Table of Contents

Framework Design Guidelines

1/19/2018 • 1 min to read • Edit Online

This section provides guidelines for designing libraries that extend and interact with the .NET Framework. The goal is to help library designers ensure API consistency and ease of use by providing a unified programming model that is independent of the programming language used for development. We recommend that you follow these design guidelines when developing classes and components that extend the .NET Framework. Inconsistent library design adversely affects developer productivity and discourages adoption.

The guidelines are organized as simple recommendations prefixed with the terms `Do`, `Consider`, `Avoid`, and `Do not`. These guidelines are intended to help class library designers understand the trade-offs between different solutions. There might be situations where good library design requires that you violate these design guidelines. Such cases should be rare, and it is important that you have a clear and compelling reason for your decision.

These guidelines are excerpted from the book *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*, by Krzysztof Cwalina and Brad Abrams.

In This Section

Naming Guidelines
Provides guidelines for naming assemblies, namespaces, types, and members in class libraries.

Type Design Guidelines
Provides guidelines for using static and abstract classes, interfaces, enumerations, structures, and other types.

Member Design Guidelines
Provides guidelines for designing and using properties, methods, constructors, fields, events, operators, and parameters.

Designing for Extensibility
Discusses extensibility mechanisms such as subclassing, using events, virtual members, and callbacks, and explains how to choose the mechanisms that best meet your framework's requirements.

Design Guidelines for Exceptions
Describes design guidelines for designing, throwing, and catching exceptions.

Usage Guidelines
Describes guidelines for using common types such as arrays, attributes, and collections, supporting serialization, and overloading equality operators.

Common Design Patterns
Provides guidelines for choosing and implementing dependency properties and the dispose pattern.

See Also

Overview

Roadmap for the .NET Framework

Development Guide

Naming Guidelines

1/5/2018 • 1 min to read • Edit Online

Following a consistent set of naming conventions in the development of a framework can be a major contribution to the framework's usability. It allows the framework to be used by many developers on widely separated projects. Beyond consistency of form, names of framework elements must be easily understood and must convey the function of each element.

The goal of this chapter is to provide a consistent set of naming conventions that results in names that make immediate sense to developers.

Although adopting these naming conventions as general code development guidelines would result in more consistent naming throughout your code, you are required only to apply them to APIs that are publicly exposed (public or protected types and members, and explicitly implemented interfaces).

In This Section

Capitalization Conventions
General Naming Conventions
Names of Assemblies and DLLs
Names of Namespaces
Names of Classes, Structs, and Interfaces
Names of Type Members
Naming Parameters
Naming Resources

See Also

Framework Design Guidelines

Capitalization Conventions

1/5/2018 • 2 min to read • Edit Online

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

Capitalization Rules for Identifiers

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

- PascalCasing
- camelCasing

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

`PropertyDescriptor`
`HtmlTag`

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

`IOStream`

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

`propertyDescriptor`
`ioStream`
`htmlTag`

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

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

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

IDENTIFIER	CASING	EXAMPLE
Namespace	Pascal	`namespace System.Security { ... }`
Type	Pascal	`public class StreamReader { ... }`
Interface	Pascal	`public interface IEnumerable { ... }`
Method	Pascal	`public class Object {` `public virtual string ToString();` `}`

IDENTIFIER	CASING	EXAMPLE

Property	Pascal	`public class String {`
		`public int Length { get; }`
		`}`

Event	Pascal	`public class Process {`
		`public event EventHandler Exited;`
		`}`

Field	Pascal	`public class MessageQueue {`
		`public static readonly TimeSpan`
		`InfiniteTimeout;`
		`}`
		`public struct UInt32 {`
		`public const Min = 0;`
		`}`

Enum value	Pascal	`public enum FileMode {`
		`Append,`
		`...`
		`}`

Parameter	Camel	`public class Convert {`
		`public static int ToInt32(string value);`
		`}`

Capitalizing Compound Words and Common Terms

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

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

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

PASCAL	CAMEL	NOT
`BitFlag`	`bitFlag`	`Bitflag`
`Callback`	`callback`	`CallBack`
`Canceled`	`canceled`	`Cancelled`
`DoNot`	`doNot`	`Don't`
`Email`	`email`	`EMail`
`Endpoint`	`endpoint`	`EndPoint`
`FileName`	`fileName`	`Filename`

PASCAL	CAMEL	NOT
`Gridline`	`gridline`	`GridLine`
`Hashtable`	`hashtable`	`HashTable`
`Id`	`id`	`ID`
`Indexes`	`indexes`	`Indices`
`LogOff`	`logOff`	`LogOut`
`LogOn`	`logOn`	`LogIn`
`Metadata`	`metadata`	`MetaData, metaData`
`Multipanel`	`multipanel`	`MultiPanel`
`Multiview`	`multiview`	`MultiView`
`Namespace`	`namespace`	`NameSpace`
`Ok`	`ok`	`OK`
`Pi`	`pi`	`PI`
`Placeholder`	`placeholder`	`PlaceHolder`
`SignIn`	`signIn`	`SignOn`
`SignOut`	`signOut`	`SignOff`
`UserName`	`userName`	`Username`
`WhiteSpace`	`whiteSpace`	`Whitespace`
`Writable`	`writable`	`Writeable`

Case Sensitivity

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

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

See Also

Framework Design Guidelines

Naming Guidelines

General Naming Conventions

1/5/2018 • 3 min to read • Edit Online

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

Word Choice

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

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

✓ **DO** favor readability over brevity.

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

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

X DO NOT use Hungarian notation.

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

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

Using Abbreviations and Acronyms

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

For example, use `GetWindow` rather than `GetWin`.

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

Avoiding Language-Specific Names

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

For example, `GetLength` is a better name than `GetInt`.

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

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

C#	VISUAL BASIC	C++	CLR
sbyte	**SByte**	**char**	**SByte**
byte	**Byte**	**unsigned char**	**Byte**

C#	VISUAL BASIC	C++	CLR
short	Short	short	Int16
ushort	UInt16	unsigned short	UInt16
int	Integer	int	Int32
uint	UInt32	unsigned int	UInt32
long	Long	__int64	Int64
ulong	UInt64	unsigned __int64	UInt64
float	Single	float	Single
double	Double	double	Double
bool	Boolean	bool	Boolean
char	Char	wchar_t	Char
string	String	String	String
object	Object	Object	Object

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

Naming New Versions of Existing APIs

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

This helps to highlight the relationship between the APIs.

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

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

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

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

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

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

Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Framework Design Guidelines
Naming Guidelines

Names of Assemblies and DLLs

1/5/2018 • 1 min to read • Edit Online

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

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

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

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

`<Company>.<Component>.dll`

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

`Litware.Controls.dll`.

See Also

Framework Design Guidelines
Naming Guidelines

Names of Namespaces

1/5/2018 • 3 min to read • Edit Online

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

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

The following are examples:

`Fabrikam.Math`
`Litware.Security`

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

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

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

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

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

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

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

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

Namespaces and Type Name Conflicts

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

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

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

- **Application model namespaces**

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

  `System.Windows*`
  `System.Web.UI*`

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

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

- **Infrastructure namespaces**

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

- **Core namespaces**

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

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

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

- **Technology namespace groups**

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

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

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

See Also

Framework Design Guidelines
Naming Guidelines

Names of Classes, Structs, and Interfaces

1/5/2018 • 3 min to read • Edit Online

The naming guidelines that follow apply to general type naming.

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

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

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

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

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

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

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

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

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

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

Names of Generic Type Parameters

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

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

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

```
public int IComparer<T> { ... }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T:struct { ... }
```

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

```
public interface ISessionChannel<TSession> where TSession : ISession{
    TSession Session { get; }
}
```

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

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

Names of Common Types

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

BASE TYPE	DERIVED/IMPLEMENTING TYPE GUIDELINE
`System.Attribute`	✓ **DO** add the suffix "Attribute" to names of custom attribute classes.
`System.Delegate`	✓ **DO** add the suffix "EventHandler" to names of delegates that are used in events. ✓ **DO** add the suffix "Callback" to names of delegates other than those used as event handlers. **X DO NOT** add the suffix "Delegate" to a delegate.
`System.EventArgs`	✓ **DO** add the suffix "EventArgs."
`System.Enum`	**X DO NOT** derive from this class; use the keyword supported by your language instead; for example, in C#, use the `enum` keyword. **X DO NOT** add the suffix "Enum" or "Flag."
`System.Exception`	✓ **DO** add the suffix "Exception."

`IDictionary`
`IDictionary<TKey,TValue>`
	✓ **DO** add the suffix "Dictionary." Note that `IDictionary` is a specific type of collection, but this guideline takes precedence over the more general collections guideline that follows.

`IEnumerable`
`ICollection`
`IList`
`IEnumerable<T>`
`ICollection<T>`
`IList<T>`
	✓ **DO** add the suffix "Collection."

`System.IO.Stream`	✓ **DO** add the suffix "Stream."
`CodeAccessPermission IPermission`	✓ **DO** add the suffix "Permission."

Naming Enumerations

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

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

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

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

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

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

See Also

Framework Design Guidelines
Naming Guidelines

Names of Type Members

1/5/2018 • 3 min to read • Edit Online

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

Names of Methods

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

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

```
public class String {
    public int CompareTo(...);
    public string[] Split(...);
    public string Trim();
}
```

Names of Properties

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

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

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

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

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

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

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

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

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

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

Names of Events

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

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

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

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

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

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

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

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

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

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

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

Names of Fields

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

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

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

X DO NOT use a prefix for field names.

For example, do not use "g_" or "s_" to indicate static fields.

See Also

Framework Design Guidelines
Naming Guidelines

Naming Parameters

1/5/2018 • 1 min to read • Edit Online

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

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

✓ **DO** use descriptive parameter names.

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

Naming Operator Overload Parameters

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

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

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

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

See Also

Framework Design Guidelines
Naming Guidelines

Naming Resources

1/5/2018 • 1 min to read • Edit Online

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

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

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

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

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

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

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

```
ArgumentExceptionIllegalCharacters
ArgumentExceptionInvalidName
ArgumentExceptionFileNameIsMalformed
```

See Also

Framework Design Guidelines
Naming Guidelines

Type Design Guidelines

1/5/2018 • 1 min to read • Edit Online

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.

In This Section

See Also

Framework Design Guidelines

Choosing Between Class and Struct

1/5/2018 • 2 min to read • Edit Online

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.

X 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.

See Also

Type Design Guidelines
Framework Design Guidelines

Abstract Class Design

1/5/2018 • 1 min to read • Edit Online

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

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

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

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

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

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

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

See Also

Type Design Guidelines
Framework Design Guidelines

Static Class Design

1/5/2018 • 1 min to read • Edit Online

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

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

✓ **DO** use static classes sparingly.

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

X DO NOT treat static classes as a miscellaneous bucket.

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

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

See Also

Type Design Guidelines
Framework Design Guidelines

Interface Design

1/5/2018 • 2 min to read • Edit Online

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

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

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

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

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

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

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

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

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

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

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

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

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

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

See Also

Type Design Guidelines
Framework Design Guidelines

Struct Design

1/5/2018 • 1 min to read • Edit Online

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

X 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.

X 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.

X 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.

See Also

Type Design Guidelines
Framework Design Guidelines
Choosing Between Class and Struct

Enum Design

1/5/2018 • 4 min to read • Edit Online

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

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

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

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

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

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

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

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

X AVOID publicly exposing enums with only one value.

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

X DO NOT include sentinel values in enums.

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

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

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

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

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

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

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

X DO NOT extend System.Enum directly.

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

Designing Flag Enums

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

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

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

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

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

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

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

Adding Value to Enums

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

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

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

See Also

Type Design Guidelines
Framework Design Guidelines

Nested Types

1/5/2018 • 2 min to read • Edit Online

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

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

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

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

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

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

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

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

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

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

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

See Also

Type Design Guidelines
Framework Design Guidelines

Member Design Guidelines

1/5/2018 • 1 min to read • Edit Online

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.

In This Section

See Also

Framework Design Guidelines

Member Overloading

1/5/2018 • 1 min to read • Edit Online

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.

X 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.

X 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.

X 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.

X 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.

Default arguments are not CLS compliant.

See Also

Member Design Guidelines
Framework Design Guidelines

Property Design

1/5/2018 • 4 min to read • Edit Online

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.

X 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.

X 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.

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.

X 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`.

X 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[Int32] property on `System.String`).

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

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

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

This is enforced by the C# compiler.

X DO NOT use nondefault indexed properties.

This is enforced by the C# compiler.

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.

See Also

Member Design Guidelines
Framework Design Guidelines

Constructor Design

1/5/2018 • 3 min to read • Edit Online

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.

X 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.

X 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.

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.

X 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.

See Also

Member Design Guidelines
Framework Design Guidelines

Event Design

1/5/2018 • 3 min to read • Edit Online

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Custom Event Handler Design

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

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

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

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

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

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

See Also

Member Design Guidelines
Framework Design Guidelines

Field Design

1/5/2018 • 1 min to read • Edit Online

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

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

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

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

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

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

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

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

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

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

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

See Also

Member Design Guidelines
Framework Design Guidelines

Extension Methods

1/5/2018 • 2 min to read • Edit Online

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

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

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

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

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

- To provide helper functionality relevant to every implementation of an interface, if said functionality can be written in terms of the core interface. This is because concrete implementations cannot otherwise be assigned to interfaces. For example, the `LINQ to Objects` operators are implemented as extension methods for all IEnumerable<T> types. Thus, any `IEnumerable<>` implementation is automatically LINQ-enabled.

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

X AVOID defining extension methods on System.Object.

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

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

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

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

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

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

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

See Also

Member Design Guidelines
Framework Design Guidelines

Operator Overloads

1/5/2018 • 3 min to read • Edit Online

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.

X 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).

X 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.

X 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`

C# OPERATOR SYMBOL	METADATA NAME	FRIENDLY NAME
`%`	`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_ModulusAssignment`	`Mod`
`+=`	`op_AdditionAssignment`	`Add`
`&=`	`op_BitwiseAndAssignment`	`BitwiseAnd`

C# OPERATOR SYMBOL	METADATA NAME	FRIENDLY NAME
`|=`	`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`

Overloading Operator ==

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

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.

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

X 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.

X 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.

X 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.

See Also

Member Design Guidelines

Parameter Design

1/5/2018 • 7 min to read • Edit Online

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

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

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

X DO NOT use reserved parameters.

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

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

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

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

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

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

This better communicates the relationship between the methods.

Choosing Between Enum and Boolean Parameters

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

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

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

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

Validating Arguments

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

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

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

✓ **DO** validate enum parameters.

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

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

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

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

Parameter Passing

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

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

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

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

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

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

X DO NOT pass reference types by reference.

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

Members with Variable Number of Parameters

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Pointer Parameters

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

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

X AVOID doing expensive argument checking of pointer arguments.

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

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

Addison-Wesley Professional as part of the Microsoft Windows Development Series.

See Also

Member Design Guidelines
Framework Design Guidelines

Designing for Extensibility

1/5/2018 • 1 min to read • Edit Online

One important aspect of designing a framework is making sure the extensibility of the framework has been carefully considered. This requires that you understand the costs and benefits associated with various extensibility mechanisms. This chapter helps you decide which of the extensibility mechanisms—subclassing, events, virtual members, callbacks, and so on—can best meet the requirements of your framework.

There are many ways to allow extensibility in frameworks. They range from less powerful but less costly to very powerful but expensive. For any given extensibility requirement, you should choose the least costly extensibility mechanism that meets the requirements. Keep in mind that it's usually possible to add more extensibility later, but you can never take it away without introducing breaking changes.

In This Section

See Also

Framework Design Guidelines

Unsealed Classes

1/5/2018 • 1 min to read • Edit Online

Sealed classes cannot be inherited from, and they prevent extensibility. In contrast, classes that can be inherited from are called unsealed classes.

✓ **CONSIDER** using unsealed classes with no added virtual or protected members as a great way to provide inexpensive yet much appreciated extensibility to a framework.

Developers often want to inherit from unsealed classes so as to add convenience members such as custom constructors, new methods, or method overloads. For example, `System.Messaging.MessageQueue` is unsealed and thus allows users to create custom queues that default to a particular queue path or to add custom methods that simplify the API for specific scenarios.

Classes are unsealed by default in most programming languages, and this is also the recommended default for most classes in frameworks. The extensibility afforded by unsealed types is much appreciated by framework users and quite inexpensive to provide because of relatively low test costs associated with unsealed types.

See Also

Framework Design Guidelines
Designing for Extensibility
Sealing

Protected Members

1/5/2018 • 1 min to read • Edit Online

Protected members by themselves do not provide any extensibility, but they can make extensibility through subclassing more powerful. They can be used to expose advanced customization options without unnecessarily complicating the main public interface.

Framework designers need to be careful with protected members because the name "protected" can give a false sense of security. Anyone is able to subclass an unsealed class and access protected members, and so all the same defensive coding practices used for public members apply to protected members.

✓ **CONSIDER** using protected members for advanced customization.

✓ **DO** treat protected members in unsealed classes as public for the purpose of security, documentation, and compatibility analysis.

Anyone can inherit from a class and access the protected members.

See Also

Framework Design Guidelines
Designing for Extensibility

Events and Callbacks

1/5/2018 • 1 min to read • Edit Online

Callbacks are extensibility points that allow a framework to call back into user code through a delegate. These delegates are usually passed to the framework through a parameter of a method.

Events are a special case of callbacks that supports convenient and consistent syntax for supplying the delegate (an event handler). In addition, Visual Studio's statement completion and designers provide help in using event-based APIs. (See Event Design.)

✓ **CONSIDER** using callbacks to allow users to provide custom code to be executed by the framework.

✓ **CONSIDER** using events to allow users to customize the behavior of a framework without the need for understanding object-oriented design.

✓ **DO** prefer events over plain callbacks, because they are more familiar to a broader range of developers and are integrated with Visual Studio statement completion.

X AVOID using callbacks in performance-sensitive APIs.

✓ **DO** use the new `Func<...>`, `Action<...>`, or `Expression<...>` types instead of custom delegates, when defining APIs with callbacks.

`Func<...>` and `Action<...>` represent generic delegates. `Expression<...>` represents function definitions that can be compiled and subsequently invoked at runtime but can also be serialized and passed to remote processes.

✓ **DO** measure and understand performance implications of using `Expression<...>`, instead of using `Func<...>` and `Action<...>` delegates.

`Expression<...>` types are in most cases logically equivalent to `Func<...>` and `Action<...>` delegates. The main difference between them is that the delegates are intended to be used in local process scenarios; expressions are intended for cases where it's beneficial and possible to evaluate the expression in a remote process or machine.

✓ **DO** understand that by calling a delegate, you are executing arbitrary code and that could have security, correctness, and compatibility repercussions.

See Also

Designing for Extensibility
Framework Design Guidelines

Virtual Members

1/5/2018 • 1 min to read • Edit Online

Virtual members can be overridden, thus changing the behavior of the subclass. They are quite similar to callbacks in terms of the extensibility they provide, but they are better in terms of execution performance and memory consumption. Also, virtual members feel more natural in scenarios that require creating a special kind of an existing type (specialization).

Virtual members perform better than callbacks and events, but do not perform better than non-virtual methods.

The main disadvantage of virtual members is that the behavior of a virtual member can only be modified at the time of compilation. The behavior of a callback can be modified at runtime.

Virtual members, like callbacks (and maybe more than callbacks), are costly to design, test, and maintain because any call to a virtual member can be overridden in unpredictable ways and can execute arbitrary code. Also, much more effort is usually required to clearly define the contract of virtual members, so the cost of designing and documenting them is higher.

X DO NOT make members virtual unless you have a good reason to do so and you are aware of all the costs related to designing, testing, and maintaining virtual members.

Virtual members are less forgiving in terms of changes that can be made to them without breaking compatibility. Also, they are slower than non-virtual members, mostly because calls to virtual members are not inlined.

✓ CONSIDER limiting extensibility to only what is absolutely necessary.

✓ DO prefer protected accessibility over public accessibility for virtual members. Public members should provide extensibility (if required) by calling into a protected virtual member.

The public members of a class should provide the right set of functionality for direct consumers of that class. Virtual members are designed to be overridden in subclasses, and protected accessibility is a great way to scope all virtual extensibility points to where they can be used.

See Also

Framework Design Guidelines
Designing for Extensibility

Abstractions (Abstract Types and Interfaces)

1/5/2018 • 2 min to read • Edit Online

An abstraction is a type that describes a contract but does not provide a full implementation of the contract. Abstractions are usually implemented as abstract classes or interfaces, and they come with a well-defined set of reference documentation describing the required semantics of the types implementing the contract. Some of the most important abstractions in the .NET Framework include Stream, IEnumerable<T>, and Object.

You can extend frameworks by implementing a concrete type that supports the contract of an abstraction and using this concrete type with framework APIs consuming (operating on) the abstraction.

A meaningful and useful abstraction that is able to withstand the test of time is very difficult to design. The main difficulty is getting the right set of members, no more and no fewer. If an abstraction has too many members, it becomes difficult or even impossible to implement. If it has too few members for the promised functionality, it becomes useless in many interesting scenarios.

Too many abstractions in a framework also negatively affect usability of the framework. It is often quite difficult to understand an abstraction without understanding how it fits into the larger picture of the concrete implementations and the APIs operating on the abstraction. Also, names of abstractions and their members are necessarily abstract, which often makes them cryptic and unapproachable without first understanding the broader context of their usage.

However, abstractions provide extremely powerful extensibility that the other extensibility mechanisms cannot often match. They are at the core of many architectural patterns, such as plug-ins, inversion of control (IoC), pipelines, and so on. They are also extremely important for testability of frameworks. Good abstractions make it possible to stub out heavy dependencies for the purpose of unit testing. In summary, abstractions are responsible for the sought-after richness of the modern object-oriented frameworks.

X DO NOT provide abstractions unless they are tested by developing several concrete implementations and APIs consuming the abstractions.

✓ DO choose carefully between an abstract class and an interface when designing an abstraction.

✓ CONSIDER providing reference tests for concrete implementations of abstractions. Such tests should allow users to test whether their implementations correctly implement the contract.

See Also

Framework Design Guidelines
Designing for Extensibility

Base Classes for Implementing Abstractions

1/5/2018 • 2 min to read • Edit Online

Strictly speaking, a class becomes a base class when another class is derived from it. For the purpose of this section, however, a base class is a class designed mainly to provide a common abstraction or for other classes to reuse some default implementation though inheritance. Base classes usually sit in the middle of inheritance hierarchies, between an abstraction at the root of a hierarchy and several custom implementations at the bottom.

They serve as implementation helpers for implementing abstractions. For example, one of the Framework's abstractions for ordered collections of items is the IList<T> interface. Implementing IList<T> is not trivial, and therefore the Framework provides several base classes, such as Collection<T> and KeyedCollection<TKey,TItem>, which serve as helpers for implementing custom collections.

Base classes are usually not suited to serve as abstractions by themselves, because they tend to contain too much implementation. For example, the `Collection<T>` base class contains lots of implementation related to the fact that it implements the nongeneric `IList` interface (to integrate better with nongeneric collections) and to the fact that it is a collection of items stored in memory in one of its fields.

As previously discussed, base classes can provide invaluable help for users who need to implement abstractions, but at the same time they can be a significant liability. They add surface area and increase the depth of inheritance hierarchies and so conceptually complicate the framework. Therefore, base classes should be used only if they provide significant value to the users of the framework. They should be avoided if they provide value only to the implementers of the framework, in which case delegation to an internal implementation instead of inheritance from a base class should be strongly considered.

✓ **CONSIDER** making base classes abstract even if they don't contain any abstract members. This clearly communicates to the users that the class is designed solely to be inherited from.

✓ **CONSIDER** placing base classes in a separate namespace from the mainline scenario types. By definition, base classes are intended for advanced extensibility scenarios and therefore are not interesting to the majority of users.

X AVOID naming base classes with a "Base" suffix if the class is intended for use in public APIs.

See Also

Framework Design Guidelines
Designing for Extensibility

Sealing

1/5/2018 • 1 min to read • Edit Online

One of the features of object-oriented frameworks is that developers can extend and customize them in ways unanticipated by the framework designers. This is both the power and danger of extensible design. When you design your framework, it is, therefore, very important to carefully design for extensibility when it is desired, and to limit extensibility when it is dangerous.

A powerful mechanism that prevents extensibility is sealing. You can seal either the class or individual members. Sealing a class prevents users from inheriting from the class. Sealing a member prevents users from overriding a particular member.

X DO NOT seal classes without having a good reason to do so.

Sealing a class because you cannot think of an extensibility scenario is not a good reason. Framework users like to inherit from classes for various nonobvious reasons, like adding convenience members. See Unsealed Classes for examples of nonobvious reasons users want to inherit from a type.

Good reasons for sealing a class include the following:

- The class is a static class. See Static Class Design.
- The class stores security-sensitive secrets in inherited protected members.
- The class inherits many virtual members and the cost of sealing them individually would outweigh the benefits of leaving the class unsealed.
- The class is an attribute that requires very fast runtime look-up. Sealed attributes have slightly higher performance levels than unsealed ones. See Attributes.

X DO NOT declare protected or virtual members on sealed types.

By definition, sealed types cannot be inherited from. This means that protected members on sealed types cannot be called, and virtual methods on sealed types cannot be overridden.

✓ CONSIDER sealing members that you override.

Problems that can result from introducing virtual members (discussed in Virtual Members) apply to overrides as well, although to a slightly lesser degree. Sealing an override shields you from these problems starting from that point in the inheritance hierarchy.

See Also

Framework Design Guidelines
Designing for Extensibility
Unsealed Classes

Design Guidelines for Exceptions

1/5/2018 • 1 min to read • Edit Online

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

In This Section

See Also

Framework Design Guidelines

Exception Throwing

1/5/2018 • 2 min to read • Edit Online

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

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

X DO NOT return error codes.

Exceptions are the primary means of reporting errors in frameworks.

✓ DO report execution failures by throwing exceptions.

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

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

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

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

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

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

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

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

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

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

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

✓ CONSIDER using exception builder methods.

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

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

X DO NOT throw exceptions from exception filter blocks.

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

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

See Also

Framework Design Guidelines
Design Guidelines for Exceptions

Using Standard Exception Types

1/5/2018 • 1 min to read • Edit Online

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

Exception and SystemException

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

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

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

ApplicationException

X DO NOT throw or derive from ApplicationException.

InvalidOperationException

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

ArgumentException, ArgumentNullException, and ArgumentOutOfRangeException

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

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

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

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

NullReferenceException, IndexOutOfRangeException, and AccessViolationException

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

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

StackOverflowException

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

X DO NOT catch `StackOverflowException`.

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

OutOfMemoryException

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

ComException, SEHException, and ExecutionEngineException

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

See Also

Framework Design Guidelines
Design Guidelines for Exceptions

Exceptions and Performance

1/5/2018 • 2 min to read • Edit Online

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

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

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

Tester-Doer Pattern

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

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

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

```
ICollection<int> numbers = ...
...
if(!numbers.IsReadOnly){
    numbers.Add(1);
}
```

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

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

Try-Parse Pattern

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

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

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

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

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

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

See Also

Framework Design Guidelines
Design Guidelines for Exceptions

Usage Guidelines

1/5/2018 • 1 min to read • Edit Online

This section contains guidelines for using common types in publicly accessible APIs. It deals with direct usage of built-in Framework types (e.g., serialization attributes) and overloading common operators.

The System.IDisposable interface is not covered in this section, but is discussed in the Dispose Pattern section.

NOTE

For guidelines and additional information about about other common, built-in .NET Framework types, see the reference topics for the following: System.DateTime, System.DateTimeOffset, System.ICloneable, System.IComparable<T>, System.IEquatable<T>, System.Nullable<T>, System.Object, System.Uri.

In This Section

See Also

Framework Design Guidelines

Arrays

1/5/2018 • 1 min to read • Edit Online

✓ **DO** prefer using collections over arrays in public APIs. The Collections section provides details about how to choose between collections and arrays.

X DO NOT use read-only array fields. The field itself is read-only and can't be changed, but elements in the array can be changed.

✓ **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.

See Also

Array
Framework Design Guidelines
Usage Guidelines

Attributes

1/5/2018 • 2 min to read • Edit Online

System.Attribute is a base class used to define custom attributes.

Attributes are annotations that can be added to programming elements such as assemblies, types, members, and parameters. They are stored in the metadata of the assembly and can be accessed at runtime using the reflection APIs. For example, the Framework defines the ObsoleteAttribute, which can be applied to a type or a member to indicate that the type or member has been deprecated.

Attributes can have one or more properties that carry additional data related to the attribute. For example, `ObsoleteAttribute` could carry additional information about the release in which a type or a member got deprecated and the description of the new APIs replacing the obsolete API.

Some properties of an attribute must be specified when the attribute is applied. These are referred to as the required properties or required arguments, because they are represented as positional constructor parameters. For example, the ConditionString property of the ConditionalAttribute is a required property.

Properties that do not necessarily have to be specified when the attribute is applied are called optional properties (or optional arguments). They are represented by settable properties. Compilers provide special syntax to set these properties when an attribute is applied. For example, the AttributeUsageAttribute.Inherited property represents an optional argument.

✓ **DO** name custom attribute classes with the suffix "Attribute."

✓ **DO** apply the AttributeUsageAttribute to custom attributes.

✓ **DO** provide settable properties for optional arguments.

✓ **DO** provide get-only properties for required arguments.

✓ **DO** provide constructor parameters to initialize properties corresponding to required arguments. Each parameter should have the same name (although with different casing) as the corresponding property.

X AVOID providing constructor parameters to initialize properties corresponding to the optional arguments.

In other words, do not have properties that can be set with both a constructor and a setter. This guideline makes very explicit which arguments are optional and which are required, and avoids having two ways of doing the same thing.

X AVOID overloading custom attribute constructors.

Having only one constructor clearly communicates to the user which arguments are required and which are optional.

✓ **DO** seal custom attribute classes, if possible. This makes the look-up for the attribute faster.

See Also

Framework Design Guidelines

Guidelines for Collections

1/5/2018 • 7 min to read • Edit Online

Any type designed specifically to manipulate a group of objects having some common characteristic can be considered a collection. It is almost always appropriate for such types to implement IEnumerable or IEnumerable<T>, so in this section we only consider types implementing one or both of those interfaces to be collections.

X 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).

X 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 not ever 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. The following two sections describe types (abstractions) designed specifically for use in public APIs.

X 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.

X 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.

X DO NOT implement both `IEnumerator<T>` and `IEnumerable<T>` on the same type. The same applies to the nongeneric interfaces `IEnumerator` and `IEnumerable`.

Collection Parameters

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

X 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

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

If `Collection<T>` does not meet some requirement (e.g., the collection must not implement IList), use a custom collection by implementing `IEnumerable<T>`, `ICollection<T>`, or IList<T>.

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

In general, prefer `ReadOnlyCollection<T>`. If it does not meet some requirement (e.g., the collection must not implement `IList`), use a custom collection by implementing `IEnumerable<T>`, `ICollection<T>`, or `IList<T>`. If you do implement a custom read-only collection, implement `ICollection<T>.ReadOnly` to return false.

In cases where you are sure that the only scenario you will ever want to support is forward-only iteration, you can simply use `IEnumerable<T>`.

✓ **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.

This will make it possible for you to add helper methods or change the collection implementation in the future.

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

Keyed collections usually have larger memory footprints and should not be used if the memory overhead outweighs the benefits of having the keys.

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

The general rule is that null and empty (0 item) collections or arrays should be treated the same.

Snapshots Versus Live Collections

Collections representing a state at some point in time are called snapshot collections. For example, a collection containing rows returned from a database query would be a snapshot. Collections that always represent the current state are called live collections. For example, a collection of `ComboBox` items is a live collection.

X DO NOT return snapshot collections from properties. Properties should return live collections.

Property getters should be very lightweight operations. Returning a snapshot requires creating a copy of an internal collection in an O(n) operation.

✓ **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.

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 arrays 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.

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

✓ **DO** use byte arrays instead of collections of bytes.

X 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.

Implementing Custom Collections

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

✓ **DO** implement `IEnumerable<T>` when designing new collections. Consider implementing `ICollection<T>` or even `IList<T>` where it makes sense.

When implementing such 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 like these two collections name them, and so on.

✓ **CONSIDER** implementing nongeneric collection interfaces (`IList` and `ICollection`) if the collection will often be passed to APIs taking these interfaces as input.

X AVOID implementing collection interfaces on types with complex APIs unrelated to the concept of a collection.

X DO NOT inherit from nongeneric base collections such as `CollectionBase`. Use `Collection<T>`, `ReadOnlyCollection<T>`, and `KeyedCollection<TKey,TItem>` instead.

Naming Custom Collections

Collections (types that implement `IEnumerable`) are created mainly for two reasons: (1) to create a new data structure with structure-specific operations and often different performance characteristics than existing data structures (e.g., List<T>, LinkedList<T>, Stack<T>), and (2) to create a specialized collection for holding a specific set of items (e.g., StringCollection). Data structures are most often used in the internal implementation of applications and libraries. Specialized collections are mainly to be exposed in APIs (as property and parameter types).

✓ **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.

X AVOID using any suffixes implying particular implementation, such as "LinkedList" or "Hashtable," in names of collection abstractions.

✓ **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`.

See Also

Framework Design Guidelines
Usage Guidelines

Serialization

1/5/2018 • 5 min to read • Edit Online

Serialization is the process of converting an object into a format that can be readily persisted or transported. For example, you can serialize an object, transport it over the Internet using HTTP, and deserialized it at the destination machine.

The .NET Framework offers three main serialization technologies optimized for various serialization scenarios. The following table lists these technologies and the main Framework types related to these technologies.

TECHNOLOGY NAME	MAIN TYPES	SCENARIOS
Data Contract Serialization	DataContractAttribute DataMemberAttribute DataContractSerializer NetDataContractSerializer DataContractJsonSerializer ISerializable	General persistence Web Services JSON
XML Serialization	XmlSerializer	XML format with full control over the shape of the XML
Runtime Serialization (Binary and SOAP)	SerializableAttribute ISerializable BinaryFormatter SoapFormatter	.NET Remoting

✓ **DO** think about serialization when you design new types.

Choosing the Right Serialization Technology to Support

✓ **CONSIDER** supporting Data Contract Serialization if instances of your type might need to be persisted or used in Web Services.

✓ **CONSIDER** supporting the XML Serialization instead of or in addition to Data Contract Serialization if you need more control over the XML format that is produced when the type is serialized.

This may be necessary in some interoperability scenarios where you need to use an XML construct that is not supported by Data Contract Serialization, for example, to produce XML attributes.

✓ **CONSIDER** supporting the Runtime Serialization if instances of your type need to travel across .NET Remoting boundaries.

X AVOID supporting Runtime Serialization or XML Serialization just for general persistence reasons. Prefer Data Contract Serialization instead.

Supporting Data Contract Serialization

Types can support Data Contract Serialization by applying the DataContractAttribute to the type and the DataMemberAttribute to the members (fields and properties) of the type.

✓ **CONSIDER** marking data members of your type public if the type can be used in partial trust.

In full trust, Data Contract serializers can serialize and deserialize nonpublic types and members, but only public members can be serialized and deserialized in partial trust.

✓ **DO** implement a getter and setter on all properties that have DataMemberAttribute. Data Contract serializers require both the getter and the setter for the type to be considered serializable. (In .NET Framework 3.5 SP1, some collection properties can be get-only.) If the type won't be used in partial trust, one or both of the property accessors can be nonpublic.

✓ **CONSIDER** using the serialization callbacks for initialization of deserialized instances.

Constructors are not called when objects are deserialized. (There are exceptions to the rule. Constructors of collections marked with CollectionDataContractAttribute are called during deserialization.) Therefore, any logic that executes during normal construction needs to be implemented as one of the serialization callbacks.

`OnDeserializedAttribute` is the most commonly used callback attribute. The other attributes in the family are OnDeserializingAttribute, OnSerializingAttribute, and OnSerializedAttribute. They can be used to mark callbacks that get executed before deserialization, before serialization, and finally, after serialization, respectively.

✓ **CONSIDER** using the KnownTypeAttribute to indicate concrete types that should be used when deserializing a complex object graph.

✓ **DO** consider backward and forward compatibility when creating or changing serializable types.

Keep in mind that serialized streams of future versions of your type can be deserialized into the current version of the type, and vice versa.

Make sure you understand that data members, even private and internal, cannot change their names, types, or even their order in future versions of the type unless special care is taken to preserve the contract using explicit parameters to the data contract attributes.

Test compatibility of serialization when making changes to serializable types. Try deserializing the new version into an old version, and vice versa.

✓ **CONSIDER** implementing IExtensibleDataObject to allow round-tripping between different versions of the type.

The interface allows the serializer to ensure that no data is lost during round-tripping. The IExtensibleDataObject.ExtensionData property is used to store any data from the future version of the type that is unknown to the current version, and so it cannot store it in its data members. When the current version is subsequently serialized and deserialized into a future version, the additional data will be available in the serialized stream.

Supporting XML Serialization

Data Contract Serialization is the main (default) serialization technology in the .NET Framework, but there are serialization scenarios that Data Contract Serialization does not support. For example, it does not give you full control over the shape of XML produced or consumed by the serializer. If such fine control is required, XML Serialization has to be used, and you need to design your types to support this serialization technology.

X AVOID designing your types specifically for XML Serialization, unless you have a very strong reason to control the shape of the XML produced. This serialization technology has been superseded by the Data Contract Serialization discussed in the previous section.

✓ **CONSIDER** implementing the IXmlSerializable interface if you want even more control over the shape of the serialized XML than what's offered by applying the XML Serialization attributes. Two methods of the interface, ReadXml and WriteXml, allow you to fully control the serialized XML stream. You can also control the XML schema that gets generated for the type by applying the `XmlSchemaProviderAttribute`.

Supporting Runtime Serialization

Runtime Serialization is a technology used by .NET Remoting. If you think your types will be transported using .NET Remoting, you need to make sure they support Runtime Serialization.

The basic support for Runtime Serialization can be provided by applying the SerializableAttribute, and more advanced scenarios involve implementing a simple Runtime Serializable Pattern (implement ISerializable and provide serialization constructor).

✓ **CONSIDER** supporting Runtime Serialization if your types will be used with .NET Remoting. For example, the System.AddIn namespace uses .NET Remoting, and so all types exchanged between `System.AddIn` add-ins need to support Runtime Serialization.

✓ **CONSIDER** implementing the Runtime Serializable Pattern if you want complete control over the serialization process. For example, if you want to transform data as it gets serialized or deserialized.

The pattern is very simple. All you need to do is implement the ISerializable interface and provide a special constructor that is used when the object is deserialized.

✓ **DO** make the serialization constructor protected and provide two parameters typed and named exactly as shown in the sample here.

```
[Serializable]
public class Person : ISerializable {
    protected Person(SerializationInfo info, StreamingContext context) {
        ...
    }
}
```

✓ **DO** implement the `ISerializable` members explicitly.

✓ **DO** apply a link demand to ISerializable.GetObjectData implementation. This ensures that only fully trusted core and the Runtime Serializer have access to the member.

See Also

Framework Design Guidelines
Usage Guidelines

System.Xml Usage

1/5/2018 • 1 min to read • Edit Online

This section talks about usage of several types residing in System.Xml namespaces that can be used to represent XML data.

X DO NOT use XmlNode or XmlDocument to represent XML data. Favor using instances of IXPathNavigable, XmlReader, XmlWriter, or subtypes of XNode instead. `XmlNode` and `XmlDocument` are not designed for exposing in public APIs.

✓ DO use `XmlReader`, `IXPathNavigable`, or subtypes of `XNode` as input or output of members that accept or return XML.

Use these abstractions instead of `XmlDocument`, `XmlNode`, or XPathDocument, because this decouples the methods from specific implementations of an in-memory XML document and allows them to work with virtual XML data sources that expose `XNode`, `XmlReader`, or XPathNavigator.

X DO NOT subclass `XmlDocument` if you want to create a type representing an XML view of an underlying object model or data source.

See Also

Framework Design Guidelines
Usage Guidelines

Equality Operators

1/5/2018 • 1 min to read • Edit Online

This section discusses overloading equality operators and refers to `operator==` and `operator!=` as equality operators.

X DO NOT overload one of the equality operators and not the other.

✓ DO ensure that Object.Equals and the equality operators have exactly the same semantics and similar performance characteristics.

This often means that `Object.Equals` needs to be overridden when the equality operators are overloaded.

X AVOID throwing exceptions from equality operators.

For example, return false if one of the arguments is null instead of throwing `NullReferenceException`.

Equality Operators on Value Types

✓ DO overload the equality operators on value types, if equality is meaningful.

In most programming languages, there is no default implementation of `operator==` for value types.

Equality Operators on Reference Types

X AVOID overloading equality operators on mutable reference types.

Many languages have built-in equality operators for reference types. The built-in operators usually implement the reference equality, and many developers are surprised when the default behavior is changed to the value equality.

This problem is mitigated for immutable reference types because immutability makes it much harder to notice the difference between reference equality and value equality.

X AVOID overloading equality operators on reference types if the implementation would be significantly slower than that of reference equality.

See Also

Framework Design Guidelines
Usage Guidelines

Common Design Patterns

1/5/2018 • 1 min to read • Edit Online

There are numerous books on software patterns, pattern languages, and antipatterns that address the very broad subject of patterns. Thus, this chapter provides guidelines and discussion related to a very limited set of patterns that are used frequently in the design of the .NET Framework APIs.

In This Section

Dependency Properties
Dispose Pattern

See Also

Framework Design Guidelines

Dependency Properties

1/5/2018 • 3 min to read • Edit Online

A dependency property (DP) is a regular property that stores its value in a property store instead of storing it in a type variable (field), for example.

An attached dependency property is a kind of dependency property modeled as static Get and Set methods representing "properties" describing relationships between objects and their containers (e.g., the position of a `Button` object on a `Panel` container).

✓ **DO** provide the dependency properties, if you need the properties to support WPF features such as styling, triggers, data binding, animations, dynamic resources, and inheritance.

Dependency Property Design

✓ **DO** inherit from DependencyObject, or one of its subtypes, when implementing dependency properties. The type provides a very efficient implementation of a property store and automatically supports WPF data binding.

✓ **DO** provide a regular CLR property and public static read-only field storing an instance of System.Windows.DependencyProperty for each dependency property.

✓ **DO** implement dependency properties by calling instance methods DependencyObject.GetValue and DependencyObject.SetValue.

✓ **DO** name the dependency property static field by suffixing the name of the property with "Property."

X DO NOT set default values of dependency properties explicitly in code; set them in metadata instead.

If you set a property default explicitly, you might prevent that property from being set by some implicit means, such as a styling.

X DO NOT put code in the property accessors other than the standard code to access the static field.

That code won't execute if the property is set by implicit means, such as a styling, because styling uses the static field directly.

X DO NOT use dependency properties to store secure data. Even private dependency properties can be accessed publicly.

Attached Dependency Property Design

Dependency properties described in the preceding section represent intrinsic properties of the declaring type; for example, the `Text` property is a property of `TextButton`, which declares it. A special kind of dependency property is the attached dependency property.

A classic example of an attached property is the Column property. The property represents Button's (not Grid's) column position, but it is only relevant if the Button is contained in a Grid, and so it's "attached" to Buttons by Grids.

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Button Grid.Column="0">Click</Button>
    <Button Grid.Column="1">Clack</Button>
</Grid>
```

The definition of an attached property looks mostly like that of a regular dependency property, except that the accessors are represented by static Get and Set methods:

```
public class Grid {

    public static int GetColumn(DependencyObject obj) {
        return (int)obj.GetValue(ColumnProperty);
    }

    public static void SetColumn(DependencyObject obj, int value) {
        obj.SetValue(ColumnProperty,value);
    }

    public static readonly DependencyProperty ColumnProperty =
        DependencyProperty.RegisterAttached(
            "Column",
            typeof(int),
            typeof(Grid)
    );
}
```

Dependency Property Validation

Properties often implement what is called validation. Validation logic executes when an attempt is made to change the value of a property.

Unfortunately dependency property accessors cannot contain arbitrary validation code. Instead, dependency property validation logic needs to be specified during property registration.

X DO NOT put dependency property validation logic in the property's accessors. Instead, pass a validation callback to `DependencyProperty.Register` method.

Dependency Property Change Notifications

X DO NOT implement change notification logic in dependency property accessors. Dependency properties have a built-in change notifications feature that must be used by supplying a change notification callback to the PropertyMetadata.

Dependency Property Value Coercion

Property coercion takes place when the value given to a property setter is modified by the setter before the property store is actually modified.

X DO NOT implement coercion logic in dependency property accessors.

Dependency properties have a built-in coercion feature, and it can be used by supplying a coercion callback to the `PropertyMetadata`.

See Also

Framework Design Guidelines
Common Design Patterns

Dispose Pattern

1/5/2018 • 10 min to read • Edit Online

All programs acquire one or more system resources, such as memory, system handles, or database connections, during the course of their execution. Developers have to be careful when using such system resources, because they must be released after they have been acquired and used.

The CLR provides support for automatic memory management. Managed memory (memory allocated using the C# operator `new`) does not need to be explicitly released. It is released automatically by the garbage collector (GC). This frees developers from the tedious and difficult task of releasing memory and has been one of the main reasons for the unprecedented productivity afforded by the .NET Framework.

Unfortunately, managed memory is just one of many types of system resources. Resources other than managed memory still need to be released explicitly and are referred to as unmanaged resources. The GC was specifically not designed to manage such unmanaged resources, which means that the responsibility for managing unmanaged resources lies in the hands of the developers.

The CLR provides some help in releasing unmanaged resources. System.Object declares a virtual method Finalize (also called the finalizer) that is called by the GC before the object's memory is reclaimed by the GC and can be overridden to release unmanaged resources. Types that override the finalizer are referred to as finalizable types.

Although finalizers are effective in some cleanup scenarios, they have two significant drawbacks:

- The finalizer is called when the GC detects that an object is eligible for collection. This happens at some undetermined period of time after the resource is not needed anymore. The delay between when the developer could or would like to release the resource and the time when the resource is actually released by the finalizer might be unacceptable in programs that acquire many scarce resources (resources that can be easily exhausted) or in cases in which resources are costly to keep in use (e.g., large unmanaged memory buffers).

- When the CLR needs to call a finalizer, it must postpone collection of the object's memory until the next round of garbage collection (the finalizers run between collections). This means that the object's memory (and all objects it refers to) will not be released for a longer period of time.

Therefore, relying exclusively on finalizers might not be appropriate in many scenarios when it is important to reclaim unmanaged resources as quickly as possible, when dealing with scarce resources, or in highly performant scenarios in which the added GC overhead of finalization is unacceptable.

The Framework provides the System.IDisposable interface that should be implemented to provide the developer a manual way to release unmanaged resources as soon as they are not needed. It also provides the GC.SuppressFinalize method that can tell the GC that an object was manually disposed of and does not need to be finalized anymore, in which case the object's memory can be reclaimed earlier. Types that implement the `IDisposable` interface are referred to as disposable types.

The Dispose Pattern is intended to standardize the usage and implementation of finalizers and the `IDisposable` interface.

The main motivation for the pattern is to reduce the complexity of the implementation of the Finalize and the Dispose methods. The complexity stems from the fact that the methods share some but not all code paths (the differences are described later in the chapter). In addition, there are historical reasons for some elements of the pattern related to the evolution of language support for deterministic resource management.

✓ **DO** implement the Basic Dispose Pattern on types containing instances of disposable types. See the Basic

Dispose Pattern section for details on the basic pattern.

If a type is responsible for the lifetime of other disposable objects, developers need a way to dispose of them, too. Using the container's `Dispose` method is a convenient way to make this possible.

✓ **DO** implement the Basic Dispose Pattern and provide a finalizer on types holding resources that need to be freed explicitly and that do not have finalizers.

For example, the pattern should be implemented on types storing unmanaged memory buffers. The Finalizable Types section discusses guidelines related to implementing finalizers.

✓ **CONSIDER** implementing the Basic Dispose Pattern on classes that themselves don't hold unmanaged resources or disposable objects but are likely to have subtypes that do.

A great example of this is the System.IO.Stream class. Although it is an abstract base class that doesn't hold any resources, most of its subclasses do and because of this, it implements this pattern.

Basic Dispose Pattern

The basic implementation of the pattern involves implementing the `System.IDisposable` interface and declaring the `Dispose(bool)` method that implements all resource cleanup logic to be shared between the `Dispose` method and the optional finalizer.

The following example shows a simple implementation of the basic pattern:

```
public class DisposableResourceHolder : IDisposable {

    private SafeHandle resource; // handle to a resource

    public DisposableResourceHolder(){
        this.resource = ... // allocates the resource
    }

    public void Dispose(){
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing){
        if (disposing){
            if (resource!= null) resource.Dispose();
        }
    }
}
```

The Boolean parameter `disposing` indicates whether the method was invoked from the `IDisposable.Dispose` implementation or from the finalizer. The `Dispose(bool)` implementation should check the parameter before accessing other reference objects (e.g., the resource field in the preceding sample). Such objects should only be accessed when the method is called from the `IDisposable.Dispose` implementation (when the `disposing` parameter is equal to true). If the method is invoked from the finalizer (`disposing` is false), other objects should not be accessed. The reason is that objects are finalized in an unpredictable order and so they, or any of their dependencies, might already have been finalized.

Also, this section applies to classes with a base that does not already implement the Dispose Pattern. If you are inheriting from a class that already implements the pattern, simply override the `Dispose(bool)` method to provide additional resource cleanup logic.

✓ **DO** declare a protected virtual void `Dispose(bool disposing)` method to centralize all logic related to releasing unmanaged resources.

All resource cleanup should occur in this method. The method is called from both the finalizer and the `IDisposable.Dispose` method. The parameter will be false if being invoked from inside a finalizer. It should be used to ensure any code running during finalization is not accessing other finalizable objects. Details of implementing finalizers are described in the next section.

```
protected virtual void Dispose(bool disposing){
    if (disposing){
        if (resource!= null) resource.Dispose();
    }
}
```

✓ **DO** implement the `IDisposable` interface by simply calling `Dispose(true)` followed by `GC.SuppressFinalize(this)`.

The call to `SuppressFinalize` should only occur if `Dispose(true)` executes successfully.

```
public void Dispose(){
    Dispose(true);
    GC.SuppressFinalize(this);
}
```

X DO NOT make the parameterless `Dispose` method virtual.

The `Dispose(bool)` method is the one that should be overridden by subclasses.

```
// bad design
public class DisposableResourceHolder : IDisposable {
    public virtual void Dispose(){ ... }
    protected virtual void Dispose(bool disposing){ ... }
}

// good design
public class DisposableResourceHolder : IDisposable {
    public void Dispose(){ ... }
    protected virtual void Dispose(bool disposing){ ... }
}
```

X DO NOT declare any overloads of the `Dispose` method other than `Dispose()` and `Dispose(bool)`.

`Dispose` should be considered a reserved word to help codify this pattern and prevent confusion among implementers, users, and compilers. Some languages might choose to automatically implement this pattern on certain types.

✓ **DO** allow the `Dispose(bool)` method to be called more than once. The method might choose to do nothing after the first call.

```
public class DisposableResourceHolder : IDisposable {

    bool disposed = false;

    protected virtual void Dispose(bool disposing){
        if(disposed) return;
        // cleanup
        ...
        disposed = true;
    }
}
```

X AVOID throwing an exception from within `Dispose(bool)` except under critical situations where the containing process has been corrupted (leaks, inconsistent shared state, etc.).

Users expect that a call to `Dispose` will not raise an exception.

If `Dispose` could raise an exception, further finally-block cleanup logic will not execute. To work around this, the user would need to wrap every call to `Dispose` (within the finally block!) in a try block, which leads to very complex cleanup handlers. If executing a `Dispose(bool disposing)` method, never throw an exception if disposing is false. Doing so will terminate the process if executing inside a finalizer context.

✓ **DO** throw an ObjectDisposedException from any member that cannot be used after the object has been disposed of.

```
public class DisposableResourceHolder : IDisposable {
    bool disposed = false;
    SafeHandle resource; // handle to a resource

    public void DoSomething(){
           if(disposed) throw new ObjectDisposedException(...);
        // now call some native methods using the resource
            ...
    }
    protected virtual void Dispose(bool disposing){
        if(disposed) return;
        // cleanup
        ...
        disposed = true;
    }
}
```

✓ **CONSIDER** providing method `Close()`, in addition to the `Dispose()`, if close is standard terminology in the area.

When doing so, it is important that you make the `Close` implementation identical to `Dispose` and consider implementing the `IDisposable.Dispose` method explicitly.

```
public class Stream : IDisposable {
    IDisposable.Dispose(){
        Close();
    }
    public void Close(){
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

Finalizable Types

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the `Dispose(bool)` method.

Finalizers are notoriously difficult to implement correctly, primarily because you cannot make certain (normally valid) assumptions about the state of the system during their execution. The following guidelines should be taken into careful consideration.

Note that some of the guidelines apply not just to the `Finalize` method, but to any code called from a finalizer. In the case of the Basic Dispose Pattern previously defined, this means logic that executes inside `Dispose(bool disposing)` when the `disposing` parameter is false.

If the base class already is finalizable and implements the Basic Dispose Pattern, you should not override `Finalize` again. You should instead just override the `Dispose(bool)` method to provide additional resource cleanup logic.

The following code shows an example of a finalizable type:

```
public class ComplexResourceHolder : IDisposable {

    private IntPtr buffer; // unmanaged memory buffer
    private SafeHandle resource; // disposable handle to a resource

    public ComplexResourceHolder(){
        this.buffer = ... // allocates memory
        this.resource = ... // allocates the resource
    }

    protected virtual void Dispose(bool disposing){
            ReleaseBuffer(buffer); // release unmanaged memory
        if (disposing){ // release other disposable objects
            if (resource!= null) resource.Dispose();
        }
    }

    ~ ComplexResourceHolder(){
        Dispose(false);
    }

    public void Dispose(){
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

X AVOID making types finalizable.

Carefully consider any case in which you think a finalizer is needed. There is a real cost associated with instances with finalizers, from both a performance and code complexity standpoint. Prefer using resource wrappers such as SafeHandle to encapsulate unmanaged resources where possible, in which case a finalizer becomes unnecessary because the wrapper is responsible for its own resource cleanup.

X DO NOT make value types finalizable.

Only reference types actually get finalized by the CLR, and thus any attempt to place a finalizer on a value type will be ignored. The C# and C++ compilers enforce this rule.

✓ DO make a type finalizable if the type is responsible for releasing an unmanaged resource that does not have its own finalizer.

When implementing the finalizer, simply call `Dispose(false)` and place all resource cleanup logic inside the `Dispose(bool disposing)` method.

```
public class ComplexResourceHolder : IDisposable {

    ~ ComplexResourceHolder(){
        Dispose(false);
    }

    protected virtual void Dispose(bool disposing){
        ...
    }
}
```

✓ DO implement the Basic Dispose Pattern on every finalizable type.

This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

X DO NOT access any finalizable objects in the finalizer code path, because there is significant risk that they will have already been finalized.

For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A's finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object (or calling a static method that might use values stored in static variables) might not be safe if Environment.HasShutdownStarted returns true.

✓ DO make your `Finalize` method protected.

C#, C++, and VB.NET developers do not need to worry about this, because the compilers help to enforce this guideline.

X DO NOT let exceptions escape from the finalizer logic, except for system-critical failures.

If an exception is thrown from a finalizer, the CLR will shut down the entire process (as of .NET Framework version 2.0), preventing other finalizers from executing and resources from being released in a controlled manner.

✓ CONSIDER creating and using a critical finalizable object (a type with a type hierarchy that contains CriticalFinalizerObject) for situations in which a finalizer absolutely must execute even in the face of forced application domain unloads and thread aborts.

See Also

IDisposable.Dispose
Object.Finalize
Framework Design Guidelines
Common Design Patterns
Garbage Collection