C# .NET Coding Standards and Best Practices

1. Formatting Guidelines

- Use consistent indentation and spacing.
- Align braces and scope blocks properly.
- One statement per line.
- Align closing braces with the line that contains the opening brace.

2. Naming Rules

- Use camelCase for variables, parameters, and private fields.
- Use PascalCase for class names, method names, properties, and public members.
- Avoid underscores except in private _camelCase fields (optional based on team style).
- Use meaningful and descriptive names.

3. Code Organization

- Group related functions together.
- Follow file-per-class rule.
- Use regions (#region) to organize large files logically.
- Keep methods short and focused on a single responsibility.

4. Whitespace and Readability

- Add vertical whitespace between logically distinct blocks.
- Avoid excessive blank lines.
- Avoid trailing whitespace.

5. Constants

- Use const when values never change.
- Use readonly when values are set at runtime only once.
- Avoid magic numbers; use named constants instead.

6. Collections

- Use the most restrictive input type: Prefer IEnumerable, IReadOnlyList for inputs.
- For outputs: Use List if ownership is transferred; else use restrictive types.
- Prefer List<T> over arrays unless array performance is necessary.
- Avoid modifying collections during iteration; use RemoveAll or create a new list.

7. Generators vs Containers

- Use containers if all data is processed.
- Use generators (yield return) for lazy evaluation or partial processing.
- Avoid generator performance traps like calling .ToList() repeatedly.

8. Properties

- Use expression-bodied properties for simple getters: int Count => _count;
- Use { get; set; } for others.

9. Structs vs Classes

- Default to classes.
- Use structs only for small, immutable types with value semantics.
- Be aware of copy behavior and property return value pitfalls.

10. Lambdas vs Named Methods

- Use lambdas for short, simple logic.
- Use named methods for clarity and reusability.

11. Field Initializers

- Prefer field initializers for setting defaults.

12. Extension Methods

- Use only when source can't be modified.
- Keep in shared core libraries.
- Avoid excessive or context-specific extensions.

13. ref and out

- Use out for additional return values, placed last.
- Avoid ref unless mutation of input is necessary.

14. LINQ Usage

- Prefer short, clear LINQ chains.
- Use extension method syntax (.Where(), not from ... where).
- Avoid ForEach() with more than one statement.

15. Tuples

- Prefer named types over Tuple<> or value tuples.
- Use tuples only for lightweight, transient results.

16. String Handling

- Prefer string interpolation $"{value}".
- Use StringBuilder when concatenating many strings.
- Avoid + for building long strings.

17. Using Directives

- Avoid type aliasing unless necessary.
- Keep using statements clean and grouped.

18. Object Initializers

- OK for simple types or POCOs.
- Avoid for complex constructor logic.

19. Namespace Conventions

- Max 2 levels deep.
- Don’t tie folder structure too tightly to namespace hierarchy.
- Shared libraries should use unique root namespaces.

20. Struct Default Values

- Prefer returning a bool success flag with out value.
- Use nullable structs sparingly and for clarity.

21. Delegates

- Use ?.Invoke() to safely call delegates: MyEvent?.Invoke();

22. The var Keyword

- Use when type is obvious or irrelevant.
- Avoid for primitive values where clarity is important.

23. Attributes

- Place on a new line above the member.
- Separate multiple attributes with newlines.

24. Argument Naming and Clarity

- Replace unclear literals with named constants.
- Prefer enums over booleans in method arguments.
- Use named arguments at call sites for clarity.
- For many config options, pass a settings object.

25. Method Guidlines

- Keep methods short and focused.
- A method should ideally perform one task.
- Avoid long chains of LINQ; prefer single-line or imperative code.
- Avoid Container.ForEach(...) if the body has more than one statement.

26. Comments and Documentation

- Write self-explanatory code, but use comments for complex logic.
- Use XML documentation comments (///) for all public members.