Handling and throwing exceptions in .NET

06/19/2018 • 2 minutes to read • +7

In this article

Applications must be able to handle errors that occur during execution in a consistent manner. .NET provides a model for notifying applications of errors in a uniform way: .NET operations indicate failure by throwing exceptions.

Exceptions

An exception is any error condition or unexpected behavior that is encountered by an executing program. Exceptions can be thrown because of a fault in your code or in code that you call (such as a shared library), unavailable operating system resources, unexpected conditions that the runtime encounters (such as code that can't be verified), and so on. Your application can recover from some of these conditions, but not from others. Although you can recover from most application exceptions, you can't recover from most runtime exceptions.

In .NET, an exception is an object that inherits from the System.Exception class. An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates.

Exceptions vs. traditional error-handling methods

Traditionally, a language's error-handling model relied on either the language's unique way of detecting errors and locating handlers for them, or on the error-handling mechanism provided by the operating system. The way .NET implements exception handling provides the following advantages:

- Exception throwing and handling works the same for .NET programming languages.
- Doesn't require any particular language syntax for handling exceptions, but allows each language to define its own syntax.
- Exceptions can be thrown across process and even machine boundaries.
- Exception-handling code can be added to an application to increase program reliability.

Exceptions offer advantages over other methods of error notification, such as return codes. Failures don't go unnoticed because if an exception is thrown and you don't handle it, the runtime terminates your application. Invalid values don't continue to propagate through the system as a result of code that fails to check for a failure return code.

Common exceptions

The following table lists some common exceptions with examples of what can cause them.

Exception type	Description	Example
Exception	Base class for all exceptions.	None (use a derived class of this exception).
IndexOutOfRangeException	Thrown by the runtime only when an array is indexed improperly.	Indexing an array outside its valid range: `arr[arr.Length+1]`
NullReferenceException	Thrown by the runtime only when a null object is referenced.	`object o = null;` `o.ToString();`
InvalidOperationException	Thrown by methods when in an invalid state.	Calling `Enumerator.MoveNext()` after removing an item from the underlying collection.
ArgumentException	Base class for all argument exceptions.	None (use a derived class of this exception).

Exception type	Description	Example

ArgumentNullException	Thrown by methods that do not allow an argument to be null.	`String s = null;`
		`"Calculate".IndexOf(s);`

ArgumentOutOfRangeException	Thrown by methods that verify that arguments are in a given range.	`String s = "string";`
		`s.Substring(s.Length+1);`

See also

- Exception Class and Properties
- How to: Use the Try-Catch Block to Catch Exceptions
- How to: Use Specific Exceptions in a Catch Block
- How to: Explicitly Throw Exceptions
- How to: Create User-Defined Exceptions
- Using User-Filtered Exception Handlers
- How to: Use Finally Blocks
- Handling COM Interop Exceptions
- Best Practices for Exceptions
- What Every Dev needs to Know About Exceptions in the Runtime

Static Classes and Static Class Members - C# Programming Guide

Static classes cannot be instantiated in C#. You access the members of a static class by using the class name itself.

Exceptions and Exception Handling

Learn about exceptions and exception handling. These C# features help deal with unexpected or exceptional situations that happen when a program is running.

Show more