classes can inherit multiple default implementations
3. **Explicit Invocation**: Default methods must be called through interface reference

```
public interface ILogger {
    void Log(string message); // Traditional abstract method

    // Default implementation
    void LogError(string error) {
      Log($"ERROR: {error}");
    }
}

class ConsoleLogger : ILogger
{
    public void Log(string message) => Console.WriteLine(message);
    // No need to implement LogError
}

// Usage:
ILogger logger = new ConsoleLogger();
logger.LogError("Something failed"); // Uses default implementation
```

2. Static Abstract Members (C# 11+)

Generic Math Support

- Enables abstract static methods in interfaces, primarily for numeric scenarios:

```
public interface IAddable<T> where T : IAddable<T>
{
    static abstract T operator +(T left, T right);
    static virtual T Zero => default;
}

public struct Point : IAddable<Point>
{
    public int X, Y;

    public static Point operator +(Point left, Point right) =>
        new Point { X = left.X + right.X, Y = left.Y + right.Y };

    public static Point Zero => new Point();
}
```

Key Benefits

1. **Type-Safe Operator Overloading**
2. **Generic Algorithm Support**
3. **Numerics Without Boxing**

3. Version-Resilient Interfaces

1. Add new methods as defaults
2. Never remove existing methods
3. Use extension methods for utility functions

4. Interface Full Member Support

- Interfaces can declare all member types:

```
public interface IObservable {
    event EventHandler Changed;
    string Name { get; set; }
    int Id { get; }
}
```

- All these must be implemented in derived classes.

5. Diamond Inheritance Resolution

- Default methods handle multiple inheritance cases:
- Example1:

```
class Intf1 {
  public void Fun() {
    Console.WriteLine("Intf1.Fun() called.");
  }
}
class Intf2 {
  public void Fun() {
    Console.WriteLine("Intf2.Fun() called.");
  }
}
class MyClass : Intf1, Intf2 {
}

// Usage : e.g. in Main()
Intf1 obj1 = new MyClass();
obj1.Fun(); // Intf1.Fun() called.

MyClass obj = new MyClass();
obj.Fun(); // Compiler error
```

- Example2:

```
interface IA { void M() => Console.Write("A"); }
interface IB : IA { void IA.M() => Console.Write("B"); }
interface IC : IA { void IA.M() => Console.Write("C"); }
class D : IB, IC
{
    // Must provide implementation to resolve ambiguity - Otherwise compiler error
    void IA.M() => Console.Write("D");
}
```

5. Best Practices

1. **Small, Focused Interfaces** (ISP)
2. **Default Methods for Backward Compatibility**
3. **Explicit Implementation for Clarity**
4. **Avoid State in Interfaces**
5. **Prefer Abstract Classes for Common Implementation**

MSDN References

- Default Interface Methods
- Static Abstract Members
- Interface Design Guidelines

`dynamic` Keyword

1. Introduction

- Introduced in **.NET Framework 4.0 (2010)** alongside C# 4.0 to simplify interoperability with **dynamic languages** (e.g., Python, JavaScript) and **COM objects**.

- Before `dynamic`, developers used **reflection** or explicit casting, which was verbose and error-prone.
- The `dynamic` keyword enables **late binding**, where type resolution happens at **runtime** instead of compile-time.
- Part of the **Dynamic Language Runtime (DLR)**, which sits atop the CLR to support dynamic operations.

2. Key Concepts & Definitions

- **Dynamic Typing**: The type of a `dynamic` variable is resolved at runtime, bypassing compile-time checks.
- **RuntimeBinder**: The component that resolves member invocations on `dynamic` objects (throws `RuntimeBinderException` on failures).
- **IDynamicMetaObjectProvider**: Interface used for custom dynamic behavior (e.g., `ExpandoObject`, `DynamicObject`).
- **DLR (Dynamic Language Runtime)**: A layer that provides services for dynamic languages in .NET (e.g., caching call sites).
- **Use Cases**:
  - COM interop (e.g., Microsoft Office automation).
  - Consuming REST APIs with unpredictable schemas.
  - Dynamic data structures (e.g., `ExpandoObject`).

3. Advantages

- **Flexibility**: Simplifies interaction with weakly-typed systems (e.g., JSON, COM).
- **Reduces Boilerplate**: Avoids complex reflection code.
- **Improves Readability**: Cleaner syntax for dynamic operations.

4. Disadvantages

- **No Compile-Time Safety**: Errors (e.g., missing methods) only surface at runtime.
- **Performance Overhead**: Dynamic dispatch is slower than static typing due to runtime resolution.
- **Tooling Limitations**: IDE features like IntelliSense don't work for `dynamic` types.

5. How `dynamic` Differs from `var` and `object`

- `var`: Still **statically typed** (type inferred at compile-time).
- `object`: Requires explicit casting and offers no dynamic dispatch.
- `dynamic`: Defers all type checks to runtime.

6. Examples

```
// Example 1: Basic dynamic usage
dynamic obj = GetExternalData(); // Could be JSON/COM
Console.WriteLine(obj.Name); // Resolved at runtime
```

```
// Example 2: ExpandoObject (dynamic dictionary)
dynamic person = new ExpandoObject();
person.Name = "Alice";
person.Age = 30;
```

```
// Example 3: COM Interop (e.g., Excel)
dynamic excel = Microsoft.Office.Interop.Excel;
excel.Application app = new excel.Application();
```

7. RuntimeBinderException

- Thrown when a member (method/property) doesn't exist at runtime.
- Example:

```
dynamic value = "hello";
value.Foo(); // RuntimeBinderException: 'string' does not contain 'Foo'
```

8. Performance Considerations

- Avoid `dynamic` in performance-critical paths (e.g., tight loops).

- Cache dynamic calls if repeated (e.g., via `Func<dynamic, object>` delegates).

MSDN References

- dynamic (C# Reference)
- DynamicObject Class
- DLR Overview

Reflection

1. Introduction

- Reflection provides the ability to inspect and interact with type information at runtime. This powerful feature enables:
  - Dynamic type discovery
  - Late binding
  - Runtime code analysis
  - Self-modifying applications

2. Core Concepts

2.1 Type Metadata

- Type metadata is binary information that describes your type (class, struct, enum, ...). It's stored alongside the compiled code (IL) in assemblies.
- **Type Metadata**
  - Name: The fully qualified name of the type.
  - Visibility: Whether the type is public, private, etc.
  - Base Class: The type that the current type inherits from.
  - Interfaces: Interfaces implemented by the type.
  - Members: Information about the methods, fields, properties, events, and nested types within the type.
  - Attributes/Flags: Additional descriptive information about the type e.g. IsAbstract, IsSealed, etc.
- **Member Metadata**
  - Name: The name of the member (e.g., method name, field name).

  - Type: The data type of the member (e.g., int, string, a custom type).
  - Visibility: Whether the member is public, private, etc.
  - Attributes: Additional descriptive information about the member.
- This metadata is loaded at runtime (when assembly is loaded).

2.2 Type Discovery

- The `System.Type` object holds type metadata.
- It can be accessed in one of the following ways:

```
Type classType = typeof(ClassName);           // Compile-time known type
Type objType = someObject.GetType();          // Runtime type
Type typeByName = Type.GetType("System.Int32"); // By name
```

2.3 Assembly Inspection

- Information from current assembly can be accessed as follows:

```
Assembly assembly = Assembly.GetExecutingAssembly();
foreach (Type type in assembly.GetTypes())
    Console.WriteLine(type.FullName);
```

- An assembly can be loaded explicitly as follows:

```
Assembly assembly = Assembly.LoadFrom(@"AssemblyPath");
foreach (Type type in assembly.GetTypes())
    Console.WriteLine(type.FullName);
```

3. Member Inspection

3.1 Exploring Members

```
Type type = typeof(MyClass);

// Get all public methods
MethodInfo[] methods = type.GetMethods();

// Get specific property
PropertyInfo[] props = type.GetProperties();

// Get all fields
FieldInfo[] field = type.GetFields(BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance |
BindingFlags.Static);

// Get field including non-public
FieldInfo field = type.GetField("_internal", BindingFlags.NonPublic | BindingFlags.Instance);
```

3.2 Dynamic Object Creation

```
object obj = Activator.CreateInstance(type);
```

3.2 Method Invocation

```
// static method invoked without object
MethodInfo method = typeof(Math).GetMethod("Max", new[] { typeof(int), typeof(int) });
object result = method.Invoke(null, new object[] { 5, 10 }); // Static method call
```

```
// non-static methods need object (usually dynamically created) as first arg to method.Invoke()
object result = method.Invoke(obj, new object[] { ... }); // Non-Static method call
```

4. Performance Considerations

4.1 Caching Strategies

```
// Cache expensive reflection operations
private static readonly MethodInfo _toStringMethod = typeof(object)
    .GetMethod("ToString");

// Reuse cached method
string result = (string)_toStringMethod.Invoke(obj, null);
```

4.2 Alternatives

- **Compiled Expressions**: Faster than pure reflection

```
Func<object, string> toString = obj => obj.ToString(); // Faster alternative
```

MSDN References

- Reflection in .NET
- Type Class
- Runtime Type Handling

Unsafe Code and P/Invoke

1. `unsafe` Code Basics

Concept

- The `unsafe` keyword allows pointer operations and direct memory access in C#. Requires compiling with `/unsafe` flag.

Minimal Example

```
unsafe void PointerDemo()
{
    int value = 10;
    int* ptr = &value;  // Pointer declaration
    Console.WriteLine(*ptr); // Dereference pointer
}
```

2. `fixed` Keyword

Concept

- Pins managed objects in memory to prevent GC relocation during pointer operations.

Minimal Example

```
unsafe void FixedDemo()
{
    int[] numbers = { 10, 20, 30 };
    fixed (int* ptr = numbers)
    {
        Console.WriteLine(ptr[1]); // Access array via pointer
    }
}
```

3. P/Invoke (Platform Invoke)

Concept

- Calls native functions from System DLLs. Requires `DllImport` attribute.

Minimal Example

```
using System.Runtime.InteropServices;

class NativeMethods
{
    [DllImport("user32.dll")]
    public static extern int MessageBox(IntPtr hWnd, string text, string caption, int type);
}

// Usage:
NativeMethods.MessageBox(IntPtr.Zero, "Hello", "Message", 0);
```

Key Notes

1. **Requirements**:
    - Enable "Allow unsafe code" in project settings
    - Mark methods with `unsafe` keyword
    - Use `fixed` when pointers reference managed objects
2. **Safety**:
    - These features bypass .NET safety checks
    - Use only when absolutely necessary

- Validate all pointer operations

3. **P/Invoke Tips**:
   - Match native types precisely (`int` vs `Int32`)
   - Consider marshaling for complex types
   - Use `CharSet` in `DllImport` for string handling

MSDN References

- Unsafe Code
- fixed Statement
- P/Invoke

String Class in C#

1. Introduction and Key Characteristics

- **Immutable**: Strings cannot be modified after creation (operations return new strings)
- **Reference Type**: Stored on the heap, but with special optimizations
- **Unicode Support**: UTF-16 encoded (2 bytes per character)
- **String Pool**: Runtime optimization for string literals

2. String Intern Pool

Concept

- Special memory area storing unique string literals
- Reuses identical strings to save memory
- Literals automatically interned at compile time

Intern Methods

```
string s1 = "hello";
string s2 = String.Intern(new StringBuilder().Append("he").Append("llo").ToString());
Console.WriteLine(ReferenceEquals(s1, s2)); // True - same reference

string s3 = String.IsInterned("hello") ?? "not interned"; // Check if interned
```

3. Commonly Used Methods

Basic Operations

```
string text = "Hello World";

// Length property
int len = text.Length; // 11

// Index access (read-only)
char first = text[0]; // 'H'

// Concatenation
string combined = String.Concat("Hello", " ", "World");
```

Searching

```
// Contains
bool hasWorld = text.Contains("World"); // true

// IndexOf
int index = text.IndexOf('W'); // 6
```

```
// StartsWith/EndsWith
bool starts = text.StartsWith("Hello"); // true
```

Modification (Returns New Strings)

```
// Substring
string part = text.Substring(6, 5); // "World"

// Replace
string updated = text.Replace("World", "C#"); // "Hello C#"

// ToUpper/ToLower
string upper = text.ToUpper(); // "HELLO WORLD"

// Trim
string clean = "  text  ".Trim(); // "text"
```

Splitting/Joining

```
// Split
string[] parts = "a,b,c".Split(','); // ["a", "b", "c"]

// Join
string joined = String.Join("-", parts); // "a-b-c"
```

4. String Comparison

```

```

```
// Culture-aware comparison
bool equal = String.Equals("hello", "HELLO", StringComparison.OrdinalIgnoreCase);

// Sorting order
int result = String.Compare("apple", "banana"); // -1
```

5. StringBuilder Introduction

Purpose

- Mutable string buffer for efficient concatenation
- Avoids multiple allocations during string building

Basic Usage

```
StringBuilder sb = new StringBuilder();
sb.Append("Hello");
sb.AppendLine(" World");
sb.AppendFormat("{0} times", 5);

string result = sb.ToString(); // "Hello World\n5 times"
```

Key Features

- **Capacity Management**: Pre-allocate buffer size
- **Chained Methods**: `Append().AppendLine()`
- **Thread Safety**: Not thread-safe by default

6. Performance Considerations

1. **String Pool Benefits**
    - Reduces memory for duplicate literals
    - Fast comparison via reference equality (`ReferenceEquals`)
2. **When to Use StringBuilder**
    - Multiple concatenations in a loop
    - Building large strings incrementally
    - When intermediate string results aren't needed

MSDN References

- String Class
- StringBuilder
- String Interning