2 Framework Design Fundamentals

DO design frameworks that are both powerful and easy to use.

DO understand and explicitly design for a broad range of developers with different programming styles, requirements, and skill levels.

DO understand and design for the broad variety of programming languages.

DO make sure that the API design specification is the central part of the design of any feature that includes a publicly accessible API.

DO define top usage scenarios for each major feature area. The API specification should include a section that describes the main scenarios and shows code samples implementing these scenarios. This section should appear immediately after the executive overview section. The average feature area (such as file I/O) should have five to ten main scenarios.

DO ensure that the scenarios correspond to an appropriate abstraction level. They should roughly correspond to the end-user use cases. For example, reading from a file is a good scenario. Opening a file, reading a line of text from a file, or closing a file are not good scenarios; they are too granular.

DO design APIs by first writing code samples for the main scenarios and then defining the object model to support the code samples.

DO write main scenario code samples in at least two different language families (e.g., C# and F#). It is best to ensure that the languages chosen have significantly different syntax, style, and capabilities.

DO organize usability studies to test APIs in main scenarios. These studies should be organized early in the development cycle, because the most severe usability problems often require substantial design changes. Most developers should be able to write code for the main scenarios without major problems; if they cannot, you need to redesign the API. Although redesign is a costly practice, we have found that it actually saves resources in the long run because the cost of fixing an unusable API without introducing changes that break existing code is enormous.

DO ensure that each main feature area namespace contains only types that are used in the most common scenarios. Types used in advanced scenarios should be placed in subnamespaces. For example, the System.Net namespace provides networking mainline-scenario APIs. The more advanced socket APIs are placed in the System.Net.Sockets subnamespace.

DO provide simple overloads of constructors and methods. A simple overload has a very small number of parameters, and all parameters are primitives.

DO provide good defaults for all properties and parameters (using convenience overloads), if possible. System.Messaging.MessageQueue is a good illustration of this concept. The component can send messages after passing just a path string to the constructor and calling the Send method. The message priority, encryption algorithms, and other message properties can be customized by adding code to the simple scenario.

DO communicate incorrect usage of APIs using exceptions. The exceptions should clearly describe their cause and the way the developer should modify the code to get rid of the problem. For example, the EventLog component requires the Source property to be set before events can be written. If the Source is not set before WriteEntry is called, an exception is thrown that states, "Source property was not set before writing to the event log."

DO ensure that APIs are intuitive and can be successfully used in basic scenarios without referring to the reference documentation.

DO provide great documentation with all APIs.

DO provide code samples illustrating how to use your most important APIs in common scenarios. Not all APIs can be self-explanatory, and some developers will want to thoroughly understand the APIs before they start using them.

DO make the discussion about identifier naming choices a significant part of specification reviews. What are the types most scenarios start with? What are the names most people will think of first when trying to implement this scenario? Are the names of the common types what users will think of first? For example, because “File” is the name most people think of when dealing with file I/O scenarios, the main type for accessing files should be named File. Also, you should discuss the most commonly used methods of the most commonly used types and all of their parameters. Can anybody who is familiar with your technology, but not this specific design, recognize and call those methods quickly, correctly, and easily?

DO use exception messages to communicate framework usage mistakes to the developer. For example, if a user forgets to set the Source property on an EventLog component, any calls to a method that requires the source to be set should state this clearly in the exception message.

DO provide strongly typed APIs if at all possible. Do not rely exclusively on weakly typed APIs such as property bags. In cases in which a property bag is required, provide strongly typed properties for the most common properties in the bag.

DO ensure consistency with .NET and other frameworks your users are likely to interact with. Consistency is great for general usability. If a user is familiar with some part of a framework that your API resembles, he or she will see your design as natural and intuitive. Your API should differ from other .NET APIs only in places where there is something unique about your particular API.

DO ensure that layers of a single feature area are well integrated. Developers should be able to start programming using one of the layers and then change their code to use the other layer without rewriting the whole application.

DO NOT rely solely on standard design methodologies when designing the public API layer of a framework. Standard design methodologies (including objectoriented design methodologies) are optimized for the maintainability of the resulting implementation, not for the usability of the resulting APIs. Scenariodriven design, together with prototyping, usability studies, and some amount of iteration, is a much better approach.

DO NOT have members intended for advanced scenarios on types intended for mainline scenarios.

DO NOT require users to explicitly instantiate more than one type in the most basic scenarios.

DO NOT require that users perform any extensive initialization before they can start programming basic scenarios.

DO NOT be afraid to use verbose identifier names when doing so makes the API self-documenting. Most identifier names should clearly state what each method does and what each type and parameter represents.

CONSIDER involving technical writers early in the design process. They can be a great resource for spotting designs with bad name choices and designs that would be difficult to explain to users.

CONSIDER reserving the best type names for the most commonly used types. If you believe you will add more high-level APIs in a future version, don't be afraid to reserve the best name in the first version of your framework for future APIs.

CONSIDER writing main scenario code samples using a dynamically typed language such as PowerShell or IronPython. It is easy to design APIs that don't work well with dynamically typed languages. Such languages often have problems dealing with some generic methods, as well as APIs that rely on applying attributes or creating strongly typed types.

CONSIDER using a layered framework with high-level APIs optimized for productivity, and using low-level APIs optimized for power and expressiveness.

AVOID many abstractions in mainline-scenario APIs.

AVOID mixing low-level and high-level APIs in a single namespace if the low-level APIs are very complex (i.e., they contain many types).

3 Naming Guidelines

Capitalization Rules

DO use PascalCasing for the names of namespaces, types, members, and generic type parameters. For example, use TextColor rather than Textcolor or Text_color. Single words, such as Button, simply have initial capitals. Compound words that are always written as a single word, like endpoint, are treated as single words and have initial capitals only.

DO use camelCasing for parameter names.

DO capitalize both characters of two-character acronyms, except the first word of a camel-cased identifier.

DO capitalize only the first character of acronyms with three or more characters, except the first word of a camel-cased identifier.

DO NOT capitalize any of the characters of any acronyms, whatever their length, at the beginning of a camel-cased identifier.

DO NOT capitalize each word in so-called closedform 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.

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

General naming conventions

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).

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

DO NOT use Hungarian notation.

DO use only ASCII characters in identifier names. All of the identifiers in .NET use ASCII characters, so anyone working with .NET is capable of typing those characters and working with files or tools that understand those characters. But the developers who want to use your library may not have easy capability to type non-ASCII characters, or may use text editors (or other tools) that do not reliably preserve non-ASCII data. When basing an identifier on a word that properly requires diacritical marks, either use (or invent) the diacritical-free approximation or choose a different identifier.

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 System.Int64 should be named ToInt64, not ToLong (because System.Int64 is a CLR name for the C#-specific alias long).

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. The following is a good example of methods of a class that support writing a variety of data types into a stream:

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.

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 (e.g., if it is an industry standard) and if adding any meaningful suffix (or changing the name) is not an appropriate option.

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. For example, various APIs on System.Diagnostics.Process return Int32 values representing memory sizes, such as PagedMemorySize or PeakWorkingSet. To appropriately support these APIs on 64-bit systems, APIs have been added that have the same name but a “64” suffix.

DO NOT use abbreviations or contractions as part of identifier names. For example, use GetWindow rather than GetWin.

DO NOT use any acronyms that are not widely accepted, and even if they are, only when necessary. For example, UI is used for User Interface and HTML is used for HyperText Markup Language. Although many framework designers feel that some recent acronym will soon be widely accepted, it is bad practice to use it in framework identifiers.

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

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

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.

Names of assemblies, DLLs and packages

DO choose names for your assembly DLLs that suggest large chunks of functionality, such as System.Data. Assembly, DLL, and package 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 can contain more than one dot-separated clauses.

Names of namespaces

DO prefix namespace names with a company name to prevent namespaces from different companies from having the same name. For example, the Microsoft Office automation APIs provided by Microsoft should be in the namespace Microsoft.Office.

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

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.

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 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.

DO NOT introduce overly general 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 general type names (e.g., FormElement, XmlNode, EventLog, SoapMessage).

DO NOT give the same name to multiple types across 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.

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. By the same token, do not add a type named EventLog to the System.Diagnostics.Events namespace, because the System.Diagnostics namespace already contains a type named EventLog.

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

DO NOT introduce type name conflicts between types in technology namespaces and an application model namespace (unless the technology is not expected to be used with the application model). For example, you should not add a type named Binding to the Microsoft.VisualBasic namespace because the System.Windows.Forms namespace already contains that type name.

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.

Names of classes, structs and interfaces

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. They might indicate that the type should be an abstract class, rather than an interface.

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.

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. Two 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, although Control doesn't appear in its name.

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.

DO prefix descriptive type parameter names with T.

DO follow the guidelines described in Table 3-5 when naming types derived from or implementing certain .NET types.

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

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.

Naming enumerations

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.

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).

Names of methods

DO give methods names that are verbs or verb phrases.

Names of properties

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

DO NOT have properties that match the name of "Get" methods, such as a property named TextWriter and a method named GetTextWriter.

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. For example, CanRead is more understandable than Readable. However, Created is actually more readable than IsCreated. Having the prefix is often too verbose and unnecessary, particularly in the face of IntelliSense in the code editors. It is just as clear to type MyObject.Enabled = and have IntelliSense give you the choice of true or false as it is to have MyObject.IsEnabled =, and the second approach is more verbose.

CONSIDER giving a property the same name as its type.

Names of events

DO name events with a verb or a verb phrase. Examples include Clicked, Painting, and DroppedDown.

DO give events names with a concept of before and after, using the present and past tenses, such as Closing and Closed. 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 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.

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

Naming fields

The field-naming guidelines apply to static public and protected fields.

DO use PascalCasing in field names.

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

DO NOT use a prefix for public or protected field names. For example, do not use “g_” or “s_” to indicate static fields. Publicly accessible fields (the subject of this section) are very similar to properties from the API design point of view; therefore, they should follow the same naming conventions as properties.

Naming parameters

DO use camelCasing in parameter names.

DO use descriptive parameter names. Parameter names should be descriptive enough to use with their types to determine their meaning in most scenarios.

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.

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

CONSIDER using names based on a parameter’s meaning rather than the parameter’s type. Development tools generally provide useful information about the type, so the parameter name can be put to better use describing semantics rather than the type. Occasional use of type-based parameter names is entirely appropriate—but it is not ever appropriate under these guidelines to revert to the Hungarian naming convention.

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

Naming resources

DO NOT directly expose localizable resources as public (or protected) members. The types and members automatically generated by a resource editor should use the internal access modifier. When it does make sense to expose a resource via public API, use intentional type and member design.

4 Type Design Guidelines

DO ensure that each type is a well-defined set of related members, not just a random collection of unrelated functionality. It is important that a type can be described in one simple sentence. A good definition should also rule out functionality that is only tangentially related.

Types and namespaces

DO use namespaces to organize types into a hierarchy of related feature areas. The hierarchy should be optimized for developers browsing the framework for desired APIs.

AVOID very deep namespace hierarchies. Such hierarchies are difficult to browse because the user has to backtrack often.

DO NOT define types without specifying their namespaces. This practice organizes related types in a hierarchy and can help resolve potential type name collisions. Of course, the fact that namespaces can help resolve name collisions does not mean that such collisions should be introduced.

AVOID having too many namespaces. Users of a framework should not have to import many namespaces in the most common scenarios. Types that are used together in common scenarios should reside in a single namespace if at all possible. This guideline does not mean “have only one namespace,” but instead urges you to seek balance. The types used by developers and the types used by code generators or IDE designers to help the developers are two different concepts and should be in different, but related, namespaces.

AVOID having types designed for advanced scenarios in the same namespace as types intended for common programming tasks. This makes it easier to understand the basics of the framework and to use the framework in the common scenarios.

Choosing between class and struct

CONSIDER defining a struct instead of a class if instances of the type are small and commonly shortlived or are commonly embedded in other objects, especially arrays.

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 less than 24 bytes.
- It is immutable. It will not have to be boxed frequently .

In all other cases, you should define your types as classes.

Choosing between class and interface

DO favor defining classes over interfaces. Class-based APIs can be evolved with much greater ease than interface-based APIs because it is possible to add members to a class without breaking existing code.

DO use abstract classes instead of interfaces to decouple the contract from implementations. Abstract classes, if designed correctly, allow for the same degree of decoupling between contract and implementation.

DO define an interface if you need to provide a polymorphic hierarchy of value types. Value types cannot inherit from other types, but they can implement interfaces. For example, IComparable, IFormattable, and IConvertible are all interfaces, so value types such as Int32, Int64, and other primitives can all be comparable, formattable, and convertible.

CONSIDER defining interfaces to achieve a similar effect to that of multiple inheritance. For example, System.IDisposable and System.ICloneable are both interfaces, so types, like System.Drawing.Image, can be both disposable and cloneable yet still inherit from the System.MarshalByRefObject class.

Abstract class design

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.

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. This also applies to protected internal constructors.

Static class design

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

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.

DO NOT treat static classes as a miscellaneous bucket. There should be a clear charter for each class. When your description of the class involves “and” or a new sentence, you need another class.

DO NOT declare or override instance members in static classes. This is enforced by the C# compiler.

Interface design

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

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, System.Collections.Generic.List is an implementation of the System.Collections.Generic.IList 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.Sort consumes the IComparer 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 to avoid versioning problems.

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.

Struct design

DO declare immutable value types with the readonly modifier. Newer compiler understand the readonly modifier on a value type and avoid making extra value copies on operations such as invoking a method on a field declared with the readonly modifier.

DO declare nonmutating methods on mutable value types with the readonly modifier.

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

DO NOT provide a default constructor for a struct. Many CLR languages do not allow developers to define default constructors on value types. Users of these languages are often surprised to learn that default(SomeStruct) and new SomeStruct() don’t necessarily produce the same value. Even if your language does allow defining a default constructor on a value type, it probably isn’t worth the confusion it would cause if you did it.

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, 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 NOT define ref-like value types (ref struct types), other than for specialized low-level purposes where performance is critical.

DO NOT explicitly extend System.ValueType. In fact, most languages prevent this. In general, structs can be very useful, but they should be used only for small, single, immutable values that will not be boxed frequently.

Enum design

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. IntelliSense provides support for specifying arguments to members with enum parameters. It does not have similar support for 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.

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.

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

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 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.

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 flagsenum 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 under lying 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.

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 practice should not be followed in managed APIs. Method overloading allows adding parameters in future releases.

Designing flag enums

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

DO use powers of 2 for the flag enum values so they can be freely combined using the bitwise OR operation.

DO name the zero value of flag enums None. For a flag enum, the value must always mean “all flags are cleared.”

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. FileAccess.ReadWrite is an example of such a special value.

AVOID creating flag enums where certain combinations of values are invalid. The System.Reflection.BindingFlags enum is an example of an incorrect design of this kind. The enum tries to represent many different concepts, such as visibility, staticness, member kind, and so on.

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.

Adding values to enums

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.

Nested types

DO use nested types when the relationship between the nested type and its outer type is such that member-accessibility semantics are desirable. For example, the nested type needs to have access to private members of the outer type.

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

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 probably should 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. In general, use nested types sparingly, and avoid their exposure as public types.

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

Types and assembly metadata

DO apply the CLSCompliant(true) attribute to assemblies with public types.

DO apply AssemblyVersionAttribute to assemblies with public types.

DO apply the following informational attributes to assemblies. These attributes are used by tools, such as Visual Studio, to inform the user of the assembly about its contents.

- [assembly:AssemblyTitle("System.Core.dll")]
- [assembly:AssemblyCompany("Microsoft Corporation")]
- [assembly:AssemblyProduct("Microsoft .NET Framework")]
- [assembly:AssemblyDescription(...)]

CONSIDER applying ComVisible(false) to your assembly. COM-callable APIs need to be designed explicitly. As a rule of thumb, .NET assemblies should not be visible to COM. If you do design the APIs to be COM-callable, you can apply ComVisible(true) either to the individual APIs or to the whole assembly.

CONSIDER applying AssemblyFileVersionAttribute and Assembly-CopyrightAttribute to provide additional information about the assembly.

CONSIDER using the format <V>.<S>.<B>.<R> for the assembly file version, where V is the major version number, S is the servicing number, B is the build number, and R is the build revision number.

Strongly typed strings

DO declare a strongly typed string as an immutable value type (struct) with a string constructor. The strongly typed string type should follow other guidance for immutable value types, such as using the readonly modifier on the type and implementing IEquatable<T>.

DO override ToString() on a strongly typed string to return the underlying string value.

DO override equality operators for strongly typed string types. Overriding the equality operators allows for a strongly typed string type to syntactically look like either a string or an enum in common code patterns. Strongly typed strings should use ordinal string equality by default. However, when a specific domain represents case-insensitive content, the IEquatable behavior can be based on other string comparisons.

DO allow null inputs to the constructor of a strongly typed string. Because value types can always be zero-initialized, the constructor of a strongly typed string should not disallow null inputs. This ensures that code logically equivalent to a copy constructor will function.

DO declare known values for a strongly typed string via static get-only properties on the type. The enum-like IntelliSense experience is a very strong motivator for strongly typed string types.

CONSIDER defining a strongly typed string value type when a base class supports a fixed set of inputs but a derived type could support more. When a strongly typed string is used only by a sealed type hierarchy, there is little to no value in supporting values other than the predefined ones, and an enum is a more consistent type choice.

CONSIDER exposing the underlying string value from a strongly typed string in a get-only property. ToString() should be overridden because the underlying string value provides an obvious “interesting human-readable string” to return, per the guidelines on overriding ToString().

AVOID creating overloads across System.String and a strongly typed string, unless the System.String overload was released in a previous version.

8 Usage Guidelines

Arrays

DO prefer using collections over arrays in public APIs.

DO remember that arrays are mutable types—even arrays of immutable types. It’s easy to fall into the logical fallacy that “one Int32 is immutable, so new Int32[] { 1, 2 } is also immutable,” especially when it comes to the readonly field modifier. But the readonly field modifier only prevents replacing the entirety of the array; changing element values via the indexer is still possible.

CONSIDER using jagged arrays instead of multidimensional arrays. A jagged array is an array with elements that are also arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data (e.g., sparse matrix) compared to multidimensional arrays. Furthermore, the CLR optimizes index operations on jagged arrays, so they might exhibit better runtime performance in some scenarios.

AVOID array properties. Array-based property get methods can be implemented in one of four basic ways: direct state return, direct unauthoritative return, shallow-copy, and deep-copy. The distinction between the four implementations only matters when a caller writes to the returned array, or modifies an object exposed via the array.

Collections

DO NOT use weakly typed collections in public APIs. The type of all return values and parameters representing collection items should be the exact item type, not any of its base types (this applies only to public members of the collection).

DO NOT use ArrayList or List<T> in public APIs. These types are data structures designed to be used in internal implementation, not in public APIs. List<T> is optimized for performance and power at the cost of cleanness of the APIs and flexibility. For example, if you return List<T>, you will never be able to receive notifications when client code modifies the collection. Also, List<T> exposes many members, such as BinarySearch, that are not useful or applicable in many scenarios. Instead of using these concrete types, consider Collection<T>, IEnumerable<T>, IList<T>, IReadOnlyList<T>, or other collection abstractions.

DO NOT use Hashtable or Dictionary<TKey, TValue> in public APIs. These types are data structures designed to be used in internal implementation. Public APIs should use IDictionary, IDictionary <TKey, TValue>, or a custom type implementing one or both of the interfaces.

DO NOT use IEnumerator<T>, IEnumerator, or any other type that implements either of these interfaces, except as the return type of a GetEnumerator method. Types returning enumerators from methods other than GetEnumerator cannot be used with the foreach statement.

DO NOT implement both IEnumerator<T> and IEnumerable<T> on the same type. The same applies to the non-generic interfaces IEnumerator and IEnumerable. In other words, a type should be either a collection or an enumerator, but not both.

Collection Parameters

DO use the least-specialized type possible as a parameter type. Most members taking collections as parameters use the IEnumerable<T> interface.

AVOID using ICollection<T> or ICollection as a parameter just to access the Count property. Instead, consider using IEnumerable<T> or IEnumerable and dynamically checking whether the object implements ICollection<T> or ICollection.

Collection properties and return values

DO use Collection<T> or a subclass of Collection<T> for properties or return values representing read/write collections.

DO use ReadOnlyCollection<T>, a subclass of ReadOnlyCollection<T>, or in rare cases IEnumerable<T> for properties or return values representing read-only collections.

DO use either a snapshot collection or a live IEnumerable<T> (or its subtype) to represent collections that are volatile (i.e., that can change without explicitly modifying the collection). In general, all collections representing a shared resource (e.g., files in a directory) are volatile. Such collections are very difficult or impossible to implement as live collections unless the implementation is simply a forward-only enumerator.

DO NOT provide settable collection properties. Users can replace the contents of the collection by clearing the collection first and then adding the new contents. If replacing the whole collection is a common scenario, consider providing the AddRange method on the collection.

DO NOT return null values from collection properties or from methods returning collections. Return an empty collection or an empty array instead.

DO NOT return snapshot collections from properties. Properties should return live collections. Property getters should be very lightweight operations.

CONSIDER using subclasses of generic base collections instead of using the collections directly. This allows for a better name and for adding helper members that are not present on the base collection types. This is especially applicable to high-level APIs.

CONSIDER returning a subclass of Collection<T> or ReadOnlyCollection<T> from very commonly used methods and properties.

CONSIDER using a keyed collection if the items stored in the collection have unique keys (e.g., names, IDs). Keyed collections are collections that can be indexed by both an integer and a key, and they are usually implemented by inheriting from KeyedCollection<TKey, TItem>.

Choosing between arrays and collections

DO prefer collections over arrays. Collections provide more control over contents, can evolve over time, and are more usable. In addition, using arrays for read-only scenarios is discouraged because the cost of cloning the array is prohibitive. Usability studies have shown that some developers feel more comfortable using collection-based APIs. However, if you are developing low-level APIs, it might be better to use array parameters for read/write scenarios. Arrays have a smaller memory footprint, which helps reduce the working set, and access to elements in an array is faster because it is optimized by the runtime.

DO use byte arrays instead of collections of bytes.

DO NOT use arrays for properties if the property would have to return a new array (e.g., a copy of an internal array) every time the property getter is called.

CONSIDER using arrays in low-level APIs to minimize memory consumption and maximize performance.

Implementing custom collections

DO implement IEnumerable<T> when designing new collections. Consider implementing ICollection<T>, IReadOnlyList<T>, or even IList<T> where it makes sense. When implementing such a custom collection, follow the API pattern established by Collection<T> and ReadOnlyCollection<T> as closely as possible. That is, implement the same members explicitly, name the parameters in the same way that these two collections name them, and so on. In other words, make your custom collection different from these two collections only when you have a very good reason to do so.

DO use the "Dictionary" suffix in names of abstractions implementing IDictionary or IDictionary<TKey, TValue>.

DO use the "Collection" suffix in names of types implementing IEnumerable (or any of its descendants) and representing a list of items.

DO use the appropriate data structure name for custom data structures.

DO NOT inherit from non-generic base collections such as CollectionBase. Use Collection<T>, ReadOnlyCollection<T>, and KeyedCollection<Tkey, TItem> instead.

CONSIDER inheriting from Collection<T>, ReadOnlyCollection<T>, or KeyedCollection<TKey, TItem> when designing new collections.

CONSIDER implementing non-generic collection interfaces (IList and ICollection) if the collection will often be passed to APIs taking these interfaces as input.

CONSIDER prefixing collection names with the name of the item type. For example, a collection storing items of type Address (implementing IEnumerable<Address>) should be named AddressCollection. If the item type is an interface, the "I" prefix of the item type can be omitted. Thus, a collection of IDisposable items can be called DisposableCollection.

CONSIDER using the "ReadOnly" prefix in names of read-only collections if a corresponding writeable collection might be added or already exists in the framework. For example, a read-only collection of strings should be called ReadOnlyStringCollection.

AVOID implementing collection interfaces on types with complex APIs unrelated to the concept of a collection. In other words, a collection should be a simple type used to store, access, and manipulate items, and not much more.

AVOID using any suffixes implying a particular implementation, such as “LinkedList” or “Hashtable,” in names of collection abstractions.

9 Common design patterns

Aggregate components guidelines

DO model high-level concepts (physical objects) rather than system-level tasks with aggregate components. For example, the components should model files, directories, and drives, rather than streams, formatters, and comparers.

DO increase the visibility of aggregate components by giving them names that correspond to well-known entities of the system, such as MessageQueue, Process, or EventLog.

DO design aggregate components so they can be used after very simple initialization. If some initialization is necessary, the exception resulting from not having the component initialized should clearly explain what needs to be done.

DO make sure aggregate components support the Create–Set–Call usage pattern, where developers expect to be able to implement most scenarios by instantiating the component, setting its properties, and calling simple methods.

DO provide a default or a very simple constructor for all aggregate components.

DO provide properties with getters and setters corresponding to all parameters of aggregate component constructors. It should always be possible to use the default constructor and then set some properties instead of calling a parameterized constructor.

DO use events instead of delegate-based APIs in aggregate components. Aggregate components are optimized for ease of use, and events are much easier to use than APIs using delegates

DO use events in aggregate components instead of virtual members that need to be overridden.

DO NOT require the users of aggregate components to explicitly instantiate multiple objects in a single scenario. Simple tasks should be done with just one object. The next best thing is to start with one object that in turn creates other supporting objects. Your top five scenario samples showing aggregate component usage should not have more than one new statement.

DO NOT require users of aggregate components to inherit, override methods, or implement any interfaces in common scenarios. Components should mostly rely on properties and composition as the means of modifying their behavior.

DO NOT require users of aggregate components to do anything besides writing code in common scenarios. For example, users should not have to configure components in the configuration file, generate any resource files, and so on.

DO NOT design factored types that have modes. Factored types should have a well-defined life span scoped to a single mode. For example, instances of Stream can either read or write, and an instantiated stream is already opened.

CONSIDER providing aggregate components for commonly used feature areas. Aggregate components provide high-level functionality and are starting points for exploring given technology. They should provide shortcuts for common operations and add significant value over what is already provided by factored types. They should not simply duplicate the functionality. Many main scenario code samples should start with an instantiation of an aggregate component.

CONSIDER making changes to aggregate components' modes automatic. For example, a single instance of MessageQueue can be used to send and receive messages, but the user should not be aware that mode switching is occurring.

CONSIDER integrating your aggregate components with Visual Studio designers. Integration allows placing the component on the Visual Studio Toolbox and adds support for drag and drop, property grid, event hookup, and so on. The integration is simple and can be done by implementing IComponent or inheriting from a type implementing this interface, such as Component or Control.

CONSIDER separating aggregate components and factored types into different assemblies. This allows the component to aggregate arbitrary functionality provided by factored types without circular dependencies.

CONSIDER exposing access to internal factored types of an aggregate component. Factored types are ideal for integrating different feature areas. For example, the SerialPort component exposes access to its stream, thus allowing integration with reusable APIs, such as compression APIs that operate on streams.

App. A C# coding style conventions

General style conventions

DO place the opening brace on the next line, indented to the same level as the block statement. if (someExpression)
{
DoSomething();
}

DO align the closing brace with the opening brace.

DO place the closing brace on its own line, except the end of a do..while statement.

DO NOT use the braceless variant of the using (dispose) statement. The braced using statement provides a strong visual indicator of when the value is being released. Just as with locking, a disposable value should be disposed as soon as it is practical to do so. The braceless variant makes it too easy to add code that unnecessarily extends the scope of the disposable value, and changing the braceless version to the braced version adds unnecessary noise to a diff representation of a change.

CONSIDER omitting braces in an argument validation preamble. For all parts of the method preamble that are of the form *if (single-LineExpression) { throw … }*, the braces and vertical whitespace can be eliminated. Once you add a blank line, or have code other than if…throw, you've left the preamble and braces become required again. Multi-line conditions should ideally be factored out into methods to maintain the visual flow of the preamble.

AVOID omitting braces, even if the language allows it. Braces should not be considered optional. Even for single-statement blocks, you should use braces. This increases code readability and maintainability. One exception to the rule is braces in case statements. These braces can be omitted, because the case and break statements indicate the beginning and the end of the block.

AVOID using the braceless variant of the await using statement, except to simulate stacked await using statements with ConfigureAwait.

Space usage

DO use one space before and after the opening and closing braces when they share a line with other code. Do not add a trailing space before a line break.

DO use a single space after a comma between parameters.

DO use a single space between arguments.

DO use spaces between flow control keywords and the open parenthesis.

DO use spaces before and after binary operators.

DO NOT use spaces after the opening parenthesis or before the closing parenthesis.

DO NOT use spaces between a member name and an opening parenthesis.

DO NOT use spaces after or before square braces.

DO NOT use spaces before or after unary operators.

Indent usage

DO use four consecutive space characters for indents.

DO indent contents of code blocks.

DO indent case blocks even if not using braces.

DO “outdent” goto labels one indentation level.

DO indent all continued lines of a single statement one indentation level.

DO chop before the first argument or parameter, indent one indentation level, and include one argument or parameter per line when a method declaration or method invocation is exceedingly long.

DO NOT use the tab character for indents.

Vertical whitespace

DO add a blank line before control flow statements.

DO add a blank line after a closing brace, unless the next line is also a closing brace.

DO add a blank line after “paragraphs” of code where it enhances readability.

Member modifiers

DO always explicitly specify visibility modifiers.

DO specify the visibility modifier as the first modifier.

DO specify the static modifier immediately after visibility, for static members or static classes.

DO specify the extern method modifier immediately after the static modifier for an extern method.

DO specify the member slot modifier (new, virtual, abstract, sealed, or override) immediately after the static modifier (if specified).

DO specify the readonly modifier for fields or methods immediately after the member slot modifier (if specified).

DO specify the unsafe modifier for methods immediately after the readonly modifier (if specified).

DO specify the volatile modifier for fields immediately after the static modifier (if specified). This guideline skips the readonly modifier for fields since volatile and readonly are mutually exclusive modifiers.

DO specify the async modifier as the last method modifier, when used.

DO use “protected internal” instead of “internal protected” for a member that can be called by derived types or types in the same assembly.

DO use “private protected” instead of “protected private” for a member that can be called by derived types within the same assembly.

Other

DO add the optional trailing comma after the last enum member.

DO add the optional trailing comma after a property assignment from an object initializer, or addition via a collection initializer, when the initializer spans multiple lines.

DO use language keywords (string, int, double, …) instead of BCL type names (String, Int32, Double, …), for both variable declarations and method invocation.

DO use object initializers where possible. This rule can be ignored when working with a type that has a required order of property assignment. As with most exceptions to a rule, however, a comment should explain why the code should not be changed to use object initializers. The object initializer does respect the order the properties are assigned, but it doesn’t “feel” as ordered as multiple assignment statements.

DO use collection initializers where possible.

DO use auto-implemented properties when the implementation is unlikely to change.

DO restrict code to ASCII characters, and use Unicode escape sequences (\uXXXX) when needed for non-ASCII values.

DO use the nameof(...) syntax, instead of a literal string, when referring to the name of a type, member, or parameter.

DO apply the readonly modifier to fields when possible. Be careful when applying the readonly modifier to fields where the field type is a struct. If the struct isn't declared as readonly, then property and method invocations might make local copies of the value and create a performance problem.

DO use if…throw instead of assignment– expression–throw for argument validation. Using expression–throw in an assignment statement can lead to subtle bugs in finalizers where some fields are set but others aren't. While finalizers aren't common, always using if…throw generalizes without exception to types with and without finalizers, as well as to both methods and constructors.

DO NOT use this. unless absolutely necessary. If the private members are named correctly, a "this." should not be required except to invoke an extension method. The use of the underscore prefix on fields already provides a distinction between field access and variable access.

DO NOT use the var keyword except to save the result of a new, as-cast or "hard" cast. In the core .NET libraries, we only use var when the type name is already specified. We don't use it when calling factory methods, calling methods for which "everyone knows" the return type (e.g., String.IndexOf), or receiving the result of a TryParse method. We also do not compel the use of var when using a new, as-cast, or "hard" cast; instead, we leave it as a judgment call on the part of the code author.

CONSIDER using expression-bodied members when the implementation of a property or method is unlikely to change.

App A.2 Naming conventions

DO follow the Framework Design Guidelines for naming identifiers, except for naming private and internal fields.

DO use PascalCasing for namespace, type, and member names, except for internal and private fields.

DO use PascalCasing for const locals and const fields except for interop code, where it should match the name from the called code exactly.

DO use camelCasing for private and internal fields.

DO use a prefix "_" (underscore) for private and internal instance fields, "s_" for private and internal static fields, and "t_" for private and internal threadstatic fields.

DO use camelCasing for local variables.

DO use camelCasing for parameters.

DO NOT use Hungarian notation (i.e., do not encode the type of a variable in its name).

App A.3 Comments

DO NOT use comments unless they describe something not obvious to someone other than the developer who wrote the code.

DO NOT place comments at the end of a line unless the comment is very short.

AVOID placing comments at the end of a line even when the comment is very short.

AVOID multiline syntax (/* ... */) for comments. The single-line syntax (// ...) is preferred even when a comment spans multiple lines.

AVOID writing "I" in comments. Code ownership changes over time, particularly in long-lived software projects, which leaves "I" ambiguous when not consulting code history.

App A.4 Organization

DO name the source file with the name of the public type it contains. For example, the String class should be in a file named "String.cs" and the List<T> class should be in a file named "List.cs".

DO name the source file for a partial type with the name of the primary file and a logical description of the contents of the file separated by a "." (period), such as "JsonDocument.Parse.cs".

DO organize the directory hierarchy just like the namespace hierarchy. For example, the source file for System.Collections.Generic.List<T> should be in the System\Collections\Generic directory. For assemblies that have a common namespace prefix for all types, removing empty top-level directories is a usual practice. Thus, a type named "SomeProject.BitManipulation.Simd" from an assembly with a common prefix of "SomeProject" would usually be found at <project root>\BitManipulation\Simd.cs.

DO group members into the following sections in the specified order:

- All const fields
- All static fields
- All instance fields
- All auto-implemented static properties
- All auto-implemented instance properties
- All constructors Remaining members
- All nested types

DO place the using directives outside the namespace declaration.

DO sort the using directives alphabetically, but place all System namespaces first.

DO NOT have more than one public type in a source file, unless they differ only in the number of generic parameters or one is nested in the other. Multiple internal types in one file are allowed, though there is a preference for "one (top-level) type per file."

CONSIDER grouping the remaining members into the following sections in the specified order:

- Public and protected properties
- Methods
- Events
- All explicit interface implementations
- Internal members
- Private membe s