

Exception Class

Namespace: [System](#)

Assemblies: mscorlib.dll, System.Runtime.dll

Represents errors that occur during application execution.

In this article

[Definition](#)

[Examples](#)

[Remarks](#)

[Constructors](#)

[Properties](#)

[Methods](#)

[Events](#)

[Applies to](#)

[See also](#)

C#

 Copy

```
public class Exception : System.Runtime.Serialization.ISerializable
```

Inheritance [Object](#) → [Exception](#)

Derived [Microsoft.Build.BuildEngine.InternalLoggerException](#)
[Microsoft.Build.BuildEngine.InvalidProjectFileException](#)
[Microsoft.Build.BuildEngine.InvalidToolsetDefinitionException](#)
[Microsoft.Build.BuildEngine.RemoteErrorException](#)
[Microsoft.Build.Exceptions.BuildAbortedException](#)
[More...](#)

Implements [ISerializable](#)

Examples

The following example demonstrates a `catch` block that is defined to handle [ArithmeticException](#) errors. This `catch` block also catches [DivideByZeroException](#) errors,

because [DivideByZeroException](#) derives from [ArithmeticException](#) and there is no catch block explicitly defined for [DivideByZeroException](#) errors.

C#	 Copy	 Run
<pre>using System; class ExceptionTestClass { public static void Main() { int x = 0; try { int y = 100 / x; } catch (ArithmeticException e) { Console.WriteLine(\$"ArithmeticException Handler: {e}"); } catch (Exception e) { Console.WriteLine(\$"Generic Exception Handler: {e}"); } } } /* This code example produces the following results: ArithmeticException Handler: System.DivideByZeroException: Attempted to di- vide by zero. at ExceptionTestClass.Main() */</pre>		

Remarks

This class is the base class for all exceptions. When an error occurs, either the system or the currently executing application reports it by throwing an exception that contains information about the error. After an exception is thrown, it is handled by the application or by the default exception handler.

In this section:

[Errors and exceptions](#)

[Try/catch blocks](#)

[Exception type features](#)

[Exception class properties](#)

[Performance considerations](#)

[Re-throwing an exception](#)


[Choosing standard exceptions](#)

[Implementing custom exceptions](#)

Errors and exceptions

Run-time errors can occur for a variety of reasons. However, not all errors should be handled as exceptions in your code. Here are some categories of errors that can occur at run time and the appropriate ways to respond to them.

- **Usage errors.** A usage error represents an error in program logic that can result in an exception. However, the error should be addressed not through exception handling but by modifying the faulty code. For example, the override of the [Object.Equals\(Object\)](#) method in the following example assumes that the `obj` argument must always be non-null.



C#	 Copy
<pre>using System; public class Person { private string _name; public string Name { get { return _name; } set { _name = value; } } public override int GetHashCode() { return this.Name.GetHashCode(); } public override bool Equals(object obj) { // This implementation contains an error in program logic: // It assumes that the obj argument is not null. Person p = (Person) obj; return this.Name.Equals(p.Name); } } public class Example { public static void Main() { Person p1 = new Person(); p1.Name = "John"; Person p2 = null; } }</pre>	

```

        // The following throws a NullReferenceException.
        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}

```

The [NullReferenceException](#) exception that results when `obj` is `null` can be eliminated by modifying the source code to explicitly test for null before calling the [Object.Equals](#) override and then re-compiling. The following example contains the corrected source code that handles a `null` argument.

C#	 Copy	 Run
----	--	---

```

using System;

public class Person
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public override int GetHashCode()
    {
        return this.Name.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        // This implementation handles a null obj argument.
        Person p = obj as Person;
        if (p == null)
            return false;
        else
            return this.Name.Equals(p.Name);
    }
}

public class Example
{
    public static void Main()
    {
        Person p1 = new Person();
        p1.Name = "John";
        Person p2 = null;

        Console.WriteLine("p1 = p2: {0}", p1.Equals(p2));
    }
}

// The example displays the following output:
//      p1 = p2: False

```

Instead of using exception handling for usage errors, you can use the [Debug.Assert](#) method to identify usage errors in debug builds, and the [Trace.Assert](#) method to identify usage errors in both debug and release builds. For more information, see [Assertions in Managed Code](#).

- **Program errors.** A program error is a run-time error that cannot necessarily be avoided by writing bug-free code.

In some cases, a program error may reflect an expected or routine error condition. In this case, you may want to avoid using exception handling to deal with the program error and instead retry the operation. For example, if the user is expected to input a date in a particular format, you can parse the date string by calling the [DateTime.TryParseExact](#) method, which returns a [Boolean](#) value that indicates whether the parse operation succeeded, instead of using the [DateTime.ParseExact](#) method, which throws a [FormatException](#) exception if the date string cannot be converted to a [DateTime](#) value. Similarly, if a user tries to open a file that does not exist, you can first call the [File.Exists](#) method to check whether the file exists and, if it does not, prompt the user whether he or she wants to create it.

In other cases, a program error reflects an unexpected error condition that can be handled in your code. For example, even if you've checked to ensure that a file exists, it may be deleted before you can open it, or it may be corrupted. In that case, trying to open the file by instantiating a [StreamReader](#) object or calling the [Open](#) method may throw a [FileNotFoundException](#) exception. In these cases, you should use exception handling to recover from the error.

- **System failures.** A system failure is a run-time error that cannot be handled programmatically in a meaningful way. For example, any method can throw an [OutOfMemoryException](#) exception if the common language runtime is unable to allocate additional memory. Ordinarily, system failures are not handled by using exception handling. Instead, you may be able to use an event such as [AppDomain.UnhandledException](#) and call the [Environment.FailFast](#) method to log exception information and notify the user of the failure before the application terminates.

Try/catch blocks

The common language runtime provides an exception handling model that is based on the representation of exceptions as objects, and the separation of program code and exception handling code into `try` blocks and `catch` blocks. There can be one or more `catch` blocks, each designed to handle a particular type of exception, or one block designed to catch a more specific exception than another block.

If an application handles exceptions that occur during the execution of a block of application code, the code must be placed within a `try` statement and is called a `try` block. Application code that handles exceptions thrown by a `try` block is placed within a `catch` statement and is called a `catch` block. Zero or more `catch` blocks are associated with a `try` block, and each `catch` block includes a type filter that determines the types of exceptions it handles.

When an exception occurs in a `try` block, the system searches the associated `catch` blocks in the order they appear in application code, until it locates a `catch` block that handles the exception. A `catch` block handles an exception of type `T` if the type filter of the `catch` block specifies `T` or any type that `T` derives from. The system stops searching after it finds the first `catch` block that handles the exception. For this reason, in application code, a `catch` block that handles a type must be specified before a `catch` block that handles its base types, as demonstrated in the example that follows this section. A `catch` block that handles `System.Exception` is specified last.

If none of the `catch` blocks associated with the current `try` block handle the exception, and the current `try` block is nested within other `try` blocks in the current call, the `catch` blocks associated with the next enclosing `try` block are searched. If no `catch` block for the exception is found, the system searches previous nesting levels in the current call. If no `catch` block for the exception is found in the current call, the exception is passed up the call stack, and the previous stack frame is searched for a `catch` block that handles the exception. The search of the call stack continues until the exception is handled or until no more frames exist on the call stack. If the top of the call stack is reached without finding a `catch` block that handles the exception, the default exception handler handles it and the application terminates.

Exception type features

Exception types support the following features:

- Human-readable text that describes the error. When an exception occurs, the runtime makes a text message available to inform the user of the nature of the error and to suggest action to resolve the problem. This text message is held in the [Message](#) property of the exception object. During the creation of the exception object, you can pass a text string to the constructor to describe the details of that particular exception. If no error message argument is supplied to the constructor, the default error message is used. For more information, see the [Message](#) property.

- The state of the call stack when the exception was thrown. The [StackTrace](#) property carries a stack trace that can be used to determine where the error occurs in the code. The stack trace lists all the called methods and the line numbers in the source file where the calls are made.

Exception class properties

The [Exception](#) class includes a number of properties that help identify the code location, the type, the help file, and the reason for the exception: [StackTrace](#), [InnerException](#), [Message](#), [HelpLink](#), [HResult](#), [Source](#), [TargetSite](#), and [Data](#).

When a causal relationship exists between two or more exceptions, the [InnerException](#) property maintains this information. The outer exception is thrown in response to this inner exception. The code that handles the outer exception can use the information from the earlier inner exception to handle the error more appropriately. Supplementary information about the exception can be stored as a collection of key/value pairs in the [Data](#) property.

The error message string that is passed to the constructor during the creation of the exception object should be localized and can be supplied from a resource file by using the [ResourceManager](#) class. For more information about localized resources, see the [Creating Satellite Assemblies](#) and [Packaging and Deploying Resources](#) topics.

To provide the user with extensive information about why the exception occurred, the [HelpLink](#) property can hold a URL (or URN) to a help file.

The [Exception](#) class uses the HRESULT COR_E_EXCEPTION, which has the value 0x80131500.

For a list of initial property values for an instance of the [Exception](#) class, see the [Exception](#) constructors.

Performance considerations

Throwing or handling an exception consumes a significant amount of system resources and execution time. Throw exceptions only to handle truly extraordinary conditions, not to handle predictable events or flow control. For example, in some cases, such as when you're developing a class library, it's reasonable to throw an exception if a method argument is invalid, because you expect your method to be called with valid parameters. An invalid method argument, if it is not the result of a usage error, means that something extraordinary has occurred. Conversely, do not throw an exception if user input is invalid, because you can expect users to occasionally enter invalid data. Instead, provide a retry mechanism so users can enter valid input. Nor should you use

exceptions to handle usage errors. Instead, use [assertions](#) to identify and correct usage errors.

In addition, do not throw an exception when a return code is sufficient; do not convert a return code to an exception; and do not routinely catch an exception, ignore it, and then continue processing.

Re-throwing an exception

In many cases, an exception handler simply wants to pass the exception on to the caller. This most often occurs in:

- A class library that in turn wraps calls to methods in the .NET Framework class library or other class libraries.
- An application or library that encounters a fatal exception. The exception handler can log the exception and then re-throw the exception.

The recommended way to re-throw an exception is to simply use the [throw](#) statement in C# and the [Throw](#) statement in Visual Basic without including an expression. This ensures that all call stack information is preserved when the exception is propagated to the caller. The following example illustrates this. A string extension method, `FindOccurrences`, wraps one or more calls to [String.IndexOf\(String, Int32\)](#) without validating its arguments beforehand.

C#

 Copy

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

public static class Library
{
    public static int[] FindOccurrences(this String s, String f)
    {
        var indexes = new List<int>();
        int currentIndex = 0;
        try {
            while (currentIndex >= 0 && currentIndex < s.Length) {
                currentIndex = s.IndexOf(f, currentIndex);
                if (currentIndex >= 0) {
                    indexes.Add(currentIndex);
                    currentIndex++;
                }
            }
        }
        catch (ArgumentNullException e) {
            // Perform some action here, such as logging this exception.
        }
    }
}
```



```

        throw;
    }
    return indexes.ToArray();
}
}

```

A caller then calls `FindOccurrences` twice. In the second call to `FindOccurrences`, the caller passes a `null` as the search string, which causes the `String.IndexOf(String, Int32)` method to throw an `ArgumentNullException` exception. This exception is handled by the `FindOccurrences` method and passed back to the caller. Because the `throw` statement is used with no expression, the output from the example shows that the call stack is preserved.

C#

 Copy

```

public class Example
{
    public static void Main()
    {
        String s = "It was a cold day when...";
        int[] indexes = s.FindOccurrences("a");
        ShowOccurrences(s, "a", indexes);
        Console.WriteLine();

        String toFind = null;
        try {
            indexes = s.FindOccurrences(toFind);
            ShowOccurrences(s, toFind, indexes);
        }
        catch (ArgumentNullException e) {
            Console.WriteLine("An exception ({0}) occurred.",
                             e.GetType().Name);
            Console.WriteLine("Message:\n  {0}\n", e.Message);
            Console.WriteLine("Stack Trace:\n  {0}\n", e.StackTrace);
        }
    }

    private static void ShowOccurrences(String s, String toFind, int[] indexes)
    {
        Console.Write("' {0}' occurs at the following character positions: ",
                      toFind);
        for (int ctr = 0; ctr < indexes.Length; ctr++)
            Console.Write("{0}{1}", indexes[ctr],
                          ctr == indexes.Length - 1 ? "" : ", ");

        Console.WriteLine();
    }
}

// The example displays the following output:
//   'a' occurs at the following character positions: 4, 7, 15
//
//   An exception (ArgumentNullException) occurred.
//   Message:

```

```
//      Value cannot be null.
//      Parameter name: value
//
//      Stack Trace:
//          at System.String.IndexOf(String value, Int32 startIndex, Int32
count, Stri
//      ngComparison comparisonType)
//          at Library.FindOccurrences(String s, String f)
//          at Example.Main()
```

In contrast, if the exception is re-thrown by using the

C#

 Copy

```
throw e;
```

statement, the full call stack is not preserved, and the example would generate the following output:

Output

 Copy

```
'a' occurs at the following character positions: 4, 7, 15

An exception (ArgumentNullException) occurred.
Message:
    Value cannot be null.
Parameter name: value

Stack Trace:
    at Library.FindOccurrences(String s, String f)
    at Example.Main()
```

A slightly more cumbersome alternative is to throw a new exception, and to preserve the original exception's call stack information in an inner exception. The caller can then use the new exception's [InnerException](#) property to retrieve stack frame and other information about the original exception. In this case, the throw statement is:

C#

 Copy

```
throw new ArgumentNullException("You must supply a search string.",
                                e);
```

The user code that handles the exception has to know that the [InnerException](#) property contains information about the original exception, as the following exception handler illustrates.

C#

 Copy

```

try {
    indexes = s.FindOccurrences(toFind);
    ShowOccurrences(s, toFind, indexes);
}
catch (ArgumentNullException e) {
    Console.WriteLine("An exception ({0}) occurred.",
        e.GetType().Name);
    Console.WriteLine("    Message:\n{0}", e.Message);
    Console.WriteLine("    Stack Trace:\n    {0}", e.StackTrace);
    Exception ie = e.InnerException;
    if (ie != null) {
        Console.WriteLine("    The Inner Exception:");
        Console.WriteLine("        Exception Name: {0}", ie.GetType().Name);
        Console.WriteLine("        Message: {0}\n", ie.Message);
        Console.WriteLine("        Stack Trace:\n    {0}\n", ie.StackTrace);
    }
}
}

// The example displays the following output:
//     'a' occurs at the following character positions: 4, 7, 15
//
//     An exception (ArgumentNullException) occurred.
//         Message: You must supply a search string.
//
//         Stack Trace:
//             at Library.FindOccurrences(String s, String f)
//             at Example.Main()
//
//         The Inner Exception:
//             Exception Name: ArgumentNullException
//             Message: Value cannot be null.
//         Parameter name: value
//
//         Stack Trace:
//             at System.String.IndexOf(String value, Int32 startIndex, Int32
count, Stri
//         ngComparison comparisonType)
//             at Library.FindOccurrences(String s, String f)

```

Choosing standard exceptions

When you have to throw an exception, you can often use an existing exception type in the .NET Framework instead of implementing a custom exception. You should use a standard exception type under these two conditions:

- You are throwing an exception that is caused by a usage error (that is, by an error in program logic made by the developer who is calling your method). Typically, you would throw an exception such as [ArgumentException](#), [ArgumentNullException](#), [InvalidOperationException](#), or [NotSupportedException](#). The string you supply to the exception object's constructor when instantiating the exception object should describe the error so that the developer can fix it. For more information, see the [Message](#) property.

- You are handling an error that can be communicated to the caller with an existing .NET Framework exception. You should throw the most derived exception possible. For example, if a method requires an argument to be a valid member of an enumeration type, you should throw an [InvalidEnumArgumentException](#) (the most derived class) rather than an [ArgumentException](#).

The following table lists common exception types and the conditions under which you would throw them.

Exception	Condition
ArgumentException	A non-null argument that is passed to a method is invalid.
ArgumentNullException	An argument that is passed to a method is <code>null</code> .
ArgumentOutOfRangeException	An argument is outside the range of valid values.
DirectoryNotFoundException	Part of a directory path is not valid.
DivideByZeroException	The denominator in an integer or Decimal division operation is zero.
DriveNotFoundException	A drive is unavailable or does not exist.
FileNotFoundException	A file does not exist.
FormatException	A value is not in an appropriate format to be converted from a string by a conversion method such as <code>Parse</code> .
IndexOutOfRangeException	An index is outside the bounds of an array or collection.
InvalidOperationException	A method call is invalid in an object's current state.
KeyNotFoundException	The specified key for accessing a member in a collection cannot be found.
NotSupportedException	A method or operation is not implemented.
NotImplementedException	A method or operation is not supported.
ObjectDisposedException	An operation is performed on an object that has been disposed.

Exception	Condition
OverflowException	An arithmetic, casting, or conversion operation results in an overflow.
PathTooLongException	A path or file name exceeds the maximum system-defined length.
PlatformNotSupportedException	The operation is not supported on the current platform.
RankException	An array with the wrong number of dimensions is passed to a method.
TimeoutException	The time interval allotted to an operation has expired.
UriFormatException	An invalid Uniform Resource Identifier (URI) is used.

Implementing custom exceptions

In the following cases, using an existing .NET Framework exception to handle an error condition is not adequate:

- When the exception reflects a unique program error that cannot be mapped to an existing .NET Framework exception.
- When the exception requires handling that is different from the handling that is appropriate for an existing .NET Framework exception, or the exception must be disambiguated from a similar exception. For example, if you throw an [ArgumentOutOfRangeException](#) exception when parsing the numeric representation of a string that is out of range of the target integral type, you would not want to use the same exception for an error that results from the caller not supplying the appropriate constrained values when calling the method.

The [Exception](#) class is the base class of all exceptions in the .NET Framework. Many derived classes rely on the inherited behavior of the members of the [Exception](#) class; they do not override the members of [Exception](#), nor do they define any unique members.

To define your own exception class:

1. Define a class that inherits from [Exception](#). If necessary, define any unique members needed by your class to provide additional information about the exception. For example, the [ArgumentException](#) class includes a [ParamName](#) property that specifies the name of the parameter whose argument caused the

exception, and the [RegexMatchTimeoutException](#) property includes a [MatchTimeout](#) property that indicates the time-out interval.

2. If necessary, override any inherited members whose functionality you want to change or modify. Note that most existing derived classes of [Exception](#) do not override the behavior of inherited members.
3. Determine whether your custom exception object is serializable. Serialization enables you to save information about the exception and permits exception information to be shared by a server and a client proxy in a remoting context. To make the exception object serializable, mark it with the [SerializableAttribute](#) attribute.
4. Define the constructors of your exception class. Typically, exception classes have one or more of the following constructors:
 - [Exception\(\)](#), which uses default values to initialize the properties of a new exception object.
 - [Exception\(String\)](#), which initializes a new exception object with a specified error message.
 - [Exception\(String, Exception\)](#), which initializes a new exception object with a specified error message and inner exception.
 - [Exception\(SerializationInfo, StreamingContext\)](#), which is a protected constructor that initializes a new exception object from serialized data. You should implement this constructor if you've chosen to make your exception object serializable.

The following example illustrates the use of a custom exception class. It defines a `NotPrimeException` exception that is thrown when a client tries to retrieve a sequence of prime numbers by specifying a starting number that is not prime. The exception defines a new property, `NonPrime`, that returns the non-prime number that caused the exception. Besides implementing a protected parameterless constructor and a constructor with [SerializationInfo](#) and [StreamingContext](#) parameters for serialization, the `NotPrimeException` class defines three additional constructors to support the `NonPrime` property. Each constructor calls a base class constructor in addition to preserving the value of the non-prime number. The `NotPrimeException` class is also marked with the [SerializableAttribute](#) attribute.

C#

 Copy

```
using System;  
using System.Runtime.Serialization;
```

```

[Serializable()]
public class NotPrimeException : Exception
{
    private int notAPrime;

    protected NotPrimeException()
        : base()
    { }

    public NotPrimeException(int value) :
        base(String.Format("{0} is not a prime number.", value))
    {
        notAPrime = value;
    }

    public NotPrimeException(int value, string message)
        : base(message)
    {
        notAPrime = value;
    }

    public NotPrimeException(int value, string message, Exception innerException) :
        base(message, innerException)
    {
        notAPrime = value;
    }

    protected NotPrimeException(SerializationInfo info,
                                StreamingContext context)
        : base(info, context)
    { }

    public int NonPrime
    { get { return notAPrime; } }
}

```

The PrimeNumberGenerator class shown in the following example uses the Sieve of Eratosthenes to calculate the sequence of prime numbers from 2 to a limit specified by the client in the call to its class constructor. The GetPrimesFrom method returns all prime numbers that are greater than or equal to a specified lower limit, but throws a NotPrimeException if that lower limit is not a prime number.

C#

 Copy

```

using System;
using System.Collections.Generic;

[Serializable]
public class PrimeNumberGenerator
{
    private const int START = 2;
    private int maxUpperBound = 10000000;
}

```

```

private int upperBound;
private bool[] primeTable;
private List<int> primes = new List<int>();

public PrimeNumberGenerator(int upperBound)
{
    if (upperBound > maxUpperBound)
    {
        string message = String.Format(
            "{0} exceeds the maximum upper bound of {1}.",
            upperBound, maxUpperBound);
        throw new ArgumentOutOfRangeException(message);
    }
    this.upperBound = upperBound;
    // Create array and mark 0, 1 as not prime (True).
    primeTable = new bool[upperBound + 1];
    primeTable[0] = true;
    primeTable[1] = true;

    // Use Sieve of Eratosthenes to determine prime numbers.
    for (int ctr = START; ctr <= (int)Math.Ceiling(Math.Sqrt(upperBound));
        ctr++)
    {
        if (primeTable[ctr]) continue;

        for (int multiplier = ctr; multiplier <= upperBound / ctr;
multiplier++)
            if (ctr * multiplier <= upperBound) primeTable[ctr * multiplier]
= true;
    }
    // Populate array with prime number information.
    int index = START;
    while (index != -1)
    {
        index = Array.FindIndex(primeTable, index, (flag) => !flag);
        if (index >= 1)
        {
            primes.Add(index);
            index++;
        }
    }
}

public int[] GetAllPrimes()
{
    return primes.ToArray();
}

public int[] GetPrimesFrom(int prime)
{
    int start = primes.FindIndex((value) => value == prime);
    if (start < 0)
        throw new NotPrimeException(prime, String.Format("{0} is not a
prime number.", prime));
    else
        return primes.FindAll((value) => value >= prime).ToArray();
}
}

```


The following example makes two calls to the `GetPrimesFrom` method with non-prime numbers, one of which crosses application domain boundaries. In both cases, the exception is thrown and successfully handled in client code.

C#

 Copy

```
using System;
using System.Reflection;

class Example
{
    public static void Main()
    {
        int limit = 10000000;
        PrimeNumberGenerator primes = new PrimeNumberGenerator(limit);
        int start = 1000001;
        try
        {
            int[] values = primes.GetPrimesFrom(start);
            Console.WriteLine("There are {0} prime numbers from {1} to {2}",
                             start, limit);
        }
        catch (NotPrimeException e)
        {
            Console.WriteLine("{0} is not prime", e.NonPrime);
            Console.WriteLine(e);
            Console.WriteLine("-----");
        }

        AppDomain domain = AppDomain.CreateDomain("Domain2");
        PrimeNumberGenerator gen =
            (PrimeNumberGenerator)domain.CreateInstanceAndUnwrap(
                typeof(Example).Assembly.FullName,
                "PrimeNumberGenerator", true,
                BindingFlags.Default, null,
                new object[] { 1000000 }, null,
                null);
        try
        {
            start = 100;
            Console.WriteLine(gen.GetPrimesFrom(start));
        }
        catch (NotPrimeException e)
        {
            Console.WriteLine("{0} is not prime", e.NonPrime);
            Console.WriteLine(e);
            Console.WriteLine("-----");
        }
    }
}
```

Windows Runtime and .NET Framework 4.5.1

In .NET for Windows 8.x Store apps for Windows 8, some exception information is typically lost when an exception is propagated through non-.NET Framework stack frames. Starting with the .NET Framework 4.5.1 and Windows 8.1, the common language runtime continues to use the original [Exception](#) object that was thrown unless that exception was modified in a non-.NET Framework stack frame.

Constructors

Exception()	Initializes a new instance of the Exception class.
Exception(SerializationInfo, StreamingContext)	Initializes a new instance of the Exception class with serialized data.
Exception(String)	Initializes a new instance of the Exception class with a specified error message.
Exception(String, Exception)	Initializes a new instance of the Exception class with a specified error message and a reference to the inner exception that is the cause of this exception.

Properties

Data	Gets a collection of key/value pairs that provide additional user-defined information about the exception.
HelpLink	Gets or sets a link to the help file associated with this exception.
HResult	Gets or sets HRESULT, a coded numerical value that is assigned to a specific exception.
InnerException	Gets the Exception instance that caused the current exception.
Message	Gets a message that describes the current exception.
Source	Gets or sets the name of the application or the object that causes the error.
StackTrace	Gets a string representation of the immediate frames on the call stack.
TargetSite	Gets the method that throws the current exception.

Methods

Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetBaseException()	When overridden in a derived class, returns the Exception that is the root cause of one or more subsequent exceptions.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(Serialization Info, StreamingContext)	When overridden in a derived class, sets the SerializationInfo with information about the exception.
GetType()	Gets the runtime type of the current instance.
MemberwiseClone()	Creates a shallow copy of the current Object . (Inherited from Object)
ToString()	Creates and returns a string representation of the current exception.

Events

SerializeObjectState	Occurs when an exception is serialized to create an exception state object that contains serialized data about the exception.
--------------------------------------	---

Applies to

Product	Versions
.NET	5.0, 6.0 Preview 6
.NET Core	1.0, 1.1, 2.0, 2.1, 2.2, 3.0, 3.1
.NET Framework	1.1, 2.0, 3.0, 3.5, 4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 2.0, 2.1
UWP	10.0

Product	Versions
Xamarin.Android	7.1
Xamarin.iOS	10.8
Xamarin.Mac	3.0

See also

- [Handling and Throwing Exceptions](#)
- [Packaging and Deploying Resources in Desktop Apps](#)
- [Assertions in Managed Code](#)

Is this page helpful?

 Yes  No
