CD EXCERPT

.NET Framework Standard Library Annotated Reference Volume 1

Base Class Library and Extended Numerics Library

Brad Abrams

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Contents | Index

Contents

PART I
Namespace Overviews

System

The `System` namespace is the root of all namespaces in the .NET Framework, containing all other namespaces as subordinates. It also contains the types that we felt to be the most fundamental and frequently used.

Basic Variable Types

The class `Object` is the root of the inheritance hierarchy in the .NET Framework. Every class in the .NET Framework ultimately derives from this class. If you define a class without specifying any other inheritance, `Object` is the implied base class. It provides the most basic methods and properties that all objects need to support, such as returning an identifying string, returning a `Type` object (think of it as a class descriptor) to use for runtime discovery of the object's contents, and providing a location for a garbage collection finalizer.

The .NET Framework provides two kinds of types, value types and reference types. Instances of value types are allocated on the stack or inline inside an object, which incurs a lower overhead than using the managed heap. Value types are most often used for small, lightweight variables accessed primarily for a single data value, while still allowing them to be treated as objects in the inheritance hierarchy (for example, having methods). All value types must derive from the abstract base class `ValueType`. Table 1 lists the value types in the System namespace.

Table 1

Name	Represents
`Boolean`	Boolean value (true or false).
`Byte`	8-bit unsigned integer.
`Char`	UTF-16 code point.
`DateTime`	An instant in time, typically expressed as a date and time of day.
`Decimal`	Decimal number.
`Double`	Double-precision floating-point number.
`Enum`	Base class for enumerations.
`Int16`	16-bit signed integer.
`Int32`	32-bit signed integer.

TABLE 1 *(continued)*

Name	Represents
`Int64`	64-bit signed integer.
`SByte`	8-bit signed integer.
`Single`	Single-precision floating-point number.
`TimeSpan`	Time interval.
`UInt16`	16-bit unsigned integer.
`UInt32`	32-bit unsigned integer.
`UInt64`	64-bit unsigned integer.

All objects that are not value types are by definition reference types. Creating an instance of a reference type allocates the new object from the managed heap and returns a reference to it, hence the name. Most objects are reference types. The class `String` is a reference type that represents an immutable series of characters. The class `CharEnumerator` supports iterating over a `String` and reading its individual characters.

The `System` namespace also contains the abstract base class `Array`, which represents a fixed-size, ordered series of objects accessed by index. It contains methods for creating, manipulating, and searching for elements within the array. Programmers will generally not use this class directly. Instead, their programming language will provide an abstraction of it.

Attributes

The .NET Framework makes extensive use of attributes, descriptive pieces of read-only information that a programmer can place in an object's metadata. Attributes can be read by any interested piece of code that has the required level of permission. Many attributes are provided and used by the system. Others are defined by programmers and used for their own purposes. All attributes derive from the abstract base class `System.Attribute`. The attributes in Table 2 were felt to be common enough to occupy the `System` namespace. Many other subordinate namespaces also define more specialized attributes.

TABLE 2

Attributes	Meaning
`AttributeUsageAttribute`	Used in the definition of other attribute classes, specifying the target types to which the other attribute class can be applied (assembly, class, method, some combination, etc.). Uses `AttributeTargets` enumeration.
`CLSCompliantAttribute`	Indicates whether a program element is compliant with the Common Language Specification (CLS).
`FlagsAttribute`	Indicates that an enumeration can be treated as a bit field; that is, a set of flags.
`ObsoleteAttribute`	Marks the program elements that are no longer in use.

Utility Objects

The class `Console` provides functions for performing input and output to a console window. It's useful for debugging and development, and any functionality for which a full Windows interface is overkill.

The class `Convert` provides static methods for converting a variable of one base type into another base type, such as `Int32` to `Double`.

The class `GC` provides a connection to the garbage collector in the automatic memory management system. It contains methods such as `Collect`, which forces an immediate garbage collection.

The utility class `Environment` provides access to environment variables, and other environment properties such as machine name.

The class `MarshalByRefObject` is the abstract base class for objects that communicate across application domain boundaries by exchanging messages using a proxy. Classes must inherit from `MarshalByRefObject` when the type is used across application domain boundaries, and the state of the object must not be copied because the members of the object are not usable outside the application domain where they were created.

The class `Math` provides access to mathematical operations such as trigonometric and logarithmic functions.

The class `Random` provides methods that generate a sequence of random numbers, starting from a specified seed. You should use specialized cryptographic functionality (in the `System.Security.Cryptography` namespace) for random number generation for cryptographic purposes.

The class `Type` is the basis for all reflection operations. Think of it as a class descriptor.

The class `Version` represents a dotted quad version number (major, minor, build, revision). It is used in the utility functions that specify versioning behavior of assemblies.

Interfaces

The `System` namespace defines a number of interfaces. An interface is a set of pure virtual function definitions, which a class can choose to implement. You define an interface to enforce a common design pattern among classes that are not hierarchically related. For example, the `IDisposable` interface contains the method `Dispose`, used for deterministic finalization. This provides a way to force an object to perform its cleanup code immediately instead of when the garbage collector feels like getting around to it. Any class anywhere in any inheritance hierarchy might reasonably need this behavior. However, most classes won't need this behavior, so it wouldn't make sense to put it in the `System.Object` base class and force all objects to implement it whether they needed it or not. Instead, a class that needs this behavior implements the interface, ensuring that it follows the same syntactic rules as all other objects that do so, without disturbing its inheritance relationships with its base classes. The interfaces in Table 3 were felt to be common enough to occupy the `System` namespace. Many other subordinate namespaces also define more specialized interfaces.

TABLE 3

Interface	Meaning
`IAsyncResult`	Represents the status of an asynchronous operation.
`ICloneable`	Supports cloning, which creates a new instance of a class with the same value as an existing instance.
`IComparable`	Defines a generalized comparison method that a value type or class implements to create a type-specific comparison method.
`IDisposable`	Defines a method to release allocated unmanaged resources.
`IFormatProvider`	Provides a mechanism for retrieving an object to control formatting.
`IFormattable`	Provides functionality to format the value of an object into a string representation.

Delegates

The .NET Framework supports callbacks from one object to another by means of the class `Delegate`. A `Delegate` represents a pointer to an individual object method or to a static class method. You generally will not use the `Delegate` class directly, but instead will use the wrapper provided by your programming language. The .NET Framework event system uses delegates. The object wanting to receive the event provides the sender with a delegate, and the sender calls the function on the delegate to signal the event.

The .NET Framework supports asynchronous method invocation for any method on any object. The caller can either poll for completion, or pass a delegate of the `AsyncCallback` class to be notified of completion by an asynchronous callback.

Exceptions

In order to provide a common, rich, easily programmed and difficult to ignore way of signaling and handling errors, the .NET Framework supports structured exception handling. A caller places an exception handler on the stack at the point at which he wants to catch the error, using the try–catch syntax of his programming language. A called function wanting to signal an error creates an object of class `System.Exception` (or one derived from it) containing information about the error and throws it. The CLR searches up the call stack until it finds a handler for the type of exception that was thrown, at which time the stack is unwound and control transferred to the catch block, which contains the error-handling code.

The class `System.Exception` is the base class from which all exception objects derive. It contains such basic information as a message provided by the thrower and the stack trace at which the exception took place. The class `System.SystemException` derives from it, and all system-provided exceptions derive from that. This allows a programmer to differentiate between system-provided and programmer-built exceptions. The system-provided exceptions in Table 4 were felt to be common enough to occupy the base `System` namespace. Many more specialized exception classes live in subordinate namespaces.

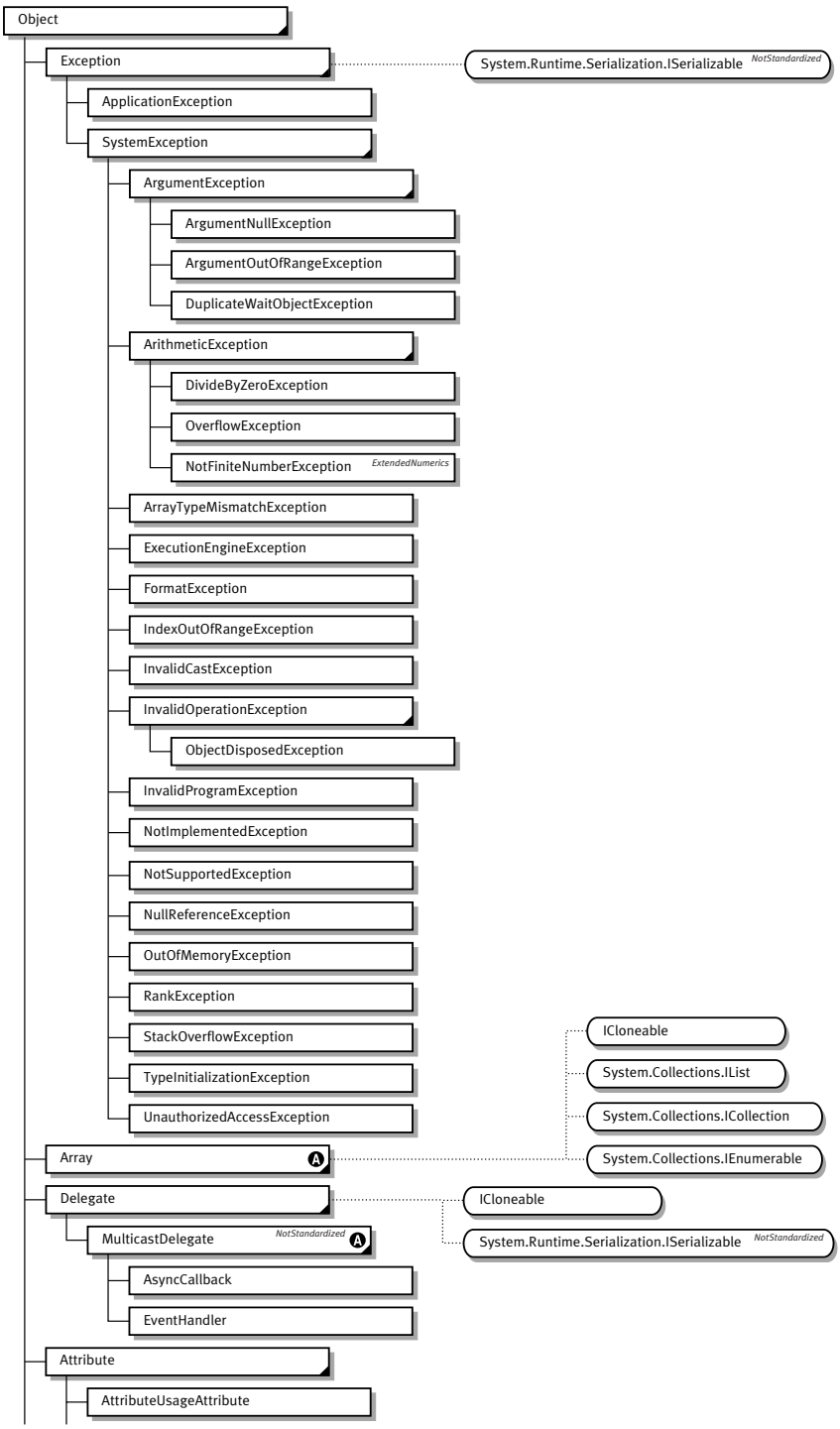
TABLE 4

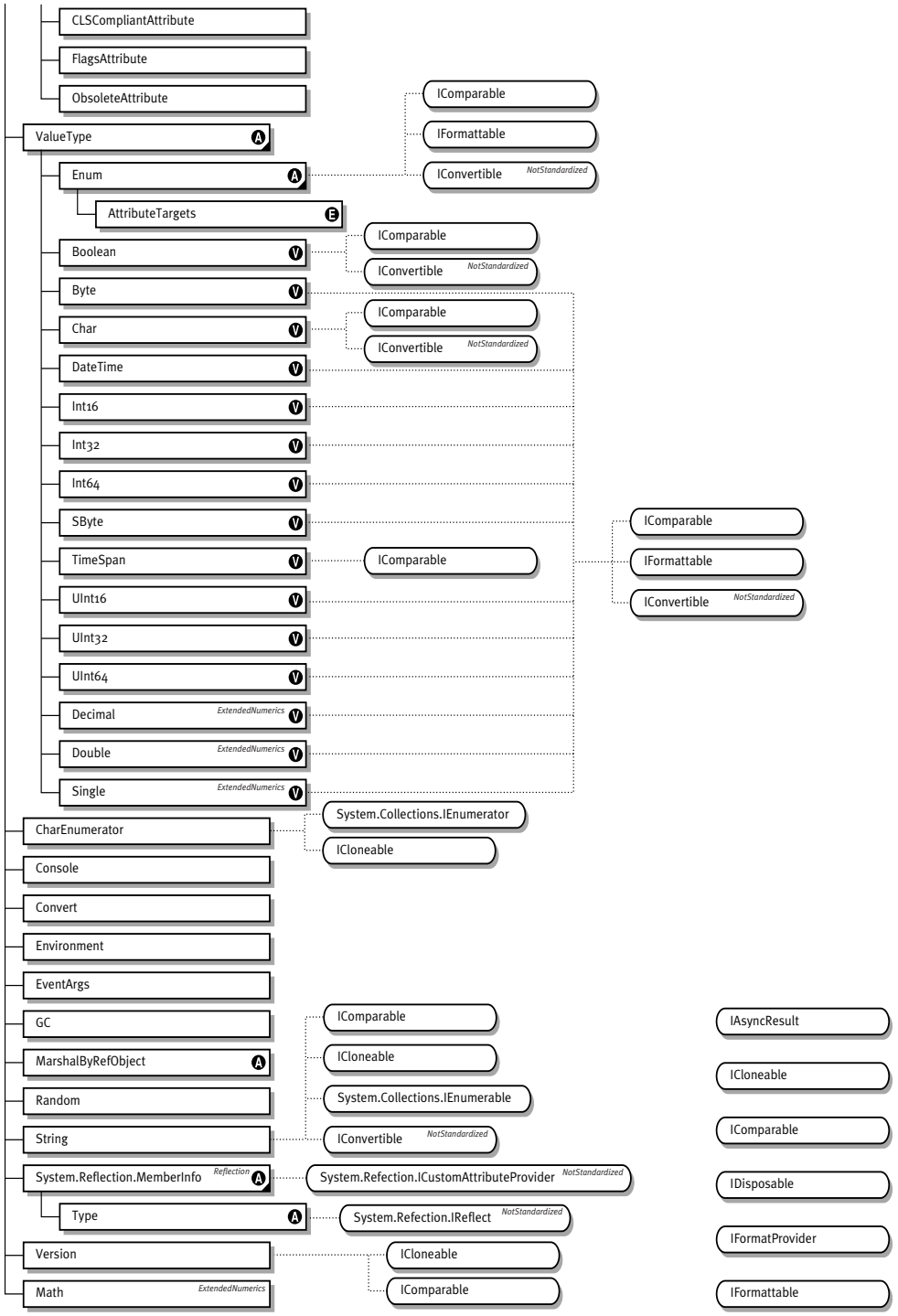
Exception	Meaning
`ApplicationException`	A non-fatal application error occurred.
`ArgumentException`	One of the arguments provided to a method is not valid.
`ArgumentNullException`	A `null` reference is passed to a member that does not accept it as a valid argument.
`ArgumentOutOfRange-Exception`	The value of an argument is outside the allowable range of values as defined by the invoked member.
`ArithmeticException`	Error in an arithmetic, casting, or conversion operation.
`ArrayTypeMismatch-Exception`	An attempt is made to store an element of the wrong type within an array.
`DivideByZeroException`	An attempt was made to divide an integral or decimal value by zero.
`DuplicateWaitObject-Exception`	An object appears more than once in an array of synchronization objects.
`ExecutionEngineException`	An internal error occurred in the execution engine of the common language runtime.

TABLE 4 *(continued)*

Exception	Meaning
`FormatException`	The format of an argument does not meet the parameter specifications of the invoked method.
`IndexOutOfRangeException`	An attempt is made to access an element of an array with an index that is outside the bounds of the array.
`InvalidCastException`	Invalid casting or explicit conversion.
`InvalidOperationException`	A method call is invalid for the object's current state.
`InvalidProgramException`	A program contains invalid Microsoft intermediate language (MSIL) or metadata. Generally this indicates a bug in a compiler.
`NotFiniteNumberException`	A floating-point value is positive infinity, negative infinity, or Not-a-Number (NaN).
`NotSupportedException`	An invoked method is not supported or not supported in the current mode of operation.
`NullReferenceException`	An attempt to dereference a `null` object reference.
`ObjectDisposedException`	An operation is performed on a disposed object.
`OutOfMemoryException`	There is not enough memory to continue the execution of a program.
`OverflowException`	An arithmetic, casting, or conversion operation in a checked context results in an overflow.
`RankException`	An array with the wrong number of dimensions is passed to a method.
`StackOverflowException`	The execution stack overflows by having too many pending method calls.
`TypeInitialization-Exception`	A wrapper around the exception thrown by the type initializer.
`UnauthorizedAccess-Exception`	The operating system denies access because of an I/O error or a specific type of security error.

Diagram


Object
Exception
System.Runtime.Serialization.ISerializable NotStandardized
ApplicationException
SystemException
ArgumentException
ArgumentNullException
ArgumentOutOfRangeException
DuplicateWaitObjectException
ArithmeticException
DivideByZeroException
OverflowException
NotFiniteNumberException ExtendedNumerics
ArrayTypeMismatchException
ExecutionEngineException
FormatException
IndexOutOfRangeException
InvalidCastException
InvalidOperationException
ObjectDisposedException
InvalidProgramException
NotImplementedException
NotSupportedException
NullReferenceException
OutOfMemoryException
RankException
StackOverflowException
TypeInitializationException
UnauthorizedAccessException
Array A
ICloneable
System.Collections.IList
System.Collections.ICollection
System.Collections.IEnumerable
Delegate
ICloneable
System.Runtime.Serialization.ISerializable NotStandardized
MulticastDelegate NotStandardized A
AsyncCallback
EventHandler
Attribute
AttributeUsageAttribute


CLSCompliantAttribute
FlagsAttribute
ObsoleteAttribute
ValueType
Enum
AttributeTargets
Boolean
Byte
Char
DateTime
Int16
Int32
Int64
SByte
TimeSpan
UInt16
UInt32
UInt64
Decimal
Double
Single
CharEnumerator
Console
Convert
Environment
EventArgs
GC
MarshalByRefObject
Random
String
System.Reflection.MemberInfo
Type
Version
Math
IComparable
IFormattable
IConvertible
NotStandardized
ExtendedNumerics
Reflection
System.Collections.IEnumerator
ICloneable
System.Collections.IEnumerable
System.Refection.ICustomAttributeProvider
System.Refection.IReflect
IAsyncResult
IDisposable
IFormatProvider

System.Collections

Organizing collections of objects is a vital but boring task that operating system designers have historically left to language implementers. Naturally, every language's and every vendor's implementation of collections varied drastically, making it essentially impossible for different applications to exchange, say, an array of objects without having intimate knowledge of each other's internal workings.

With the `System.Collections` namespace, Microsoft has brought the common implementation philosophy to the mundane task of organizing collections of objects. Rather than depend on individual languages to implement such common concepts as arrays and hash tables, Microsoft decided to bring them into the .NET Framework, thereby standardizing them for all applications. This namespace contains classes that are used to organize collections of objects, and also the interfaces that you can use to write your own collection classes while still retaining a common interface to callers.

The two main classes of collection are `ArrayList` and `Hashtable`. Each is dynamically sizable and can hold any type of object, even mixing contained object types within the same collection object. They differ in their organization strategies. The `ArrayList` is an ordered, numerically indexed collection of objects. When you place an object into an `ArrayList` or fetch an object from it, you specify which element location to put it in or fetch it from ("Put this object in slot 2," "Get the object from slot 5"). Think of it as a set of pigeonholes. It differs from the basic array class `System.Array` by being dynamically sizable. The architects felt that the basic fixed-size array was fundamental enough to join the most basic types in the `System` namespace.

A `Hashtable` is an unordered collection in which objects are identified by keys. When you place an object in a `Hashtable`, you specify the key that you want used to identify it. When you fetch an object from a `Hashtable`, you provide the key and the `Hashtable` returns the object that the key identifies. The key is most often a string, but it can be any type of object.

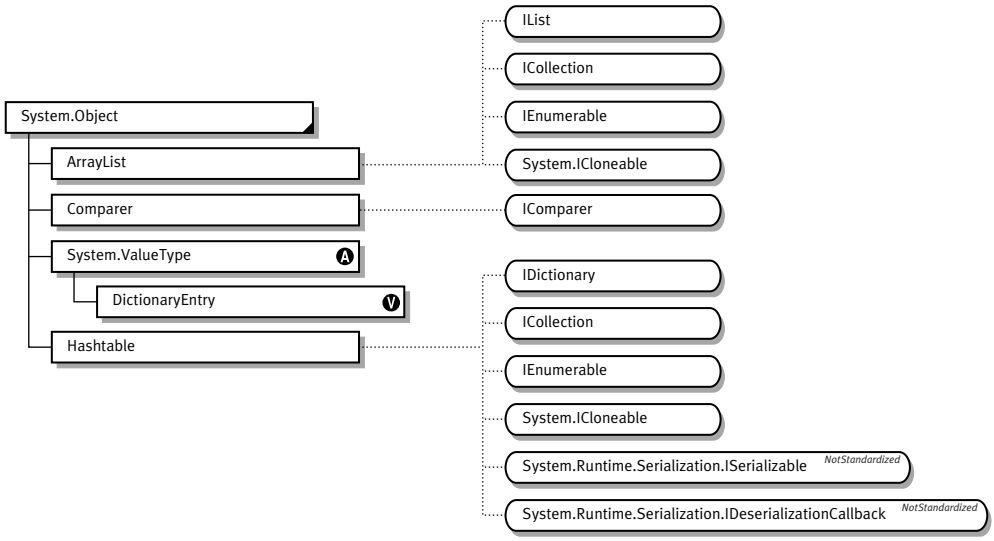
As you examine the individual member functions, you will notice that the collection classes share many common methods. For example, the `ArrayList` and `Hashtable` classes each contain the method `GetEnumerator`. These common behaviors ease the tasks of implementers and consumers alike. The collection classes obtain this commonality of behavior by implementing standardized interfaces. You probably want to do the same with your derived classes. The standardized interfaces, and their usages in the collection classes, are shown in Table 5.

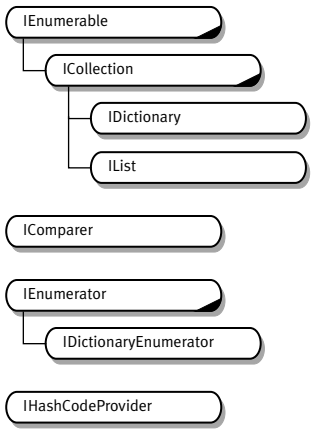
Note that a number of the interfaces are not implemented directly on the collection classes that I've listed. For example, the `IEnumerator` interface is not implemented directly on `ArrayList` or `Hashtable` object, but instead is returned by the `IEnumerable` interface, which is. Also note that the collection classes listed implement interfaces from other namespaces, such as `System.ICloneable`.

TABLE 5

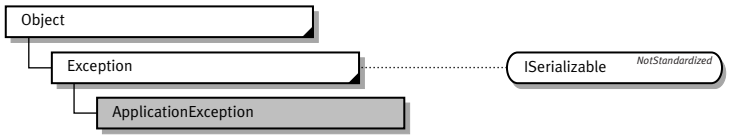
Interface	Description	ArrayList	HashTable
`ICollection`	Defines size, enumerators and synchronization methods for all collections.	Y	Y
`IComparer`	Exposes a method that compares two objects.		
`IDictionary`	Represents a collection of key-and-value pairs.		Y
`IDictionaryEnumerator`	Enumerates the elements of a dictionary.		
`IEnumerable`	Exposes the enumerator, which supports a simple iteration over a collection.	Y	Y
`IEnumerator`	Supports a simple iteration over a collection.		
`IHashCodeProvider`	Supplies a hash code for an object, using a custom hash function.		
`IList`	Represents a collection of objects that can be individually accessed by index.	Y	

Diagram


System.Object
ArrayList
Comparer
System.ValueType
A
DictionaryEntry
V
Hashtable
IList
ICollection
IEnumerable
System.ICloneable
IComparer
IDictionary
ICollection
IEnumerable
System.ICloneable
System.Runtime.Serialization.ISerializable
NotStandardized
System.Runtime.Serialization.IDeserializationCallback
NotStandardized


IEnumerable
ICollection
IDictionary
IList
IComparer
IEnumerator
IDictionaryEnumerator
IHashCodeProvider

PART II
Class Libraries



Summary

`System.ApplicationException` is the base class for all exceptions defined by applications.

Type Summary

```
public class ApplicationException : Exception
{
  // Constructors
     public ApplicationException ();
     public ApplicationException (string message);
     public ApplicationException (string message,
                                  Exception innerException);
MS ~~CF~~ protected ApplicationException (SerializationInfo info,
                                    StreamingContext context);
}
```

KC Designing exception hierarchies is tricky. Well-designed exception hierarchies are wide, not very deep, and contain only those exceptions for which there is a programmating scenario for catching. We added `ApplicationException` thinking it would add value by grouping exceptions declared outside of the .NET Framework, but there is no scenario for catching `ApplicationException` and it only adds unnecessary depth to the hierarchy.

JR You should not define new exception classes derived from `ApplicationException`; use Exception instead. In addition, you should not write code that catches `ApplicationException`.

Description

This class represents application-defined errors detected during the execution of an application. It is provided as means to differentiate between exceptions defined by applications versus exceptions defined by the system. [*Note:* For more information on exceptions defined by the system, see `System.SystemException`.]

[*Note:* `System.ApplicationException` does not provide information as to the cause of the exception. In most scenarios, instances of this class should not be thrown. In

cases where this class is instantiated, a human-readable message describing the error should be passed to the constructor.]

Example

The following example demonstrates catching an exception type that derives from `ApplicationException`. There is, however, no valid scenerio for catching an `ApplicationException` type.

```
using System;
using System.Reflection;

namespace Samples
{
  public class ApplicationExceptionSample
  {
    public static void Main()
    {
      try
      {
        Type t = typeof(string);
        MethodInfo m = t.GetMethod("EndsWith");
        string s = "Hello world!";
        object[] arguments = {"world!", "!"};
        Console.WriteLine(m.Invoke(s, arguments));
      }
      catch(ApplicationException e)
      {
        Console.WriteLine("Exception: {0}", e);
      }
    }
  }
}
```

The output is

```
Exception: System.Reflection.TargetParameterCountException: Parameter count mismatch.
   at System.Reflection.RuntimeMethodInfo.InternalInvoke(Object obj, BindingFlags
invokeAttr, Binder binder, Object[] parameters, CultureInfo culture, Boolean
isBinderDefault, Assembly caller, Boolean verifyAccess)
   at System.Reflection.RuntimeMethodInfo.InternalInvoke(Object obj, BindingFlags
invokeAttr, Binder binder, Object[] parameters, CultureInfo culture, Boolean
verifyAccess)
   at System.Reflection.RuntimeMethodInfo.Invoke(Object obj, BindingFlags
invokeAttr, Binder binder, Object[] parameters, CultureInfo culture)
   at System.Reflection.MethodBase.Invoke(Object obj, Object[] parameters)
   at Samples.ApplicationExceptionSample.Main() in C:\Books\BCL\Samples\System\
ApplicationException\ApplicationException.cs:line 16
```

ApplicationException() Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor()
[C#]
public ApplicationException()
```

Summary

Constructs and initializes a new instance of the `System.ApplicationException` class.

Description

This constructor initializes the `System.ApplicationException.Message` property of the new instance to a system-supplied message that describes the error, such as "An application error has occurred." This message takes into account the current system culture.

The `System.ApplicationException.InnerException` property is initialized to `null`.

ApplicationException(System.String) Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor(string message)
[C#]
public ApplicationException(string message)
```

Summary

Constructs and initializes a new instance of the `System.ApplicationException` class.

Parameters

Parameter	Description
message	A `System.String` that describes the error. The content of *message* is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.

Description

This constructor initializes the `System.ApplicationException.Message` property of the new instance using *message*. If *message* is `null`, the `System.ApplicationException.Message` property is initialized to the system-supplied message provided by the constructor that takes no arguments. The `System.ApplicationException.InnerException` property is initialized to `null`.

ApplicationException(System.String, System.Exception) Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor(string message, class
System.Exception innerException)
[C#]
public ApplicationException(string message, Exception innerException)
```

Summary

Constructs and initializes a new instance of the `System.ApplicationException` class.

Parameters

Parameter	Description
message	A `System.String` that describes the error. The content of *message* is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.
innerException	An instance of `System.Exception` that is the cause of the current Exception. If *innerException* is non-`null`, then the current Exception was raised in a catch block handling *innerException*.

Description

This constructor initializes the `System.ApplicationException.Message` property of the new instance using *message*, and the `System.ApplicationException.InnerException` property using *innerException*. If *message* is `null`, the `System.ApplicationException.Message` property is initialized to the system-supplied message provided by the constructor that takes no arguments.

[*Note:* For information on inner exceptions, see `System.Exception.InnerException`.]

ApplicationException(System.Runtime.Serialization.SerializationInfo, System.Runtime.Serialization.StreamingContext) Constructor

```
[ILASM]
family rtspecialname specialname instance void .ctor(class
System.Runtime.Serialization.SerializationInfo info, valuetype
System.Runtime.Serialization.StreamingContext context)
[C#]
protected ApplicationException(SerializationInfo info, StreamingContext context)
```

Summary

Initializes a new instance of the `System.ApplicationException` class with serialized data.

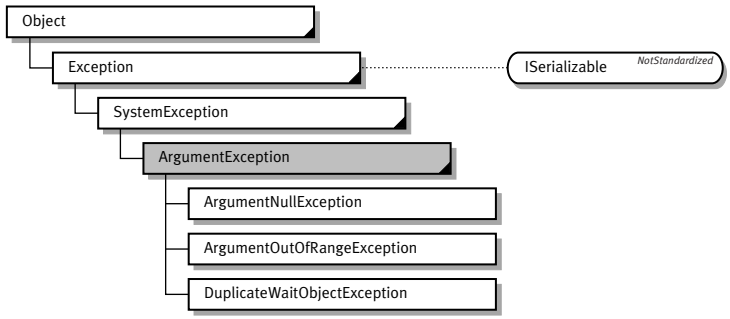
Parameters

Parameter	Description
info	The object that holds the serialized object data.
context	The contextual information about the source or destination.

Description

This constructor is called during deserialization to reconstitute the exception object transmitted over a stream.

System ArgumentException



Summary

Represents the error that occurs when an argument passed to a method is invalid.

Type Summary

```
public class ArgumentException : SystemException,
                                 ISerializable
{
  // Constructors
     public ArgumentException ();
     public ArgumentException (string message);
     public ArgumentException (string message,
                               string paramName);
  CF public ArgumentException (string message,
                               string paramName,
                               Exception innerException);
     public ArgumentException (string message,
                               Exception innerException);
MS CF protected ArgumentException (SerializationInfo info,
                                   StreamingContext context);

  // Properties
MS CF public override string Message { get; }
  CF public virtual string ParamName { get; }

  // Methods
MS CF public override void GetObjectData (SerializationInfo info,
                                          StreamingContext context);
}
```

Description

`System.ArgumentException` is thrown when a method is invoked and at least one of the passed arguments does not meet the method's parameter specification.

[*Note:* The Base Class Library includes three derived types: When appropriate, use these types instead of `System.ArgumentException`.]

Example

```
using System;

namespace Samples
{
  public class ArgumentExceptionSample
  {
    public static void Main()
    {
      try
      {
        string s = "one";
        s.CompareTo(1);
      }
      catch(ArgumentException e)
      {
        Console.WriteLine("Exception: {0}", e);
      }
    }
  }
}
```

The output is

```
Exception: System.ArgumentException: Object must be of type String.
   at System.String.CompareTo(Object value)
   at Samples.ArgumentExceptionSample.Main() in C:\Books\BCL\Samples\System\
ArgumentException\ArgumentException.cs:line 12
```

ArgumentException() Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor()
[C#]
public ArgumentException()
```

Summary

Constructs and initializes a new instance of the `System.ArgumentException` class.

Description

This constructor initializes the `System.ArgumentException.Message` property of the new instance to a system-supplied message that describes the error, such as "An invalid argument was specified." This message takes into account the current system culture.

The `System.ArgumentException.InnerException` property is initialized to `null`.

ArgumentException(System.String) Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor(string message)
[C#]
public ArgumentException(string message)
```

Summary

Constructs and initializes a new instance of the `System.ArgumentException` class.

Parameters

Parameter	Description
message	A `System.String` that describes the error. The content of *message* is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.

Description

This constructor initializes the `System.ArgumentException.Message` property of the new instance using *message*. If *message* is `null`, the `System.ArgumentException.Message` property is initialized to the system-supplied message provided by the constructor that takes no arguments. The `System.ArgumentException.InnerException` and `System.ArgumentException.ParamName` properties are initialized to `null`.

ArgumentException(System.String, System.String) Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor(string message, string
paramName)
[C#]
public ArgumentException(string message, string paramName)
```

Summary

Constructs and initializes a new instance of the `System.ArgumentException` class.

Parameters

Parameter	Description
message	A `System.String` that describes the error. The content of *message* is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.
paramName	A `System.String` that contains the name of the parameter that caused the exception.

Description

This constructor initializes the `System.ArgumentException.Message` property of the new instance using *message*, and the `System.ArgumentException.ParamName` property using *paramName*. If *message* is `null`, the `System.ArgumentException.Message` property is initialized to the system-supplied message provided by the constructor that takes no arguments. The `System.ArgumentException.InnerException` property is initialized to `null`.

ArgumentException(System.String, System.String, System.Exception) Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor(string message, string
paramName, class System.Exception innerException)
[C#]
public ArgumentException(string message, string paramName, Exception
innerException)
```

Summary

Constructs and initializes a new instance of the `System.ArgumentException` class.

Parameters

Parameter	Description
message	A `System.String` that describes the error. The content of *message* is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.
paramName	A `System.String` that contains the name of the parameter that caused the current exception.
innerException	An instance of `System.Exception` that is the cause of the current Exception. If *innerException* is non-`null`, then the current Exception was raised in a catch block handling *innerException*.

Description

This constructor initializes the `System.ArgumentException.Message` property of the new instance using *message*, the `System.ArgumentException.ParamName` property using *paramName*, and the `System.ArgumentException.InnerException` property using *innerException*. If *message* is `null`, the `System.ArgumentException.Message` property is initialized to the system-supplied message provided by the constructor that takes no arguments.

[*Note:* For information on inner exceptions, see `System.Exception.InnerException`.]

ArgumentException(System.String, System.Exception) Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor(string message, class
System.Exception innerException)
[C#]
public ArgumentException(string message, Exception innerException)
```

Summary

Constructs and initializes a new instance of the `System.ArgumentException` class.

Parameters

Parameter	Description
message	A `System.String` that describes the error. The content of *message* is intended to be understood by humans. The caller of this constructor is required to ensure that this string has been localized for the current system culture.
innerException	An instance of `System.Exception` that is the cause of the current Exception. If *innerException* is non-`null`, then the current Exception was raised in a catch block handling *innerException*.

Description

This constructor initializes the `System.ArgumentException.Message` property of the new instance using *message*, and the `System.ArgumentException.InnerException` property using *innerException*. If *message* is `null`, the `System.ArgumentException.Message` property is initialized to the system-supplied message provided by the constructor that takes no arguments. The `System.ArgumentException.ParamName` property is initialized to `null`.

[*Note:* For information on inner exceptions, see `System.Exception.InnerException`.]

ArgumentException(System.Runtime.Serialization.SerializationInfo, System.Runtime.Serialization.StreamingContext) Constructor

```
[ILASM]
family rtspecialname specialname instance void .ctor(class System.Runtime.
Serialization.SerializationInfo info, valuetype System.Runtime.Serialization.
StreamingContext context)
[C#]
protected ArgumentException(SerializationInfo info, StreamingContext context)
```

Summary

Initializes a new instance of the `System.ArgumentException` class with serialized data.

Parameters

Parameter	Description
info	The object that holds the serialized object data.
context	The contextual information about the source or destination.

Description

This constructor is called during deserialization to reconstitute the exception object transmitted over a stream.

ArgumentException.Message Property

```
[ILASM]
.property string Message { public hidebysig virtual specialname string
get_Message() }
[C#]
public override string Message { get; }
```

Summary

Gets the error message and the parameter name, or only the error message if no parameter name is set.

Property Value

A text string describing the details of the exception. The value of this property takes one of two forms:

Condition	Value
The *paramName* is a `null` reference or of zero length.	The *message* string passed to the constructor.
The *paramName* is not a `null` reference and it has a length greater than zero.	The *message* string appended with the name of the invalid parameter.

Description

This property overrides `System.Exception.Message`. The error message should be localized.

ArgumentException.ParamName Property

```
[ILASM]
.property string ParamName { public hidebysig virtual specialname string
get_ParamName() }
[C#]
public virtual string ParamName { get; }
```

Summary

Gets the name of the parameter that caused the current Exception.

Property Value

A `System.String` that contains the name of the parameter that caused the current Exception, or `null` if no value was specified to the constructor for the current instance.

Behaviors

The `System.ArgumentException.ParamName` property returns the same value as was passed into the constructor.

How and When to Override

Override the `System.ArgumentException.ParamName` property to customize the content or format of the parameter name.

ArgumentException.GetObjectData(System.Runtime.Serialization.SerializationInfo, System.Runtime.Serialization.StreamingContext) Method

```
[ILASM]
.method public hidebysig virtual void GetObjectData(class System.Runtime.
Serialization.SerializationInfo info, valuetype System.Runtime.Serialization.
StreamingContext context)
[C#]
public override void GetObjectData(SerializationInfo info, StreamingContext
context)
```

Summary

Sets the `System.Runtime.Serialization.SerializationInfo` object with the parameter name and additional exception information.

Parameters

Parameter	Description
info	The object that holds the serialized object data.
context	The contextual information about the source or destination.

Description

`System.ArgumentException.GetObjectData` sets a `System.Runtime.Serialization.SerializationInfo` with all the exception object data targeted for serialization. During deserialization, the exception object is reconstituted from the `System.Runtime.Serialization.SerializationInfo` transmitted over the stream.

For more information, see `System.Runtime.Serialization.SerializationInfo`.