C# Coding Style Guide and Naming Guidelines.

Note:

The Naming guidelines section is based on Microsoft C# Ecma documentation.

Some things of the Coding Style section are based on Java conventions.

Carlos Guzmán Álvarez.

A. Coding Style

A.1 Line Length

Consider avoiding (if possible) lines longer than 110 characters, switch on the ruler in your editor to get that managed, wrap lines if necessary.

A.2 Wrapping lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break after an operator.
- Prefer higher-level breaks to low-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.

Example of breaking method calls:

```
someMethod(longExpression1, longExpression2, longExpression3,
          longExpression4, longExpression5);

var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                     longExpression3));
```

Examples of breaking an arithmetic expression. The first is preferred, since the break occurs outside of the paranthesized expression (higher level rule).

```
       var =  a * b / (c - g + f) +
              4 * z; // PREFER

       var =  a * b / (c - g +
              f) + 4 * z;  // AVOID
```

A.3 Comments

A.3.1 Block Comments

Use the following style for block comments:

```
/* Line 1
 * Line 2
 * Line 3
 */
```

A.3.2 Single line comments

Use this style for end of line comments:

```
/* Comment Line */
```

or

```
// Comment Line
```

A.3.3 End of Line Comments

Use this style for end of line comments:

```
System.Int32 intValue; // Integer value
```

A.3.4 Documentation comments

Place documentation comments on separate XML files and use <include ... /> for make reference to the XML documentation file.

Example:

```
/// <include
file='Documentation.xml'path='doc/member[@name="T:Example"]/*'/>
public sealed class Example
{
    ...
}
```

A.4 Declarations

A.4.1 Number per line.

One declaration per line is recommended since it encourages commenting, example:

```
int    value;
string name;  // PREFER
int    value; string name;     // AVOID
```

A.4.2 Initialization

Try to initialize local variables where they’re declared.

Example:

```
int    value  = 0;
string name   = String.Empty;
bool   flag   = false;
```

A.4.3 Placement

Put declarations only at the beginning of blocks.

Example:

```
void myMethod()
{
    int int1 = 0;       // beginning of method block
    if (condition)
    {
        int int2 = 0;    // beginning of "if" block
        ...
    }
}
```

A.4.4 Class and Interfaces

Use this style for end of line comments:

- No space between a method name and the parenthesis “(“ starting its parameter list.

- Open brace "{"starts a line by itself indented to match its declaration statement.
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement.

Example:

```
public class MyClass : MyBaseClass
{
    public MyClass()
    {
    }

    public string ReadData()
    {
        /* Access a custom resource. */
    }
}
```

Example of Interface:

```
/* C#
 * Code for the IAccount interface module.
 */
public interface IAccount
{
    void PostInterest();
    void DeductFees(IFeeSchedule feeSchedule);
}
```

A.5 Statements

A.5.1 Simple statements

Each line should contain at most one statement.

Example:

```
var1++;          // Correct
var2++;          // Correct
var1++; var2--;  // AVOID!
```

A.5.2 Return statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way.

Example:

```
return;

return var1;

return (var1 > var2 ? 0 : 1);
```

A.5.3 if statement

Use always the open and close bracket for if and else blocks, put an space between the if and the open parenthesis.

Examples:

```
if (x > 10)
{
    if (y > 20)
    {
        Console.WriteLine("Statement_1");
    }
    else
    {
        Console.WriteLine("Statement_2");
    }
}

if (Condition_1)
{
    Console.WriteLine("Statement_1");
}
else
{
    if (Condition_2)
    {
        Console.WriteLine("Statement_2");
    }
    else
    {
        if (Condition_3)
        {
            Console.WriteLine("Statement_1");
        }
    }
}

if (condition)   // AVOID! THIS OMITS THE BRACES {}!
    statement;
```

A.5.4 switch statement

A `switch` statement should have the following form:

```
switch (condition)
{
    case 0:
    case 1:
    {
        statements;
    }
    break;

    case 2:
    {
        statements;
    }
    break;

    case 3:
    {
        statements;
    }
    break;
```

```
        default:
        {
            statements;
        }
        break;
    }
```

A.5.5 for statement

A `for` statement should have the following form:

```
    for (initializers; expression; iterators)
    {
        statements;
    }
```

A.5.6 foreach statement

A `foreach` statement should have the following form:

```
    foreach (type identifier in expression)
    {
        statements;
    }
```

A.5.7 while and do...while statements

A `while` statement should have the following form:

```
    while (expression)
    {
        statements;
    }
```

A `do ... while` statement should have the following form:

```
    do
    {
        statements;
    } while (expression);
```

A.5.8 try...catch statements

A `try ... catch` statement should have the following form:

```
    try
    {
        statements;
    }
    catch (ExceptionClass ex)
    {
        statements;
    }
```

or

```
try
{
    statements;
}
catch (ExceptionClass ex)
{
    statements;
}
finally
{
    statements;
}
```

A.6 White space

A.6.1 Blank lines

Blank lines improve readability by setting off sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

One blank line should always be used in the following circumstances:

- Between methods
- Between logical sections inside a method to improve readability

A.6.2 Blank spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true)
{
    ...
}
```

  Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands. Example:

```
a += c + d;
a = (a + b) / (c * d);
while (d++ == s++)
{
    n++;
}
```

- The expressions in a for statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

B. Naming guidelines

B.1 Capitalization styles

The following section describes different ways of capitalizing identifiers.

B.1.1 Pascal casing

This convention capitalizes the first character of each word. For example:

```
Color    BitConverter
```

B.1.2 Camel casing

This convention capitalizes the first character of each word except the first word. For example:

```
backgroundColor    totalValueCount
```

B.1.3 All uppercase

Only use all uppercase letters for an identifier if it contains an abbreviation. For example:

```
System.IO
System.WinForms.UI
```

B.1.4 Capitalization summary

The following table summarizes the capitalization style for the different kinds of identifiers:

Type	Case	Notes
Class	PascalCase	
Attribute Class	PascalCase	Has a suffix of Attribute
Exception Class	PascalCase	Has a suffix of Exception
Constant	PascalCase	
Enum type	PascalCase	
Enum values	PascalCase	
Event	PascalCase	
Interface	PascalCase	Has a prefix of **I**
Local variable	camelCase	
Method	PascalCase	
Namespace	PascalCase	
Property	PascalCase	
Public Instance Field	PascalCase	Rarely used (use a property instead)
Protected Instance Field	camelCase	Rarely used (use a property instead)
Private instance Field	camelCase	
Parameter	camelCase	

B.2 Word choice

- Do avoid using class names duplicated in heavily used namespaces. For example, don't use the following for a class name.

```
System    Collections    Forms    UI
```

- Do not use abbreviations in identifiers.
- If you must use abbreviations, do use camelCase for any abbreviation containing more than two characters, even if this is not the usual abbreviation.

B.3 Namespaces

The general rule for namespace naming is: CompanyName.TechnologyName.

- Do avoid the possibility of two published namespaces having the same name, by prefixing namespace names with a company name or other well-established brand. For example, Microsoft.Office for the Office Automation classes provided by Microsoft.
- Do use **PascalCase**, and separate logical components with periods (as in Microsoft.Office.PowerPoint). If your brand employs non-traditional casing, do follow the casing defined by your brand, even if it deviates from normal namespace casing (for example, NeXT.WebObjects, and ee.cummings).
- Do use plural namespace names where appropriate. For example, use System.Collections rather than System.Collection.Exceptions to this rule are brand names and abbreviations. For example, use System.IO not System.IOs.
- Do not have namespaces and classes with the same name.

B.4 Classes

- Do name classes with nouns or noun phrases.
- Do use PascalCase.
- Do use sparingly, abbreviations in class names.
- Do not use any prefix (such as **"C"**, for example). Where possible, avoid starting with the letter **"I"**, since that is the recommended prefix for interface names. If you must start with that letter, make sure the second character is lowercase, as in IdentityStore.
- Do not use any underscores.

```
public class FileStream
{
    …
}
public class Button
{
    …
}
public class String
{
    …
}
```

B.5 Interfaces

- Do name interfaces with nouns or noun phrases, or adjectives describing behavior. For example, IComponent (descriptive noun), ICustomAttributeProvider (noun phrase), and IPersistable (adjective).
- Do use PascalCase.
- Do use sparingly, abbreviations in interface names.
- Do not use any underscores.
- Do prefix interface names with the letter **"I"**, to indicate that the type is an interface.
- Do use similar names when defining a class/interface pair where the class is a standard implementation of the interface. The names should differ only by the "I" prefix in the interface name. This approach is used for the interface IComponent and its standard implementation, Component.

```
public interface IComponent
{
    …
}
public class Component : IComponent
{
    …
}
public interface IserviceProvider
{
    …
}
public interface IFormatable
{
    …
}
```

B.6 Enums

- Do use **PascalCase** for enums.
- Do use **PascalCase** for enum value names.
- Do use sparingly, abbreviations in enum names.
- Do not use a family-name prefix on enum.
- Do not use any "Enum" suffix on enum types.
- Do use a singular name for enums
- Do use a plural name for bit fields
- Do define enumerated values using an enum if they are used in a parameter or property. This gives development tools a chance at knowing the possible values for a property or parameter.

```
public enum FileMode
```

```
{
    Create,
    CreateNew,
    Open,
    OpenOrCreate,
    Truncate
}
```

- Do use the Flags custom attribute if the numeric values are meant to be bitwise ORed together

```
[Flags]
public enum Bindings
{
    CreateInstance,
    DefaultBinding,
    ExcatBinding,
    GetField,
    GetProperty,
    IgnoreCase,
    InvokeMethod,
    NonPublic,
    OABinding,
    SetField
    SetProperty,
    Static
}
```

- Do use int as the underlying type of an enum. (An exception to this rule is if the enum represents flags and there are more than 32 flags, or the enum may grow to that many flags in the future, or the type needs to be different from int for backward compatibility.)
- Do use enums only if the value can be completely expressed as a set of bit flags. Do not use enums for open sets (such as operating system version).

B.7 Static fields

- Do name static members with nouns, noun phrases, or abbreviations for nouns.
- Do name static members using **PascalCase**.
- Do not use Hungarian-type prefixes on static member names.

B.8 Parameters

- Do use descriptive names such that a parameter's name and type clearly imply its meaning.
- Do name parameters using **camelCase**.
- Do prefer names based on a parameter's meaning, to names based on the parameter's type. It is likely that development tools will provide the information about type in a convenient way, so the parameter name can be put to better use describing semantics rather than type.
- Do not reserve parameters for future use. If more data is need in the next version, a new overload can be added.
- Do not use Hungarian-type prefixes.

```
Type   GetType(string typeName)
String Format(string format, object[] args)
```

B.9 Methods

- Do name methods with verbs or verb phrases.
- Do name methods with **PascalCase**

```
RemoveAll(), GetCharArray(), Invoke()
```

B.10 Properties

- Do name properties using noun or noun phrases
- Do name properties with **PascalCase**.
- Consider having a property with the same as a type. When declaring a property with the same name as a type, also make the type of the property be that type. In other words, the following is okay

```
public enum Color
{
    ...
}
public class Control
{
    public Color Color
    {
        get {...}
        set {...}
    }
}
```

but this is not

```
public enum Color
    {
        //..
    }
public class Control
{
    public int Color
    {
        get {...}
        set {...}
    }
}
```

In the latter case, it will not be possible to refer to the members of the Color enum because Color.Xxx will be interpreted as being a member access that first gets the value of the Color property (of type `int`) and then accesses a member of that value (which would have to be an instance member of System.Int32).

B.11 Events

- Do name event handlers with the “EventHandler” suffix.

```
public delegate void MouseEventHandler(object sender, MouseEvent
e);
```

- Do use two parameters named sender and e. The sender parameter represents the object that raised the event, and this parameter is always of type object, even if it is possible to employ a more specific type. The state associated with the event is encapsulated in an instance e of an event class. Use an appropriate and specific event class for its type.

```
public delegate void MouseEventHandler(object sender, MouseEvent
e);
```

- Do name event argument classes with the “EventArgs” suffix.

```
public class MouseEventArgs : EventArgs
{
    int x;
    int y;
    public MouseEventArgs(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
        public int X
    {
        get { return x; }
    }
        public int Y
        {
        get { return y; }
    }
}
```

- Do name event names that have a concept of pre- and post-operation using the present and past tense (do not use BeforeXxx/AfterXxx pattern). For example, a close event that could be canceled would have a Closing and Closed event.

```
public event ControlEventHandler ControlAdded
{
   //..
}
```

- Consider naming events with a verb.

B.12 Case sensitivity

- Don’t use names that require case sensitivity. Components might need to be usable from both case-sensitive and case-insensitive languages. Since case-insensitive languages cannot distinguish between two names within the same context that differ only by case, components must avoid this situation.

Examples of what not to do:

- Don't have two namespaces whose names differ only by case.

```
namespace ee.cummings;
namespace Ee.Cummings;
```

- Don't have a method with two parameters whose names differ only by case.

```
void F(string a, string A)
```

- Don't have a namespace with two types whose names differ only by case.

```
System.WinForms.Point p;
System.WinForms.POINT pp;
```

- Don't have a type with two properties whose names differ only by case.

```
int F {get, set};
int F {get, set}
```

- Don't have a type with two methods whose names differ only by case.

```
void f();
void F();
```

B.13 Avoiding type name confusion

Different languages use different names to identify the fundamental managed types, so in a multi-language environment, designers must take care to avoid language-specific terminology. This section describes a set of rules that help avoid type name confusion.

- Do use semantically interesting names rather than type names.
- In the rare case that a parameter has no semantic meaning beyond its type, use a generic name. For example, a class that supports writing a variety of data types into a stream might have:

```
void Write(double value);
void Write(float value);
void Write(long value);
void Write(int value);
void Write(short value);
```

  rather than a language-specific alternative such as:

```
void Write(double doubleValue);
void Write(float floatValue);
void Write(long longValue);
void Write(int intValue);
void Write(short shortValue);
```

- In the extremely rare case that it is necessary to have a uniquely named method for each fundamental data type, do use the following universal type names: *Sbyte, Byte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, Boolean, Char, String, and Object*. For example, a class that supports reading a variety of data types from a stream might have:

```
Double ReadDouble();
float  ReadSingle();
long   ReadIn64();
int    ReadInt32();
short  ReadInt16();
```

  rather than a language-specific alternative such as:

```
double ReadDouble();
float  ReadFloat();
long   ReadLong();
```

```
int    ReadInt();
short  ReadShort();
```