C# Coding Guidelines

1. General Best Practices

- Use meaningful variable and method names (CamelCase for methods, PascalCase for classes).
- Avoid hard-coded values; use constants or configuration files.
- Follow the DRY (Don't Repeat Yourself) principle.
- Keep methods short and focused on a single responsibility.
- A method should not be more than 30 lines of code.
- A class should not have more than 5 methods.
- Use proper indentation and spacing for readability.
- Always catch specific exceptions instead of using catch (Exception e) {}.
- Don’t suppress exceptions.

2. Naming Conventions

- Classes: PascalCase (e.g., CustomerService)
- Methods: PascalCase (e.g., GetCustomerById)
- Variables: camelCase (e.g., customerId)
- Constants: UPPER_SNAKE_CASE (e.g., MAX_RETRIES)
- Interfaces: Prefix with I (e.g., IService)

3. Logging Best Practices

- Use structured logging (e.g., ILogger).
- Log at appropriate levels: Debug, Info, Warning, Error.
- Never log sensitive data (e.g., passwords, PHI, PII).

4. Code Structure and Formatting

- Use {} for all conditional statements, even for single-line blocks.
- Avoid deeply nested code; use guard clauses instead.
- Remove unused imports and references.

Bad Example:

```
if (user != null) user.Process();
```

Good Example:

```
if (user != null)
{
   user.Process();
}
```

5. Comments and Documentation

- Use XML comments (///).
- Keep comments meaningful—explain why, not what.

Example:

```
/// <summary>
/// Retrieves a customer by ID.
/// </summary>
public Customer GetCustomerById(int id) { ... }
```

6. Performance Optimization

- Use async/await for asynchronous processing.
- Avoid excessive object creation; reuse instances where possible.
- Optimize database queries—avoid SELECT *, use pagination.

7. Other Objectives

- Ensure maintainability and scalability in the codebase.
- Improve code reusability through modular design.
- Enhance security measures, including data encryption and secure authentication.
- Follow coding standards for better collaboration and readability.

Java Coding Guidelines

1. General Best Practices

- Use meaningful variable and method names (camelCase for methods, PascalCase for classes).
- Avoid hard-coded values; use constants or configuration files.
- Follow the DRY (Don't Repeat Yourself) principle.
- Keep methods short and focused on a single responsibility.
- A method should not be more than 30 lines of code.
- A class should not have more than 5 methods.
- Use proper indentation and spacing for readability.
- Always catch specific exceptions instead of using catch (Exception e) {}.
- Don’t suppress exceptions.

2. Naming Conventions

- Classes: PascalCase (e.g., CustomerService)
- Methods: camelCase (e.g., getCustomerById)
- Variables: camelCase (e.g., customerId)
- Constants: UPPER_SNAKE_CASE (e.g., MAX_RETRIES)
- Interfaces: No I prefix (e.g., Service)

3. Logging Best Practices

- Use structured logging (e.g., SLF4J).
- Log at appropriate levels: Debug, Info, Warning, Error.
- Never log sensitive data (e.g., passwords, PHI, PII).

4. Code Structure and Formatting

- Use {} for all conditional statements, even for single-line blocks.
- Avoid deeply nested code; use guard clauses instead.
- Remove unused imports and references.

Bad Example:

```
if (user != null) user.process();
```

Good Example:

```
if (user != null) {
   user.process();
}
```

5. Comments and Documentation

- Use Javadoc (/** ... */).
- Keep comments meaningful—explain why, not what.

Example:

```
/**
 * Retrieves a customer by ID.
 */
public Customer getCustomerById(int id) { ... }
```

6. Performance Optimization

- Use CompletableFuture for asynchronous processing.
- Avoid excessive object creation; reuse instances where possible.
- Optimize database queries—avoid SELECT *, use pagination.

7. Object-Oriented Principles

- Encapsulation: Keep data private and expose only necessary behavior.
- Abstraction: Use abstract classes and interfaces to define reusable behaviors.
- Inheritance: Reuse common behavior via class hierarchies but avoid deep inheritance trees.
- Polymorphism: Use method overriding and interfaces to provide flexible code.
- SOLID Principles: Ensure code follows principles like Single Responsibility and Open/Closed principle.

8. Other Objectives

- Ensure maintainability and scalability in the codebase.

- Improve code reusability through modular design.
- Enhance security measures, including data encryption and secure authentication.
- Follow coding standards for better collaboration and readability.

By following these guidelines, you can ensure high-quality, maintainable, and efficient code.