# Programming Guidelines (C#.NET)

# Capitalization Conventions

# The guidelines in this chapter lay out a simple method for using case that, when applied consistently, make identifiers for types, members, and parameters easy to read.

## Capitalization Rules for Identifiers

## To differentiate words in an identifier, capitalize the first letter of each word in the identifier. Do not use underscores to differentiate words, or for that matter, anywhere in identifiers. There are two appropriate ways to capitalize identifiers, depending on the use of the identifier:

* PascalCasing
* camelCasing

The PascalCasing convention, used for all identifiers except parameter names, capitalizes the first character of each word (including acronyms over two letters in length), as shown in the following examples:

PropertyDescriptor  
HtmlTag

A special case is made for two-letter acronyms in which both letters are capitalized, as shown in the following identifier:

IOStream

The camelCasing convention, used only for parameter names, capitalizes the first character of each word except the first word, as shown in the following examples. As the example also shows, two-letter acronyms that begin a camel-cased identifier are both lowercase.

propertyDescriptor  
ioStream  
htmlTag

**✓ DO** use PascalCasing for all public member, type, and namespace names consisting of multiple words.

**✓ DO** use camelCasing for parameter names.

The following table describes the capitalization rules for different types of identifiers.

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Casing** | **Example** |
| Namespace | Pascal | namespace System.Security { ... } |
| Type | Pascal | public class StreamReader { ... } |
| Interface | Pascal | public interface IEnumerable { ... } |
| Method | Pascal | public class Object  {  public virtual string ToString(); } |
| Property | Pascal | public class String {  public int Length { get; } } |
| Event | Pascal | public class Process {  public event EventHandler Exited; } |
| Field | Pascal | public class MessageQueue {  public static readonly TimeSpan  InfiniteTimeout; } public struct UInt32 {  public const Min = 0; } |
| Enum value | Pascal | public enum FileMode {  Append,  ... } |
| Parameter | Camel | public class Convert {  public static int ToInt32(string value); } |

## Capitalizing Compound Words and Common Terms

Most compound terms are treated as single words for purposes of capitalization.

**X DO NOT** capitalize each word in so-called closed-form compound words.

These are compound words written as a single word, such as endpoint. For the purpose of casing guidelines, treat a closed-form compound word as a single word. Use a current dictionary to determine if a compound word is written in closed form.

|  |  |  |
| --- | --- | --- |
| **Pascal** | **Camel** | **Not** |
| BitFlag | bitFlag | Bitflag |
| Callback | callback | CallBack |
| Canceled | canceled | Cancelled |
| DoNot | doNot | Don't |
| Email | email | EMail |
| Endpoint | endpoint | EndPoint |
| FileName | fileName | Filename |
| Gridline | gridline | GridLine |
| Hashtable | hashtable | HashTable |
| Indexes | indexes | Indices |
| LogOff | logOff | LogOut |
| LogOn | logOn | LogIn |
| Metadata | metadata | MetaData, metaData |
| Multipanel | multipanel | MultiPanel |
| Multiview | multiview | MultiView |
| Namespace | namespace | NameSpace |
| Ok | ok | OK |
| Pi | pi | PI |
| Placeholder | placeholder | PlaceHolder |
| SignIn | signIn | SignOn |
| SignOut | signOut | SignOff |
| UserName | userName | Username |
| WhiteSpace | whiteSpace | Whitespace |
| Writable | writable | Writeable |

## Case Sensitivity

Languages that can run on the CLR are not required to support case-sensitivity, although some do. Even if your language supports it, other languages that might access your framework do not. Any APIs that are externally accessible, therefore, cannot rely on case alone to distinguish between two names in the same context.

**X DO NOT** assume that all programming languages are case sensitive. They are not. Names cannot differ by case alone.

# General Naming Conventions

# This section describes general naming conventions that relate to word choice, guidelines on using abbreviations and acronyms, and recommendations on how to avoid using language-specific names.

## Word Choice

**✓ DO** choose easily readable identifier names.

For example, a property named HorizontalAlignment is more English-readable than AlignmentHorizontal.

**✓ DO** favor readability over brevity.

The property name CanScrollHorizontally is better than ScrollableX (an obscure reference to the X-axis).

**X DO NOT** use underscores, hyphens, or any other nonalphanumeric characters.

**X DO NOT** use Hungarian notation.

**X AVOID** using identifiers that conflict with keywords of widely used programming languages.

According to Rule 4 of the Common Language Specification (CLS), all compliant languages must provide a mechanism that allows access to named items that use a keyword of that language as an identifier. C#, for example, uses the @ sign as an escape mechanism in this case. However, it is still a good idea to avoid common keywords because it is much more difficult to use a method with the escape sequence than one without it.

## Using Abbreviations and Acronyms

**X DO NOT** use abbreviations or contractions as part of identifier names.

For example, use GetWindow rather than GetWin.

**X DO NOT** use any acronyms that are not widely accepted, and even if they are, only when necessary.

## Avoiding Language-Specific Names

**✓ DO** use semantically interesting names rather than language-specific keywords for type names.

For example, GetLength is a better name than GetInt.

**✓ DO** use a generic CLR type name, rather than a language-specific name, in the rare cases when an identifier has no semantic meaning beyond its type.

For example, a method converting to Int64 should be named ToInt64, not ToLong (because Int64 is a CLR name for the C#-specific alias **long**). The following table presents several base data types using the CLR type names (as well as the corresponding type names for C#, Visual Basic, and C++).

|  |  |  |  |
| --- | --- | --- | --- |
| **C#** | **Visual Basic** | **C++** | **CLR** |
| **sbyte** | **SByte** | **char** | **SByte** |
| **byte** | **Byte** | **unsigned char** | **Byte** |
| **short** | **Short** | **short** | **Int16** |
| **ushort** | **UInt16** | **unsigned short** | **UInt16** |
| **int** | **Integer** | **int** | **Int32** |
| **uint** | **UInt32** | **unsigned int** | **UInt32** |
| **long** | **Long** | **\_\_int64** | **Int64** |
| **ulong** | **UInt64** | **unsigned \_\_int64** | **UInt64** |
| **float** | **Single** | **float** | **Single** |
| **double** | **Double** | **double** | **Double** |
| **bool** | **Boolean** | **bool** | **Boolean** |
| **char** | **Char** | **wchar\_t** | **Char** |
| **string** | **String** | **String** | **String** |
| **object** | **Object** | **Object** | **Object** |

**✓ DO**use a common name, such as value or item, rather than repeating the type name, in the rare cases when an identifier has no semantic meaning and the type of the parameter is not important.

## Naming New Versions of Existing APIs

**✓ DO** use a name similar to the old API when creating new versions of an existing API.

This helps to highlight the relationship between the APIs.

**✓ DO** prefer adding a suffix rather than a prefix to indicate a new version of an existing API.

This will assist discovery when browsing documentation, or using Intellisense. The old version of the API will be organized close to the new APIs, because most browsers and Intellisense show identifiers in alphabetical order.

**✓ CONSIDER** using a brand new, but meaningful identifier, instead of adding a suffix or a prefix.

**✓ DO** use a numeric suffix to indicate a new version of an existing API, particularly if the existing name of the API is the only name that makes sense (i.e., if it is an industry standard) and if adding any meaningful suffix (or changing the name) is not an appropriate option.

**X DO NOT** use the "Ex" (or a similar) suffix for an identifier to distinguish it from an earlier version of the same API.

**✓ DO** use the "64" suffix when introducing versions of APIs that operate on a 64-bit integer (a long integer) instead of a 32-bit integer. You only need to take this approach when the existing 32-bit API exists; don’t do it for brand new APIs with only a 64-bit version.

# Names of Assemblies and DLLs

An assembly is the unit of deployment and identity for managed code programs. Although assemblies can span one or more files, typically an assembly maps one-to-one with a DLL. Therefore, this section describes only DLL naming conventions, which then can be mapped to assembly naming conventions.

**✓ DO** choose names for your assembly DLLs that suggest large chunks of functionality, such as System.Data.

Assembly and DLL names don’t have to correspond to namespace names, but it is reasonable to follow the namespace name when naming assemblies. A good rule of thumb is to name the DLL based on the common prefix of the assemblies contained in the assembly. For example, an assembly with two namespaces,MyCompany.MyTechnology.FirstFeature and MyCompany.MyTechnology.SecondFeature, could be called MyCompany.MyTechnology.dll.

**✓ CONSIDER** naming DLLs according to the following pattern:

<Company>.<Component>.dll

where <Component> contains one or more dot-separated clauses. For example:

Litware.Controls.dll.

# Names of Namespaces

# As with other naming guidelines, the goal when naming namespaces is creating sufficient clarity for the programmer using the framework to immediately know what the content of the namespace is likely to be. The following template specifies the general rule for naming namespaces:

<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]

The following are examples:

Fabrikam.Math  
Litware.Security

**✓ DO** prefix namespace names with a company name to prevent namespaces from different companies from having the same name.

**✓ DO** use a stable, version-independent product name at the second level of a namespace name.

**X DO NOT** use organizational hierarchies as the basis for names in namespace hierarchies, because group names within corporations tend to be short-lived. Organize the hierarchy of namespaces around groups of related technologies.

**✓ DO** use PascalCasing, and separate namespace components with periods (e.g., Microsoft.Office.PowerPoint). If your brand employs nontraditional casing, you should follow the casing defined by your brand, even if it deviates from normal namespace casing.

**✓ CONSIDER** using plural namespace names where appropriate.

For example, use System.Collections instead of System.Collection. Brand names and acronyms are exceptions to this rule, however. For example, use System.IO instead of System.IOs.

**X DO NOT** use the same name for a namespace and a type in that namespace.

For example, do not use Debug as a namespace name and then also provide a class named Debug in the same namespace. Several compilers require such types to be fully qualified.

## Namespaces and Type Name Conflicts

**X DO NOT** introduce generic type names such as Element, Node, Log, and Message.

There is a very high probability that doing so will lead to type name conflicts in common scenarios. You should qualify the generic type names (FormElement, XmlNode, EventLog, SoapMessage).

There are specific guidelines for avoiding type name conflicts for different categories of namespaces.

* **Application model namespaces**

Namespaces belonging to a single application model are very often used together, but they are almost never used with namespaces of other application models. For example, the System.Windows.Forms namespace is very rarely used together with the System.Web.UI namespace. The following is a list of well-known application model namespace groups:

System.Windows\*  
System.Web.UI\*

**X DO NOT** give the same name to types in namespaces within a single application model.

For example, do not add a type named Page to the System.Web.UI.Adapters namespace, because the System.Web.UI namespace already contains a type named Page.

* **Infrastructure namespaces**

This group contains namespaces that are rarely imported during development of common applications. For example, .Design namespaces are mainly used when developing programming tools. Avoiding conflicts with types in these namespaces is not critical.

* **Core namespaces**

Core namespaces include all System namespaces, excluding namespaces of the application models and the Infrastructure namespaces. Core namespaces include, among others, System, System.IO, System.Xml, and System.Net.

**X DO NOT** give types names that would conflict with any type in the Core namespaces.

For example, never use Stream as a type name. It would conflict with System.IO.Stream, a very commonly used type.

* **Technology namespace groups**

This category includes all namespaces with the same first two namespace nodes (<Company>.<Technology>\*), such as Microsoft.Build.Utilities and Microsoft.Build.Tasks. It is important that types belonging to a single technology do not conflict with each other.

**X DO NOT** assign type names that would conflict with other types within a single technology.

**X DO NOT** introduce type name conflicts between types in technology namespaces and an application model namespace (unless the technology is not intended to be used with the application model).

# Names of Classes, Structures, and Interfaces

The naming guidelines that follow apply to general type naming.

**✓ DO** name classes and structures with nouns or noun phrases, using PascalCasing.

This distinguishes type names from methods, which are named with verb phrases.

**✓ DO** name interfaces with adjective phrases, or occasionally with nouns or noun phrases.

Nouns and noun phrases should be used rarely and they might indicate that the type should be an abstract class, and not an interface.

**X DO NOT** give class names a prefix (e.g., "C").

**✓ CONSIDER** ending the name of derived classes with the name of the base class.

This is very readable and explains the relationship clearly. Some examples of this in code are:ArgumentOutOfRangeException, which is a kind of Exception, and SerializableAttribute, which is a kind ofAttribute. However, it is important to use reasonable judgment in applying this guideline; for example, the Button class is a kind of Control event, although Control doesn’t appear in its name.

**✓ DO** prefix interface names with the letter I, to indicate that the type is an interface.

For example, IComponent (descriptive noun), ICustomAttributeProvider (noun phrase), and IPersistable(adjective) are appropriate interface names. As with other type names, avoid abbreviations.

**✓ DO** ensure that the names differ only by the "I" prefix on the interface name when you are defining a class–interface pair where the class is a standard implementation of the interface.

## Names of Generic Type Parameters

Generics were added to .NET Framework 2.0. The feature introduced a new kind of identifier called type parameter.

**✓ DO** name generic type parameters with descriptive names unless a single-letter name is completely self-explanatory and a descriptive name would not add value.

**✓ CONSIDER** using T as the type parameter name for types with one single-letter type parameter.

public int IComparer<T> { ... }

public delegate bool Predicate<T>(T item);

public struct Nullable<T> where T:struct { ... }

**✓ DO** prefix descriptive type parameter names with T.

public interface ISessionChannel<TSession> where TSession : ISession

{

TSession Session { get; }

}

**✓ CONSIDER** indicating constraints placed on a type parameter in the name of the parameter.

For example, a parameter constrained to ISession might be called TSession.

## [Names of Common Types](javascript:void(0))

**✓ DO** follow the guidelines described in the following table when naming types derived from or implementing certain .NET Framework types.

|  |  |
| --- | --- |
| **Base Type** | **Derived/Implementing Type Guideline** |
| System.Attribute | **✓ DO** add the suffix "Attribute" to names of custom attribute classes. Add the suffix "Attribute" to names of custom attribute classes. |
| System.Delegate | **✓ DO** add the suffix "EventHandler" to names of delegates that are used in events.  **✓ DO** add the suffix "Callback" to names of delegates other than those used as event handlers.  **X DO NOT** add the suffix "Delegate" to a delegate. |
| System.EventArgs | **✓ DO** add the suffix "EventArgs.” |
| System.Enum | **X DO NOT** derive from this class; use the keyword supported by your language instead; for example, in C#, use the enum keyword.  **X DO NOT** add the suffix "Enum" or "Flag.” |
| System.Exception | **✓ DO** add the suffix "Exception.” |
| IDictionary IDictionary<TKey,TValue> | **✓ DO** add the suffix "Dictionary." Note that IDictionary is a specific type of collection, but this guideline takes precedence over the more general collections guideline that follows. |
| IEnumerable ICollection IList IEnumerable<T> ICollection<T> IList<T> | **✓ DO** add the suffix "Collection.” |
| System.IO.Stream | **✓ DO** add the suffix "Stream.” |
| CodeAccessPermission IPermission | **✓ DO** add the suffix "Permission.” |

## Naming Enumerations

Names of enumeration types (also called enums) in general should follow the standard type-naming rules (PascalCasing, etc.). However, there are additional guidelines that apply specifically to enums.

**✓ DO** use a singular type name for an enumeration unless its values are bit fields.

**✓ DO** use a plural type name for an enumeration with bit fields as values, also called flags enum.

**X DO NOT** use an "Enum" suffix in enum type names.

**X DO NOT** use "Flag" or “Flags" suffixes in enum type names.

**X DO NOT** use a prefix on enumeration value names (e.g., "ad" for ADO enums, "rtf" for rich text enums, etc.).

# Names of Type Members

# Types are made of members: methods, properties, events, constructors, and fields. The following sections describe guidelines for naming type members.

## Names of Methods

Because methods are the means of taking action, the design guidelines require that method names be verbs or verb phrases. Following this guideline also serves to distinguish method names from property and type names, which are noun or adjective phrases.

**✓ DO** give methods names that are verbs or verb phrases.

public class String

{

public int CompareTo(...);

public string[] Split(...);

public string Trim();

}

## Names of Properties

Unlike other members, properties should be given noun phrase or adjective names. That is because a property refers to data, and the name of the property reflects that. PascalCasing is always used for property names.

**✓ DO** name properties using a noun, noun phrase, or adjective.

**X DO NOT** have properties that match the name of "Get" methods as in the following example:

public string TextWriter { get {...} set {...} }  
public string GetTextWriter(int value) { ... }

This pattern typically indicates that the property should really be a method.

**✓ DO** name collection properties with a plural phrase describing the items in the collection instead of using a singular phrase followed by "List" or "Collection."

**✓ DO** name Boolean properties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with "Is," "Can," or "Has," but only where it adds value.

**✓ CONSIDER** giving a property the same name as its type.

For example, the following property correctly gets and sets an enum value named Color, so the property is named Color:

public enum Color {...}

public class Control

{

public Color Color { get {...} set {...} }

}

## Names of Events

Events always refer to some action, either one that is happening or one that has occurred. Therefore, as with methods, events are named with verbs, and verb tense is used to indicate the time when the event is raised.

**✓ DO** name events with a verb or a verb phrase.

Examples include Clicked, Painting, DroppedDown, and so on.

**✓ DO** give events names with a concept of before and after, using the present and past tenses.

For example, a close event that is raised before a window is closed would be called Closing, and one that is raised after the window is closed would be called Closed.

**X DO NOT** use "Before" or "After" prefixes or postfixes to indicate pre- and post-events. Use present and past tenses as just described.

**✓ DO** name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

public delegate void ClickedEventHandler(object sender, ClickedEventArgs e);

**✓ DO** use two parameters named sender and e in event handlers.

The sender parameter represents the object that raised the event. The sender parameter is typically of type object, even if it is possible to employ a more specific type.

**✓ DO** name event argument classes with the "EventArgs" suffix.

## Names of Fields

The field-naming guidelines apply to static public and protected fields. Internal and private fields are not covered by guidelines, and public or protected instance fields are not allowed by the member design guidelines.

**✓ DO** use PascalCasing in field names.

**✓ DO** name fields using a noun, noun phrase, or adjective.

**X DO NOT** use a prefix for field names.

For example, do not use "g\_" or "s\_" to indicate static fields.

# Naming Parameters

# Beyond the obvious reason of readability, it is important to follow the guidelines for parameter names because parameters are displayed in documentation and in the designer when visual design tools provide Intellisense and class browsing functionality.

**✓ DO** use camelCasing in parameter names.

**✓ DO** use descriptive parameter names.

**✓ CONSIDER** using names based on a parameter’s meaning rather than the parameter’s type.

## Naming Operator Overload Parameters

**✓ DO** use left and right for binary operator overload parameter names if there is no meaning to the parameters.

**✓ DO** use value for unary operator overload parameter names if there is no meaning to the parameters.

**✓ CONSIDER** meaningful names for operator overload parameters if doing so adds significant value.

**X DO NOT** use abbreviations or numeric indices for operator overload parameter names.

**References**

<https://msdn.microsoft.com/en-us/library/ms229042(v=vs.110).aspx>