Diartis Guidlines

Microsoft
Developer Network

Table Of Contents

Naming

Capitalization Conventions

.NET Framework (current version)

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`
`Id`	`id`	`ID`
`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.

See Also

Framework Design Guidelines
Naming Guidelines

General Naming Conventions

.NET Framework (current version)

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.

Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Framework Design Guidelines
Naming Guidelines

Names of Namespaces

.NET Framework (current version)

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:

```
Klib[<Feature>][.<Subnamespace>]*
```

The following are examples:

```
KlibDatabrowser.Forms
KlibRessources.Icons.Files
```

✓ **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).

See Also

Framework Design Guidelines
Naming Guidelines

Names of Classes, Structs, and Interfaces

.NET Framework (current version)

The naming guidelines that follow apply to general type naming.

✓ **DO** name classes and structs 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 of `Attribute`. 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

✓ **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.).

See Also

Framework Design Guidelines

Naming Guidelines

Names of Type Members

.NET Framework (current version)

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.

See Also

Framework Design Guidelines

Naming Guidelines

Naming Parameters

.NET Framework (current version)

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.

See Also

Framework Design Guidelines
Naming Guidelines

Naming Resources

.NET Framework (current version)

Because localizable resources can be referenced via certain objects as if they were properties, the naming guidelines for resources are similar to property guidelines.

✓ **DO** use PascalCasing in resource keys.

✓ **DO** provide descriptive rather than short identifiers.

X DO NOT use language-specific keywords of the main CLR languages.

✓ **DO** use only alphanumeric characters and underscores in naming resources.

✓ **DO** use the following naming convention for exception message resources.

The resource identifier should be the exception type name plus a short identifier of the exception:

```
ArgumentExceptionIllegalCharacters
ArgumentExceptionInvalidName
ArgumentExceptionFileNameIsMalformed
```

See Also

Framework Design Guidelines

Naming Guidelines

Type Design

Abstract Class Design

.NET Framework (current version)

X DO NOT define public or protected internal constructors in abstract types.

Constructors should be public only if users will need to create instances of the type. Because you cannot create instances of an abstract type, an abstract type with a public constructor is incorrectly designed and misleading to the users.

✓ DO define a protected or an internal constructor in abstract classes.

A protected constructor is more common and simply allows the base class to do its own initialization when subtypes are created.

An internal constructor can be used to limit concrete implementations of the abstract class to the assembly defining the class.

✓ DO provide at least one concrete type that inherits from each abstract class that you ship.

Doing this helps to validate the design of the abstract class. For example, System.IO.FileStream is an implementation of the System.IO.Stream abstract class.

See Also

Type Design Guidelines
Framework Design Guidelines

Static Class Design

.NET Framework (current version)

A static class is defined as a class that contains only static members (of course besides the instance members inherited from System.Object and possibly a private constructor). Some languages provide built-in support for static classes. In C# 2.0 and later, when a class is declared to be static, it is sealed, abstract, and no instance members can be overridden or declared.

Static classes are a compromise between pure object-oriented design and simplicity. They are commonly used to provide shortcuts to other operations (such as System.IO.File), holders of extension methods, or functionality for which a full object-oriented wrapper is unwarranted (such as System.Environment).

✓ **DO** use static classes sparingly.

Static classes should be used only as supporting classes for the object-oriented core of the framework.

X DO NOT treat static classes as a miscellaneous bucket.

X DO NOT declare or override instance members in static classes.

✓ **DO** declare static classes as sealed, abstract, and add a private instance constructor if your programming language does not have built-in support for static classes.

See Also

Type Design Guidelines
Framework Design Guidelines

Interface Design

.NET Framework (current version)

Although most APIs are best modeled using classes and structs, there are cases in which interfaces are more appropriate or are the only option.

The CLR does not support multiple inheritance (i.e., CLR classes cannot inherit from more than one base class), but it does allow types to implement one or more interfaces in addition to inheriting from a base class. Therefore, interfaces are often used to achieve the effect of multiple inheritance. For example, IDisposable is an interface that allows types to support disposability independent of any other inheritance hierarchy in which they want to participate.

The other situation in which defining an interface is appropriate is in creating a common interface that can be supported by several types, including some value types. Value types cannot inherit from types other than ValueType, but they can implement interfaces, so using an interface is the only option in order to provide a common base type.

✓ **DO** define an interface if you need some common API to be supported by a set of types that includes value types.

✓ **CONSIDER** defining an interface if you need to support its functionality on types that already inherit from some other type.

X AVOID using marker interfaces (interfaces with no members).

If you need to mark a class as having a specific characteristic (marker), in general, use a custom attribute rather than an interface.

✓ **DO** provide at least one type that is an implementation of an interface.

Doing this helps to validate the design of the interface. For example, List<T> is an implementation of the IList<T> interface.

✓ **DO** provide at least one API that consumes each interface you define (a method taking the interface as a parameter or a property typed as the interface).

Doing this helps to validate the interface design. For example, List<T>.Sort consumes the System.Collections.Generic.IComparer<T> interface.

X DO NOT add members to an interface that has previously shipped.

Doing so would break implementations of the interface. You should create a new interface in order to avoid versioning problems.

Except for the situations described in these guidelines, you should, in general, choose classes rather than interfaces in designing managed code reusable libraries.

See Also

Type Design Guidelines
Framework Design Guidelines

Enum Design

.NET Framework (current version)

Enums are a special kind of value type. There are two kinds of enums: simple enums and flag enums.

Simple enums represent small closed sets of choices. A common example of the simple enum is a set of colors.

Flag enums are designed to support bitwise operations on the enum values. A common example of the flags enum is a list of options.

✓ **DO** use an enum to strongly type parameters, properties, and return values that represent sets of values.

✓ **DO** favor using an enum instead of static constants.

X DO NOT use an enum for open sets (such as the operating system version, names of your friends, etc.).

X DO NOT provide reserved enum values that are intended for future use.

You can always simply add values to the existing enum at a later stage. See Adding Values to Enums for more details on adding values to enums. Reserved values just pollute the set of real values and tend to lead to user errors.

X AVOID publicly exposing enums with only one value.

A common practice for ensuring future extensibility of C APIs is to add reserved parameters to method signatures. Such reserved parameters can be expressed as enums with a single default value. This should not be done in managed APIs. Method overloading allows adding parameters in future releases.

X DO NOT include sentinel values in enums.

Although they are sometimes helpful to framework developers, sentinel values are confusing to users of the framework. They are used to track the state of the enum rather than being one of the values from the set represented by the enum.

✓ **DO** provide a value of zero on simple enums.

Consider calling the value something like "None." If such a value is not appropriate for this particular enum, the most common default value for the enum should be assigned the underlying value of zero.

✓ **CONSIDER** using Int32 (the default in most programming languages) as the underlying type of an enum unless any of the following is true:

- The enum is a flags enum and you have more than 32 flags, or expect to have more in the future.
- The underlying type needs to be different than Int32 for easier interoperability with unmanaged code expecting different-size enums.
- A smaller underlying type would result in substantial savings in space. If you expect the enum to be used mainly as an argument for flow of control, the size makes little difference. The size savings might be significant if:
  - You expect the enum to be used as a field in a very frequently instantiated structure or class.
  - You expect users to create large arrays or collections of the enum instances.
  - You expect a large number of instances of the enum to be serialized.

For in-memory usage, be aware that managed objects are always **DWORD**-aligned, so you effectively need multiple enums or other small structures in an instance to pack a smaller enum with in order to make a difference, because the total instance size is always going to be rounded up to a **DWORD**.

✓ **DO** name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases.

X DO NOT extend System.Enum directly.

System.Enum is a special type used by the CLR to create user-defined enumerations. Most programming languages provide a programming element that gives you access to this functionality. For example, in C# the **enum** keyword is used to define an enumeration.

Designing Flag Enums

✓ **DO** apply the System.FlagsAttribute to flag enums. Do not apply this attribute to simple enums.

✓ **DO** use powers of two for the flag enum values so they can be freely combined using the bitwise OR operation.

✓ **CONSIDER** providing special enum values for commonly used combinations of flags.

Bitwise operations are an advanced concept and should not be required for simple tasks. ReadWrite is an example of such a special value.

X AVOID creating flag enums where certain combinations of values are invalid.

X AVOID using flag enum values of zero unless the value represents "all flags are cleared" and is named appropriately, as prescribed by the next guideline.

✓ **DO** name the zero value of flag enums `None`. For a flag enum, the value must always mean "all flags are cleared."

Adding Value to Enums

It is very common to discover that you need to add values to an enum after you have already shipped it. There is a potential application compatibility problem when the newly added value is returned from an existing API, because poorly written applications might not handle the new value correctly.

✓ **CONSIDER** adding values to enums, despite a small compatibility risk.

If you have real data about application incompatibilities caused by additions to an enum, consider adding a new API that returns the new and old values, and deprecate the old API, which should continue returning just the old values. This will ensure that your existing applications remain compatible.

See Also

Type Design Guidelines
Framework Design Guidelines

Event Design

.NET Framework (current version)

Events are the most commonly used form of callbacks (constructs that allow the framework to call into user code). Other callback mechanisms include members taking delegates, virtual members, and interface-based plug-ins. Data from usability studies indicate that the majority of developers are more comfortable using events than they are using the other callback mechanisms. Events are nicely integrated with Visual Studio and many languages.

It is important to note that there are two groups of events: events raised before a state of the system changes, called pre-events, and events raised after a state changes, called post-events. An example of a pre-event would be `Form.Closing`, which is raised before a form is closed. An example of a post-event would be `Form.Closed`, which is raised after a form is closed.

✓ **DO** use the term "raise" for events rather than "fire" or "trigger."

✓ **DO** use System.EventHandler<TEventArgs> instead of manually creating new delegates to be used as event handlers.

✓ **CONSIDER** using a subclass of EventArgs as the event argument, unless you are absolutely sure the event will never need to carry any data to the event handling method, in which case you can use the `EventArgs` type directly.

If you ship an API using `EventArgs` directly, you will never be able to add any data to be carried with the event without breaking compatibility. If you use a subclass, even if initially completely empty, you will be able to add properties to the subclass when needed.

✓ **DO** use a protected virtual method to raise each event. This is only applicable to nonstatic events on unsealed classes, not to structs, sealed classes, or static events.

The purpose of the method is to provide a way for a derived class to handle the event using an override. Overriding is a more flexible, faster, and more natural way to handle base class events in derived classes. By convention, the name of the method should start with "On" and be followed with the name of the event.

The derived class can choose not to call the base implementation of the method in its override. Be prepared for this by not including any processing in the method that is required for the base class to work correctly.

✓ **DO** take one parameter to the protected method that raises an event.

The parameter should be named `e` and should be typed as the event argument class.

X DO NOT pass null as the sender when raising a nonstatic event.

✓ **DO** pass null as the sender when raising a static event.

X DO NOT pass null as the event data parameter when raising an event.

You should pass `EventArgs.Empty` if you don’t want to pass any data to the event handling method. Developers expect this parameter not to be null.

✓ **CONSIDER** raising events that the end user can cancel. This only applies to pre-events.

Use System.ComponentModel.CancelEventArgs or its subclass as the event argument to allow the end user to cancel events.

Custom Event Handler Design

There are cases in which `EventHandler<T>` cannot be used, such as when the framework needs to work with earlier versions of the CLR, which did not support Generics. In such cases, you might need to design and develop a custom event handler delegate.

✓ **DO** use a return type of void for event handlers.

An event handler can invoke multiple event handling methods, possibly on multiple objects. If event handling methods were allowed to return a value, there would be multiple return values for each event invocation.

✓ **DO** use `object` as the type of the first parameter of the event handler, and call it `sender`.

✓ **DO** use System.EventArgs or its subclass as the type of the second parameter of the event handler, and call it `e`.

X DO NOT have more than two parameters on event handlers.

See Also

Member Design Guidelines
Framework Design Guidelines

Field Design

.NET Framework (current version)

The principle of encapsulation is one of the most important notions in object-oriented design. This principle states that data stored inside an object should be accessible only to that object.

A useful way to interpret the principle is to say that a type should be designed so that changes to fields of that type (name or type changes) can be made without breaking code other than for members of the type. This interpretation immediately implies that all fields must be private.

We exclude constant and static read-only fields from this strict restriction, because such fields, almost by definition, are never required to change.

X DO NOT provide instance fields that are public or protected.

You should provide properties for accessing fields instead of making them public or protected.

✓ **DO** use constant fields for constants that will never change.

The compiler burns the values of const fields directly into calling code. Therefore, const values can never be changed without the risk of breaking compatibility.

✓ **DO** use public static `readonly` fields for predefined object instances.

If there are predefined instances of the type, declare them as public read-only static fields of the type itself.

X DO NOT assign instances of mutable types to `readonly` fields.

A mutable type is a type with instances that can be modified after they are instantiated. For example, arrays, most collections, and streams are mutable types, but System.Int32, System.Uri, and System.String are all immutable. The read-only modifier on a reference type field prevents the instance stored in the field from being replaced, but it does not prevent the field's instance data from being modified by calling members changing the instance.

See Also

Member Design Guidelines
Framework Design Guidelines

Extension Methods

.NET Framework (current version)

Extension methods are a language feature that allows static methods to be called using instance method call syntax. These methods must take at least one parameter, which represents the instance the method is to operate on.

The class that defines such extension methods is referred to as the "sponsor" class, and it must be declared as static. To use extension methods, one must import the namespace defining the sponsor class.

X AVOID frivolously defining extension methods, especially on types you don't own.

If you do own source code of a type, consider using regular instance methods instead. If you don't own, and you want to add a method, be very careful. Liberal use of extension methods has the potential of cluttering APIs of types that were not designed to have these methods.

✓ **CONSIDER** using extension methods in any of the following scenarios:

- To provide helper functionality relevant to every implementation of an interface, if said functionality can be written in terms of the core interface. This is because concrete implementations cannot otherwise be assigned to interfaces. For example, the `LINQ to`

`Objects` operators are implemented as extension methods for all IEnumerable<T> types. Thus, any `IEnumerable<>` implementation is automatically LINQ-enabled.

- When an instance method would introduce a dependency on some type, but such a dependency would break dependency management rules. For example, a dependency from String to System.Uri is probably not desirable, and so `String.ToUri()` instance method returning `System.Uri` would be the wrong design from a dependency management perspective. A static extension method `Uri.ToUri(this string str)` returning `System.Uri` would be a much better design.

X AVOID defining extension methods on System.Object.

VB users will not be able to call such methods on object references using the extension method syntax. VB does not support calling such methods because, in VB, declaring a reference as Object forces all method invocations on it to be late bound (actual member called is determined at runtime), while bindings to extension methods are determined at compile-time (early bound).

Note that the guideline applies to other languages where the same binding behavior is present, or where extension methods are not supported.

X DO NOT put extension methods in the same namespace as the extended type unless it is for adding methods to interfaces or for dependency management.

X AVOID defining two or more extension methods with the same signature, even if they reside in different namespaces.

✓ **CONSIDER** defining extension methods in the same namespace as the extended type if the type is an interface and if the extension methods are meant to be used in most or all cases.

X DO NOT define extension methods implementing a feature in namespaces normally associated with other features. Instead, define them in the namespace associated with the feature they belong to.

X AVOID generic naming of namespaces dedicated to extension methods (e.g., "Extensions"). Use a descriptive name (e.g., "Routing") instead.

See Also

Member Design Guidelines
Framework Design Guidelines

Parameter Design

.NET Framework (current version)

This section provides broad guidelines on parameter design, including sections with guidelines for checking arguments. In addition, you should refer to the guidelines described in Naming Parameters.

✓ **DO** use the least derived parameter type that provides the functionality required by the member.

For example, suppose you want to design a method that enumerates a collection and prints each item to the console. Such a method should take IEnumerable as the parameter, not ArrayList or IList, for example.

X DO NOT use reserved parameters.

If more input to a member is needed in some future version, a new overload can be added.

X DO NOT have publicly exposed methods that take pointers, arrays of pointers, or multidimensional arrays as parameters.

Pointers and multidimensional arrays are relatively difficult to use properly. In almost all cases, APIs can be redesigned to avoid taking these types as parameters.

✓ **DO** place all `out` parameters following all of the by-value and `ref` parameters (excluding parameter arrays), even if it results in an inconsistency in parameter ordering between overloads (see Member Overloading).

The `out` parameters can be seen as extra return values, and grouping them together makes the method signature easier to understand.

✓ **DO** be consistent in naming parameters when overriding members or implementing interface members.

This better communicates the relationship between the methods.

Choosing Between Enum and Boolean Parameters

✓ **DO** use enums if a member would otherwise have two or more Boolean parameters.

X DO NOT use Booleans unless you are absolutely sure there will never be a need for more than two values.

Enums give you some room for future addition of values, but you should be aware of all the implications of adding values to enums, which are described in Enum Design.

✓ **CONSIDER** using Booleans for constructor parameters that are truly two-state values and are simply used to initialize Boolean properties.

Validating Arguments

✓ **DO** validate arguments passed to public, protected, or explicitly implemented members. Throw System.ArgumentException, or one of its subclasses, if the validation fails.

Note that the actual validation does not necessarily have to happen in the public or protected member itself. It could happen at a lower level in some private or internal routine. The main point is that the entire surface area that is exposed to the end users checks the arguments.

✓ **DO** throw ArgumentNullException if a null argument is passed and the member does not support null arguments.

✓ **DO** validate enum parameters.

Do not assume enum arguments will be in the range defined by the enum. The CLR allows casting any integer value into an enum value even if the value is not defined in the enum.

X DO NOT use Enum.IsDefined for enum range checks.

✓ **DO** be aware that mutable arguments might have changed after they were validated.

If the member is security sensitive, you are encouraged to make a copy and then validate and process the argument.

Parameter Passing

From the perspective of a framework designer, there are three main groups of parameters: by-value parameters, `ref` parameters, and `out` parameters.

When an argument is passed through a by-value parameter, the member receives a copy of the actual argument passed in. If the argument is a value type, a copy of the argument is put on the stack. If the argument is a reference type, a copy of the reference is put on the stack. Most popular CLR languages, such as C#, VB.NET, and C++, default to passing parameters by value.

When an argument is passed through a `ref` parameter, the member receives a reference to the actual argument passed in. If the argument is a value type, a reference to the argument is put on the stack. If the argument is a reference type, a reference to the reference is put on the stack. `Ref` parameters can be used to allow the member to modify arguments passed by the caller.

`Out` parameters are similar to `ref` parameters, with some small differences. The parameter is initially considered unassigned and cannot be read in the member body before it is assigned some value. Also, the parameter has to be assigned some value before the member returns.

X AVOID using `out` or `ref` parameters.

Using `out` or `ref` parameters requires experience with pointers, understanding how value types and reference types differ, and handling methods with multiple return values. Also, the difference between `out` and `ref` parameters is not widely understood. Framework architects designing for a general audience should not expect users to master working with `out` or `ref` parameters.

X DO NOT pass reference types by reference.

There are some limited exceptions to the rule, such as a method that can be used to swap references.

Members with Variable Number of Parameters

Members that can take a variable number of arguments are expressed by providing an array parameter. For example, String provides the following method:

```
public class String {
    public static string Format(string format, object[] parameters);
}
```

A user can then call the String.Format method, as follows:

```
String.Format("File {0} not found in {1}",new object[]{filename,directory});
```

Adding the C# params keyword to an array parameter changes the parameter to a so-called params array parameter and provides a shortcut to creating a temporary array.

```
public class String {
    public static string Format(string format, params object[] parameters)
}
```

Doing this allows the user to call the method by passing the array elements directly in the argument list.

```
String.Format("File {0} not found in {1}",filename,directory);
```

Note that the params keyword can be added only to the last parameter in the parameter list.

✓ **CONSIDER** adding the params keyword to array parameters if you expect the end users to pass arrays with a small number of elements. If it's expected that lots of elements will be passed in common scenarios, users will probably not pass these elements inline anyway, and so the params keyword is not necessary.

X AVOID using params arrays if the caller would almost always have the input already in an array.

For example, members with byte array parameters would almost never be called by passing individual bytes. For this reason, byte array parameters in the .NET Framework do not use the params keyword.

X DO NOT use params arrays if the array is modified by the member taking the params array parameter.

Because of the fact that many compilers turn the arguments to the member into a temporary array at the call site, the array might be a temporary object, and therefore any modifications to the array will be lost.

✓ **CONSIDER** using the params keyword in a simple overload, even if a more complex overload could not use it.

Ask yourself if users would value having the params array in one overload even if it wasn't in all overloads.

✓ **DO** try to order parameters to make it possible to use the params keyword.

✓ **CONSIDER** providing special overloads and code paths for calls with a small number of arguments in extremely performance-sensitive APIs.

This makes it possible to avoid creating array objects when the API is called with a small number of arguments. Form the names of the parameters by taking a singular form of the array parameter and adding a numeric suffix.

You should only do this if you are going to special-case the entire code path, not just create an array and call the more general method.

✓ **DO** be aware that null could be passed as a params array argument.

You should validate that the array is not null before processing.

X DO NOT use the `varargs` methods, otherwise known as the ellipsis.

Some CLR languages, such as C++, support an alternative convention for passing variable parameter lists called `varargs` methods. The convention should not be used in frameworks, because it is not CLS compliant.

Pointer Parameters

In general, pointers should not appear in the public surface area of a well-designed managed code framework. Most of the time, pointers should be encapsulated. However, in some cases pointers are required for interoperability reasons, and using pointers in such cases is appropriate.

✓ **DO** provide an alternative for any member that takes a pointer argument, because pointers are not CLS-compliant.

X AVOID doing expensive argument checking of pointer arguments.

✓ **DO** follow common pointer-related conventions when designing members with pointers.

For example, there is no need to pass the start index, because simple pointer arithmetic can be used to accomplish the same result.

See Also

Member Design Guidelines
Framework Design Guidelines

Nested Types

.NET Framework (current version)

A nested type is a type defined within the scope of another type, which is called the enclosing type. A nested type has access to all members of its enclosing type. For example, it has access to private fields defined in the enclosing type and to protected fields defined in all ascendants of the enclosing type.

In general, nested types should be used sparingly. There are several reasons for this. Some developers are not fully familiar with the concept. These developers might, for example, have problems with the syntax of declaring variables of nested types. Nested types are also very tightly coupled with their enclosing types, and as such are not suited to be general-purpose types.

Nested types are best suited for modeling implementation details of their enclosing types. The end user should rarely have to declare variables of a nested type and almost never should have to explicitly instantiate nested types. For example, the enumerator of a collection can be a nested type of that collection. Enumerators are usually instantiated by their enclosing type, and because many languages support the foreach statement, enumerator variables rarely have to be declared by the end user.

✓ **DO** use nested types when the relationship between the nested type and its outer type is such that member-accessibility semantics are desirable.

X DO NOT use public nested types as a logical grouping construct; use namespaces for this.

X AVOID publicly exposed nested types. The only exception to this is if variables of the nested type need to be declared only in rare scenarios such as subclassing or other advanced customization scenarios.

X DO NOT use nested types if the type is likely to be referenced outside of the containing type.

For example, an enum passed to a method defined on a class should not be defined as a nested type in the class.

X DO NOT use nested types if they need to be instantiated by client code. If a type has a public constructor, it should probably not be nested.

If a type can be instantiated, that seems to indicate the type has a place in the framework on its own (you can create it, work with it, and destroy it without ever using the outer type), and thus should not be nested. Inner types should not be widely reused outside of the outer type without any relationship whatsoever to the outer type.

X DO NOT define a nested type as a member of an interface. Many languages do not support such a construct.

See Also

Type Design Guidelines
Framework Design Guidelines

Exceptions

Design Guidelines for Exceptions

.NET Framework (current version)

Exception handling has many advantages over return-value-based error reporting. Good framework design helps the application developer realize the benefits of exceptions. This section discusses the benefits of exceptions and presents guidelines for using them effectively.

In This Section

See Also

Framework Design Guidelines

Exception Throwing

.NET Framework (current version)

Exception-throwing guidelines described in this section require a good definition of the meaning of execution failure. Execution failure occurs whenever a member cannot do what it was designed to do (what the member name implies). For example, if the `OpenFile` method cannot return an opened file handle to the caller, it would be considered an execution failure.

Most developers have become comfortable with using exceptions for usage errors such as division by zero or null references. In the Framework, exceptions are used for all error conditions, including execution errors.

X DO NOT return error codes.

Exceptions are the primary means of reporting errors in frameworks.

✓ **DO** report execution failures by throwing exceptions.

✓ **CONSIDER** terminating the process by calling `System.Environment.FailFast` (.NET Framework 2.0 feature) instead of throwing an exception if your code encounters a situation where it is unsafe for further execution.

X DO NOT use exceptions for the normal flow of control, if possible.

Except for system failures and operations with potential race conditions, framework designers should design APIs so users can write code that does not throw exceptions. For example, you can provide a way to check preconditions before calling a member so users can write code that does not throw exceptions.

The member used to check preconditions of another member is often referred to as a tester, and the member that actually does the work is called a doer.

There are cases when the Tester-Doer Pattern can have an unacceptable performance overhead. In such cases, the so-called Try-Parse Pattern should be considered (see Exceptions and Performance for more information).

✓ **CONSIDER** the performance implications of throwing exceptions. Throw rates above 100 per second are likely to noticeably impact the performance of most applications.

✓ **DO** document all exceptions thrown by publicly callable members because of a violation of the member contract (rather than a system failure) and treat them as part of your contract.

Exceptions that are a part of the contract should not change from one version to the next (i.e., exception type should not change, and new exceptions should not be added).

X DO NOT have public members that can either throw or not based on some option.

X DO NOT have public members that return exceptions as the return value or an `out` parameter.

Returning exceptions from public APIs instead of throwing them defeats many of the benefits of exception-based error reporting.

✓ **CONSIDER** using exception builder methods.

It is common to throw the same exception from different places. To avoid code bloat, use helper methods that create exceptions and initialize their properties.

Also, members that throw exceptions are not getting inlined. Moving the throw statement inside the builder might allow the member to be inlined.

X DO NOT throw exceptions from exception filter blocks.

When an exception filter raises an exception, the exception is caught by the CLR, and the filter returns false. This behavior is indistinguishable from the filter executing and returning false explicitly and is therefore very difficult to debug.

X AVOID explicitly throwing exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition by Krzysztof Cwalina and Brad Abrams, published Oct 22, 2008 by Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Framework Design Guidelines
Design Guidelines for Exceptions

Using Standard Exception Types

.NET Framework (current version)

This section describes the standard exceptions provided by the Framework and the details of their usage. The list is by no means exhaustive. Please refer to the .NET Framework reference documentation for usage of other Framework exception types.

Exception and SystemException

X DO NOT throw System.Exception or System.SystemException.

X DO NOT catch `System.Exception` or `System.SystemException` in framework code, unless you intend to rethrow.

X AVOID catching `System.Exception` or `System.SystemException`, except in top-level exception handlers.

ApplicationException

X DO NOT throw or derive from ApplicationException.

InvalidOperationException

✓ **DO** throw an InvalidOperationException if the object is in an inappropriate state.

ArgumentException, ArgumentNullException, and ArgumentOutOfRangeException

✓ **DO** throw ArgumentException or one of its subtypes if bad arguments are passed to a member. Prefer the most derived exception type, if applicable.

✓ **DO** set the `ParamName` property when throwing one of the subclasses of `ArgumentException`.

This property represents the name of the parameter that caused the exception to be thrown. Note that the property can be set using one of the constructor overloads.

✓ **DO** use `value` for the name of the implicit value parameter of property setters.

NullReferenceException, IndexOutOfRangeException, and AccessViolationException

X DO NOT allow publicly callable APIs to explicitly or implicitly throw NullReferenceException, AccessViolationException, or IndexOutOfRangeException. These exceptions are reserved and thrown by the execution engine and in most cases indicate a bug.

Do argument checking to avoid throwing these exceptions. Throwing these exceptions exposes implementation details of your method that might change over time.

StackOverflowException

X DO NOT explicitly throw StackOverflowException. The exception should be explicitly thrown only by the CLR.

X DO NOT catch `StackOverflowException`.

It is almost impossible to write managed code that remains consistent in the presence of arbitrary stack overflows. The unmanaged parts of the CLR remain consistent by using probes to move stack overflows to well-defined places rather than by backing out from arbitrary stack overflows.

OutOfMemoryException

X DO NOT explicitly throw OutOfMemoryException. This exception is to be thrown only by the CLR infrastructure.

ComException, SEHException, and ExecutionEngineException

X DO NOT explicitly throw COMException, ExecutionEngineException, and SEHException. These exceptions are to be thrown only by the CLR infrastructure.

See Also

Framework Design Guidelines
Design Guidelines for Exceptions

Exceptions and Performance

.NET Framework (current version)

One common concern related to exceptions is that if exceptions are used for code that routinely fails, the performance of the implementation will be unacceptable. This is a valid concern. When a member throws an exception, its performance can be orders of magnitude slower. However, it is possible to achieve good performance while strictly adhering to the exception guidelines that disallow using error codes. Two patterns described in this section suggest ways to do this.

X DO NOT use error codes because of concerns that exceptions might affect performance negatively.

To improve performance, it is possible to use either the Tester-Doer Pattern or the Try-Parse Pattern, described in the next two sections.

Tester-Doer Pattern

Sometimes performance of an exception-throwing member can be improved by breaking the member into two. Let's look at the Add method of the ICollection<T> interface.

```
ICollection<int> numbers = ...
numbers.Add(1);
```

The method Add throws if the collection is read-only. This can be a performance problem in scenarios where the method call is expected to fail often. One of the ways to mitigate the problem is to test whether the collection is writable before trying to add a value.

```
ICollection<int> numbers = ...
```

```
...
if(!numbers.IsReadOnly){
    numbers.Add(1);
}
```

The member used to test a condition, which in our example is the property `IsReadOnly`, is referred to as the tester. The member used to perform a potentially throwing operation, the `Add` method in our example, is referred to as the doer.

✓ **CONSIDER** the Tester-Doer Pattern for members that might throw exceptions in common scenarios to avoid performance problems related to exceptions.

Try-Parse Pattern

For extremely performance-sensitive APIs, an even faster pattern than the Tester-Doer Pattern described in the previous section should be used. The pattern calls for adjusting the member name to make a well-defined test case a part of the member semantics. For example, DateTime defines a Parse method that throws an exception if parsing of a string fails. It also defines a corresponding TryParse method that attempts to parse, but returns false if parsing is unsuccessful and returns the result of a successful parsing using an `out` parameter.

```
public struct DateTime {
    public static DateTime Parse(string dateTime){
        ...
    }
    public static bool TryParse(string dateTime, out DateTime result){
        ...
    }
}
```

When using this pattern, it is important to define the try functionality in strict terms. If the member fails for any reason other than the well-defined try, the member must still throw a corresponding exception.

✓ **CONSIDER** the Try-Parse Pattern for members that might throw exceptions in common scenarios to avoid performance problems related to exceptions.

✓ **DO** use the prefix "Try" and Boolean return type for methods implementing this pattern.

✓ **DO** provide an exception-throwing member for each member using the Try-Parse Pattern.

See Also

Framework Design Guidelines
Design Guidelines for Exceptions