.NET Framework
Standard Library
Annotated Reference
Volume 1

Base Class Library and Extended Numerics Library

Brad Abrams

Addison-Wesley
Contents

Foreword xiii
Preface xvii

Part I  Namespace Overviews  1
System  3
System.Collections  11
System.Diagnostics  14
System.Globalization  15
System.IO  17
System.Security  21
System.Text  25
System.Threading  27

Part II  Class Libraries  31
System.ApplicationException  33
System.ArgumentException  38
System.ArgumentNullException  47
System.ArgumentOutOfRangeException  53
System.ArithmeticException  61
System.Array  65
System.Collections.ArrayList  191
System.ArrayTypeMismatchException  283
System.Text.ASCIIEncoding  288
System.AsyncCallback Delegate  309
System.Attribute  311
PART I

Namespace Overviews
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

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

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttributeUsageAttribute</td>
<td>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.</td>
</tr>
<tr>
<td>CLSCompliantAttribute</td>
<td>Indicates whether a program element is compliant with the Common Language Specification (CLS).</td>
</tr>
<tr>
<td>FlagsAttribute</td>
<td>Indicates that an enumeration can be treated as a bit field; that is, a set of flags.</td>
</tr>
<tr>
<td>ObsoleteAttribute</td>
<td>Marks the program elements that are no longer in use.</td>
</tr>
</tbody>
</table>

TABLE 2
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>
<thead>
<tr>
<th>Interface</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>IAsyncResult</code></td>
<td>Represents the status of an asynchronous operation.</td>
</tr>
<tr>
<td><code>ICloneable</code></td>
<td>Supports cloning, which creates a new instance of a class with the same value as an existing instance.</td>
</tr>
<tr>
<td><code>IComparable</code></td>
<td>Defines a generalized comparison method that a value type or class implements to create a type-specific comparison method.</td>
</tr>
<tr>
<td><code>IDisposable</code></td>
<td>Defines a method to release allocated unmanaged resources.</td>
</tr>
<tr>
<td><code>IFormatProvider</code></td>
<td>Provides a mechanism for retrieving an object to control formatting.</td>
</tr>
<tr>
<td><code>IFormattable</code></td>
<td>Provides functionality to format the value of an object into a string representation.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Exception</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationException</td>
<td>A non-fatal application error occurred.</td>
</tr>
<tr>
<td>ArgumentException</td>
<td>One of the arguments provided to a method is not valid.</td>
</tr>
<tr>
<td>ArgumentNullException</td>
<td>A null reference is passed to a member that does not accept it as a valid argument.</td>
</tr>
<tr>
<td>ArgumentOutOfRangeException</td>
<td>The value of an argument is outside the allowable range of values as defined by the invoked member.</td>
</tr>
<tr>
<td>ArithmeticException</td>
<td>Error in an arithmetic, casting, or conversion operation.</td>
</tr>
<tr>
<td>ArrayTypeMismatchException</td>
<td>An attempt is made to store an element of the wrong type within an array.</td>
</tr>
<tr>
<td>DivideByZeroException</td>
<td>An attempt was made to divide an integral or decimal value by zero.</td>
</tr>
<tr>
<td>DuplicateWaitObjectException</td>
<td>An object appears more than once in an array of synchronization objects.</td>
</tr>
<tr>
<td>ExecutionEngineException</td>
<td>An internal error occurred in the execution engine of the common language runtime.</td>
</tr>
<tr>
<td>Exception</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FormatException</td>
<td>The format of an argument does not meet the parameter specifications of the invoked method.</td>
</tr>
<tr>
<td>IndexOutOfRangeException</td>
<td>An attempt is made to access an element of an array with an index that is outside the bounds of the array.</td>
</tr>
<tr>
<td>InvalidCastException</td>
<td>Invalid casting or explicit conversion.</td>
</tr>
<tr>
<td>InvalidOperationException</td>
<td>A method call is invalid for the object's current state.</td>
</tr>
<tr>
<td>InvalidProgramException</td>
<td>A program contains invalid Microsoft intermediate language (MSIL) or metadata. Generally this indicates a bug in a compiler.</td>
</tr>
<tr>
<td>NotFiniteNumberException</td>
<td>A floating-point value is positive infinity, negative infinity, or Not-a-Number (NaN).</td>
</tr>
<tr>
<td>NotSupportedException</td>
<td>An invoked method is not supported or not supported in the current mode of operation.</td>
</tr>
<tr>
<td>NullReferenceException</td>
<td>An attempt to dereference a null object reference.</td>
</tr>
<tr>
<td>ObjectDisposedException</td>
<td>An operation is performed on a disposed object.</td>
</tr>
<tr>
<td>OutOfMemoryException</td>
<td>There is not enough memory to continue the execution of a program.</td>
</tr>
<tr>
<td>OverflowException</td>
<td>An arithmetic, casting, or conversion operation in a checked context results in an overflow.</td>
</tr>
<tr>
<td>RankException</td>
<td>An array with the wrong number of dimensions is passed to a method.</td>
</tr>
<tr>
<td>StackOverflowException</td>
<td>The execution stack overflows by having too many pending method calls.</td>
</tr>
<tr>
<td>TypeInitializationException</td>
<td>A wrapper around the exception thrown by the type initializer.</td>
</tr>
<tr>
<td>UnauthorizedAccessException</td>
<td>The operating system denies access because of an I/O error or a specific type of security error.</td>
</tr>
</tbody>
</table>
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 IEnumerable 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>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
<th>ArrayList</th>
<th>HashTable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICollection</td>
<td>Defines size, enumerators and synchronization methods for all collections.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>IComparer</td>
<td>Exposes a method that compares two objects.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDictionary</td>
<td>Represents a collection of key-and-value pairs.</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>IDictionaryEnumerator</td>
<td>Enumerates the elements of a dictionary.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEnumerable</td>
<td>Exposes the enumerator, which supports a simple iteration over a collection.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>IEnumerator</td>
<td>Supports a simple iteration over a collection.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IHashCodeProvider</td>
<td>Supplies a hash code for an object, using a custom hash function.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IList</td>
<td>Represents a collection of objects that can be individually accessed by index.</td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>
PART II
Class Libraries
BCL

System

ApplicationException

Summary

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

Type Summary

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

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

**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 programming 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.
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 scenario for catching an ApplicationException type.

```csharp
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

```
  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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>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.</td>
</tr>
</tbody>
</table>
**ApplicationException() Constructor**

**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**

**Summary**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>message</code></td>
<td>A <code>System.String</code> that describes the error. The content of <code>message</code> 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.</td>
</tr>
<tr>
<td><code>innerException</code></td>
<td>An instance of <code>System.Exception</code> that is the cause of the current Exception. If <code>innerException</code> is non-null, then the current Exception was raised in a catch block handling <code>innerException</code>.</td>
</tr>
</tbody>
</table>

**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`.]

[ILASM]

[C#]
protected ApplicationException(SerializationInfo info, StreamingContext context)

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

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The object that holds the serialized object data.</td>
</tr>
<tr>
<td>context</td>
<td>The contextual information about the source or destination.</td>
</tr>
</tbody>
</table>

Description
This constructor is called during deserialization to reconstitute the exception object transmitted over a stream.
Summary
Represents the error that occurs when an argument passed to a method is invalid.

Type Summary

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

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

    // Methods
    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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>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.</td>
</tr>
</tbody>
</table>

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)
**ArgumentException() Constructor**

**Summary**

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

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>A <code>System.String</code> that describes the error. The content of <code>message</code> 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.</td>
</tr>
<tr>
<td>paramName</td>
<td>A <code>System.String</code> that contains the name of the parameter that caused the exception.</td>
</tr>
</tbody>
</table>

**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.
ArgumentException System

ArgumentException() Constructor

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>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.</td>
</tr>
<tr>
<td>paramName</td>
<td>A System.String that contains the name of the parameter that caused the current exception.</td>
</tr>
<tr>
<td>innerException</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>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.</td>
</tr>
<tr>
<td>innerException</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.]


[ILASM]
[C#]
protected ArgumentException(SerializationInfo info, StreamingContext context)

Summary

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

Message Property

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The object that holds the serialized object data.</td>
</tr>
<tr>
<td>context</td>
<td>The contextual information about the source or destination.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>The paramName is a null reference or of zero length.</td>
<td>The message string passed to the constructor.</td>
</tr>
<tr>
<td>The paramName is not a null reference and it has a length greater than zero.</td>
<td>The message string appended with the name of the invalid parameter.</td>
</tr>
</tbody>
</table>
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.


[ILASM]
[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**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info</td>
<td>The object that holds the serialized object data.</td>
</tr>
<tr>
<td>context</td>
<td>The contextual information about the source or destination.</td>
</tr>
</tbody>
</table>

**Description**


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