README.md

A C# Developer's Handbook

The focus of this document is on providing a reference for writing C#. It includes naming, structural and formatting conventions as well as best practices for writing clean, safe and maintainable code.

Many of the best practices and conventions apply equally well to other languages.

Goals

This handbook has the following aims and guiding principles:

- Increase readability and maintainability with a unified style.
- Minimize complexity with proven design principles
- Increase code safety and prevent hard-to-find errors through best practices.
- Maximize effectiveness of coding tools.
- Accommodate IDE- or framework-generated code.
- Provide justifications and examples for rules.

Scope

This handbook includes:

- General programming advice and best practices
- General formatting and style recommendations
- C#-specific sections

Environment

The recommended environment and tools at this time are:

- *Microsoft Visual Studio 2017*
- *JetBrains ReSharper 2016.3.2*
- *StyleCop 4.7*
- *StyleCop by JetBrains* extension for ReSharper

- *Cyclomatic Complexity* extension for ReSharper
- *EditorConfig*
- *C# 7.0*

For older versions of Visual Studio and C#, use what you can or refer to older versions of this handbook.

Improvements

This document is a work-in-progress. Please speak up or contribute if you think there is something missing.

- If a guideline is not sufficiently clear, recommend a clearer formulation.
- If you don’t like a guideline, try to get it changed or removed, but don’t just ignore it. Your code reviewer is most likely unaware that you are special and not subject to the rules.

In Practice

Applying the Guidelines

- Unless otherwise noted, these guidelines are not optional, nor are they up to interpretation.
- A reviewer always has the right to correct mistakes and aberrations, but is not obligated to do so in every review.
- Please note issues with the guidelines during a review. These changes should flow into the guidelines if enough parties agree.

The handbook defines the goal. Use iterations and refactoring to incrementally bring the code closer to full compliance.

Fixing Problems in Code

Fix non-conforming code at the earliest opportunity.

- Fix small and localized errors immediately, in a "cleanup" commit.
- Always use a separate commit to rename or move files.
- Create an issue for larger problems that cannot be fixed quickly.

Working with an IDE

Modern IDEs generate code; this is very helpful and saves a lot of work. Within reason, the generated code should satisfy the coding guidelines. If the generated code is not under your control, then it's OK to turn a blind eye to style infractions.

- Configure your IDE to produce code that is as close to the guidelines as possible. StyleCop and ReSharper are an enormous help.
- Update names for visual-design elements and event handlers manually, if needed.
- Write code generators to produce conforming code.
- Do not update code generated by tools not under your control (e.g. *.Designer files).
- Use “Format Document” to reformat auto-generated code.
- Use the highest warning level available (level 4 in Visual Studio) and address all warnings (either by fixing the code or explicitly ignoring them).

Settings files

This repository includes configuration files that set up the rules outlined in this handbook for *StyleCop* and *ReSharper* and *EditorConfig*.

Table of Contents

Overview

1. Terminology
2. History
3. References

Naming

1. Characters
2. Words
3. Semantics
4. Case
5. Grouping
6. Algorithm
7. Structure
    i. Assemblies
    ii. Files
    iii. Namespaces
8. Types
    i. Classes
    ii. Interfaces
    iii. `enums`
    iv. Generic Parameters
    v. Sequences and Lists
9. Members
    i. Properties
    ii. Methods
    iii. Extension Methods
    iv. Parameters
    v. Lambdas
    vi. Events
    vii. Delegates
10. Statements and Expressions
    i. Local Variables
    ii. Return Values
    iii. Compiler Variables

Formatting

1. Whitespace and Symbols
    i. Blank Lines
    ii. Line Breaks
    iii. Indenting and Spacing
    iv. Braces
    v. Parentheses
2. Language Elements
    i. Methods
    ii. Constructors
    iii. Initializers
    iv. Lambdas
    v. Multi-line Text

Usage

vii. Resource Strings
viii. Floating Point and Integral Types
ix. Local Variables
x. Local Functions
xi. `var`
xii. `out` variables
xiii. Loops
xiv. Conditions
xv. `switch`
xvi. Pattern-matching
xvii. `continue`
xviii. `return`
xix. `goto`
xx. `unsafe`
xxi. Ternary and Coalescing Operators
xxii. Null-conditional Operator
xxiii. `throw`-Expressions
xxiv. Lambdas
xxv. System.Linq
xxvi. Casting
xxvii. `checked`
xxviii. Compiler Variables
xxix. Comments

Best Practices

1. Design
   i. Abstractions
   ii. Inheritance vs. Composition
   iii. Interfaces vs. Abstract Classes
   iv. Open vs. Closed APIs
   v. Controlling API Size
   vi. Read-only Interfaces
   vii. Single-Responsibility Principle
   viii. Loose vs. Tight Coupling
   ix. Methods vs. Properties
   x. Code > Comments
2. Safe Programming
   i. Be Functional
   ii. Avoid `null` references
   iii. Local variables
   iv. Side Effects
   v. “Access to Modified Closure”
   vi. "Collection was modified; enumeration operation may not execute."
   vii. "Possible multiple enumeration of IEnumerable"
3. Error Handling
   i. Strategies
   ii. Terms
   iii. Errors

Overview

Terminology

Term	Definition
IDE	Integrated Development Environment
VS	Microsoft Visual Studio
R#	JetBrains ReSharper
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
DRY	Don't Repeat Yourself
YAGNI	You Ain't Gonna Need It

History

This document started out as a Microsoft Word document and was originally published at CodePlex in 2008 by Encodo Systems AG. It has been updated a few times over the years, but not nearly often enough to keep pace with changes.

This most recent version reflects a conversion to Markdown format and storage in a Git repository to improve version-tracking, collaboration and updates.

It is also a complete overhaul in both content and structure to provide maximum benefit to various readers. It is also now maintained by Marco von Ballmoos (partner and senior developer at Encodo Systems AG) instead of by Encodo itself.

Versions

Version	Date	Author	Comments
7.0.0	29.04.2017	MvB	Converted the manual from Microsoft Word to Markdown format; rewrote most chapters; removed redundancy; upgraded all styles/patterns from C# 3.5 to C# 7; matched version number to language-version number
2.0.0	Unreleased	MvB	Updated text throughout the "1 – General" and "2 – Design Guide" sections; added "7.17 – Using System.Linq", "7.29 – Refactoring Names and Signatures" and "7.30 – Loose vs. Tight Coupling" best practices;
1.5.3	Unreleased	MvB	Added reference and notes from "The Little Manual of API Design"; added section "7.15 – Using Partial Classes".
1.5.2	19.10.2009	MvB	Expanded "8.1 – Documentation" with examples; added more tips to the "2.3 – Interfaces vs. Abstract Classes" section; added "7.21 – Restricting Access with Interfaces"; added "5.3.7 – Extension Methods" and "7.18 – Using Extension Methods".

Version	Date	Author	Comments
1.5.1	24.10.2008	MvB	Incorporated feedback from the forums at the MSDN Code Gallery.
1.5	20.05.2008	MvB	Updated line-breaking section; added more tips for generic methods; added tips for naming delegates and delegate parameters; added rules for object and array initializers; added rules and best practices for return statements; added tips for formatting complex conditions; added section on formatting switch statements.
1.4	18.04.2008	MvB	Updated formatting for code examples; added section on using the var keyword; improved section on choosing names; added naming conventions for lambda expressions; added examples for formatting methods; re-organized error handling/exceptions section; updated formatting.
1.3	31.03.2008	MvB	Added more tips for documentation; added white-space rules for regions; expanded rules for line-breaking; updated event naming section.
1.2	07.03.2008	MvB	Change to empty methods; added conditional compilation section; updated section on comments; made some examples customer-neutral; fixed some syntax-highlighting; reorganized language elements.
1.1	06.02.2008	MvB	Updated sections on error handling and naming
1.0	28.01.2008	MvB	First Draft
0.1	03.12.2007	MvB	Document Created

Referenced Documents

The style and formatting guidelines draw mostly from in-house programming experience. Most of the following documents were more heavily used in prior versions. References have been included for completeness.

Date	Title	Authors	Version
13.04.2017	Patterns and Practices in C# 7	Jonathan Allen	
20.07.2015	Microsoft C# Coding Conventions	Microsoft	VS2015
01.07.2011	IDesign C# Coding Standards	Juval Lowy	2.4
19.05.2011	Optional argument corner cases, part four	Eric Lippert	
01.11.2008	Microsoft Framework Design Guidelines	Krzysztof Cwalina and Brad Abrams	2.0
19.06.2008	The Little Manual of API Design (PDF)	Jasmin Blanchette	
19.05.2005	Coding Standard: C# (PDF)	Philips Medical Systems	1.3
26.01.2005	Microsoft Internal Coding Guidelines	Brad Abrams	2.0

1_naming.md

Naming

Characters

- Names contain only alphabetic characters.
- The underscore is allowed only as a leading character for `private` fields.
- Numbers are allowed only for local variables in tests and then only as a suffix.
- Do not use the @-symbol

Words

- Use US-English (e.g. “color” rather than “colour”).
- Use English grammar (e.g. use `ImportableDatabase` instead of `DatabaseImportable`).
- Use only standard abbreviations (e.g. “XML” or “SQL”).
- Use correct capitalization. If a word is not hyphenated, then it does not need a capital letter in the camel- or Pascal-cased form. For example, “metadata” is written as `Metadata` in Pascal-case, not `MetaData`.
- Use number names instead of numbers (e.g. `partTwo` instead of `part2`).
- Do not use Hungarian notation or any other prefixing notation to "group" types or members.
- Use whole words or stick to accepted short forms (e.g. you may use `max` for `maximum` but prefer the suffix `Count` to the prefix `num`).

Semantics

- A name must be semantically meaningful in its scope.
- A name should be as short as possible.
- A name communicates intent; prefer `UpdatesAutomatically` to `AutoUpdate`.
- A name reflects semantics, not storage details; prefer `InterestRate` to `DecimalRate`.
- A name should include explicit units where possible. The following property isn't clearly defined.

```
public Timeout { get; } = 3600;
```

This looks like a timeout of either an hour or 3.6 seconds. Timeouts are *usually* expression in milliseconds. We can fix this with documentation.

```
/// <summary>
/// The number of milliseconds to wait before aborting the operation.
/// </summary>
public Timeout { get; } = 3600;
```

This is a good start. Even better is to include the units in the name of the property.

```
/// <summary>
/// The number of milliseconds to wait before aborting the operation.
/// </summary>
public TimeoutInMilliseconds { get; } = 3600;
```

Case

The following table lists the capitalization and naming rules for different language elements.

- Pascal-case capitalizes every word in a name.
- Camel-case capitalizes all but the first word in a name.
- Acronyms longer than two letters are in Pascal-case (e.g. `Xml` or `Sql`). Acronyms at the beginning of a camel-case name are always all lowercase (e.g. `html`).

Language Element	Case
Class	Pascal
Interface	Pascal w/leading `I`
Struct	Pascal
Enumerated type	Pascal
Enumerated element	Pascal
Properties	Pascal
Generic parameters	Pascal
Tuple field	Pascal
Public or protected `readonly` or `const` field	Pascal
Private field	Camel with leading underscore
Method argument	Camel
Local variable	Camel
Attributes	Pascal with `Attribute` suffix
Exceptions	Pascal with `Exception` suffix
Event handlers	Pascal with `EventHandler` suffix

Collision and Matching

- An element may not have the same name as its containing element (e.g. class `Expressions` in namespace `Expressions` or property `Company` on class `Company`).
- The most appropriate name for a property is often the same as its type (e.g. for `enum` properties).
- Names differing only by case may be defined within the same scope only for different language elements (e.g. a local variable and a property or a method parameter and a local parameter).

```
public void UpdateLength(int newLength, bool refreshViews)
{
  int length = Length;
  // ...
}
```

Grouping

- Do not use a "library" prefix for types (e.g. instead of `QnoDatabase`, use a more descriptive name, like `MetaDatabase` or `RelationalDatabase`).
- You may use a prefix when it's more convenient for disambiguation, as for UI-control libraries. If you do use a prefix, use a whole word (e.g. prefer `Quino` to `Qno`) and apply it consistently.
- Avoid very generic type names (e.g. `Element`, `Node`, `Message` or `Log`), which collide with types from the framework or other commonly-used libraries. Use a more specific name, if at all possible.

- If there are multiple types encapsulating similar concepts (but with different implementations, for example), you should use a common suffix to group them. For example, all the expression node types in the Encodo expressions library end in the word Expression.

Algorithm

Local Variables and Parameters

A parameter name or local variable should have the same name as the type. It is valid to use shorter forms for longer type names, as long as the context is clear.

- `IMetaExpressionFactory`: can be `metaExpressionFactory`, `expressionFactory` or `factory`.
- `f` is not acceptable for parameters, but is acceptable for a local variable in a short body.

Structure

Assemblies

- Assemblies should be named after their content.
- Group assemblies with a common prefix (e.g. `Encodo` or `Quino`).
- Separate identifiers in assembly names with a period.
- The root namespace of an assembly does not have to match the assembly name.
- The `AssemblyInfo.cs` file must contain company, copyright and version information.

Files

- The name of the file should match the type name.
- Generated partial classes belong in a separate file, using the same root name as the user-editable file, but extended by an identifier to indicate its purpose or origin (as in the example below). This extra part must be Pascal-cased. For example:

```
Company.cs          // user-modifiable file
Company.Metadata.cs // properties generated from metadata
```

- If a generic type is the only type with that name, then do not include the parameter names in the filename. If there is a non-generic type and a generic type with the same name, then the filename should include the generic argument names enclosed in {}. E.g. If the types `Atom` and `Atom<TInput>` both exist, then the filenames should be `Atom.c` and `Atom{TInput}.cs`, respectively.
- Tests for a file go in `<FileName>Tests.cs` (if there are a lot of tests, they should be split into several files, but always using the form `<Extra><FileName>Tests.cs`) where `Extra` identifies the group of tests found in the file. Tests should be defined in their own assembly to avoid dependencies on unit-testing assemblies. The tests for a class should appear in the same location as the class being tested. That is, the tests for the class `Encodo.Tools.Csv.CsvParser` should be in `Encodo.Testing.Tools.Csv.CsvParser`.

Namespaces

- The namespace must match the file location in the project. For example, if the project's root namespace is `Encodo.Parsers`, then the file located at `Csv/CsvParser.cs` should have the namespace `Encodo.Parsers.Csv`.
- Namespaces start with `<CustomerName>.<ProductName>` (e.g. `Encodo.Quino.*` or `Encodo.Punchclock.*`)
- Namespaces should be plural, as they will contain multiple types (e.g. `Encodo.Expressions` instead of `Encodo.Expression`). This also reduces the risk of collision.
- If your framework or application encompasses more than one tier, use the same namespace identifiers for similar tasks. For example, common data-access code goes in `Encodo.Data`, but metadata-based data-access code goes in `Encodo.Quino.Data`.

- Do not use “reserved” namespace names like `System` because these will conflict with standard .NET namespaces and require resolution using the `global::` namespace prefix.

Types

Classes

- If a class implements a single interface, it should reflect this by incorporating the interface name into its own (e.g. `MetaList` implements `IList`).
- Static classes containing extension methods end in `Extensions`
- All other static classes should use the suffix `Tools`.
- `abstract` classes should use the suffix `Base`.

Interfaces

- Prefix interfaces with the letter “I”.

`enums`

- Simple enumerations have singular names, whereas bit-sets have plural names.

Generic Parameters

- If a class or method has a single generic parameter, use the letter `T`.
- If there are two generic parameters and they correspond to a key and a value, then use `K` and `V`.
- Generic methods on classes with a generic parameter should use `TResult`, where appropriate. The example below shows a generic class with such a method.

```
public class ListBookmarkSelection<T> : IBookmarkSelection
{
  public IList<TResult> GetObjects<TResult>()
  {
    // Convert list contents from T to TResult
  }
}
```

- Generic conversion functions should use `TInput` and `TOutput`, respectively.

```
public static IList<TOutput> ConvertList<TInput, TOutput>(IList<TInput> input)
{
  // Convert list contents from TInput to TOutput
}
```

- If there are multiple parameters, but no pattern, name the “contained” element `T` (if there is one) and the other parameters something specific starting with the letter T.

Sequences and Lists

- Prefer the plural form (e.g. `appointments`) for sequences (e.g. `IEnumerable<T>`) and lists or arrays
- Use the suffix “List” when you want to emphasize that a parameter or property is a list (e.g. `appointmentList`).

Members

Properties

- Properties should be nouns or adjectives.

- Prepend “Is” to the name for Boolean properties only if the intent is unclear without it. The next example shows such a case:

  ```
  public bool Empty { get; }
  public bool IsEmpty { get; }
  ```

  Even though it’s a property not a method, the first example might still be interpreted as a verb rather than an adjective. The second example adds the verb “Is” to avoid confusion, but both formulations are acceptable.
- A property’s backing field (if present) must be an underscore followed by the name of the property in camel case.
- Use common names, like `Item` or `Value`, for accessing the central property of a type.
- Do not include type information in property names. For example, for a property of type `IMetaRelation`, use the name Relation instead of the name `MetaRelation`.
- Make the identifier as short as possible without losing information. For example, if a class named `IViewContext` has a property of type `IViewContextHandler`, that property should be called `Handler`.
- If there are two properties that could be shortened in this way, then neither of them should be. If the class in the example above has another property of type `IEventListHandler`, then the properties should be named something like `ViewContextHandler` and `EventListHandler`, respectively.
- Avoid repeating information in a class member that is already in the class name. Suppose, there is an interface named `IMessages`; instances of this interface are typically named messages. That interface should not have a property named `Messages` because that would result in calls to `messages.Messages.Count`, which is redundant and not very readable. Instead, name the property something more semantically relevant, like `All`, so the call would read `messages.All.Count`.

Methods

- Methods names should include a verb.
- Method names should not repeat information from the enclosing type. For example, an interface named `IMessages` should not have a method named `LogMessage`; instead name the method `Log`.
- State what a method does; do not describe the parameters (let code-completion and the signature do that for you).
- Methods that return values should indicate this in their name, like `GetList()`, `GetItem()` or `CreateDefaultDatabase()`. Though there is garbage collection in C#, you should still use `Get` to indicate retrieval of a local value and `Create` to indicate a factory method, which always creates a new reference. For example, instead of writing:

  ```
  public IDataList<GenericObject> GetList(IMetaClass cls)
  {
    return ViewApplication.Application.CreateContext<GenericObject>(cls);
  }
  ```

You should write:

```
public IDataList<GenericObject> CreateList(IMetaClass cls)
{
  return ViewApplication.Application.CreateContext<GenericObject>(cls);
}
```

- Avoid defining everything as a noun or a manager. Prefer names that are logically relevant, like `Missile.Launch()` rather than `MissileLauncher.Execute(missile)`.
- Methods that set a single property value should begin with the verb `Set`.
- The most generalized version of a method name should be reserved for the method that the framework wishes to encourage or that is used most often. An example from [6] is reproduced below:

> Suppose you have two event-delivery mechanisms, one for immediate (synchronous) delivery and one for delayed (asynchronous) delivery. The names `sendEventNow()` and `sendEventLater()` suggest themselves. Now, if you want to encourage your users to use synchronous delivery (e.g., because it is more lightweight), you could name the synchronous method `sendEvent()` and keep `sendEventLater()` for the asynchronous case.

Extension Methods

- Extension methods for a given class or interface should appear in a class named after the class or interface being extended, plus the suffix "Tools". For example, extension methods for the class `Enum` should appear in a class named `EnumTools`.
- In the case of interfaces, the leading "I" should be dropped from the class name. For example, extension methods for the interface `IEnumerable<T>` should appear in a class named `EnumerableTools`.

Parameters

- Prefer whole words instead of abbreviations (use `index` instead of `idx`).
- Parameter names should be based on their intended use or purpose rather than their type (unless the type indicates the purpose adequately).
- Do not simply repeat the type for the parameter name; use a name that is as short as possible, but doesn't lose meaning. (E.g. a parameter of type `IDataContext` should be called `context` instead of `dataContext`.)
- However, if the method also, at some point, makes use of an `IViewContext`, you should make the parameter name more specific, using `dataContext` instead.
- For copy constructors or equality operators, name the object to be copied or compared `other`.

Lambdas

- Do not use the highly non-expressive `x` as a parameter name.
- Instead, use a single-letter variable starting with the first letter of the type or formal parameter.
- Parameters in a lambda expression should follow the same conventions as for parameters in standard methods.

Events

- Single events should be named with a noun followed by a descriptive verb in the past tense.

```
event EventHandler MessageDispatched;
```

- For paired events—one raised before executing some code and one raised after—use the gerund form (i.e. ending in "ing") for the first event and the past tense (i.e. ending in "ed") for the second event.

```
event EventHandler MessageDispatching;
event EventHandler MessageDispatched;
```

- Event receivers are like any other methods and should be named according to their task, not the event to which they are attached. The following method updates the user interface; it does this regardless of whether it is attached as an event receiver for the `MessageDispatched` event.

```
void UpdateUserInterface(object sender, EventArgs args)
{
  // Implementation
}
```

- Never start an event receiver method with "On" because Microsoft uses that convention for raising events.
- To trigger an event, use `Raise[EventName]`; this method must be `protected` and `virtual` to allow descendants to perform work before and after calling the base method.
- If you are raising events for changes made to properties, use the pattern `Raise[Property]Changed`.

Delegates

- Use a descriptive verb for delegate names, like the following examples:

```
delegate string TransformString(T item);
delegate bool ItemExists(T item);
delegate int CompareItems(T first, T second);
```

- Delegate method parameter names should use the same grammar as the type, so that it sound natural when executed:

```
public string[] ToStrings(TransformToString transform)
{
  // ...
  result[] = transform(item);
  // ...
}
```

- If a delegate is only used once or twice and has a relatively simple syntax, use `Func<>` for the parameter signature instead of declaring a delegate. The example above can be rewritten as:

```
public string[] ToStrings(Func<T, string> transform)
{
 // ...
 result[] = transform(item);
 // ...
}
```

  This practice makes it much easier to determine the expected signature in code-completion as well.

Statements and Expressions

Local Variables

Since local variables are limited to a much smaller scope and are not documented, the rules for name-selection are somewhat more relaxed.

- Avoid using `temp` or `i` or `idx` for loop indexes. Use the suffix `Index` together with a descriptive prefix, as in `colIndex` or `itemIndex` or `memberIndex`.
- Names need only be as specific as the scope requires.
- The more limited the scope, the more abbreviated the variable may be.
- Use real words where there is enough space to do so.
- Use single letters in lambdas where you're trying to save space.
- Use `_` instead of declaring useless local variables during deconstruction.

Return Values

- If a method creates a local variable expressly for the purpose of returning it as the result, that variable should be named `result`.

```
object result = this[Fields.Id];

if (result == null) { return null; }

return (Int32)result;
```

Compiler Variables

- Compiler variables are all capital letters, with words separated by underscores.

```
#if ENCODO_DEVELOPER
      return true;
#else
      return false;
#endif
```

Formatting

Whitespace and Symbols

Blank Lines

In the following list, the phrase “surrounding code” refers to a line consisting of more than just an opening or closing brace. That is, no new line is required when an element is at the beginning or end of a methods or other block-level element. Always place an empty line in the following places:

- Between the file header and the `namespace` declaration or the first `using` statement.
- Between the last `using` statement and the `namespace` declaration.
- Between types (`classes` , `structs` , `interfaces` , `delegates` or `enums`).
- Between public, protected and internal members.
- Between preconditions and ensuing code.
- Between post-conditions and preceding code.
- Between a call to a `base` method and ensuing code.
- Between `return` statements and surrounding code.
- Between block constructs (e.g. `while` loops or `switch` statements) and surrounding code.
- Between documented `enum` values; undocumented values may be grouped together.
- Between logical groups of code in a method; this notion is subjective and more a matter of style. You should use empty lines to improve readability, but should not overuse them.
- Between the last line of code in a block and a comment for the next block of code.
- Between statements that are broken up into multiple lines.
- Between a `#region` tag and the first line of code in that region. See next section.
- Between the last line of code in a region and the `#endregion` tag. See next section.

Do not place an empty line in the following places:

- After another empty line.
- Between retrieval code and handling for that code. Instead, they should be formatted together.

```
IMetaReadableObject obj = context.Find<IMetaReadableObject>();
if (obj == null)
{
  context.Recorder.Log(Level.Fatal, String.Format("Error!"));

  return null;
}
```

- Between any line and a line that has only an opening or closing brace on it (i.e. there should be no leading or trailing newlines in a block).
- Between undocumented fields (usually private); if there are many such fields, you may use empty lines to group them logically.

Line Breaks

- Use line-breaking only when necessary, as outlined below.

- No line should exceed 120 characters.
- Use as few line-breaks as possible.
- Line-breaking should occur at natural boundaries; the most common such boundary is between parameters in a method call or definition.
- A line-break at a boundary that defines a new block should be indented one more level.
- A line-break at any other boundary should be indented at the same level as the original line.
- The separator (e.g. a comma) between elements formatted onto multiple lines goes on the same line after the element. The IDE is much more helpful when formatting that way.
- The most natural line-breaking boundary is often before and after a list of elements. For example, the following method call has line-breaks at the beginning and end of the parameter list.

```
people.DataSource = CurrentCompany.Employees.GetList(
  connection, metaClass, GetFilter(), null
);
```

- If one of the parameters is much longer, then you add line-breaking between the parameters; in that case, all parameters are formatted onto their own lines:

```
people.DataSource = CurrentCompany.Employees.GetList(
  connection,
  metaClass,
  GetFilter("Global.Applications.Updater.PersonList.Search"),
  null
);
```

- Note in the examples above that the parameters are indented. If the assignment or method call was longer, they would no longer fit on the same line. In that case, you should use two levels of indenting.

```
Application.Model.people.DataSource =
  Global.ApplicationEnvironment.CurrentCompany.Employees.GetList(
    connection,
    metaClass,
    GetFilter("Global.Applications.Updater.PersonList.Search"),
    null
  );
```

- Even if there is a logical grouping for parameters, you should still apply line-breaking using the all-on-one-line or each-on-its-own-line rules stated above. For example, the following method specifying Cartesian coordinates feels natural, but is not well-supported by automatic formatting rules:

```
Geometry.PlotInBox(
  "Global.Applications.MainWindow",
  topLeft.X, topLeft.Y,
  bottomRight.X, bottomRight.Y
);
```

  As nice as it might look, *do not use this line-breaking technique*.

Indenting and Spacing

- An indent is two spaces.
- Use a single space after a comma (e.g. between function arguments).
- There is no space after the leading parenthesis/bracket or before the closing parenthesis/bracket.
- There is no space between a method name and the leading parenthesis, but there is a space before the leading parenthesis of a flow-control statement.
- Use a single space to surround *all* infix operators; there is no space between a prefix operator (e.g. “-” or “!”) and its single parameter.
- Do not use spacing to align type members on the same column (e.g. as with the explicitly assigned values for members of an enumerated type).

Braces

- Curly braces should—with a few exceptions outlined below—go on their own line.
- A line with only an opening brace should never be followed by an empty line.
- A line with only a closing brace should never be preceded by an empty line.

Properties

- Simple getters and setters should go on the same line as all brackets.
- Abstract properties should have get, set and all braces on the same line.
- Complex getters and setters should have each bracket on its own line.
- Place Auto-Property Initializers on the same line.

```
public int Maximum { get; } = 45;
```

Methods

- Completely empty methods should have brackets placed on separate lines:

```
SomeClass(string name)
  : base(name)
{
}
```

Parentheses

- Use parentheses only to improve clarity.
- Do not use parentheses for simple expressions. The following expression is clear enough without extra parentheses.

```
if (context != null && context.Count > 0)
{
}
```

Language Elements

Methods

Definitions

- Stay consistent with line-breaking in related methods within a class; if one is broken up onto multiple lines, then all related methods should be broken up onto multiple lines.
- The closing brace of a method definition goes on the same line as the last parameter (unlike method calls). This avoids having a line with a solitary closing parenthesis followed by a line with a solitary opening brace.

```
public static void SetupLookupDefinition(
  RepositoryItemLookUpEdit lookupOptions,
  IMetaClass metaClass)
{
  // Implementation...
}
```

- Generic method constraints should always be on their own line, with a single indent.

```
string GetNames<T>(IMetaCollection<T> elements, string separator, NameOption options
  where T : IMetaBase;
```

- The indent for a generic-method constraint stays the same, even with wrapped parameters.

```
public static void SetupLookupFromData<T>(
  RepositoryItemLookUpEdit lookupOptions,
  IDataList<T> dataList)
  where T : IMetaReadable
{
  SetupLookupFromData<T>(lookupOptions, dataList, dataList.MetaClass);
}
```

Calls

- The closing parenthesis of a method call goes on its own line to “close” the block (see example below).

```
result.Messages.Log(
  Level.Error,
  String.Format(
    "Class [{0}] has the same name as class [{1}].",
    dbCls.Identifier,
    classMap[cls.MetaId]
  )
);
```

- If the result of calling a method is assigned to a variable, the call may be on the same line as the assignment if it fits.

```
people.DataSource = CurrentCompany.Employees.GetList(
  connection,
  ViewAspectTools.GetViewableWrapper(cls),
  GetFilter().Where(String.Format(“PersonId = {0}”, personId)),
  null
);
```

- If the call does not fit easily—or if the method call is “too far away” from the ensuing parameters—you should move the call to its own line and indent it:

```
WindowTools.GetActiveWindow().GetActivePanel().GetActiveList().DataSource =
  CurrentCompany.Organization.MainOffice.Employees.GetList(
    connection,
    ViewAspectTools.GetViewableWrapper(cls),
    GetFilter().Where(String.Format(“PersonId = {0}”, personId)),
    null
  );
```

Chaining

- Chained method calls can be formatted onto multiple lines; if one chained method-call is formatted onto its own line, then they must all be on separate lines.

```
string contents = header
  .Replace("{Year}", DateTime.Now.Year.ToString())
  .Replace("{User}", "ENCODO")
  .Replace("{DateTime}", DateTime.Now.ToString());
```

- If a line of a chained method-call opens a new logical context, then ensuing lines should be indented to indicate this. For example, the following example joins tables together, with the last three statements applied to the last joined table. The indenting helps make this clear.

```
query
  .Join(Settings.Relations.Company)
  .Join(Company.Relations.Office)
  .Join(Office.Relations.Employees)
    .WhereEquals(Employee.Fields.Id, employee.Id)
    .OrderBy(Employee.Fields.LastName, SortDirection.Ascending)
    .OrderBy(Employee.Fields.FirstName, SortDirection.Ascending);
```

Constructors

- Base constructors should be on a separate line, indented one level.

```
public class B : A
{
  B(string name)
    : base(name)
  {
  }
}
```

Initializers

- Don't add a trailing comma for the last element.

Object Initializers

Longer initialization blocks should go on their own line; the rest can be formatted in the following ways (depending on line-length and preference):

- Shorter initialization blocks (one or two properties) can be specified on the same line:

```
var personOne = new Person { LastName = "Miller", FirstName = "John" };
```

- The `new`-clause is on the same line as the declaration:

```
var sizeAspect = new ViewPropertySizeAspect
{
  VerticalSizeMode = SizeMode.Absolute,
  VerticalUnits = height
};

prop.Aspects.Add(sizeAspect);
```

- The new-clause is on its own line:

```
var sizeAspect =
  new ViewPropertySizeAspect
  {
    VerticalSizeMode = SizeMode.Absolute,
    VerticalUnits = height
  };

prop.Aspects.Add(sizeAspect);
```

- The initializer is nested within the method-call on the same line as the declaration:

```
prop.Aspects.Add(new ViewPropertySizeAspect { VerticalUnits = height });
```

- The initializer is nested within the method-call on its own line:

```
prop.Aspects.Add(
  new ViewPropertySizeAspect
  {
    VerticalSizeMode = SizeMode.Absolute,
    VerticalUnits = height
  });
```

- Putting the `new` operator on the same line is also fine, but then you shouldn't indent the braces.

```
prop.Aspects.Add(new ViewPropertySizeAspect
{
  VerticalSizeMode = SizeMode.Absolute,
```

```
  VerticalUnits = height
});
```

If the initializer goes spans multiple lines, then the new-clause must also go on its own line.

Array Initializers

The type is usually optional (unless you're initializing an empty array), so you should leave it empty.

Lambdas

- Do not use anonymous delegates; use lambda notation instead.
- Do not use parentheses around a single parameter in a lambda expression.
- All rules for standard method calls also apply to method calls with lambdas.
- Longer lambdas should be written with an indent, as follows:

```
var persistentProperties =
  model.ReferencedProperties.FindAll(
    prop =>
    {
      // More code...

      return prop.Persistent;
    }
  );
```

- Short lambdas benefit can just be inlined:

```
public string[] Keys
{
  get
  {
    return ToStrings(i => i.Name);
  }
}
```

  Also OK, since the `get` body is short:

```
public string[] Keys
{
  get { return ToStrings(i => i.Name); }
}
```

  Even better, use expression-bodied members:

```
public string[] Keys => ToStrings(i => i.Name);
```

- Short lambdas are just parameters; treat them as you would any other parameters:

```
_context = new DataContext(
  Settings.Default.ConfigFileName,
  DatabaseType.PostgreSql,
  () => ModelGenerator.CreateModel()
);
```

  In the example above each parameter is on its own line, as required.
- Keep parameters short in constructor bases.

```
public Application()
  : base(DatabaseType.PostgreSql, () => ModelGenerator.CreateModel())
{
}
```

- All rules for standard method calls also apply to method calls with lambda expressions.
- Very short lambda expressions may be written as a simple parameter on the same line as the method call:

```
ReportError(msg => MessageBox.Show(msg));
```

- Longer lambda expressions should go on their own line, with the closing parenthesis of the method call closing the block on another line. Any calls attached to the result—like `ToList()` or `Count()` —should go on the same line as the closing parenthesis.

```
people.DataSource = CurrentCompany.Employees.Where(
  item => item.LessonTimeId == null
).ToList();
```

- Longer lambda expressions should not be both wrapped and used in a `foreach` -statement; instead, use two statements as shown below. Use short parameter names in lambdas since they are used very close to their declaration (by definition).

```
var appointmentsForDates = data.Appointments.FindAll(
  a => a.StartTime >= startDate && a.EndTime <= endDate
);

foreach (var appointment in appointmentsForDates)
{
  // Do something with each appointment
}
```

Multi-Line Text

- Longer string-formatting statements with newlines should be formatted using the verbatim strings (`@""`) and should avoid using concatenation:

```
result.SqlText = String.Format(
  @"FROM person
      LEFT JOIN
        employee
          ON person.employee_id = employee.id
      LEFT JOIN
        company
          ON person.company_id = company.id
      LEFT JOIN
        program
          ON company.program_id = program.id
      LEFT JOIN
        settings
          ON settings.program_id = program.id
    WHERE
      program.id = {0} AND person.hire_date <= '{2}';
  ",
  settings.ProgramId,
  state,
  offset.ToString("yyyy-MM-dd")
);
```

- If the indenting in the string argument above is important, you may break indenting rules and place the text all the way to the left of the source in order to avoid picking up extra, unwanted spaces. However, you should consider externalizing such text to resources or text files.
- The trailing double-quote in the example above is not required, but is permitted; in this case, the code needs to include a newline at the end of the SQL statement.

`return` Statements

- If a `return` statement is not the only statement in a method, it should be separated from other code by a single newline.
- Always use multi-line formatting for `return` statements so they're easy to see.

```
if (Count != other.Count)
{
  return false;
}
```

- Do *not* use else with `return` statements.

```
if (a == 1)
{
  return true;
}
else  // Not necessary
{
  return false;
}
```

  Instead, you should write the `return` as shown below.

```
if (a == 1)
{
  return true;
}

return false;
```

  In this case, return the condition instead.

```
return a == 1;
```

`switch` Statements

The following rules apply for all `switch` statements, including pattern-matching.

- Contents under `switch` statements should be indented.
- Braces for a case-label are not indented; this maintains a nice alignment with the brackets from the switch-statement.
- Use braces for longer code blocks under case-labels; leave a blank line above the break-statement to improve clarity.

```
switch (flavor)
{
  case Flavor.Up:
  case Flavor.Down:
  {
    if (someConditionHolds)
    {
      // Do some work
    }

    // Do some more work

    break;
  }
  default:
    break;
}
```

- Use braces to enforce tighter scoping for local variables used for only one `case`-label.

```
switch (flavor)
{
  case Flavor.Up:
  case Flavor.Down:
  {
    int quarkIndex = 0; // valid within scope of case statement
    break;
  }
  case Flavor.Charm:
  case Flavor.Strange:
    int isTopOrBottom = false;  // valid within scope of switch statement
    break;
  default:
    break;
}
```

- If brackets are used for a case-label, the break-statement should go inside those brackets so that the bracket provides some white-space from the next case-label.

```
switch (flavor)
{
  case Flavor.Up:
  case Flavor.Down:
  {
    int quarkIndex = 0;
    break;
  }
  case Flavor.Charm:
  case Flavor.Strange:
  {
    int isTopOrBottom = false;
    break;
  }
  default:
  {
    handled = false;
    break;
  }
}
```

Ternary and Coalescing Operators

- Do not use line-breaking to format statements containing ternary and coalescing operators; instead, convert to an `if` / `else` statement.
- Do not use parentheses around the condition of a ternary expression. If the condition is not immediately recognizable, extract it to a local variable.

```
return _value != null ? _value.ToString() : "NULL";
```

- Prefix operators (e.g. `!`) and method calls should not have parentheses around them.

```
return !HasValue ? Value.ToString() : "EMPTY";
```

Comments

- Place comments above referenced code.
- Indent comments at the same level as referenced code.

Regions

- Do not use regions.

- A historical usage of `#region` tags is to delineate code regions to be ignored by ReSharper in generated files. Instead, tell ReSharper to ignore files with the pattern of the generated filenames (e.g. `*.Class.cs`).

Usage

Structure

Assemblies

- Use a separate assembly to improve decoupling and reduce dependencies.
- Top-level application assemblies should have as little code as possible. Most logic goes in class libraries.

The example below illustrates the projects for a solution called “Calculator” with a *WPF* application, a web-API application and a console application.

- `Calculator.Core`
- `Calculator.Core.Web`
- `Calculator.Core.Wpf`
- `Calculator.Web.Api`
- `Calculator.Wpf`
- `Calculator.Console`

The first three define libraries of functionality that is used by the next four applications. The server and console only use the `Calculator.Core` library whereas the *Winform* and *WPF* applications use their respective libraries. Separating the renderer-dependent code into a separate library makes it much easier to add another application using the same renderer but performing a slightly different task. Only highly application-dependent code should be defined directly in an application project.

Files

- Place each type (classes, interfaces, enums, etc.) in a separate file.
- Each file should include a header, with the following form:

  ```
  // <copyright file="Filename.cs" company="Encodo Systems AG">
  //   Copyright (c) 2017 Encodo Systems AG. All rights reserved.
  // </copyright>
  // <license>
  //   This file is subject to the terms and conditions defined in the 'LICENSE' file.
  // </license>
  ```

  A solution should include licensing conditions in a file named `LICENSE`.
- Namespace `using` statements should go at the very top of the file, just after the header and just before the namespace declaration.
- The namespaces at the top of the file should be in alphabetical order, with the exception that `System.*` assemblies come first.

Namespaces

Referencing

- Do not use the global namespace.
- Avoid fully-qualified type names; use the `using` statement instead.

- Use aliases to resolve ambiguities.
- Use static classes where they improve clarity of code.

Defining

- Put abstract/high-level types in outer namespaces (e.g. `Encodo.Quino.Data`).
- Put concrete types in inner ones (e.g. `Encodo.Quino.Data.Ado`).
- Group types in specific namespaces.
- Avoid deep hierarchies, as they are more difficult to browse and understand.
- For general-use types, it's OK to use “Utilities”, “Core” or “General”.
- Refactor types into new namespaces if a clear presents itself.

Types

Classes

- Declare at most one field per line.
- Do not use public or protected fields; use properties instead.

Abstract Classes

- Define constructors for abstract classes as `protected`.
- Consider providing a partial implementation of an abstract class that handles some of the abstraction in a standard way; implementors can use this class as a base and avoid having to repeat code in their own implementations.

Static Classes

- Use static classes only for extension methods *or* constants, but not both.
- Group functionality into logical static classes.

Inner Classes

- Inner classes should be `private` or `protected`.
- Inner types should not replace namespaces for organization.
- Use nested types if the inner type is logically within the other type (e.g. a `TableOfContents` class may have an `Options` inner class or a `Builder` inner class).
- Use an inner class to group private or protected constants.

Alternatives to consider:

- Consider using composition to inject the class instead, to allow customization and testing.
- Consider *local methods* as an alternative to a `private` class.

Partial Classes

To control file size, partial classes can be useful to separate

- Generated code.
- `private` or `protected` inner classes.
- larger blocks of `private` or `protected` methods.

Interfaces

Design

- Use interfaces to clearly define your API surface, separate from implementation.

- Use interfaces so that tests can mock behavior and test more precisely.
- Remove interfaces that are not used outside of testing code.
- Always create an interface for composed objects (i.e. those that are injected into other constructors) to ease testing and mocking.
- An interface should be as concise as possible. This allows consumers to precisely mock or override functionality.
- Remember the single-responsibility principle.
- If the standard implementation of a method can always be written in terms of other interface methods, consider defining an extension method for the interface instead. This is a nice way of providing a default implementation for all implementors.
- Avoid *marker* interfaces. Attributes are a more appropriate way to mark types without *changing* the type.

Usage

- Avoid similar interfaces that cause confusion as to which one should be used where. Re-use interfaces wherever possible and appropriate.
- Provide a standard implementation or an abstract base. This provides both an implementation example and some protection from future changes to the interface.
- Use explicit interface implementation where appropriate to avoid expanding a class API unnecessarily.

`structs`

Consider defining a structure instead of a class if most of the following conditions apply:

- Instances of the type are small (16 bytes or less) and commonly short-lived.
- The type is commonly embedded in other types.
- The type logically represents a single value and is similar to a primitive type, like an `int` or a `double`.
- The type is immutable.
- The type will not be boxed frequently.

Use the following rules when defining a `struct`.

- Avoid methods; at most, have only one or two methods other than equality overrides and operator overloads.
- Provide parameterized constructors for initialization.
- Overload operators and equality as expected; implement `IEquatable` instead of overriding `Equals` in order to avoid the negative performance impact of boxing and un-boxing the value.
- A `struct` should be valid when uninitialized so that consumers can declare an instance without calling a constructor.
- Public fields are allowed (even encouraged) for structures used to communicate with external APIs through unmanaged code.

Generics

- Use generic collection types (e.g. use `IList<T>` instead of `IList`).
- Use generic constraints instead of casting or using the `is`-operator.
- Use interfaces as generic constraints wherever possible.
- When inheriting from both a generic and non-generic interface (e.g. `IEnumerable` and `IEnumerable<T>`), implement the non-generic version explicitly and implement it using the generic interface.

`enums`

Design

- Use enumerations for strongly typed sets of values
- Use a singular name (e.g. `MigrationPhase` instead of `MigrationPhases`)

- Use enumerations for a list of constants that is not logically open-ended; otherwise, use a `static` class with constants so that consumers can extend the list.
- Use the default type of `Int32` whenever possible.
- Do not include sentinel values, such as `FirstValue` or `LastValue`.
- The first value in an enumeration is the default; make sure that the most appropriate simple enumeration value is listed first.
- Do not assign explicit values except to enforce specific values for storage in a database or to match an external API.

Usage

- Enumerations are like interfaces; be extremely careful of changing them when they are already included in code that is not under your control (e.g. used by a framework that is, in turn, used by external application code). If the enumeration must be changed, use the `ObsoleteAttribute` to mark members that are no longer in use.

Bit-sets

- Use the `[Flags]` attribute to make a bit-set instead of a simple enumeration.
- Use plural names for bit-sets.
- Assign explicit values for bit-sets in powers of two; use hexadecimal notation.
- The first value of a bit-set should always be `None` and equal to `0x00`.
- In bit-sets, feel free to include commonly-used aliases or combinations of flags to improve readability and maintainability. One such common value is `All`, which includes all available flags and, if included, should be defined last. For example:

  ```
  [Flags]
  public enum QuerySections
  {
    None = 0x00,
    Select = 0x01,
    From = 0x02,
    Where = 0x04,
    OrderBy = 0x08,
    NotOrderBy = All & ~OrderBy,
    All = Select | From | Where | OrderBy,
  }
  ```

  The values `NotOrderBy` and `All` are aliases defined in terms of the other values. Note that the elements here are not aligned because it is expected that they will be documented, in which case column-alignment won't make a difference in legibility.
- Avoid designing a bit-set when certain combinations of flags are invalid; in those cases, consider dividing the enumeration into two or more separate enumerations that are internally valid.

Members

Modifiers

- The visibility modifier is required for all types, methods and fields; this makes the intention explicit and consistent.
- The visibility keyword is always the first modifier.
- The `const` or `readonly` keyword, if present, comes immediately after the visibility modifier.
- The static keyword, if present, comes after the visibility modifier and readonly modifier.

  ```
  private readonly static string DefaultDatabaseName = "admin";
  ```

`sealed`

- Do not declare protected or virtual members on sealed classes
- Avoid sealing classes unless there is a very good reason for doing so (e.g. to improve reflection performance).
- Consider sealing only selected members instead of sealing an entire class.
- Consider sealing members that you have overridden if you don't want descendants to avoid your implementation.

`internal`

- Wherever possible, make implementation classes `internal` to reduce the surface area of the API.
- Prefer `private` and `protected` to `internal` methods.
- Instead of using `internal` to mean *assembly-local*, use composition to provide access to shared functionality

Most historical uses for `internal` methods can be implemented with other, better patterns. One use is to allow overriding of a method inside an assembly, but not outside. Two questions: why are you overriding instead of composing? And, if it's useful for your library or framework to be able to override, why deny this to consumers?

Declaration Order

- Constructors, in descending order of complexity
- public constants
- public properties
- public methods
- protected constants
- protected properties
- protected methods
- private constants
- private properties
- private methods
- private fields

Constants

- Declare all constants other than `0`, `1`, `true`, `false` and `null`.
- Use `true` and `false` only for assignment, never for comparison.
- Avoid passing `true` or `false` for parameters; use an `enum` or constants to impart meaning instead.
- If there is a logical connection between two constants, indicate this by making the initialization of one dependent on the other.

```
public const int DefaultCacheSize = 25;
public const int DefaultGranularity = DefaultCacheSize / 5;
```

- Use `const` only when the value really is constant (e.g. `NumberDaysInWeek`); otherwise, use `readonly`.
- Use `readonly` as much as possible.

Constructors

- Do not include a call to the default `base()`
- Avoid doing more than setting properties in a constructor; provide a well-named method on the class to perform any extra work after the object has been constructed.
- Avoid calling virtual methods from a constructor. The initialization order for constructors can lead to crashes. The example below illustrates this problem, where the override `CaffeineAddict.GoToWork()` uses Coffee before it has been initialized.

```
public abstract class Employee
{
  public Employee()
```

```
  {
    Notify();
  }

  protected abstract void Notify();
}

public class CaffeineAddict : Employee
{
  public CaffeineAddict([NotNull] Employee boss)
    : base()
  {
    if (boss == null) { throw new ArgumentNullException(nameof(beverage)); }

    Boss = boss;
  }

  [NotNull]
  public Employee Boss { get; }

  protected override void Notify()
  {
    // Crashes when called from the constructor
    Boss.Notify();
  }
}
```

- Constructors should "funnel" so that initialization code is written only once. For example:

```
protected Query()
{
  _restrictions = new List<IRestriction>();
  _sorts = new List<ISort>();
}

public Query([NotNull] IMetaClass model)
  : this()
{
  if (model == null) { throw new ArgumentNullException(nameof(model)); }

  Model = model;
}

public Query([NotNull] IDataRelation relation)
  : this()
{
  if (relation == null) { throw new ArgumentNullException(nameof(relation)); }

  Relation = relation;
}
```

Properties

- Prefer immutable properties.
- Prefer automatic properties.
- Prefer getter-only auto-properties.
- Prefer auto-property initializers.

Declaration

For example, the first version below is not only verbose, but B is mutable (within the class):

```
// Do not use this style
public class A
{
  A()
```

```
  {
    B = 1;
  }

  int B { get; private set; }
}
```

This is also quite verbose, but B is now read-only:

```
public class A
{
  int B
  {
    get { return _b; }
  }

  private readonly int _b = 1;
}
```

Finally, this is the recommended syntax (C#6 and higher):

```
public class A
{
  int B { get; } = 1;
}
```

Commutativity

Properties should be commutative; that is, it should not matter in which order you set them. Avoid enforcing an ordering by using a method to execute code that you would want to execute from the property setter. The following example is incorrect because setting the password before setting the name causes a login failure.

```
class SecuritySystem
{
  private string _userName;

  public string UserName
  {
    get { return _userName; }
    set { _userName = value; }
  }

  private int _password;

  public int Password
  {
    get { return _password; }
    set
    {
      _password = value;
      LogIn();
    }
  }

  protected void LogIn()
  {
    IPrincipal principal = Authenticate(UserName, Password);
  }

  private IPrincipal Authenticate(string UserName, int Password)
  {
    // Authenticate the user
  }
}
```

Instead, you should take the call LogIn() out of the setter for Password and make the method public, so the class can be used like this instead:

```
var system = new SecuritySystem()
{
  Password = "knock knock";
  UserName = "Encodo";
}
system.LogIn();
```

In this case, Password can be set before the UserName without causing any problems.

Indexers

- Avoid indexers. They are difficult to navigate, even with good tools. Use well-named methods instead (e.g. `Get*()` and `Set*()`).
- Provide an indexed property only if it really makes sense.
- Indexes should be 0-based.

Methods

- Methods should not exceed a cyclomatic complexity of 20.
- Prefer private methods.
- Use the same names and positions for parameters shared by similar methods.
- Avoid returning `null` for methods that return collections or strings. Instead, return an empty collection (declare a static empty list) or an empty string (`String.Empty`).
- Methods that are explicitly left empty should be marked with NOP:

  ```
  protected override void DoBeforeSave()
  {
    // NOP
  }
  ```

Extension Methods

- Use extension methods for methods that can be defined in terms of public members of that interface.
- Do so only for methods for which the implementation is certain to be the same for all implementations. That is, do not restrict an implementation's efficiency because an extension instead of interface method was used.
- Do not extend `object` or `string` in commonly used namespaces.
- Do not mix extension methods with other static methods. If a class contains extension methods, it should contain only extension methods and private support methods.
- Do not mix extension methods for different types in one class.
- Define useful, but more rarely used extension methods in a separate namespace or assembly to force callers to "opt in".

Bodies

- Do not *make decisions* in extension methods. Instead, declare components and inject them where needed.
- Do not use static code that does *make decisions* in extension methods.
- Do not use a global service locator in extension methods.
- Do not pass an IOC to extension methods.

Parameters

- Use at most 5 parameters per method. Otherwise, use a `class`, `struct` or `tuple`.
- Use at most 1 `out` or `ref` parameter. Otherwise, return a `class`, `struct` or `tuple`.

- A `ref` and `out` parameter belongs at the end of the list of non-optional parameters.
- Use the same name for parameters in interface implementations or overrides.
- Avoid re-assigning the value of a parameter. Instead, use a local variable.
- A valid re-assignment is for nullable parameters, of the form shown below:

```
void Apply([NotNull] IMigrationPlan plan, [CanBeNull] ILogger logger = null)
{
  if (plan != null) { throw new ArgumentNullException(nameof(plan)); }

  logger = logger ?? NullLogger.Default;

  // ...
}
```

Optional Parameters

- Do not use optional parameters in constructors.
- Do not use optional parameters that might change in public APIs; instead, use overloaded methods.
- Do not use more than one or two optional parameters, even for internal APIs.

This blog post Optional argument corner cases, part four by Eric Lippert discusses the problem in more detail.

> [Optional arguments can lead to] fairly serious versioning issue[s]. [...] The lesson here is to think carefully about the scenario with the long term in mind. If you suspect that you will be changing a default value and you want the callers to pick up the change without recompilation, don't use a default value in the argument list; make two overloads, where the one with fewer parameters calls the other.

Expression-bodied Members

- Use expression-bodied members for simple properties and methods.
- The same rules apply as for any other expression; use a standard property or method body for complex logic.
- Do not use expression-bodied members for constructors and finalizers.

`tuples`

- Do not use tuples with more than 3 fields.
- In C# 7 or higher, consider using a `Tuple<T, bool>` instead of an `out` parameter.
- In C# 6 or lower, use the Try* pattern because tuple fields cannot have names.
- Avoid `tuple` return types for public APIs

Naming

- Always name the parameters in a tuple.
- If the method returns a single constant tuple, then specify the names there (to keep the method declaration shorter). If there are several exit points, don't repeat the names in the literal tuples; instead, include the names only in the return-type declaration.
- Prefer external `var (first, second, third)` declaration to the internal one `(var first, var second, var third)`

Deconstruction

- Provide a custom `deconstruct` method for structs.
- Match the field order in a class's constructor, ToString override, and Deconstruct method.
- Use deconstruction where appropriate to consume tuples. If the tuple has named members, then you can just use it; otherwise, use deconstruction to assign the members to variables with logical names.

Overloads

- Use overloads for methods that have similar behavior. Do not include parameter names in the method name. For example, the following is incorrect

```
void Update();
void UpdateUsingQuery(IQuery query);
void UpdateUsingSql(string sql);
```

  The overloaded version below reduces the perceived size of the API and makes it easier to understand.

```
void Update();
void Update(IQuery query);
void Update(string sql);
```

- Avoid putting a lot of logic in overloaded methods.
- Try to make overloads "funnel" to a single overload or other method.
- Make at most one overload `virtual`. For example:

```
public void Update()
{
  Update(QueryTools.NullQuery);
}

public void UpdateUsingSql(string sql)
{
  Update(new Query(sql));
}

public virtual void Update(IQuery query)
{
  // Perform update
}
```

Virtual

- `virtual` is a code smell; consider *composition* as an alternative.
- Avoid `public virtual` methods, but do not create an extra layer of method call either.
- If a method has logical pre-conditions or post-conditions, consider wrapping a `protected virtual` method in a `public` method, as shown below:

```
public void Update(IQuery query)
{
  if (query == null) { throw new ArgumentNullException(nameof(query)); }
  if (!query.Valid) { throw new ArgumentException("Query is not valid.",  nameof(query)); }
  if (!query.Updatable) { throw new ArgumentException("Query is not updatable.",  nameof(query)); }

  DoUpdate(query);

  if (!query.UpToDate) { throw new ArgumentException("Query should have been updated.",  nameof(query))
}

protected virtual void DoUpdate(IQuery query)
{
  // Perform update
}
```

  Use a `Do` or `Internal` prefix for these methods.
- Wrap multiple parameters in an "arguments" class to avoid changing the signature when more data is needed in future versions.

`new` Properties

- Do not use the `new` keyword to force overrides; use `override` instead or restructure the code to avoid it.

Event Handlers

Be aware of the following when raising events.

- Event handlers can affect performance.
- Event handlers can change the calling object.
- Event handlers can throw exceptions.
- Event handlers are not guaranteed to run in the calling thread.

Alternatives

- Do not use event handlers other than in legacy code (e.g. Winform)
- For user interfaces, use the MVVM pattern instead
- For back-end objects, use messaging or event-aggregator patterns

Rules

- Declare events using the `event` keyword.
- Do not use delegate members.
- Use delegate inference instead of writing `new EventHandler(…)`.
- Declare events with `EventHandler<T>`.
- An event has two parameters named `sender` of type `object` and `args` with an event-specific type.
- Do not allow `null` for either the `sender` or the `args` parameters.
- Use `EventArgs` as the base class for custom arguments.
- Use `CancelEventArgs` as the base class if you need to be able to cancel an event.
- Custom arguments should include only properties, but no logic.

Race conditions

To avoid null-reference exceptions, get a reference to the handler in a local variable before checking it and calling it. The so-called "elvis" operator in C# 6 and higher is recommended.

```
protected virtual void RaiseMessageDispatched()
{
  MessageDispatched?.(this, EventArgs.Empty);
}
```

For C# 5 and lower, use:

```
protected virtual void RaiseMessageDispatched()
{
  EventHandler handler = MessageDispatched;
  if (handler != null)
  {
    handler(this, EventArgs.Empty);
  }
}
```

The following code is an example of a simple event handler and receiver.

```
public class Safe
{
  public event EventHandler Locked;

  public void Lock()
  {
    // Perform work
```

```
    RaiseLocked(EventArgs.Empty);
  }

  protected virtual void RaiseLocked(EventArgs args)
  {
    Locked?.(this, args);
  }
}

public static class StoreManager
{
  private static void SendMailAboutSafe(object sender, EventArgs args)
  {
    // Respond to the event
  }

  public static void TestSafe()
  {
    Safe safe = new Safe();
    safe.Locked += SendMailAboutSafe;
    safe.Lock();
  }
}
```

Operators

Caveats

- Avoid overloading operators in general; it's not appropriate for most problem domains.
- Do not override the `==` -operator for reference types; instead, override the `Equals()` method to avoid redefining reference equality.
- Do not provide a conversion operator unless it can be logically expected by consumers of the API.

Recommendations

- If an operator is needed, re-use operator conventions from other languages or the problem domain of the API (e.g. mathematical operators)
- If you do override Equals(), you must also override `GetHashCode()`.
- If you do override the == operator, consider overriding the other comparison operators (!=, <, <=, >, >=) as well.
- You should return `false` from the `Equals()` function if the objects cannot be compared. However, if they are different types, you may throw an exception.
- Do not mix and match conversion operators and types. The type `DynamicString` can convert to `System.String` but should not convert to `System.Int32`. Instead, use a constructor to initialize from types that are not in the same domain.
- Use `implicit` operators *only* where the conversion *cannot* result in a loss of data or an exception.
- Use an `explicit` operator where data-loss or exceptions are possible.

`ref` Returns and Properties

`ref` returns are a feature that can improve memory-management in applications that uses larger structures. Taking references avoids copying values where not necessary.

- Use ref properties for mutable structs.
- Do not use ref properties for immutable structs or read-only classes.

Statements and Expressions

`base`

- Use `base` only from a constructor or to call a predecessor method.

- You may only call the `base` of the method being executed; do not call other `base` methods. In the following example, the call to `CheckProcess()` is not allowed, whereas the call to `RunProcess()` is.

```
public override void RunProcess()
{
  base.CheckProcess();  // Not allowed
  base.RunProcess();
}
```

`this`

- Use `this` only when referring to other constructors.
- Do not use `this` to resolve name-clashes; instead, change one of the conflicting names.

Value Types

- Always use the lower-case primitive type.
  - Use `int` instead of `Int32`
  - Use `string` instead of `String`
  - Use `bool` instead of `Boolean`
  - Use `short` instead of `Int16`
  - Use `byte` instead of `Byte`
  - Use `long` instead of `Int64`
  - And so on.

Strings

- Use `string.Empty` instead of `""`.
- Use `string.IsNullOrEmpty` instead of `s == null` or `s.Length == 0`.

Concatenation

- Prefer string-interpolation for C#6 or higher.
- Prefer `string.Format` for C#5 or lower.
- Use string-concatenation or `string.Concat` only if you have identified a performance bottleneck.
- Use a `StringBuilder` for more complex situations or when concatenation occurs over multiple statements.

Interpolation

- Prefer embedding variables rather than expressions.
- Avoid embedding longer expressions.

The following example uses short expressions and is legible.

```
var s = $"The total [{total + shipping}] is higher than the balance [{balance - fees}].";
```

However, the following interpolated string isn't very easy to read.

```
var s = $"The total [{allOrders.Sum(o => o.Total + (o.Tax * o.Vat.Rate)) + shipping}] is higher than the ba
```

Instead, extract variables to reduce complexity to the previous formulation.

```
var total = allOrders.Sum(o => o.Total + (o.Tax * o.Vat.Rate));
var balance = accounts.Sum(a => a.Balance);
var s = $"The total [{total + shipping}] is higher than the balance [{balance - fees}].";
```

`nameof`

- Use `nameof` for passing names to `ArgumentExceptions`.
- Use `nameof` wherever possible to avoid constant strings.

Resource Strings

- Use resources for all localizable strings.
- Do not use resources for strings that will not be localized (e.g. log messages)
- Resource identifiers follow the same rules as all other identifiers.
- Do not waste time localizing strings until code is reviewed and stable.

Floating Point and Integral Types

- Be careful when comparing floating-point constants; unless you are using `decimals`, the representation will have limited precision and can lead to subtle bugs.
- One exception to this rule is when you are comparing constants of fixed, known value (like `0.0` or `1.0`, but not `3.14`).
- Be careful when casting representations with different sizes (e.g. `long` to `int`).
- Use the literal `_` as a reasonable separator (e.g. to delineate hex groups or as a thousands-separator).

Local Variables

- Use `var` and initialization wherever possible.
- Declare local variables individually.
- Initialize a local variable on the same line as the declaration
- Use standard line-breaking rules outline elsewhere for longer, fluent initialization.

Local Functions

- Use local functions for short private methods that are used only once.
- If a local function body must be repeated, then use a private method instead.
- Use local functions instead of anonymous functions.
- Use local iterators when returning an IEnumerator when parameters need to be validated.
- Put local functions at the beginning or end of its containing body.

`var`

- Use `var` everywhere.

The justification is that the rest of this handbook encourages a style where:

- methods are small
- parameters and variables are well-named

So the types, where relevant, will be obvious from the relatively small and local context.

The following examples show calls without using `var`.

```
IList<Airplane> planes1 = new List<Airplane>();
IDataList<Airplane> planes2 = connection.GetList<Airplane>();
IDataList<Airplane> planes3 = hanger.GetAirplanes(connection);
```

In which cases is the type relevant or non-obvious? The following version using `var` only gains in clarity.

```
var planes1 = new List<Airplane>();
var planes2 = connection.GetList<Airplane>();
var planes3 = hanger.GetAirplanes(connection);
```

`out` variables

- Use `out`-parameter declaration to define parameters.

  ```
  if (list.TryGetValue("One", var out item))
  {
    // use 'item'
  }
  ```

  For C# 5 and older, use the standalone variable declaration.

  ```
  Item item;
  if (list.TryGetValue("One", out item))
  {
    // use 'item'
  }
  ```

Loops

- Do not change the loop variable of a for-loop.
- Update while loop variables either at the beginning or the end of the loop.
- Keep loop bodies short; avoid excessive nesting.

Conditions

- Do not compare to `true` or `false`.
- Use parentheses only if the precedence isn't relatively obvious
- Use *StyleCop* or *ReSharper* to indicate where parentheses are needed and stick to it.
- Initialize Boolean values with simple expressions rather than using an if-statement.

  ```
  bool needsUpdate = Count > 0 && Objects.Any(o => o.Modified);
  ```

- Always use brackets for flow-control blocks (`switch`, `if`, `while`, `for`, etc.)
- Do not add useless `else` blocks. An `if` statement may stand alone and an `else if` statement may be the last condition.

  ```
  if (a == b)
  {
    // Do something
  }
  else if (a > b)
  {
    // Do something else
  }
  // No final "else" required
  ```

- Do not force really complicated logic into an `if` statement; instead, use local variables to make the intent clearer. For example, imagine we have a lesson planner and want to find all unsaved lessons that are either unscheduled or are scheduled within a given time-frame. The following condition is too long and complicated to interpret quickly:

  ```
  if (!lesson.Stored && ((StartTime <= lesson.StartTime && lesson.EndTime <= EndTime) || !lesson.Schedule
  {
    // Do something with the lesson
  }
  ```

Even trying to apply the line-breaking rules results in an unreadable mess:

```
if (!lesson.Stored &&
  ((StartTime <= lesson.StartTime && lesson.EndTime <= EndTime) ||
  ! lesson.Scheduled))
{
  // Do something with the lesson
}
```

Even with this valiant effort, the intent of the ||-operator is difficult to discern. With local variables, however, the logic is much clearer:

```
bool lessonInTimeSpan = StartTime <= lesson.StartTime && lesson.EndTime <= EndTime;
if (!lesson.Stored && (lessonInTimeSpan || !lesson.Scheduled))
{
  // Do something with the lesson
}
```

`switch` Statements

- Use `throw new UnexpectedEnumException(value)` in the `default` branch. This is more semantically correct than `InvalidEnumArgumentException`, which does not allow you to indicate the unexpected value *and* erroneously suggests that the value was invalid.
- The following code is correct:

```
switch (type)
{
  case DatabaseType.PostgreSql:
    return new PostgreSqlMetaDatabase();
  case DatabaseType.SqlServer:
    return new SqlServerMetaDatabase();
  case DatabaseType.SQLite:
    return new SQLiteMetaDatabase();
  default:
    throw new UnexpectedEnumException(value);
}
```

- The `default` label must always be the last label in the statement. In C# 7, the default label is always interpreted last anyway.

`continue` Statements

- Do not use `continue`.
- The following example is not allowed.

```
foreach (var search in searches)
{
  if (!search.Path.Contains("CN="))
  {
    continue;
  }

  // Work with valid searches
}
```

Instead, use a condition to filter elements.

```
foreach (var search in searches)
{
  if (search.Path.Contains("CN="))
  {
    // Work with valid searches
```

```
  }
}
```

Even better, use `Where()` to filter elements.

```
foreach (var search in searches.Where(s => s.Path.Contains("CN=")))
{
  // Work with valid searches
}
```

`return` Statements

- Prefer multiple return statements to local variables and nesting.

```
if (specialTaxRateApplies)
{
  return CalculateSpecialTaxRate();
}

return CalculateRegularTaxRate();
```

- Compose smaller methods to avoid local variables for return values. For example, the following method uses a local variable rather than multiple returns.

```
bool result;

if (SomeConditionHolds())
{
  PerformOperationsForSomeCondition();

  result = false;
}
else
{
  PerformOtherOperations();

  if (SomeOtherConditionHolds())
  {
    PerformOperationsForOtherCondition();

    result = false;
  }
  else
  {
    PerformFallbackOperations();

    result = true;
  }
}

return result;
```

This method can be rewritten to return the value instead.

```
if (SomeConditionHolds())
{
  PerformOperationsForSomeCondition();

  return false;
}

PerformOtherOperations();

if (SomeOtherConditionHolds())
```

```
{
  PerformOperationsForOtherCondition();

  return false;
}

PerformFallbackOperations();

return true;
```

- The only code that may follow the last `return` statement is the body of an exception handler.

```
try
{
  // Perform operations

  return true;
}
catch (Exception exception)
{
  throw new DirectoryAuthenticatorException(exception);
}
```

Pattern-matching

- The same rules apply for `when` expressions as for other conditions: short expressions are fine; extract more complex logic to local or private methods.

`goto` Statements

- Do not use `goto`.

`unsafe` Blocks

- Do not use `unsafe`.

Ternary and Coalescing Operators

- Use these operators for simple expressions and results.
- Do not use these operators with long conditions and values. Instead, use local variables and/or standard conditional statements.

Null-conditional Operator

- Use the `?.`-operator only when no handling for `null` cases is required.
- Most code should not use this operator; instead, enforce non-null values.

For example, the following example is only allowed when the data comes from a dynamic source (e.g. JSON).

```
company.People?[0]?.ContactInfo?.BusinessAddress.Street
```

If the data is not dynamic, then `People`, `ContactInfo` and `BusinessAddress` should never be `null`.

`throw`-Expressions

- Only use `throw`-expressions at the end of an expression.
- Do not use `throw`-expressions anywhere else in an compound expression.
- Do not use `throw`-expressions as actual arguments.

Lambdas

- Do not make overly-complex lambda expressions; instead, define a method or use a delegate.

System.Linq

- Pay attention to the order of your Linq expressions to improve performance.
  - Filter before sorting
  - Apply filters that are likely to remove more items first
  - Apply filters with low performance impact first
- Do not use `List.Foreach()`.
- Use multiple lines and indenting to to make expressions more legible.

```
var result = elements
  .Where(e => e.Enabled)
  .Where(e => LastUsed > clock.Now.AddWeeks(-2))
  .OrderBy(e => e.LastUsed)
  .ThenBy(e => e.Name);
```

  Here `Enabled` is tested first because it's cheaper to check.
- Use Linq syntax to share temporary variables instead of re-declaring them in several lambdas.

```
var result =
  from e in elements
  where e.Enabled
  where e.LastUsed > clock.Now.AddWeeks(-2)
  orderby e.LastUsed, e.Name;
```

Casting

- Use a direct cast if you are sure of the type.

```
((IWeapon)item).Fire();
```

- Use the `is`-operator when *testing* but not *using* the result of the cast.

```
return item is IWeapon;
```

- To use the result of the cast, use an `is`-expression in C# 7 and higher.

```
if (item == null) { throw new ArgumentNullException(nameof(item)); }

if (item is IWeapon weapon)
{
  return weapon.Fire();
}

return NullTurn.Default;
```

- Use a `switch` statement to match more than one or two patterns. Keep the argument precondition separate (even though it *could* be the penultimate `case`).

```
if (item == null) { throw new ArgumentNullException(nameof(item)); }

switch (item)
{
  case IWeapon weapon:
    return weapon.Fire();
  case IMagic magic:
    return magic.Cast();
  default:
```

```
    return NullTurn.Default;
}
```

- In C# 6 and lower, use the `as`-operator.

```
if (item == null) { throw new ArgumentNullException(nameof(item)); }

var weapon = item as IWeapon;
if (weapon != null)
{
  return weapon.Fire();
}

var magic = item as IMagic;
if (magic != null)
{
  return magic.Cast();
}

return NullTurn.Default;
```

checked

- Enable range-checking during development and debugging.
- Disable range-checking in release builds only if there is a valid performance reason for doing so.
- Use explicit `checked` blocks for overflow- and underflow-prone operations. I.e. if there was a range-checking problem at some point, the block should be marked with a `checked` block. This guarantees checking for these blocks even when range-checking is disabled.

Compiler Variables

- Avoid using compiler variables.
- Avoid using `#define` in the code; use a compiler define in the project settings instead.
- Avoid suppressing compiler warnings.

The [Conditional] Attribute

Use the `ConditionalAttribute` instead of the `#ifdef` / `#endif` pair wherever possible (i.e. for methods or classes).

```
public class SomeClass
{
  [Conditional("TRACE_ON")]
  public static void Msg(string msg)
  {
    Console.WriteLine(msg);
  }
}
```

#if/#else/#endif

For other conditional compilation, use a static method in a static class instead of scattering conditional options throughout the code.

```
public static class EncodoCompilerOptions
{
  public static bool DeveloperBuild()
  {
#if ENCODO_DEVELOPER
    return true;
#else
    return false;
#endif
```

```
    }
}
```

This approach has the following advantages:

- The compiler checks all code paths instead of just the one satisfying the current options; this avoids unknowingly retaining incompatible code in a library or application.
- Code formatting and indenting is not broken up by (possibly overlapping) compile conditions; the name `EncodoCompilerOptions` makes the connection to the compiler obvious enough.
- The compiler option is referenced only once, avoiding situations in which some code uses one compiler option (e.g. `ENCODO_DEVELOPER`) and other code uses another, misspelled option (e.g. `ENCODE_DEVELOPER`).

Comments

Styles

- Use the single-line comment style— `//` —to indicate a comment.
- Use four slashes — `////` —to indicate a single line of code that has been temporarily commented.
- Use the multi-line comment style— `/*` ... `*/` —to indicate a commented-out block of code. Do not push these comments to the master branch.
- Consider using a compiler variable to define a non-compiling block of code; this practice avoids misusing a comment.

  ```
  #if FALSE
        // commented code block
  #endif
  ```

- Use the single-line comment style with `TODO` to indicate an issue that must be addressed. Before a check-in, these issues must either be addressed or documented in the issue tracker, adding the URL of the issue to the TODO as follows:

  ```
  // TODO http://issue-tracker.encodo.com/?id=5647: Title of the issue in the issue tracker
  ```

Placement

- Longer comments should always precede the line being commented. Separate multi-line comments with an additional newline before the code.
- Short comments may appear to the right of the code being commented, but only for lines ending in semicolon (i.e. marking the end of a statement). For example:

  ```
  int Granularity = Size / 5; // More than 50% is not valid!
  ```

- Comments on the same line as code should *never* be wrapped to multiple lines.

Use Cases

- Use comments to explain algorithms or tricky bits that aren't immediately obvious from a quick read.
- Use comments to indicate where a hard-won bug-fix was added; if possible, include a reference to a URL in an issue tracker.
- Use comments to indicate assumptions not already evident from assertions or thrown exceptions.
- Comments are in US-English; prefer a short style that gets right to the point.
- A comment need not be a full, grammatically-correct sentence. For example, the following comment is too long

  ```
  // Using a granularity that is more than 50% of the size is not valid!
  int Granularity = Size / 5;
  ```

  Instead, you should stick to the essentials so that the warning is immediately clear:

```
int Granularity = Size / 5; // More than 50% is not valid!
```

- Comments should not explain the obvious. In the following example, the comment is superfluous.

```
public const int Granularity = Size / 5; // granularity is 20% of size
```

Design

Design decisions should not be made alone. You should apply these principles to come up with a design, but should always seek the advice and approval of at least one other team member before proceeding.

Abstractions

The first rule of design is: don't overdo it (YAGNI). Overdesign leads to a framework that offers unused functionality and has interfaces that are difficult to understand and implement. Only create abstractions where there will be more than one implementation or where there is a reasonable need to provide for other implementations in the future.

This leads directly to the second rule of design: "don't under-design". Understand your problem domain well enough before starting to code so that you accommodate reasonably foreseeable additional requirements. For example, you need to figure out whether multiple implementations will be required (in which case you should define interfaces) and whether any of those implementations will share code (in which case abstract interfaces or one or more abstract base classes are in order). You should create abstractions where they prevent repeated code—applying the DRY principle—or where they provide decoupling.

If you do create an abstraction, make sure that there are tests which run against the abstraction rather than a concrete implementation so that all future implementations can be tested. For example, database access for a particular database should include an abstraction and tests for that abstraction that can be used to verify all supported databases.

Inheritance vs. Composition

The rule here is to only use inheritance where it makes semantic sense to do so. If two classes could share code because they perform similar tasks, but aren't really related, do not give them a common ancestor just to avoid repeating yourself. Extract the shared code into a helper class and use that class from both implementations. Prefer composition of instances over `static` helpers.

Interfaces vs. Abstract Classes

Whether or not to use interfaces is a hotly-debated topic. On the one hand, interfaces offer a clean abstraction and "interface" to a library component and, on the other hand, they restrict future upgrades by forcing new methods or properties on existing implementations. In a framework or library, you can safely add members to classes that have descendants in application code without forcing a change in that application code. However, abstract methods—which are necessary for very low-level objects because the implementation can't be known—run into the same problems as new interface methods. Creating new, virtual methods with no implementation to avoid this problem is strongly frowned upon, as it fails to properly impart the intent of the method.

One exception to this rather strictly enforced rule is for classes that simply cannot be abstract. This will be the case for user-interface components that interact with a visual designer. The Visual Studio designer, for example, requires that all components be non-abstract and include a default constructor in order to be used. In these cases, empty virtual methods that throw a `NotImplementedException` are the only alternative. If you must use such a method, include a comment explaining the reason.

Where interfaces can also be very useful is in restricting write-access to certain properties or containers. That is, an interface can be declared with only a getter, but the implementation includes both a getter and setter. This allows an

application to set the property when it works with an internal implementation, but to restrict the code receiving the interface to a read-only property.

Open vs. Closed APIs

This section used to be called “Virtual Methods” but could just as easily have been called “Virtual vs. Regular Methods” or “Protected vs. Private Methods” or “Sealed Classes and Methods vs. anyone using your APIs”. That last one was—kind of—a joke.

The point is that any element that is exposed to other code imposes a maintenance burden of some kind. Public and protected elements:

- Must be documented
- Must be tested (either with automated tests or manual testing)
- Must be properly designed (private members don’t need as much attention)
- Can only expose public types
- Cannot be refactored as mercilessly

By nature, C# is more closed in this regard because classes are internal and methods are non-virtual by default. This makes for APIs that require less maintenance but are also less open to other coders. This can be a very frustrating experience for those coders when they inherit from these classes only to find that misbehaving methods cannot be overridden, vital functionality is hidden inside private or virtual methods or classes are sealed so as to prevent inheritance entirely.

Very often, this leads to code duplication as entire swaths of code are copied via reverse-engineering just in order to apply a minor tweak. A historical case is the Silverlight framework from Microsoft where many classes are sealed and many methods are internal or private and very little of the API can be overridden.

Naturally, it would be nice to avoid frustrating other coders in this way, but making everything public or protected and virtual is also not the solution. Such blanket approaches fail to guide users of your API sufficiently. It is up to you to find a happy medium, exposing exactly the functionality that any user might need: no more, no less. Naturally, this will force you to actually think about the purpose of the class that you are writing and consider the use-cases for it.

With use-cases in mind, here are some points to consider when building a class.

- What are the odds that anyone will want to create a descendant of your class?
- If these odds are non-zero, do not seal the class.
- Does your class include functionality that descendants will want to reuse? In that case, make the method or property protected.
- Is it possible that descendants will want to change how that functionality works? In that case, make the method or property virtual.
- Anything else should be made private in order to avoid exposing too large an interface to both users of the public interface and descendants, which use the protected interface.

Controlling API Size

- Be as stingy as possible when making methods public; smaller APIs are easier to understand.
- If another assembly needs a type to be public, consider whether that type could not remain internalized if the API were higher-level. Use the Object Browser to examine the public API.
- To this end, frameworks that are logically split into multiple assemblies can use the `InternalsVisibleTo` attributes to make “friend assemblies” and avoid making elements public. Given three assemblies, `Quino`, `QuinoWinform` and `QuinoWeb` (of which a standard Windows application would include only the first two), the `Quino` assembly can make its internals visible to `QuinoWinform`.

Read-only Interfaces

One advantage in using interfaces over abstract classes is that interfaces can enforce immutability even though the backing implementation is mutable.

The following example illustrates this principle for properties:

```
interface IMaker
{
  bool Enabled { get; }
}

class Maker : IMaker
{
  bool Enabled { get; set; }
}
```

The principle extends to making read-only sequences as well, using `IEnumerable<T>` in the interface and exposing `IList<T>` in the backing class, using explicit interface implementation, as shown below:

```
interface IProcessedCommands
{
  IEnumerable<ISwitchCommand> SwitchCommands { get; }
}

class ProcessedCommands
{
  IEnumerable<ISwitchCommand> IProcessedCommands.SwitchCommands
  {
    get { return SwitchCommands.ToList(); }
  }

  IList<ISwitchCommand> SwitchCommands { get; private set; }
}
```

In this way, the client of the interface cannot call `Add()` or `Remove()` on the list of commands, but the provider has the convenience of doing so as it prepares the backing object.

Using Linq, the client is free to convert the result to a list using `ToList()`, but will create its own copy, leaving the `SwitchCommands` on the object represented by the interface unaffected.

Note, though, that the implementation of `IProcessedCommands.SwitchCommands` calls `ToList()` to avoid passing its internal representation back to the caller. If it didn't do this, then caller could alter the internal state with the following code.

```
var processedCommands = GetProcessedCommands();
var switches = processedCommands.SwitchCommands as IList<ISwitchCommand>;
if (switches != null)
{
  switches.Clear();
}
```

Single-Responsibility Principle

Design types so that the caller cannot use them incorrectly.

The example below shows a typical class indicates usage in documentation rather than the API.

```
/// <remarks>
/// Make sure to set the <see cref="ServerName">, <see cref="UserName">
/// and <see cref="Password"> before calling <see cref="Connect">. Call <see cref="Connect"> before calling
/// </remarks>
public interface IBackEnd
```

```
{
  string ServerName { get; set; }
  string UserName { get; set; }
  string Password { get; set; }
  void Connect();
  void LogIn();
  void RunTask([NotNull] ITask task);
}

internal static Main()
{
  var backEnd = CreateBackEnd();
  var tasks = GetTasks();

  backEnd.ServerName = GetServerName();
  backEnd.UserName = GetUserName();
  backEnd.Password = GetPassword();
  backEnd.Connect();
  backEnd.LogIn();

  foreach (var task in tasks)
  {
    backEnd.RunTask(task);
  }
}
```

The block in `Main()` *should* always look the same, but the pattern is not enforced.

Instead of a single `IBackEnd` type with multiple methods, move the configuration parameters to separate objects and define *three* interfaces, each of which has a single responsibility:

```
interface IDisconnectedBackEnd
{
  [NotNull]
  IConnectedBackEnd Connect([NotNull] IConnectionSettings settings);
}

interface IConnectedBackEnd
{
  [NotNull]
  ILoggedInBackEnd LogIn([NotNull] IUser user);
}

interface ILoggedInBackEnd
{
  void RunTask([NotNull] ITask task);
}

internal static Main()
{
  var settings = GetConnectionSettings();
  var user = GetUser();
  var tasks = GetTasks();

  var backEnd =
    CreateDisconnectedBackEnd()
    .Connect(settings)
    .LogIn(user);

  foreach (var task in tasks)
  {
    backEnd.RunTask(task);
  }
}
```

In this case, the code in `Main()` will still always look the same, *but it can now no longer take any other shape*. The caller *cannot* call `LogIn()` without having called `Connect()` first. With these types, a method can require a

`IConnectedBackEnd` or `ILoggedInBackEnd` rather than a generic `IBackEnd` in an unknown state.

For example, we can extract a method for running tasks where the type of the parameter enforces the state of the back end.

```
internal static Main()
{
  var settings = GetConnectionSettings();
  var user = GetUser();
  var tasks = GetTasks();

  var backEnd =
    CreateDisconnectedBackEnd()
    .Connect(settings)
    .LogIn(user);

  RunTasks(backEnd, tasks);
}

private static void RunTasks([NotNull] ILoggedInBackEnd backEnd, [NotNull] IEnumerable<ITask> tasks)
{
  if (backEnd != null) { throw new ArgumentNullException(nameof(backEnd)); }
  if (tasks != null) { throw new ArgumentNullException(nameof(tasks)); }

  foreach (var task in tasks)
  {
    backEnd.RunTask(task);
  }
}
```

Loose vs. Tight Coupling

Whether to use loose or tight coupling for components depends on several factors. If a component on a lower-level must access functionality on a higher level, then you're doing something wrong.

Callbacks vs. Interfaces

Both callbacks and interfaces can be used to connect components in a loosely-coupled way. A delegate is more loosely-coupled than an interface because it specifies the absolute minimum amount of information needed in order to inter-operate whereas an interface forces the implementing component to satisfy a set of clearly-defined functionality.

If the bridge between two components is truly that of an event sink communicating with an event listener, then you should use event handlers and delegates to communicate. However, if you start to have many such delegate connections between two components, you'll want to improve clarity by defining an interface to more completely describe this relationship.

Another consideration is the prevailing model of the surrounding components. For example, the Windows Forms components and designer use events exclusively and so should your components if they are to integrate with the designer.

In other cases, where you have a handful of events that are highly interrelated, it is useful to create an interface so that components can be sure that they are handling the events correctly (and not forgetting to handle one or the other).

An Example

If the component on the higher level needs to be coupled to a component on a lower level, then it's possible to have them be more tightly coupled by using an interface. The advantage of using an interface over a set or one or more callbacks is that changes to the semantics of how the coupling should occur can be enforced. The example below should make this much clearer.

Imagine a class that provides a single event to indicate that it has received data from somewhere.

```
public class DataTransmitter
{
  public event EventHandler<DataBundleEventArgs> DataReceived;
}
```

This is the class way of loosely coupling components; any component that is interested in receiving data can simply attach to this event, like this:

```
public class DataListener
{
  public DataListener(DataTransmitter transmitter)
  {
    transmitter.DataReceived += TransmitterDataReceived;
  }

  private TransmitterDataReceived(object sender, DataBundleEventArgs args)
  {
    // Do something when data is received
  }
}
```

Another class could combine these two classes in the following, classic way:

```
var transmitter = new DataTransmitter();
var listener = new DataListener(transmitter);
```

The transmitter and listener can be defined in completely different assemblies and need no dependency on any common code (other than the .NET runtime) in order to compile and run. If this is an absolute *must* for your component, then this is the pattern to use for all events. Just be aware that the loose coupling may introduce *semantic* errors—errors in usage that the compiler will not notice. For example, suppose the transmitter is extended to include a new event, NoDataAvailableReceived.

```
public class DataTransmitter
{
  public event EventHandler<DataBundleEventArgs> DataReceived;
  public event EventHandler NoDataAvailableReceived;
}
```

Let's assume that the previous version of the interface threw a timeout exception when it had not received data within a certain time window. Now, instead of throwing an exception, the transmitter triggers the new event instead. The code above will no longer indicate a timeout error (because no exception is thrown) nor will it indicate that no data was transmitted.

One way to fix this problem (once detected) is to hook the new event in the `DataListener` constructor. If the code is to remain highly decoupled—or if the interface cannot be easily changed—this is the only real solution.

Imagine now that the transmitter becomes more sophisticated and defines more events, as shown below.

```
public class DataTransmitter
{
  public event EventHandler<DataBundleEventArgs> DataReceived;
  public event EventHandler NoDataAvailableReceived;
  public event EventHandler ConnectionOpened;
  public event EventHandler ConnectionClosed;
  public event EventHandler<DataErrorEventArgs> ErrorOccurred;
}
```

Clearly, a listener that attaches and responds appropriately to *all* of these events will provide a much better user experience than one that does not. The loose coupling of the interface thus far requires all clients of this interface to be proactively aware that something has changed and, once again, the compiler is no help at all.

If we can change the interface—and if the components can include references to common code—then we can introduce tight coupling by defining an interface with methods instead of individual events.

```
public interface IDataListener
{
  void DataReceived(IDataBundle bundle);
  void NoDataAvailableReceived();
  void ConnectionOpened();
  void ConnectionClosed();
  void ErrorOccurred(Exception exception, string message);
}
```

With a few more changes, we have a more tightly coupled system, but one that will enforce changes on clients:

- Add a list of listeners on the `DataTransmitter`
- Add code to copy and iterate the listener list instead of triggering events from the `DataTransmitter`.
- Make `DataListener` implement `IDataListener`
- Add the listener to the transmitter's list of listeners.

Now when the transmitter requires changes to the `IDataListener` interface, the compiler will required that all listeners also be updated.

Methods vs. Properties

Use methods instead of properties in the following situations:

- For transformations or conversions, like `ToXml()` or `ToSql()`.
- If the value of the property is not cached internally, but is expensive to calculate, indicate this with a method call instead of a property (properties generally give the impression that they reference information stored with the object).
- If the result is not idempotent (yields the same result no matter how often it is called), it should be a method.
- If the property returns a copy of an internal state rather than a direct reference; this is especially significant with array properties, where repeated access is very inefficient.
- When a getter is not desired, use a method instead of a set-only property.

For all other situations in which both a property and a method are appropriate, properties have the following advantages over methods:

- Properties don't require parentheses and result in cleaner code when called (especially when many calls are chained together).
- It clearly indicates that the value is a logical property of the construct instead of an operation.

Code > Comments

Good variable and method names go a long way to making comments unnecessary.

Replace comments with private methods with descriptive names. For example, the following code is commented, but a bit wordy.

```
public void MethodOne()
{
  // Collect and aggregate results
  var projections = new List<Result>();
  foreach (var p in OriginalProjections)
```

```
  {
    // Do a bunch of stuff with p
    projections.Add(new Projection(p, i))
  }

  // Format the projections into a text report
  var lines = new List<string>();
  foreach (var projection in projections)
  {
    var line = string.Format($"Some text with a {projection}");

    // Work with the line

    lines.Add(line);
  }

  // Save the lines to file
  File.WriteAllLines(OutputPath, lines);
}
```

Instead, use methods to achieve the same clarity without comments. At the same time, we make use of better types (`IEnumerable<T>`) and constructs (`Select()` , `NotNull`) to streamline and improve the code even more.

```
public void MethodOne()
{
  var projections = CalculateFinalProjections();
  var lines = CreateReport(projections);

  StoreReport(lines);
}

[NotNull]
private IEnumerable<Projection> CalculateFinalProjections()
{
  return OriginalProjections.Select(CreateFinalProjection);
}

[NotNull]
private Projection CreateFinalProjection([NotNull] Projection p)
{
  if (p == null) { throw new ArgumentNullException(nameof(p)); }

  // Do a bunch of stuff with p

  return new Projection(p, i));
}

[NotNull]
private IEnumerable<string> CreateReport([NotNull] IEnumerable<Projection> projections)
{
  if (projections == null) { throw new ArgumentNullException(nameof(projections)); }

  return projections.Select(p => string.Format($"Some text with a {p}"));
}

private void StoreReport([NotNull] IEnumerable<string> lines)
{
  if (lines == null) { throw new ArgumentNullException(nameof(lines)); }

  File.WriteAllLines(OutputPath, lines);
}
```

This version obviously needs no comments: it is now clear what `MethodOne` does. In fact, we can streamline `MethodOne` to a single line without losing legibility. Using the syntactic constructs of the language eliminates the need for many, if not all, comments.

```
public void MethodOne()
{
  StoreReport(CreateReport(CalculateFinalProjections()));
}
```

Convert to Variables

Replace comments with local variables with descriptive names.

```
// For new lessons that are within the time-span or not scheduled
if (!lesson.Stored && ((StartTime <= lesson.StartTime && lesson.EndTime <= EndTime) || !lesson.Scheduled))
```

With local variables, however, the logic is much clearer and no comment is required:

```
bool lessonInTimeSpan = StartTime <= lesson.StartTime && lesson.EndTime <= EndTime;

if (!lesson.Stored && (lessonInTimeSpan || !lesson.Scheduled)) { }
```

Safe Programming

- Use static typing wherever possible.

Be Functional

- Make data immutable wherever possible.
- Make methods pure wherever possible.

Avoid `null` references

- Make references non-nullable wherever possible.
- Use the `[NotNull]` attribute for parameters, fields and results. Enforce it with a runtime check.
- Always test parameters, local variables and fields that can be `null`.

Instead of allowing `null` for a parameter, avoid null-checks with a null implementation.

```
interface ILogger
{
  bool Log(string message);
}

class NullLogger : ILogger
{
  void Log(string message)
  {
    // NOP
  }
}
```

Local variables

- Do not re-use local variable names, even though the scoping rules are well-defined and allow it. This prevents surprising effects when the variable in the inner scope is removed and the code continues to compile because the variable in the outer scope is still valid.
- Do not modify a variable with a prefix or suffix operator more than once in an expression. The following statement is not allowed:

  ```
  items[propIndex++] = ++propIndex;
  ```

Side Effects

A side effect is a change in an object as a result of reading a property or calling a method that causes the result of the property or method to be different when called again.

- Prefer pure methods.
- Void methods have side effects by definition.
- Writing a property must cause a side effect.
- Reading a property should not cause a side effect. An exception is lazy-initialization to cache the result.

- Avoid writing methods that return results *and* cause side effects. An exception is lazy-initialization to cache the result.

"Access to Modified Closure"

`IEnumerable<T>` sequences are evaluated lazily. ReSharper will warn of multiple enumeration.

You can accidentally change the value of a captured variable before the sequence is evaluated. Since *ReSharper* will complain about this behavior even when it does not cause unwanted side-effects, it is important to understand which cases are actually problematic.

```
var data = new[] { "foo", "bar", "bla" };
var otherData = new[] { "bla", "blu" };
var overlapData = new List<string>();

foreach (var d in data)
{
  if (otherData.Where(od => od == d).Any())
  {
    overlapData.Add(d);
  }
}

Assert.That(overlapData.Count, Is.EqualTo(1)); // "bla"
```

The reference to the variable `d` will be flagged by *ReSharper* and marked as an *"access to a modified closure"*. This indicates that a variable referenced—or "captured"—by the lambda expression—closure—will have the last value assigned to it rather than the value that was assigned to it when the lambda was created.

In the example above, the lambda is created with the first value in the sequence, but since we only use the lambda once, and then always before the variable has been changed, we don't have to worry about side-effects. *ReSharper* can only detect that a variable referenced in a closure is being changed within its scope.

Even though there isn't a problem in this case, rewrite the `foreach`-statement above as follows to eliminate the *access to modified closure* warning.

```
var data = new[] { "foo", "bar", "bla" };
var otherData = new[] { "bla", "blu" };
var overlapData = data.Where(d => otherData.Where(od => od == d).Any()).ToList();

Assert.That(overlapData.Count, Is.EqualTo(1)); // "bla"
```

Finally, use library functionality wherever possible. In this case, we should use `Intersect` to calculate the overlap (intersection).

```
var data = new[] { "foo", "bar", "bla" };
var otherData = new[] { "bla", "blu" };
var overlapData = data.Intersect(otherData).ToList();

Assert.That(overlapData.Count, Is.EqualTo(1)); // "bla"
```

Remember to be aware of how items are compared. The `Intersects` method above compares using `Equals`, not reference-equality.

The following example does not yield the expected result:

```
var data = new[] { "foo", "bar", "bla" };

var threshold = 2;
var twoLetterWords = data.Where(d => d.Length == threshold);
```

```
threshold = 3;
var threeLetterWords = data.Where(d => d.Length == threshold);

Assert.That(twoLetterWords.Count(), Is.EqualTo(0));
Assert.That(threeLetterWords.Count(), Is.EqualTo(3));
```

The lambda in `twoLetterWords` *references* `threshold`, which is then changed before the lambda is evaluated with `Count()`. There is nothing wrong with this code, but the results can be surprising. Use `ToList()` to evaluate the lambda in `twoLetterWords` *before* the threshold is changed.

```
var data = new[] { "foo", "bar", "bla" };
var threshold = 2;
var twoLetterWords = data.Where(d => d.Length == threshold).ToList();

threshold = 3;
var threeLetterWords = data.Where(d => d.Length == threshold);

Assert.That(twoLetterWords.Count(), Is.EqualTo(0));
Assert.That(threeLetterWords.Count(), Is.EqualTo(3));
```

"Collection was modified; enumeration operation may not execute."

Changing a sequence during enumeration causes a runtime error.

The following code will fail whenever `data` contains an element for which `IsEmpty` returns `true`.

```
foreach (var d in data.Where(d => d.IsEmpty))
{
  data.Remove(d);
}
```

To avoid this problem, use an in-memory copy of the sequence instead. A good practice is to use `ToList()` to create the copy and to call it in the `foreach` statement so that it's clear why it's being used.

```
foreach (var d in data.Where(d => d.IsEmpty).ToList())
{
  data.Remove(d);
}
```

"Possible multiple enumeration of IEnumerable"

Suppose, in the example above, that we also want to know how many elements were empty. Let's start by extracting `emptyElements` to a variable.

```
var emptyElements = data.Where(d => d.IsEmpty);
foreach (var d in emptyElements.ToList())
{
  data.Remove(d);
}

return emptyElements.Count();
```

Since `emptyElements` is evaluated lazily, the call to `Count()` to return the result will evaluate the iterator again, producing a sequence that is now empty—because the `foreach`-statement removed them all from data. The code above will always return zero.

A more critical look at the code above would discover that the `emptyElements` iterator is triggered twice: by the call to `ToList()` and `Count()` (ReSharper will helpfully indicate this with an inspection). Both `ToList()` and `Count()` logically iterate the entire sequence.

To fix the problem, we lift the call to `ToList()` out of the `foreach` statement and into the variable.

```
var emptyElements = data.Where(d => d.IsEmpty).ToList();
foreach (var d in emptyElements)
{
  data.Remove(d);
}

return emptyElements.Count;
```

We can eliminate the `foreach` by directly re-assigning `data`, as shown below.

```
var dataCount = data.Count;
var data = data.Where(d => !d.IsEmpty).ToList();

return dataCount - data.Count;
```

The first algorithm is more efficient when the majority of item in `data` are empty. The second algorithm is more efficient when the majority is non-empty.

Error Handling

Strategies

- Prefer exceptions to return codes.
- Use a return code only where all results are valid.
- Use the `Try*`-pattern (illustrated below) to encapsulate methods that can fail.
- If errors/warnings are expected, then use an `ILogger` or similar construct to record those warnings rather than throwing and multiple catching exceptions.

Terms

The following definitions are used below:

- *errors* are thrown when handling input. An application must *handle* and *recover* from these.
- *bugs* result from programming error. An application *may not* handle or recover from these.
- An exception is *caught* with a `catch` block that matches it
- An application *re-throws* an exception with a naked `throw` statement
- An application *wraps* an exception by throwing a new exception with the original exception as its inner exception.
- An exception is *handled* if it is neither re-thrown nor wrapped
- An exception is *logged* by writing it to a logging handler

Errors

The following are examples of errors.

- Timed-out calls over a network
- Storage failure on a database
- Invalid input-file format
- Invalid user input

Bugs

The following are examples of bugs.

- The system runs out of memory
- Dereferencing a `null` variable
- An invalid cast
- Any unexpected situation for which the software is not prepared

The most common exceptions in C# are bugs.

- `ArgumentException` and descendants
- `NullReferenceException`
- `ClassCastException`
- `OutOfMemoryException`

- `StackOverflowException`
- `InvalidOperationException`
- `AccessViolationException`
- `NotSupportedException`
- `NotImplementedException`

Design-by-Contract

Use assertions at the beginning of a method to assert preconditions; assert post-conditions where appropriate.

- Throw `ArgumentNullExceptions` for preconditions and post-conditions.
- Do not use `Debug.Assert`.
- Do not remove constracts in release code unless you can prove a performance issue.
- Throw the exception on the same line as the check, to mirror the formatting of the assertion.

```
if (connection == null) { throw new ArgumentNullException("connection"); }
```

- If the assertion cannot be formulated in code, add a comment describing it instead.
- All methods and properties used to test pre-conditions must have the same visibility as the method being called.

Throwing Exceptions

- If a member cannot satisfy its post-condition (or, absent a post-condition, fulfill the promise implied in its name or specified in its documentation), it should throw an exception.
- Use standard exceptions.
- Never throw `Exception`. Instead, use one of the standard .NET exceptions when possible. These include `InvalidOperationException`, `NotSupportedException`, `ArgumentException`, `ArgumentNullException` and `ArgumentOutOfRangeException`.
- When using an `ArgumentException` or descendent thereof, make sure that the `ParamName` property is non-empty.
- Your code should not explicitly or implicitly throw `NullReferenceException`, `System.AccessViolationException`, `System.InvalidCastException`, or `System.IndexOutOfRangeException` as these indicate implementation details and possible attack points in your code. These exceptions are to be avoided with pre-conditions and/or argument-checking and should never be documented or accepted as part of the contract of a method.
- Do not throw `StackOverflowException` or `OutOfMemoryException`; these exceptions should only be thrown by the runtime.
- Do not explicitly throw exceptions from `finally` blocks (implicit exceptions are fine).

Catching Exceptions

- Do not handle bugs.
- Handle only specific, expected errors.
- Always log handled errors.
- Catch and re-throw an error in order to reset the internal state of an object.
- Do not log exceptions to an event handler; this practice separates the point-of-failure from the logging/collection point, increasing the likelihood that an exception is ignored and making debugging very difficult.

Buggy Third-party code

The *only time* it is appropriate to handle a bug is when third-party code has a bug *and you are sure that possibly corrupt state will not be re-used*. If the component is long-lived, you should not continue to use it after it has encountered a bug.

If the component is short-lived, it will be recycled and you can more-or-less safely ignore the bug. In the case of a misbehaving third-party component, catch the specific, known exception that is causing the problem and note it.

```
try
{
  return new BuggyComponent().GenerateReport(data);
}
catch (NullReferenceException exception)
{
  logger.Log("Buggy Component encountered known bug when processing expression.", exception);
}
```

Defining Exceptions

- Re-use exception types wherever possible.
- Do not simply create an exception type for every different error.
- Create a new type only if you want to expose additional properties or catch a specific class of exception.
- Don't expose additional properties unless you're actually going to use them.
- Use a custom exception to hold any information that more completely describes the error (e.g. error codes or structures).

  ```
  throw new DatabaseException(errorInfo);
  ```

- Custom exceptions should always inherit from `Exception`.
- Custom exceptions should be `public` so that other assemblies can catch them and extract information.
- Avoid constructing complex exception hierarchies; use your own exception base-classes only if you actually will have code that needs to catch all exceptions of a particular sub-class.
- An exception should provide the two standard constructors and should use the given parameter names:

  ```
  public class ConfigurationException : Exception
  {
    public ConfigurationException(string message)
      : base(message)
    { }

    public ConfigurationException(string message, Exception innerException)
      : base(message, innerException)
    { }
  }
  ```

- Only implement serialization for exceptions if you're going to use it.
- If an exception must be able to work across network boundaries, then it must be serializable.
- Do not cause exceptions during the construction of another exception (this sometimes happens when formatting custom messages) as this will subsume the original exception and cause confusion.

Wrapping Exceptions

- Only catch an exception to wrap it in another exception, log it or set an internal state
- Use an empty throw statement to re-throw the original exception in order to preserve the stack-trace.
- Wrapped exceptions should *always* include the original exception in order to preserve the stack-trace.
- Lower-level exceptions from an implementation-specific subsection should be caught and wrapped before being allowed to bubble up to implementation-independent code (e.g. when handling database exceptions).

The Try* Pattern

The Try* pattern is used by the .NET framework. Generally, Try*-methods accept an `out` parameter on which to attempt an operation, returning `true` if successful.

- The parameter should be named “result”. The method should be prefixed with “Try”.
- If you provide a method using the Try* pattern (), you should also provide a non-try-based, exception-throwing variant as well. The exception-throwing variant should call the Try* variant, never the other way around.

```
public IExpression Parse(string text)
{
  if (TryParse(text, var out expression))
  {
    return expression;
  }

  throw new InvalidOperationException($"The expression [{text}] contains a syntax error.");
}

public bool TryParse(string text, out IExpression result)
{
  if (text == "true")
  {
    expression = BooleanExpression(true);

    return true;
  }

  if (text == "false")
  {
    expression = BooleanExpression(false);

    return true;
  }

  expression = null;

  return false;
}
```

Error Messages

Content

- Use complete sentences that end in a period.
- Do not use question marks or exclamation points.
- Be brief.
- Be specific.
- Provide information on how to prevent the error in the future.

The following message is too vague and wordy.

```
"The file that the application was looking for in order to load the configuration could not be loaded from
```

This message leaves a lot of questions open.

- Does the file exist?
- Is it empty?
- Is it corrupted?
- Can the application read it?
- Where exactly was the application looking?

Instead, use something like the following.

```
"Permission to read file [~/.appConfig] was denied."
```

From this message the problem is clear and the user has many clues as to how to address the issue.

Exceptions

- Log all exceptions.
- Include technical detail in a separate message in the exception (e.g. stored in the `Data` array with a standard key).
-
- Lower-level, developer messages should be logged to sources that are available only to those with permission to view lower-level details.
- Applications should avoid showing sensitive information to end-users. This applies especially to web applications, which must never show exception traces in production code. The exact message returned by an exception can vary depending on the permission level of the executing code.
- If data included in a message *could* be empty, consider wrapping it in braces so that the message is clear even when the data is empty. For example, the following code might produce a confusing error message:

  ```
  var message = $"The following expression {data} could not be parsed.";
  ```

  If `data` is empty, the caller (user or developer) sees only *The following expression could not be parsed*. If the message was instead defined as follows:

  ```
  var message = $"The following expression [{data}] could not be parsed.";
  ```

  Then the caller sees *The following expression [] could not be parsed*. In this case it's more obvious that the expression was empty.

Object Lifetime

- Use the `using` statement to precisely delineate the lifetime of `IDisposable` objects.
- Use `try` / `finally` blocks to manage other state (e.g. executing a matching `EndUpdate()` for a `BeginUpdate()`)
- Do not set local variables to `null`. They will be automatically de-referenced and cleaned up.

`IDisposable`

- Implement `IDisposable` if your object uses disposable objects or system resources.
- Be careful about making your interfaces `IDisposable`; that pattern is a leaky abstraction and can be quite invasive.
- Implement `Dispose()` so that it can be safely called multiple times (see example below).
- Don't hide `Dispose()` in an explicit implementation. It confuses callers unnecessarily. If there is a more appropriate domain-specific name for `Dispose`, feel free to make a synonym.

`Finalize`

- There are performance penalties for implementing `Finalize`. Implement it only if you actually have costly external resources.
- Call the `GC.SuppressFinalize` method from `Dispose` to prevent `Finalize` from being executed if `Dispose()` has already been called.
- `Finalize` should include only calls to `Dispose` and `base.Finalize()`.
- `Finalize` should never be `public`

Destructors

- Avoid using destructors because they incur a performance penalty in the garbage collector.
- Do not access other object references inside the destructor as those objects may already have been garbage-collected (there is no guaranteed order-of-destruction in the IL or .NET runtime).

Best Practices

The following class expects the caller to use a non-standard pattern to avoid holding open a file handle.

```
public class Weapon
{
  public Load(string file)
  {
    _file = File.OpenRead(file);
  }

  // Aim(), Fire(), etc.

  public EjectShell()
  {
    _file.Dispose();
  }
```

```
  private File _file;
}
```

Instead, make `Weapon` disposable. The following implementation follows the recommended pattern. R# can help you create this pattern.

> *Note that `_file` is set to `null` so that `Dispose` can be called multiple notes, not to clear the reference.*

```
public class Weapon : IDisposable
{
  public Weapon(string file)
  {
    _file = File.OpenRead(file);
  }

  // Aim(), Fire(), etc.

  public Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
  }

  protected virtual void Dispose(bool disposing)
  {
    if (disposing)
    {
      if (_file != null)
      {
        _file.Dispose();
        _file = null;
      }
    }
  }

  private File _file;
}
```

Using the standard pattern, R# and Code Analysis will detect when an `IDisposable` object is not disposed of properly.

Managing Change

Modifying Interfaces

- Avoid introducing breaking changes. Wherever possible, retain old overloads/names for one extra major version.
- Document breaking changes in the release notes, including upgrade instructions.

Marking Members as Obsolete

Include the version number in the message. For example:

```
[Obsolete(""Since 4.0: Use IMetaElementSearchAspect instead.")]
```

Refactoring Names and Signatures

The whole point of having an agile process and lots of automated tests is to be able to quickly improve designs and accommodate new functionality. Very often, this involves quite aggressive refactoring. Aggressive refactoring means that code that compiled with a previous version of framework may no longer compile with the latest version. In each case where such a change is to be made, the following points must be considered:

- Will customer code be affected? A "customer" in this case is *anyone* who consumes your code: both internal and external developers.
- Do you have access to all uses of the code in order to be able to update it?
- Is all of the code that depends on code integrated at least daily in order to catch any usages you might have forgotten?
- Is the area you are changing covered by automated tests?
- Just how important is the change?
- Can you make the change in a non-destructive manner by introducing an optional parameter or an overload instead?
- Are you willing to introduce a compile error into customer code?

With an agile methodology, the answer to the question "should you?" is quite often "yes". Cleaner, tighter and more logical code is more maintainable and self-explanatory code.

Roadmap for Safe Obsolescence

If the change is localized, you can of course make it right away. If not, you may need to go the long route described below:

1. Mark the old version of the feature as obsolete.
2. Create the new feature with a different name so that it does not collide with the existing feature.
3. Rewrite the code so that the main code path uses the new feature but also incorporates the old version as well.
4. Add an issue to complete the next stage of refactoring in the next release.
5. Make and distribute a point release.
6. In the next release, remove the obsolete feature and all handling for it.

7. Rename the new feature to the desired name but retain a copy with the temporary name as well, marking it as obsolete.
8. Update the issue to indicate that it should be completed in the next release.
9. Make and distribute a point release.
10. In the next release, remove the obsolete version with the temporary name and the refactoring is complete.
11. Close the issue.

Because the steps outlined above require the patience of a saint, they should really only be used for features that absolutely *must* be refactored but that absolutely *cannot* break customer code. In all other cases, refactor away and make sure that repair instructions for the compile error are included in the release notes.

Documentation

Files

- Include a `README.md` file at the root of the project that includes the following information:
  - Dependencies
  - Basic configuration
  - Basic command line
  - Links to other documentation
- Include a `LICENSE` file at the root of the project that describes licensing restrictions.

Language

- Use U.S. English spelling and grammar.
- Use full sentences or clauses; do not use lists of keywords or short phrases.
- Use the prepositional possessive for code elements (i.e. write "the value of `<paramref name="prop">`" instead of "`<paramref name="prop">`'s value").

Style

- An API should document itself. Code documentation can sometimes be very obvious and simple. This indicates to the caller that it really *is* that simple.
- Document similar members consistently; it's better to repeat yourself or to use the same structure for all members as long as the documentation is useful for each member.
- Include conceptual documentation for each concept/component to provide an overview and examples of how to use the product.
- Move longer documentation out of the code and into higher-level conceptual documentation or examples.

XML Documentation

- Include XML documentation to enhance code-completion.
- Document `public` and `protected` elements.
- Do not document `private` or `internal` members.
- Include references to important members from class documentation.

Dependencies

- Do not introduce dependencies for documentation.
- Do not add `using` statements for documentation; if necessary, include the required namespace in the documentation reference itself.

Tags

- Format block tags onto separate lines. E.g. `<summary>`, `<param>`, `<remarks>` and `<returns>`
- Use `<c>` tags for the keywords `null`, `false` and `true`.
- Use `<see>` tags to refer to properties, methods and classes.

- Use `<paramref>` and `<typeparamref>` tags to refer to method parameters.
- Use the `<inheritdoc/>` tag for method overrides or interface implementations.
- Use a `<remarks>` section to indicate usage and to link to related members. It's sometimes good to include references to other types or methods in descriptive sentences in addition to listing them in the `<seealso>` section.

Examples

Classes

- An abstract implementation of an interface should use the following form:

```
/// <summary>
/// A base implementation of the <see cref="IMaker"/> interface.
/// </summary>
public abstract class MakerBase : IMaker { }
```

- The standard (or only) implementation of an interface should use the following form:

```
/// <summary>
/// The standard implementation of the <see cref="IMaker"/> interface.
/// </summary>
public class Maker : IMaker { }
```

- For one of several implementations, use the following form:

```
/// <summary>
/// An implementation of the <see cref="IMaker"/> interface that works with a
/// Windows service.
/// </summary>
public class WindowsServerBasedMaker : IMaker { }
```

Methods

- Document parameters in declaration order.
- Do not document exceptions that are bugs (e.g. `ArgumentNullException`).
- Refer to the first parameter as "given"; subsequent parameter references do not need to be qualified. For example,

```
/// Gets the value of the given <paramref name="prop"/> in <paramref name="obj">.`
```

- The documentation should indicate which values are acceptable inputs for a parameter (e.g. whether or not it can be `null` or empty (for strings) or the range of acceptable values (for numbers). The example below demonstrates all of these principles:

```
/// <summary>
/// Fills the <see cref="Body"> with random text using the given
/// <paramref name="generator"> and <paramref name="seedValues">.
/// </summary>
/// <param name="generator">
/// The generator to use to create the random text; cannot be <c>null</c>.
/// </param>
/// <param name="seedValues">
/// The values with which to seed the random generator; cannot be <c>null</c>.
/// </param>
void FillWithRandomText(IRandomGenerator generator, string seedValues);
```

- For methods that return a `bool`, use the following form:

```
/// <summary>
/// Gets a value indicating whether the value of the given <paramref name="prop"/>
```

```
/// in <paramref name="obj"> has changed since it was loaded from or last stored.
/// </summary>
/// <param name="obj">
/// The object to test; cannot be <c>null</c>.
/// </param>
/// <param name="prop">
/// The property to test; cannot be <c>null</c>.
/// </param>
/// <returns>
/// <c>true</c> if the value has been modified; otherwise <c>false</c>.
/// </returns>
bool ValueModified(object obj, IMetaProperty prop);
```

- Exceptions should begin with “If…” as shown in the example below:

```
/// <summary>
/// Gets the <paramref name="value"/> as formatted according to its type.
/// </summary>
/// <param name="value">
/// The value to format; can be <c>null</c>.
/// </param>
/// <param name="hints">
/// Hints indicating context and formatting requirements.
/// </param>
/// <returns>
/// The <paramref name="value"/> as formatted according to its type.
/// </returns>
/// <exception cref="FormatException">
/// If <paramref name="value"/> cannot be formatted.
/// </exception>
string FormatValue(object value, CommandTextFormatHints hints);
```

Constructors

- Do not use `<inheritdoc/>` for constructors.
- Documentation for parameters that initialize `public` or `protected` properties should reference that property instead of repeating documentation for properties in the documentation for the initializing parameter in the constructor.
- Parameters assigned to read-only properties should use the form "Initializes the value of ..."
- Parameters assigned to read/write properties should use the form "Sets the initial value of ..."

```
class SortOrderAspect
{
  /// <summary>
  /// Initializes a new instance of the <see cref="SortOrderAspect"/> class.
  /// </summary>
  /// <param name="sortOrderProperty">
  /// Initializes the value of <see cref="SortOrderProperty"/>.
  /// </param>
  /// <param name="sortOrderProperty">
  /// Sets the initial value of <see cref="Enabled"/>.
  /// </param>
  public SortOrderAspect(IMetaProperty sortOrderProperty, bool enabled)
  {
  }

  /// <summary>
  /// Gets the property used to create a manual sorting for objects using the
  /// <see cref="IMetaClass"/> to which this aspect is attached.
  /// </summary>
  IMetaProperty SortOrderProperty { get; private set; }

  /// <summary>
  /// Gets or sets a value indicating whether this <see cref="SortOrderAspect"/> is
  /// enabled.
  /// </summary>
```

```
  IMetaProperty Enabled { get; set; }
}
```

Properties

- If a property has a non-public setter, do not include the "or sets" part to avoid confusion for public users of the property.
- Include only the `<summary>` tag. The `<value>` tag is not needed.
- The documentation for read-only properties must begin with “Gets”.

```
/// <summary>
/// Gets the environment within which the database runs.
/// </summary>
IDatabaseEnvironment Environment { get; private set; }
```

- The documentation for read/write properties should begin with “Gets or sets”, as follows:

```
/// <summary>
/// Gets or sets the database type.
/// </summary>
DatabaseType DatabaseType { get; }
```

- Boolean properties should have the following form, formatting the value element as follows:

```
/// <summary>
/// Gets a value indicating whether the database in <see cref="Settings"/> exists.
/// </summary>
bool Exists { get; }
```

- For properties with generic names, take care to specify exactly what the property does, rather than writing vague documentation like “gets or sets a value indicating whether this object is enabled”. Tell the user what “enabled” means in the context of the property being documented:

```
/// <summary>
/// Gets or sets a value indicating whether automatic updating of the sort-order
/// is enabled.
/// </summary>
bool Enabled { get; }
```

Full Example

The example below includes many of the best practices outlined in the previous sections. It includes `<seealso>`, `<exception>` and several `<paramref>` tags as well as clearly stating what it does with those parameters and their acceptable values. Finally, it includes extra detail in the `<remarks>` section instead of the `<summary>`.

```
/// <summary>
/// Copies the entire contents of the given <paramref name="input"/> stream to the
/// given <paramref name="output"/> stream.
/// <param name="input">
/// The stream from which to copy data; cannot be <c>null</c>.
/// </param>
/// <param name="output">
/// The stream to which to copy data; cannot be <c>null</c>.
/// </param>
/// <remarks>
/// Uses a 32KB buffer; use the <see cref="CopyTo(Stream,Stream,int)"/> overload to
/// use a different buffer size.
/// </remarks>
/// <seealso cref="CopyTo(Stream,Stream,int)"/>
/// <exception cref="IOException">If the <paramref name="input"/> cannot be read
/// or is not at the head of the stream and cannot perform a seek or if the
```

```
/// <paramref name="output"/> cannot be written.</exception>
public static void CopyTo(this Stream input, Stream output)
```

Miscellaneous

Generated Code

- Do not commit manual changes to generated code. Temporary changes for debugging are fine, but always be aware that your changes will be overwritten when code is re-generated.

Configuration and File System

- An assembly should not assume its location.
- Do not use the registry to store application information; save user settings to a user-accessible file instead.

Logging

- Do not log directly to any output (e.g. a file or the console).
- Avoid logging directly to global or static constructs.
- Instead, inject an interface into the method where needed (e.g. `ILogger`).

`ValueTask<T>`

The `ValueTask<T>` is also a performance-improvement feature to improve task-handling when there are a *lot* of tasks. Do not use these unless you have performance-sensitive code.

- Use `ValueTask<T>` where results will usually be returned synchronously.
- Use `ValueTask<T>` when Tasks cannot be cached and you memory-usage is a problem.
- Do not expose `ValueTask<T>` in public APIs.