C# CODING STANDARD RULE DOCUMENT

INTRODUCTION

This document presents a set of coding standards and best practices for writing C# code. Adhering to these standards will help maintain code quality, improve readability, and ensure consistency across projects.

Table Of Contents

1. Naming Conventions

1.1. General Naming Guidelines

- Use descriptive and meaningful names for all identifiers.
- Avoid single-letter variable names except for loop counters.
- Choose clarity over brevity.

1.2. Classes and Types

- Use PascalCase for class names (e.g., `MyClass`).
- Use nouns or noun phrases for class names.

1.3. Methods and Functions

- Use PascalCase for method names (e.g., `CalculateTotalPrice`).
- Use verbs or verb phrases for method names.

Example:

```
public class Calculator
{
   public int Add(int operand1, int operand2)
   {
      return operand1 + operand2;
   }
}
```

1.4. Variables and Fields

- Use camelCase for variable names (e.g., `myVariable`).
- Prefix member variables with an underscore (e.g., `_myField`).

1.5. Constants

- Use PascalCase for constant names (e.g., `MaxAttempts`).
- Prefix constants with "k" (e.g., `kMaxAttempts`).

1.6. Properties

- Use PascalCase for property names (e.g., `FullName`).
- Avoid exposing fields directly as properties unless necessary.

1.7. Enums

- Use PascalCase for enum names.
- Use singular nouns for enum values.
- Use explicit values for enum members when necessary.

1.8. Interfaces

- Prefix interfaces with "I" (e.g., `IComparable`).
- Use meaningful interface names.

1.9. Events and Delegates

- Use PascalCase for event and delegate names.
- Include the word "EventHandler" for event handler delegate names.

1.10. Namespaces

- Use meaningful and hierarchical namespace names.
- Avoid global namespace pollution.

1.11. Abbreviations

- Avoid abbreviations in identifier names unless widely accepted (e.g., XML).
- If abbreviations are used, make them consistent and documented.

2. Coding Style

2.1. Indentation and Spacing

- Use consistent indentation (e.g., 4 spaces per level).
- Use spaces, not tabs, for indentation.
- Maintain proper spacing around operators and after commas
- Limit lines to a reasonable length (e.g., 120 characters).

2.2. Braces and Formatting

- Use the "K&R" style for braces (e.g., opening brace on the same line).
- Use a new line for each method, class, or logical block.
- Keep code blocks concise and well-organized.

2.3. Comments

- Use meaningful comments for code explanations, not redundant comments.
  Follow a consistent commenting style
  (e.g., `//` for single-line comments, `/* */` for multi-line comments).

2.4. Using Statements

- Organize using statements alphabetically and logically.
- Remove unused using statements.

2.5. Exception Handling

- Catch specific exceptions rather than using generic catch blocks.
- Provide informative error messages in exceptions.

2.6. Null Checking

- Use null conditional operators (e.g., `?.`) for safe null checking.
- Avoid unnecessary null checks.

3. Object-Oriented Programming (OOP)

3.1. Inheritance and Polymorphism

- Prefer composition over inheritance when possible.
- Implement polymorphism using interfaces and abstract classes.

3.2. Encapsulation

- Encapsulate class members with appropriate access modifiers (e.g., private, protected).
- Use properties to encapsulate fields when necessary.

3.3. Abstraction

- Create abstract classes or interfaces for defining contracts.
- Provide clear and meaningful abstractions.

3.4. Interfaces and Contracts

- Follow the Single Responsibility Principle (SRP) when defining interfaces.
- Use interfaces to define contracts and promote loose coupling.

3.5. Composition over Inheritance

- Prefer composition and favor object composition over class inheritance.

3.6. Design Patterns

- Familiarize yourself with common design patterns (e.g., Singleton, Factory, Observer) and apply them when appropriate.

4. Error Handling and Logging

4.1. Exception Handling

- Use structured exception handling with `try`, `catch`, and `finally` blocks.
- Log exceptions for debugging and auditing purposes.
- Avoid swallowing exceptions without proper handling.

- **Example :**

```
try
{
   // Code that may throw an exception
}
catch (Exception ex)
{
   Logger.LogError(ex);
   throw; // Re-throw the exception
}
```

4.2. Custom Exception Types

- Create custom exception types for specific error conditions.
- Include relevant properties in custom exceptions for additional context.

5. Code Formatting

5.1. Line Length

- Limit the length of lines of code to improve readability. A common guideline is 80-120 characters per line.

5.2. Code Alignment

- Use consistent indentation and align related code elements (e.g., assignment operators, method calls) for improved readability.
- **Example:**

```
int firstNumber = 10;
string secondString = "Hello";
double thirdValue = 3.14;
```

6. Documentation

6.1. XML Documentation Comments

Use XML documentation comments for documenting public APIs.
Include `<summary>`, `<param>`, `<returns>`, and `<example>` tags where applicable.

6.2. README and Documentation Files

Include a README file with project documentation and setup instructions.
Maintain up-to-date documentation for libraries and projects.