.NET

Partial classes

1. Introduction

- A partial class in C# allows you to split the definition of a class, struct, or interface across multiple source files.
- When the application is compiled, these parts are combined into a single class. This feature is useful for organizing large classes, especially when multiple developers are working on the same class, or when dealing with automatically generated code.

2. Key Features

- Splitting Class Definition: A partial class's definition is divided into multiple files, each marked with the partial keyword.
- Compilation: The compiler combines all the partial class definitions into a single class during compilation.
- Same Namespace and Class Name: All parts of a partial class must belong to the same namespace and have the same name.
- Accessibility: All parts of a partial class must have the same accessibility (e.g., public, private, internal).
- Partial Structures and Interfaces: The partial keyword can also be used with structures and interfaces.

3. Benefits

- Organization: Large classes can be broken down into smaller, more manageable files.
- Collaboration: Multiple developers can work on different parts of the same class simultaneously.
- Generated Code: Partial classes are often used with automatically generated code (e.g., in Visual Studio) to add custom logic without modifying the generated code directly.
- Readability: Splitting a class into logical parts can make the code easier to read and understand.

4. Example:

```
// File: Person.cs
public partial class Person {
```

```
  public string FirstName { get; set; }
  public string LastName { get; set; }
}
```

```
// File: Person.Methods.cs
public partial class Person {
  public void DisplayName() {
    Console.WriteLine($"Name: {FirstName} {LastName}");
  }
}
```

```
// Usage:
Person person = new Person();
person.FirstName = "John";
person.LastName = "Doe";
person.DisplayName(); // Output: Name: John Doe
```

MSDN References

- Partial Classes and Members

Partial Methods

1. Introduction

- A partial method is a special type of method that allows you to split its declaration and implementation across multiple parts of a partial class.
- If the implementation is not provided, the method and all calls to it are removed at compile time, making it useful for scenarios like code generation or optional functionality.

2. Key characteristics:

- A partial method has a declaration (signature) in one part of a partial class and an optional definition (implementation) in the same or another part.
- If the implementation is not provided, the method and all calls to it are removed during compilation. This is a key feature for code generation and optional features.
- Partial Class Requirement: Partial methods must be declared within a partial class or struct. Partial methods not possible in interfaces.
- Partial methods must have a void return type.
- Partial methods cannot have access modifiers like public, private, etc.
- They cannot have out parameters

3. Example

```
public partial class MyPartialClass {
  partial void OnNameChanged(string oldName, string newName);
}

public partial class MyPartialClass {
  partial void OnNameChanged(string oldName, string newName) {
    Console.WriteLine($"Name changed from {oldName} to {newName}");
  }
}

public class Example {
  public void ChangeName(string newName) {
    string oldName = "Initial Name";
    // If OnNameChanged is implemented, this line will call the implementation
    // If it's not, this line will be removed.
    OnNameChanged(oldName, newName);
  }
}
```

`static` Keyword

Introduction

- Traditionally, class `static` members represent `shared` members i.e. they are shared among all objects of the class.
- Indicates a member belongs to the type itself rather than to specific instances.
- Static members are allocated memory once when the program starts and exist for the application's lifetime.
- Apart from static members, `static` is also useful in many other cases.
- All possible uses of `static` are given below.

1. Static Fields

- Class-level variables shared across all instances:

```
public class Counter {
    public static int TotalCount; // Shared across all instances
    public Counter() {
        TotalCount++;
    }
}
```

2. Static Methods

- Utility methods that don't require instance data:

```
public class MathUtils {
    public static double CalculateCircleArea(double radius) {
        return Math.PI * radius * radius;
    }
    // ...
}
// Usage: double area = MathUtils.CalculateCircleArea(5);
```

3. Static Constructors

- Run once when the class is first accessed.
- No access specifier (implicitly private).

```
public class ConfigLoader {
    public static readonly string ConnectionString;
    static ConfigLoader() {
        ConnectionString = ConfigurationManager.AppSettings["DBConnection"];
    }
}
```

- App.config

```
<configuration>
<connectionStrings>
    <add name="DBConnection"
        connectionString="Data Source=serverName;Initial Catalog=databaseName;User Id=userName;Password=password;"
        providerName="System.Data.SqlClient" />
</connectionStrings>
</configuration>
```

4. Static Readonly vs Const

- Const keyword represent compile-time constant (replaced by compiler). Cannot be static.
- Readonly keyword represent run-time constant - can be initialized only once - field initializer or constructor.

```
public class AppConstants {
    public const double PI = 3.14159; // Compile-time constant
```

```
    public static readonly DateTime StartupTime = DateTime.Now; // Runtime constant
}
```

5. Static Classes

- Contain only static members and cannot be instantiated.
- Typically used to implement helper/utility classes.

```
public static class StringUtils {
    public static string SwapCase(string value) {
        StringBuilder swapped = new StringBuilder(value.Length);
        foreach (char c in value) {
            if (char.IsUpper(c))
                swapped.Append(char.ToLower(c));
            else if (char.IsLower(c))
                swapped.Append(char.ToUpper(c));
            else
                swapped.Append(c);
        }
        return swapped.ToString();
    }
}

// Usage:
string result = StringUtils.SwapCase("SunBeam InfoTech");
```

6. Extension Methods (C# 3.0+)

- Enable adding methods to existing types without inheritance or modification
- Guidelines:

- Must be defined in a static class
- First parameter uses `this` modifier
- Appear as instance methods on target type

- Example:

```
public static class StringExtensions {
    public static bool IsValidEmail(this string input) {
        return Regex.IsMatch(input, @"^[^@\s]+@[^@\s]+\.[^@\s]+$");
    }
}

// Usage:
string email = "test@example.com";
bool isValid = email.IsValidEmail();
```

7. Static Local Functions (C# 8.0+)

- Prevents accidental capture of enclosing scope variables:

```
public void ProcessData() {
    int baseValue = 10;
    static int AddNumbers(int a, int b) {
        // Cannot access baseValue here
        return a + b;
    }
}
```

8. Operator Overloading

- Define custom behavior for operators on custom types

- Characteristics
  - Must be declared as public static
  - At least one parameter must be containing type
  - Certain operators must be overloaded in pairs (==/!=, </>, etc.)
- Example:

```
public struct Vector {
    public double X, Y;

    public static Vector operator +(Vector a, Vector b) {
        return new Vector { X = a.X + b.X, Y = a.Y + b.Y };
    }

    public static bool operator ==(Vector a, Vector b) {
        return a.X == b.X && a.Y == b.Y;
    }

    public static bool operator !=(Vector a, Vector b) {
        return !(a == b);
    }
}
```

```
// Usage
Vector v1 = new Vector() { X = 3, Y = 2 };
Vector v2 = new Vector() { X = 1, Y = 4 };
Vector v3 = v1 + v2; // invokes overloaded operator+
  // ...
if(v1 == v2)
  Console.WriteLine("Same");
else
  Console.WriteLine("Different");
```

9. Static Abstract Members (C# 11+)

- Enables static polymorphism in interfaces
- Key Features:
  - Interface can require implementing types to provide static members
  - Enables generic math scenarios
  - Supports operators, methods, and properties
- Example:

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

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

    public static Vector operator +(Vector left, Vector right) {
        return new Vector { X = left.X + right.X, Y = left.Y + right.Y };
    }
}
```

10. Static Imports (C# 6.0+)

- Import static members directly into scope
- Highlights:
  - Avoid repetitive class name qualification
  - Works with both types and enums

    - Can lead to naming conflicts if overused

- Example:

```
using static System.Math;
using static System.Console;

double radius = 10;
double area = PI * Pow(radius, 2);
WriteLine($"Area: {area}");
```

11. Module Initializers (C# 9.0+)

- Run code when assembly loads
- Highlights:
    - Marked with `[ModuleInitializer]` attribute
    - Must be static void parameterless method
    - Execution order not guaranteed
- Example:

```
internal static class ModuleInit {
    [ModuleInitializer]
    public static void Initialize() {
        // Runs when assembly loads
        Console.WriteLine("Module initialized");
    }
}
```

12. Singleton Design Pattern

- Ensures a class has only one instance with global access point:
- Example

```
public sealed class Logger {
    private static readonly Logger _instance = new Logger();

    private Logger() {} // Private constructor

    public static Logger Instance {
        get { return _instance; }
    }

    public void Log(string message) {
        Console.WriteLine($"{DateTime.Now}: {message}");
    }
}

// Usage:
Logger.Instance.Log("Application started");
```

Best Practices and Considerations

1. **Extension Methods**
    - Keep in dedicated namespace
    - Follow naming conventions (Extensions suffix)
    - Document behavior clearly
2. **Operator Overloading**
    - Only overload where operation is intuitive
    - Maintain mathematical invariants

- Provide corresponding instance methods

3. **Static Imports**
   - Use sparingly to avoid confusion
   - Prefer for commonly used constants (Math.PI)
   - Avoid with frequently conflicting names
4. **Static Abstracts**
   - Primarily for advanced generic scenarios
   - Consider performance implications
   - Document requirements thoroughly
5. **Memory Allocation**
   - Static members are allocated in a special high-frequency heap
   - Exist for the application's entire lifetime
   - Cannot be garbage collected
6. **Thread Safety**
   - Static fields are shared across threads
   - Requires synchronization for mutable state
   - Prefer immutable static data where possible
7. **Testing Challenges**
   - Static dependencies make unit testing difficult
   - Consider dependency injection for testable code
   - Use interfaces when mocking is needed
8. **static: Appropriate Use Cases**

- Utility methods that don't need instance data
- Shared configuration values

- Factory methods
- Extension methods

9. **Anti-Patterns to Avoid**

- Overusing static for state management
- Creating "god classes" with many static methods
- Using static as a shortcut to avoid proper DI

MSDN References

- Static (C# Reference)
- Static Classes and Members
- Extension Methods
- Operator Overloading
- Static Imports
- Static Abstract Members
- Singleton Implementation

Nullable Types

1. Introduction

1.1 Definition

- Added in C# 2.0.
- Primarily designed for **value types** (e.g., `int`, `DateTime`) to hold `null`.
- The **reference types** can be null. C# 8.0+, reference types can be explicitly defined with the ? suffix.
- Useful for:
  - Database fields (where values can be `NULL`).
  - Optional parameters.

1.2 Syntax

```
int? nullableInt = null;                // Shorthand (preferred)
Nullable<double> nullableDouble = 3.14;  // Full syntax
```

1.3 Underlying Representation

- The `Nullable<T>` struct wraps value types:

```
public struct Nullable<T> where T : struct
{
    public T Value { get; }
    public bool HasValue { get; }
}
```

2. Checking for `null`

2.1 `HasValue` Property

- HasValue is true/false indicate value is present/absent.

```
int? age = 25;
if (age.HasValue)
    Console.WriteLine($"Age: {age.Value}");
```

2.2 `GetValueOrDefault()`

- Returns the value or a default (avoiding `InvalidOperationException`).

```
int? count = null;
int safeCount = count.GetValueOrDefault();  // 0
int customDefault = count.GetValueOrDefault(100);  // 100
```

3. Null-Coalescing Operator (??)

- Provides a fallback value if the left-hand side is null.

```
int? userId = null;
int actualId = userId ?? -1;  // -1
```

- ?? can be chained for multiple values.

```
string name = GetName();
string showName = GetDisplayName();
string displayName = name ?? showName ?? "Anonymous";
```

4. Null-Conditional Operator (?.)

4.1 Safe Member Access

- Short-circuits to null if the object is null.

```
Person person = null;
int? length = person?.Name?.Length;  // null (no exception)
```

4.2 Combining with Null-Coalescing

```
int nameLength = person?.Name?.Length ?? 0;
```

5. Null-Forgiving Operator (`!`)

- Suppresses nullable warnings (use cautiously!).

```
string name = null!;  // Assert: "I know this is null"
```

- Common Usage

```
public string Process(string? input) {
    return input!.ToUpper();  // Trust developer's judgment
}
```

---

6. Casting Nullable Types

6.1 Explicit Cast (Throws if `null`)

```
int? nullableNum = 42;
int num = (int)nullableNum;  // Works
```

```
int? badNum = null;
int crash = (int)badNum;     // ❌ InvalidOperationException
```

6.2 `as` Operator

```
object obj = "Hello";
string? str = obj as string;  // Safe (returns null if fails)
```

```
object obj = null;
int? num = obj as int?;     // num will be null obj is null - No exception
```

7. Arithmetic with Nullables

- Operations return `null` if any operand is `null`.

```
int? a = 10;
int? b = null;
int? sum = a + b;  // null
```

8. Best Practices

✔ **Use `?.` and `??`** to avoid `NullReferenceException`.
✔ **Prefer `GetValueOrDefault()`** over direct `.Value` access.
✔ **Limit `!` operator** (only when certain of non-null).

9. MSDN References

- Nullable Value Types
- Null-Conditional Operator

---

var and Anonymous Types

1. `var` Keyword (C# 3.0+)

Introduction

- Introduced in C# 3.0 (2007) as part of LINQ, `var` enables implicit typing where the compiler determines the variable type at compile-time. It does not create dynamically typed variables - C# remains statically typed.

Key Characteristics

1. **Type Inference**: Compiler determines type from initialization expression
2. **Static Typing**: Still enforces type safety at compile time
3. **Local Variables Only**: Cannot be used for fields, parameters, or return types
4. **Requires Initialization**: Must declare and initialize in one statement

Valid Use Cases

```
var numbers = new List<int>(); // Clear type from initialization
var name = "John Doe";         // Obvious string type
var result = Calculate();      // When type is evident from method call
```

Invalid Use Cases

```
var value;                     // Error: must be initialized
public var Id { get; set; }    // Error: can't use for properties
```

```
var x = null;                   // Error: can't infer type
```

Best Practices

1. Use when type is obvious from right-hand side
2. Avoid when type isn't immediately clear
3. Prefer explicit types for public API signatures
4. Consider readability in team environments

2. Anonymous Types (C# 3.0+)

Definition and Purpose

- Anonymous types are compiler-generated, immutable reference types typically used in LINQ queries for temporary results. They contain read-only properties inferred from initialization.

Key Features

1. **Compiler-Generated**: Created at compile time
2. **Immutable**: Properties are read-only
3. **Reference Type**: Allocated on heap despite value-type syntax
4. **Limited Scope**: Primarily for local use in methods

Creation Syntax

```
var person = new { Name = "Alice", Age = 30 };
var product = new { ID = 1001, Price = 19.99m };
```

Type Characteristics

1. **Property Inference**: Names and types from initialization
2. **Equals/GetHashCode**: Overridden for value equality
3. **ToString**: Generates property listing

Common LINQ Usage

```
var query = from p in products
            select new { p.Name, DiscountedPrice = p.Price * 0.9 };
```

Advanced Scenarios

1. **Nested Anonymous Types**

```
var order = new {
    ID = 1001,
    Customer = new { Name = "Bob", Email = "bob@example.com" }
};
```

2. **Array of Anonymous Types**

```
var people = new[] {
    new { Name = "Alice", Age = 30 },
    new { Name = "Bob", Age = 25 }
};
```

3. Combined Usage in LINQ

The `var` keyword becomes essential when working with anonymous types from LINQ queries:

```
var results = from e in employees
              where e.Salary > 50000
              select new { e.Name, e.Department };
```

4. Important Limitations

1. **Anonymous Type Limitations**
   - Cannot add methods/events
   - Only contains read-only properties
   - Cannot be passed as parameters (without using `dynamic`)
2. `var` **Limitations**
   - Not the same as JavaScript `var` (still static typing)
   - Doesn't work with `dynamic` types
   - Can't be used in explicit interface implementation

5. Performance Considerations

1. **Anonymous Types**
   - No runtime performance penalty
   - Generated class has optimized Equals/GetHashCode
   - Garbage collected like regular reference types
2. `var` **Keyword**
   - Zero runtime impact (compile-time only)
   - Doesn't affect generated IL
   - No memory or CPU overhead

MSDN References

- var Keyword
- Anonymous Types
- Type Inference

LINQ (Language Integrated Query)

1. Introduction to LINQ

Historical Context

- Introduced in C# 3.0 (.NET Framework 3.5, 2007), LINQ revolutionized data querying in .NET by bringing SQL-like syntax to C#.
- Provides a unified model for querying various data sources including collections, databases, XML, and more.

Key Benefits

1. **Declarative Syntax**: Express what you want, not how to get it
2. **Type Safety**: Compile-time checking of queries
3. **IntelliSense Support**: Full IDE integration
4. **Standardized Patterns**: Consistent across data sources

2. Core LINQ Operators

Filtering Operations

```
var highScores = students.Where(s => s.Score > 90);
var topStudent = students.FirstOrDefault(s => s.Score == 100);
```

Projection Operations

```
var names = students.Select(s => s.Name);
var studentDetails = students.Select(s => new { s.Name, s.Age });
```

Sorting Operations

```
var orderedStudents = students.OrderBy(s => s.Name);
var multiLevelSort = students.OrderBy(s => s.Grade).ThenByDescending(s => s.Score);
```

Grouping Operations

```
var studentsByGrade = students.GroupBy(s => s.Grade);
var gradeAverages = students.GroupBy(s => s.Grade)
                            .Select(g => new { Grade = g.Key, Avg = g.Average(s => s.Score) });
```

Aggregation Operations

```
var totalScore = students.Sum(s => s.Score);
var averageAge = students.Average(s => s.Age);
var maxScore = students.Max(s => s.Score);
```

3. Query Syntax vs Method Syntax

Query Syntax (SQL-like)

```
var results = from s in students
              where s.Score > 80
              orderby s.Name
              select new { s.Name, s.Score };
```

Method Syntax (Lambda-based)

```
var results = students
              .Where(s => s.Score > 80)
              .OrderBy(s => s.Name)
              .Select(s => new { s.Name, s.Score });
```

When to Use Each

1. **Query Syntax**: Better for joins, complex queries
2. **Method Syntax**: Better for method chaining, simple queries
3. **Mixed Syntax**: Can combine both styles

4. Deferred Execution

Lazy Evaluation Concept

- Queries aren't executed until enumerated
- Enables query composition and optimization
- Can lead to multiple enumerations if not careful

Immediate Execution Methods

```
foreach(var s in students)      // enumeration
  System.Console.WriteLine(s);

var list = students.ToList();  // Forces execution
var array = students.ToArray();
var count = students.Count();
```

5. Performance Considerations

Optimization Techniques

1. **Add `.AsParallel()` for CPU-bound operations**
2. **Use `.Any()` instead of `.Count() > 0` for existence checks**
3. **Consider `.ToLookup()` for repeated key-based access**
4. **Pre-size collections when possible with `.ToList(capacity)`**

Common Pitfalls

1. **N+1 queries in nested iterations**
2. **Multiple enumerations of same query**
3. **Unnecessary sorting operations**
4. **Inefficient joins on large collections**

6. Advanced LINQ Patterns

Joins Between Collections

```
var studentCourses = from s in students
                     join c in courses on s.CourseId equals c.Id
                     select new { s.Name, c.CourseName };
```

Partitioning Operations

```
var firstPage = students.Take(20);
var secondPage = students.Skip(20).Take(20);
```

Set Operations

```
var distinctAges = students.Select(s => s.Age).Distinct();
var commonStudents = classA.Intersect(classB);
```

7. Best Practices

Do's

1. **Chain methods properly** (filter first, then project)
2. **Use meaningful variable names** in queries
3. **Consider readability** over cleverness
4. **Profile performance** of complex queries

Don'ts

1. **Don't misuse LINQ** for complex business logic
2. **Avoid deep nesting** of queries
3. **Don't ignore deferred execution** implications
4. **Don't mix LINQ with side effects**

MSDN References

- LINQ Overview

- Standard Query Operators
- LINQ Performance

File and Stream I/O

1. Fundamental Concepts

- The System.IO namespace has been a core part of .NET since version 1.0 (2002), providing comprehensive APIs for file system operations and stream-based I/O.
    - .NET Framework 2.0 (2005): Added FileSystemWatcher and improved serialization
    - .NET Framework 4.0 (2010): Introduced memory-mapped files
    - .NET Core 3.0 (2019): Added high-performance System.IO.Pipelines
    - .NET 6 (2021): Improved async file operations

1.1 Core Namespaces

- `System.IO`: Basic file and directory operations
- `System.Text`: Encoding/decoding text streams
- `System.IO.Compression`: For ZIP/GZIP handling

1.2 Stream Based Architecture

- The .NET I/O system is built on a stream-based model that provides:
    - **Abstraction layer** over various storage mediums
    - **Uniform interface** for sequential byte access
    - **Buffering capabilities** for performance optimization

2. File System Operations

- The File and Directory classes provide static methods for common operations:
    - File: Create, copy, delete, move, and open files
    - Directory: Create, move, and enumerate directories

- Path: Cross-platform path manipulation methods

2.1 File Class (Static Methods)

```
// Basic operations
File.WriteAllText("data.txt", "Hello World\n");
File.AppendAllText("data.txt", "Bye World\n");
string content = File.ReadAllText("data.txt");
bool exists = File.Exists("data.txt");

// Advanced scenarios
File.Copy("source.txt", "dest.txt", overwrite: true);
File.SetAttributes("data.txt", FileAttributes.Hidden);
```

2.2 Directory Management

```
// Directory operations
Directory.CreateDirectory("logs");
var files = Directory.EnumerateFiles("docs", "*.pdf");

// Special folders
string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

3. Stream-Based Operations

3.1 Core Stream Classes

Class	Purpose
`FileStream`	Physical file access

Class	Purpose
`MemoryStream`	In-memory byte storage
`BufferedStream`	Performance optimization
`NetworkStream`	Network communication
`CryptoStream`	For encryption/decryption

3.2 Traditional Synchronous IO

- Traditional blocking operations suitable for:
  - Small files
  - Non-UI applications
  - Simple scripting scenarios

3.3 Write/Read Binary Files

```
// Writing to file
using (FileStream fs = File.Create("data.bin"))
using (BinaryWriter writer = new BinaryWriter(fs))
{
    writer.Write(42);
    writer.Write(3.14);
}

// Reading from file
using (FileStream fs = File.OpenRead("data.bin"))
using (BinaryReader reader = new BinaryReader(fs))
{
    int number = reader.ReadInt32();
    double pi = reader.ReadDouble();
}
```

3.4 Write/Read Text Files

```
// Writing text
using (StreamWriter writer = new StreamWriter("log.txt", append: true))
{
    writer.WriteLine($"{DateTime.Now}: Application started");
}

// Reading lines
using (StreamReader reader = new StreamReader("data.csv"))
{
    while (!reader.EndOfStream)
    {
        string line = reader.ReadLine();
        // Process line
    }
}
```

3.5 Stream Processing Patterns

A. Basic Stream Usage

- The standard disposable pattern for streams ensures:
  - Proper resource cleanup
  - Exception safety
  - Buffer management

B. Buffering Strategies

- Critical for performance optimization:

    - Default buffer sizes (4KB typically)
    - Custom buffering for large files
    - Flush considerations for writers
- Example:

```
using (FileStream fileStream = new FileStream("output.txt", FileMode.Create)) {
  using (BufferedStream bufferedStream = new BufferedStream(fileStream)) {
    using (StreamWriter streamWriter = new StreamWriter(bufferedStream))
    {
        streamWriter.WriteLine("First line of text.");
        streamWriter.WriteLine("Second line of text.");
    }
  }
}
```

C. Position and Seeking

- Random access capabilities:
    - Position property tracking
    - Seek() for arbitrary access
    - Length monitoring

4. Serialization and Formatters

- Serialization is converting each given object into sequence of bytes.
- This sequence of bytes can be written into any stream e.g. FileStream, NetworkStream, etc.
- Deserialization is converting sequence of bytes back to the object.

4.1 Binary Serialization

- `[Serializable]` attribute enables:

  - Object graph preservation
  - Type fidelity
  - Compact binary representation

- `[NonSerialized]` attribute on field:
  - makes that field non serialized (like `transient` in Java).
- **Note**:
  - BinaryFormatter is deprecated as Deserialization is insecure (may cause attack).
  - Removed in .NET 9. Needs to be enabled explicitly in .NET 8 project settings (still with a warning).
    - `<EnableUnsafeBinaryFormatterSerialization>true</EnableUnsafeBinaryFormatterSerialization>`
- Example:

```
[Serializable]
public class Person { /* ... */ }

// Serialize
BinaryFormatter formatter = new BinaryFormatter();
using (FileStream fs = File.Create("person.bin"))
{
    formatter.Serialize(fs, new Person());
}

// Deserialize
using (FileStream fs = File.OpenRead("person.bin"))
{
    Person p = (Person)formatter.Deserialize(fs);
}
```

4.2 XML Serialization

- XmlSerializer provides:
  - Human-readable output
  - Schema generation
  - Interoperability
- Example:

```
XmlSerializer serializer = new XmlSerializer(typeof(Person));

// Serialize to file
using (TextWriter writer = new StreamWriter("person.xml"))
{
    serializer.Serialize(writer, new Person());
}

// Deserialize from file
using (TextReader reader = new StreamReader("person.xml"))
{
    Person p = (Person)serializer.Deserialize(reader);
}
```

4.3 JSON Serialization (System.Text.Json)

- Modern default choice with:
  - System.Text.Json (high performance)
  - Newtonsoft.Json (rich features)
  - Async streaming support
- Example:

```

```

```
// Serialize
string json = JsonSerializer.Serialize(new Person());
File.WriteAllText("person.json", json);

// Deserialize
string jsonText = File.ReadAllText("person.json");
Person p = JsonSerializer.Deserialize<Person>(jsonText);
```

5. Advanced Scenarios

5.1 Asynchronous I/O

- Modern non-blocking pattern offering:
  - Better scalability
  - UI responsiveness
  - Efficient resource utilization
- Example:

```
async Task ProcessFileAsync() {
    using (StreamReader reader = new StreamReader("largefile.txt")) {
        string content = await reader.ReadToEndAsync();
        // Process content
    }
}
```

5.2 File System Watcher

- FileSystemWatcher provides events for:
  - File creations/modifications
  - Directory changes

    - Rename operations
- Example:

```
FileSystemWatcher watcher = new FileSystemWatcher("C:\\WatchFolder");
watcher.Created += (s, e) => Console.WriteLine($"Created: {e.Name}");
watcher.EnableRaisingEvents = true;
```

6.3 Memory-Mapped Files

- For extremely large files:
    - Efficient random access
    - Shared memory between processes
    - Native OS integration
- Example:

```
using (var mmf = MemoryMappedFile.CreateFromFile("large.bin"))
using (var accessor = mmf.CreateViewAccessor()) {
    int value = accessor.ReadInt32(position: 0);
}
```

6. Best Practices

6.1 Resource Management

1. **Always dispose streams** (use `using` blocks)
2. **Flush writers** when needed (or use auto-flush)
3. **Handle exceptions** (FileNotFoundException, UnauthorizedAccessException)

6.2 Performance Considerations

1. **Buffer sizes**: Optimal defaults exist (typically 4KB)
2. **Async vs Sync**: Use async for UI apps and services
3. **File sharing**: Consider `FileShare` modes for concurrent access

6.3 Security

1. **Validate file paths** (avoid path traversal attacks)
2. **Handle sensitive data** carefully in memory
3. **Set proper permissions** when creating files

MSDN References

- File and Stream I/O
- System.Text.Json
- Asynchronous File Access