C Style Guide

Release 0.1

Adam Greene

Nov 12, 2017

Contents

Welcome to the Cognitive X Solutions C# Style Guide. The follow guide will outline to you how Cognitive X writes C# code. The guidelines are laid out in a simple format prefixed with the terms DO, CONSIDER, AVOID, DON'T. There are times when these guidelines might need to be violated, but there should be a clear and compelling reason to do so (and should be discussed with team before doing so).

This guide is based on https://msdn.microsoft.com/en-us/library/ms229042.aspx

CHAPTER 1

Naming Conventions

1.1 Identifiers

DO follow the these rules about capitalization when naming identifiers:

Identifier	Casing	Example
Namespace	Pascal	namespace **System.Security** { ... }
Type	Pascal	public class **StreamReader** { ... }
Interface	Pascal	public interface **IEnumerable** { ... }
Method	Pascal	public virtual string **ToString()** { ... }
Property	Pascal	public string **Name** { get { ... } }
Event	Pascal	public event EventHandler **Exited**;
Public Field	Pascal	public static readonly TimeSpan **InfiniteTimeout**;
Public Const	Pascal	public const **Min** = 0;
Private Field	Camel	private string **url**;
Enum Value	Pascal	public enum FileMode { **Append**, ... }
Parameter	Camel	public static int ToInt32(string **value**);

1.1.1 Rules for Compound Words

DO follow these examples about compound words:

Pascal	Camel	Not
BitFlag	bitFlag	Bitflag
Callback[1]	Callback	CallBack
Canceled	Canceled	Cancelled
DoNot	doNot	Don't
Email	Email	EMail
Endpoint[1]	Endpoint	EndPoint
FileName	filename	Filename
Gridline[1]	Gridline	GridLine
Hashtable[1]	Hashtable	HashTable
Id	id	ID
Indexes	indexes	Indices
LogOff	logOff	LogOut
LogOn	logOn	LogIn
Metadata[1]	metadata	MetaData
Multipanel[1]	multipanel	MultiPanel
Multiview[1]	multiview	MultiView
Namespace[1]	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

1.1.2 Case Sensitivity

DO NOT vary identifiers based solely on capitalization. The CLR in general does not require that identifiers vary based on sensitivity (Age is the same as age). C# does support it (as it is a C type language), but the only time it is acceptable to do so is in field backed properties, as follows:

```
class Person {
    private int age;

    public int Age {
        return age;
    }
}
```

1.2 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.

[1] These are what are called closed-form compound words and should be treated as a single word. If you would write it as a single word in a sentence, do so in an identifier as well.

1.2.1 Word choice

DO choose easily readable, descriptive names. HorizontalAlignment rather than AlignmentHorizontal.

DO favour readability over brevity. CanScrollHorizontally rather than ScrollableX.

DO NOT use underscores, hyphens, or any other non-alphanumeric characters.

DO NOT use Hungarian notation. Cost should be cost not dCost (where the "d" would indicate that its decimal).

AVOID naming variables the same as language keywords. You should not have an identifier called **private** as that is a reserved word. Most languages won't allow it anyways. In C#, however, you can prefix a name with a "commercial at" (@) to escape reserved words. Avoid this except in cases where it doesn't make sense. Example: @class is preferable to clazz in code dealing with CSS generation.

1.2.2 Abbreviations and Acronyms

DO NOT use abbreviations and/or acronyms as part of identifier names. GetWindow not GetWin. The most obvious places where this might need to be violated is dealing with 3rd party REST APIs. But in those cases there are override mechanisms available in the JSON and XML serialization mechanisms to override naming. Example:

```
public class Screen {
    [XmlAttribute("Win")]
    public string Window { get; set; }
}
```

Sometimes, based on business rules, it might be appropriate to utilize acronyms. But this, again, is only when part of an API and is in the public contract of the API (internal code should strive to avoid use of the acronyms).

1.2.3 Avoid Language Specific Names

DO use semantically interesting names rather than language-specific keywords for type names. Use GetLength not GetInt.

DO use generic CLR type names, rather than language specific ones, in the rare cases when an identifier has no semantic meaning beyond its type. Given a class called Convert, the method would be ToInt64 not ToLong as long is the C# keyword for the underlying CLR type Int64. So when naming methods that need to include a type name:

Use this, instead of:	C#	Visual Basic	C++
SByte	sbyte	SByte	char
Byte	byte	Byte	unsigned char
Int16	short	Short	short
UInt16	ushort	UInt16	unsigned short
Int32	int	Integer	int
UInt32	uint	UInt32	unsigned int
Int64	long	Long	__int64
UInt64	ulong	UInt64	unsigned __int64
Single	float	Single	float
Double	double	Double	double
Boolean	bool	Boolean	bool
Char	char	Char	wchar_t
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.

1.3 Naming New Versions of an Existing API

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.

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.

1.4 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 namespaces 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.

1.5 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: CognitiveX.Math, Dovico.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.

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.

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.

1.6 Names of Classes, Structs, and Interfaces

The naming guidelines that follow apply to general type naming.

DO name classes and structs with nouns or noun phrases, using PascalCasing. Examples: User, Database, Order, LineItem.

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.

For example, IComponent (descriptive noun), ICustomAttributeProvider (noun phrase), and IPersistable (adjective) are appropriate interface names. As with other type names, avoid abbreviations. Names of interfaces end many times in -able, -Provider.

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. Example: Component (class), IComponent (interface).

Nouns and noun phrases should be used rarely and they might indicate that the type should be an abstract class, and not an interface. Our example of Component and IComponent could also be solved by simply making Component an abstract class.

DO NOT give class names a prefix (e.g., “C”).

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

Base Class	Derived Class
Validator	StringValidator NumberValidator RegexValidator
Field	StringField NumberField RegexField

This is very readable and explains the relationship clearly.

DO use reasonable judgment in applying this guideline; for example, the Button class is a kind of Control, although Control doesn’t appear in its name.

1.6.1 Names of Generic Type Parameters

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. Example, a type parameter constrainted to ISession might be called TSession.

1.6.2 Names of Common Types

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

System.Attribute

DO add the suffix "Attribute" to names of custom attribute classes. add the suffix "Attribute" to names of custom attribute classes. Example:

```
public class IgnoreAttribute : Attribute {}
```

System.Delegate

DO add the suffix "EventHandler" to names of delegates that are used in events. Example:

```
public delegate void OnClickEventHandler(object sender, ClickEventArgs args);
```

CONSIDER adding the suffix "Callback" to names of delegates other than those used as event handlers. Example:

```
public delegate bool HandleMessageCallback(Message message);
```

DO NOT add the suffix "Delegate" to a delegate.

System.EventArgs

DO add the suffix "EventArgs." Example:

```
public class ClickEventArgs : EventArgs {}
```

System.Enum

DO NOT derive from this class; use the keyword supported by your language instead; for example, in C#, use the enum keyword.

DO NOT add the suffix "Enum" or "Flag."

System.Exception

DO add the suffix "Exception." Example:

```
public class InvalidStateException : 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. Example:

```
public class ClientAddressDictionary : IDictionary<Client, Address> { ... }
```

IEnumerable ICollection IList IEnumerable<T> ICollection<T> IList<T>

DO add the suffix "Collection." Example:

```
public class UserCollection : IList<User> { ... }
```

System.IO.Stream

DO add the suffix "Stream." Example:

```
public class TCPIPStream : Stream { ... }
```

CodeAccessPermission IPermission

DO add the suffix "Permission."

1.6.3 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. Example:

```
public enum Severity { Mild, Medium, Serious, Nuclear }
```

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

```
[Flags]
enum Visibilities {
    Private = 1,
    Internal = 2,
    Public = 4
}
```

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

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

DO NOT use a prefix on enumeration value names (e.g., "ad" for ADO enums, "rtf" for rich text enums, etc.). Don't do:

```
enum Visibility { visPublic, visInternal, visPublic }
```

1.7 Names of Type Members

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

1.7.1 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();
}
```

1.7.2 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.

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 {...} }
}
```

1.7.3 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.

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.

1.7.4 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.

DO NOT use a prefix for field names.

For example, do not use “**g_**” or “**s_**” to indicate static fields.

1.8 Naming Parameters

Beyond the obvious reason of readability, it is important to follow the guidelines for parameter names because parameters are displaysed 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.

1.8.1 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.

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

1.9 Naming Resources

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.

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
```

CHAPTER 2

Type Design Guidelines

From the CLR perspective, there are only two categories of types—reference types and value types—but for the purpose of a discussion about framework design, we divide types into more logical groups, each with its own specific design rules.

Classes are the general case of reference types. They make up the bulk of types in the majority of frameworks. Classes owe their popularity to the rich set of object-oriented features they support and to their general applicability. Base classes and abstract classes are special logical groups related to extensibility.

Interfaces are types that can be implemented by both reference types and value types. They can thus serve as roots of polymorphic hierarchies of reference types and value types. In addition, interfaces can be used to simulate multiple inheritance, which is not natively supported by the CLR.

Structs are the general case of value types and should be reserved for small, simple types, similar to language primitives.

Enums are a special case of value types used to define short sets of values, such as days of the week, console colors, and so on.

Static classes are types intended to be containers for static members. They are commonly used to provide shortcuts to other operations.

Delegates, exceptions, attributes, arrays, and collections are all special cases of reference types intended for specific uses, and guidelines for their design and usage are discussed elsewhere in this book.

DO ensure that each type is a well-defined set of related members, not just a random collection of unrelated functionality.

2.1 Choosing Between Class and Struct

One of the basic design decisions every framework designer faces is whether to design a type as a class (a reference type) or as a struct (a value type). Good understanding of the differences in the behavior of reference types and value types is crucial in making this choice.

The first difference between reference types and value types we will consider is that reference types are allocated on the heap and garbage-collected, whereas value types are allocated either on the stack or inline in containing types and deallocated when the stack unwinds or when their containing type gets deallocated. Therefore, allocations and deallocations of value types are in general cheaper than allocations and deallocations of reference types.

Next, arrays of reference types are allocated out-of-line, meaning the array elements are just references to instances of the reference type residing on the heap. Value type arrays are allocated inline, meaning that the array elements are the actual instances of the value type. Therefore, allocations and deallocations of value type arrays are much cheaper than allocations and deallocations of reference type arrays. In addition, in a majority of cases value type arrays exhibit much better locality of reference.

The next difference is related to memory usage. Value types get boxed when cast to a reference type or one of the interfaces they implement. They get unboxed when cast back to the value type. Because boxes are objects that are allocated on the heap and are garbage-collected, too much boxing and unboxing can have a negative impact on the heap, the garbage collector, and ultimately the performance of the application. In contrast, no such boxing occurs as reference types are cast.

Next, reference type assignments copy the reference, whereas value type assignments copy the entire value. Therefore, assignments of large reference types are cheaper than assignments of large value types.

Finally, reference types are passed by reference, whereas value types are passed by value. Changes to an instance of a reference type affect all references pointing to the instance. Value type instances are copied when they are passed by value. When an instance of a value type is changed, it of course does not affect any of its copies. Because the copies are not created explicitly by the user but are implicitly created when arguments are passed or return values are returned, value types that can be changed can be confusing to many users. Therefore, value types should be immutable.

As a rule of thumb, the majority of types in a framework should be classes. There are, however, some situations in which the characteristics of a value type make it more appropriate to use structs.

CONSIDER defining a struct instead of a class if instances of the type are small and commonly short-lived or are commonly embedded in other objects.

AVOID defining a struct unless the type has all of the following characteristics:

- It logically represents a single value, similar to primitive types (int, double, etc.).
- It has an instance size under 16 bytes.
- It is immutable.
- It will not have to be boxed frequently.
- In all other cases, you should define your types as classes.

2.2 Abstract Class Design

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.

2.3 Static Class Design

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.

DO NOT treat static classes as a miscellaneous bucket.

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.

2.4 Interface Design

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.

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.

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.

2.5 Struct Design

The general-purpose value type is most often referred to as a struct, its C# keyword. This section provides guidelines for general struct design.

DO NOT provide a default constructor for a struct.

Following this guideline allows arrays of structs to be created without having to run the constructor on each item of the array. Notice that C# does not allow structs to have default constructors.

DO NOT define mutable value types.

Mutable value types have several problems. For example, when a property getter returns a value type, the caller receives a copy. Because the copy is created implicitly, developers might not be aware that they are mutating the copy, and not the original value. Also, some languages (dynamic languages, in particular) have problems using mutable value types because even local variables, when dereferenced, cause a copy to be made.

DO ensure that a state where all instance data is set to zero, false, or null (as appropriate) is valid.

This prevents accidental creation of invalid instances when an array of the structs is created.

DO implement IEquatable<T> on value types.

The Object.Equals method on value types causes boxing, and its default implementation is not very efficient, because it uses reflection. Equals can have much better performance and can be implemented so that it will not cause boxing.

DO NOT explicitly extend ValueType. In fact, most languages prevent this.

In general, structs can be very useful but should only be used for small, single, immutable values that will not be boxed frequently.

2.6 Enum Design

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.

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

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.

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.

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.

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.

2.6.1 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.

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

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."

2.6.2 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.

2.7 Nested Types

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.

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

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.

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.

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.

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

CHAPTER 3

Member Design Guidelines

Methods, properties, events, constructors, and fields are collectively referred to as members. Members are ultimately the means by which framework functionality is exposed to the end users of a framework.

Members can be virtual or nonvirtual, concrete or abstract, static or instance, and can have several different scopes of accessibility. All this variety provides incredible expressiveness but at the same time requires care on the part of the framework designer.

This chapter offers basic guidelines that should be followed when designing members of any type.

3.1 Member Overloading

Member overloading means creating two or more members on the same type that differ only in the number or type of parameters but have the same name. For example, in the following, the WriteLine method is overloaded:

```
public static class Console {
    public void WriteLine();
    public void WriteLine(string value);
    public void WriteLine(bool value);
    ...
}
```

Because only methods, constructors, and indexed properties can have parameters, only those members can be overloaded.

Overloading is one of the most important techniques for improving usability, productivity, and readability of reusable libraries. Overloading on the number of parameters makes it possible to provide simpler versions of constructors and methods. Overloading on the parameter type makes it possible to use the same member name for members performing identical operations on a selected set of different types.

DO try to use descriptive parameter names to indicate the default used by shorter overloads.

AVOID arbitrarily varying parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name.

AVOID being inconsistent in the ordering of parameters in overloaded members. Parameters with the same name should appear in the same position in all overloads.

DO make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.

DO NOT use ref or out modifiers to overload members.

Some languages cannot resolve calls to overloads like this. In addition, such overloads usually have completely different semantics and probably should not be overloads but two separate methods instead.

DO NOT have overloads with parameters at the same position and similar types yet with different semantics.

DO allow null to be passed for optional arguments.

DO use member overloading rather than defining members with default arguments when you are creating libraries that will be used outside the company (or from a different language, as default arguments are not CLS compliant).

3.2 Property Design

Although properties are technically very similar to methods, they are quite different in terms of their usage scenarios. They should be seen as smart fields. They have the calling syntax of fields, and the flexibility of methods.

DO create get-only properties if the caller should not be able to change the value of the property.

Keep in mind that if the type of the property is a mutable reference type, the property value can be changed even if the property is get-only.

DO NOT provide set-only properties or properties with the setter having broader accessibility than the getter.

For example, do not use properties with a public setter and a protected getter.

If the property getter cannot be provided, implement the functionality as a method instead. Consider starting the method name with Set and follow with what you would have named the property. For example, AppDomain has a method called SetCachePath instead of having a set-only property called CachePath.

DO provide sensible default values for all properties, ensuring that the defaults do not result in a security hole or terribly inefficient code.

DO allow properties to be set in any order even if this results in a temporary invalid state of the object.

It is common for two or more properties to be interrelated to a point where some values of one property might be invalid given the values of other properties on the same object. In such cases, exceptions resulting from the invalid state should be postponed until the interrelated properties are actually used together by the object.

DO preserve the previous value if a property setter throws an exception. Which means that you should check the validity of a value before storing the new value to the underlying field.

AVOID throwing exceptions from property getters.

Property getters should be simple operations and should not have any preconditions. If a getter can throw an exception, it should probably be redesigned to be a method. Notice that this rule does not apply to indexers, where we do expect exceptions as a result of validating the arguments.

3.2.1 Indexed Property Design

An indexed property is a special property that can have parameters and can be called with special syntax similar to array indexing.

Indexed properties are commonly referred to as indexers. Indexers should be used only in APIs that provide access to items in a logical collection. For example, a string is a collection of characters, and the indexer on System.String was added to access its characters.

CONSIDER using indexers to provide access to data stored in an internal array.

CONSIDER providing indexers on types representing collections of items.

AVOID using indexed properties with more than one parameter.

If the design requires multiple parameters, reconsider whether the property really represents an accessor to a logical collection. If it does not, use methods instead. Consider starting the method name with Get or Set.

AVOID indexers with parameter types other than System.Int32, System.Int64, System.String, System.Object, or an enum.

If the design requires other types of parameters, strongly reevaluate whether the API really represents an accessor to a logical collection. If it does not, use a method. Consider starting the method name with Get or Set.

DO use the name Item for indexed properties unless there is an obviously better name (e.g., see the Chars property on System.String).

In C#, indexers are by default named Item. The IndexerNameAttribute can be used to customize this name.

DO NOT provide both an indexer and methods that are semantically equivalent.

DO NOT provide more than one family of overloaded indexers in one type.

This is enforced by the C# compiler.

DO NOT use nondefault indexed properties.

This is enforced by the C# compiler.

3.2.2 Property Change Notification Events

Sometimes it is useful to provide an event notifying the user of changes in a property value. For example, System.Windows.Forms.Control raises a TextChanged event after the value of its Text property has changed.

CONSIDER raising change notification events when property values in high-level APIs (usually designer components) are modified.

If there is a good scenario for a user to know when a property of an object is changing, the object should raise a change notification event for the property.

However, it is unlikely to be worth the overhead to raise such events for low-level APIs such as base types or collections. For example, List<T> would not raise such events when a new item is added to the list and the Count property changes.

CONSIDER raising change notification events when the value of a property changes via external forces.

If a property value changes via some external force (in a way other than by calling methods on the object), raise events indicate to the developer that the value is changing and has changed. A good example is the Text property of a text box control. When the user types text in a TextBox, the property value automatically changes.

3.3 Constructor Design

There are two kinds of constructors: type constructors and instance constructors.

Type constructors are static and are run by the CLR before the type is used. Instance constructors run when an instance of a type is created.

Type constructors cannot take any parameters. Instance constructors can. Instance constructors that don't take any parameters are often called default constructors.

Constructors are the most natural way to create instances of a type. Most developers will search and try to use a constructor before they consider alternative ways of creating instances (such as factory methods).

CONSIDER providing simple, ideally default, constructors.

A simple constructor has a very small number of parameters, and all parameters are primitives or enums. Such simple constructors increase usability of the framework.

CONSIDER using a static factory method instead of a constructor if the semantics of the desired operation do not map directly to the construction of a new instance, or if following the constructor design guidelines feels unnatural.

DO use constructor parameters as shortcuts for setting main properties.

There should be no difference in semantics between using the empty constructor followed by some property sets and using a constructor with multiple arguments.

DO use the same name for constructor parameters and a property if the constructor parameters are used to simply set the property.

The only difference between such parameters and the properties should be casing.

DO minimal work in the constructor.

Constructors should not do much work other than capture the constructor parameters. The cost of any other processing should be delayed until required.

DO throw exceptions from instance constructors, if appropriate.

DO explicitly declare the public default constructor in classes, if such a constructor is required.

If you don't explicitly declare any constructors on a type, many languages (such as C#) will automatically add a public default constructor. (Abstract classes get a protected constructor.)

Adding a parameterized constructor to a class prevents the compiler from adding the default constructor. This often causes accidental breaking changes.

AVOID explicitly defining default constructors on structs.

This makes array creation faster, because if the default constructor is not defined, it does not have to be run on every slot in the array. Note that many compilers, including C#, don't allow structs to have parameterless constructors for this reason.

AVOID calling virtual members on an object inside its constructor.

Calling a virtual member will cause the most derived override to be called, even if the constructor of the most derived type has not been fully run yet.

3.3.1 Type Constructor Guidelines

DO make static constructors private.

A static constructor, also called a class constructor, is used to initialize a type. The CLR calls the static constructor before the first instance of the type is created or any static members on that type are called. The user has no control over when the static constructor is called. If a static constructor is not private, it can be called by code other than the CLR. Depending on the operations performed in the constructor, this can cause unexpected behavior. The C# compiler forces static constructors to be private.

DO NOT throw exceptions from static constructors.

If an exception is thrown from a type constructor, the type is not usable in the current application domain.

CONSIDER initializing static fields inline rather than explicitly using static constructors, because the runtime is able to optimize the performance of types that don't have an explicitly defined static constructor.

3.4 Event Design

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 (the 'O' in SOLID). 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.

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

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

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.

3.4.1 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.

DO NOT have more than two parameters on event handlers.

3.5 Field Design

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. This is to help enforce the 'O' in SOLID.

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

DO NOT provide instance fields that are public or protected.

You should provide properties for accessing fields instead of making them public or protected. If you feel you need to violate this guideline, it might indicate that you should consider using a struct rather than a class.

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.

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. If you need to enforce read-only on an object / complex type, you should look into PostSharp's Immutable aspect.

3.6 Extension Methods

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.

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.

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.

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.

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.

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.

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

3.7 Operator overloads

Operator overloads allow framework types to appear as if they were built-in language primitives.

Although allowed and useful in some situations, operator overloads should be used cautiously. There are many cases in which operator overloading has been abused, such as when framework designers started to use operators for operations that should be simple methods. The following guidelines should help you decide when and how to use operator overloading.

AVOID defining operator overloads, except in types that should feel like primitive (built-in) types.

CONSIDER defining operator overloads in a type that should feel like a primitive type.

For example, System.String has operator== and operator!= defined.

DO define operator overloads in structs that represent numbers (such as System.Decimal).

DO NOT be cute when defining operator overloads.

Operator overloading is useful in cases in which it is immediately obvious what the result of the operation will be. For example, it makes sense to be able to subtract one DateTime from another DateTime and get a TimeSpan. However, it is not appropriate to use the logical union operator to union two database queries, or to use the shift operator to write to a stream.

DO NOT provide operator overloads unless at least one of the operands is of the type defining the overload.

DO overload operators in a symmetric fashion.

For example, if you overload the operator==, you should also overload the operator!=. Similarly, if you overload the operator<, you should also overload the operator>, and so on.

CONSIDER providing methods with friendly names that correspond to each overloaded operator.

Many languages do not support operator overloading. For this reason, it is recommended that types that overload operators include a secondary method with an appropriate domain-specific name that provides equivalent functionality.

The following table contains a list of operators and the corresponding friendly method names.

C# Operator Symbol	Metadata Name	Friendly Name
N/A	op_Implicit	To<TypeName>/From<TypeName>
N/A	op_Explicit	To<TypeName>/From<TypeName>
+ (binary)	op_Addition	Add
- (binary)	op_Subtraction	Subtract
* (binary)	op_Multiply	Multiply
/	op_Division	Divide
%	op_Modulus	Mod or Remainder
^	op_ExclusiveOr	Xor
& (binary)	op_BitwiseAnd	BitwiseAnd
|	op_BitwiseOr	BitwiseOr
&&	op_LogicalAnd	And
||	op_LogicalOr	Or
=	op_Assign	Assign
<<	op_LeftShift	LeftShift
>>	op_RightShift	RightShift
N/A	op_SignedRightShift	SignedRightShift
N/A	op_UnsignedRightShift	UnsignedRightShift
==	op_Equality	Equals
!=	op_Inequality	Equals
>	op_GreaterThan	CompareTo
<	op_LessThan	CompareTo
>=	op_GreaterThanOrEqual	CompareTo
<=	op_LessThanOrEqual	CompareTo
*=	op_MultiplicationAssignment	Multiply
-=	op_SubtractionAssignment	Subtract
^=	op_ExclusiveOrAssignment	Xor
<<=	op_LeftShiftAssignment	LeftShift
>>=	op_RightShiftAssignment	RightShift
%=	op_ModulusAssignment	Mod
+=	op_AdditionAssignment	Add
&=	op_BitwiseAndAssignment	BitwiseAnd
|=	op_BitwiseOrAssignment	BitwiseOr
,	op_Comma	Comma
/=	op_DivisionAssignment	Divide
- -	op_Decrement	Decrement
++	op_Increment	Increment
- (unary)	op_UnaryNegation	Negate
+ (unary)	op_UnaryPlus	Plus
~	op_OnesComplement	OnesComplement

3.7.1 Overloading Operator ==

Overloading operator == is quite complicated. The semantics of the operator need to be compatible with several other members, such as Object.Equals.

3.7.2 Conversion Operators

Conversion operators are unary operators that allow conversion from one type to another. The operators must be defined as static members on either the operand or the return type. There are two types of conversion operators: implicit and explicit.

DO NOT provide a conversion operator if such conversion is not clearly expected by the end users.

DO NOT define conversion operators outside of a type's domain.

For example, Int32, Double, and Decimal are all numeric types, whereas DateTime is not. Therefore, there should be no conversion operator to convert a Double(long) to a DateTime. A constructor is preferred in such a case.

DO NOT provide an implicit conversion operator if the conversion is potentially lossy.

For example, there should not be an implicit conversion from Double to Int32 because Double has a wider range than Int32. An explicit conversion operator can be provided even if the conversion is potentially lossy.

DO NOT throw exceptions from implicit casts.

It is very difficult for end users to understand what is happening, because they might not be aware that a conversion is taking place.

DO throw System.InvalidCastException if a call to a cast operator results in a lossy conversion and the contract of the operator does not allow lossy conversions.

3.8 Parameter Design

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.

DO NOT use reserved parameters.

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

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.

3.8.1 Choosing Between Enum and Boolean Parameters

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

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.

3.8.2 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.

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.

3.8.3 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.

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.

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.

3.8.4 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.

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.

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.

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.

3.8.5 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.

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.