

# Best practices for exceptions

12/05/2018 • 10 minutes to read •      +12

## In this article

[Use try/catch/finally blocks to recover from errors or release resources](#)

[Handle common conditions without throwing exceptions](#)

[Design classes so that exceptions can be avoided](#)

[Throw exceptions instead of returning an error code](#)

[Use the predefined .NET exception types](#)

[End exception class names with the word Exception](#)

[Include three constructors in custom exception classes](#)

[Ensure that exception data is available when code executes remotely](#)

[Use grammatically correct error messages](#)

[Include a localized string message in every exception](#)

[In custom exceptions, provide additional properties as needed](#)

[Place throw statements so that the stack trace will be helpful](#)

[Use exception builder methods](#)

[Restore state when methods don't complete due to exceptions](#)

[See also](#)

A well-designed app handles exceptions and errors to prevent app crashes. This section describes best practices for handling and creating exceptions.


## Use try/catch/finally blocks to recover from errors or release resources

Use `try/catch` blocks around code that can potentially generate an exception *and* your code can recover from that exception. In `catch` blocks, always order exceptions from the most derived to the least derived. All exceptions derive from [Exception](#). More derived exceptions are not handled by a `catch` clause that is preceded by a `catch` clause for a base exception class. When your code cannot recover from an exception, don't catch that exception. Enable methods further up the call stack to recover if possible.


Clean up resources allocated with either `using` statements, or `finally` blocks. Prefer `using` statements to automatically clean up resources when exceptions are thrown. Use `finally` blocks to clean up resources that don't implement [IDisposable](#). Code in a `finally` clause is almost always executed even when exceptions are thrown.

# Handle common conditions without throwing exceptions

For conditions that are likely to occur but might trigger an exception, consider handling them in a way that will avoid the exception. For example, if you try to close a connection that is already closed, you'll get an `InvalidOperationException`. You can avoid that by using an `if` statement to check the connection state before trying to close it.

C#	 Copy
<pre>if (conn.State != ConnectionState.Closed) {     conn.Close(); }</pre>	

If you don't check connection state before closing, you can catch the `InvalidOperationException` exception.


C#	 Copy
<pre>try {     conn.Close(); } catch (InvalidOperationException ex) {     Console.WriteLine(ex.GetType().FullName);     Console.WriteLine(ex.Message); }</pre>	

The method to choose depends on how often you expect the event to occur.

- Use exception handling if the event doesn't occur very often, that is, if the event is truly exceptional and indicates an error (such as an unexpected end-of-file). When you use exception handling, less code is executed in normal conditions.
- Check for error conditions in code if the event happens routinely and could be considered part of normal execution. When you check for common error conditions, less code is executed because you avoid exceptions.

## Design classes so that exceptions can be avoided

A class can provide methods or properties that enable you to avoid making a call that would trigger an exception. For example, a [FileStream](#) class provides methods that help determine whether the end of the file has been reached. These can be used to avoid the exception that is thrown if you read past the end of the file. The following example shows how to read to the end of a file without triggering an exception.

C#	 Copy
<pre>class FileRead {     public void ReadAll(FileStream fileToRead)     {         // This if statement is optional         // as it is very unlikely that         // the stream would ever be null.         if (fileToRead == null)         {             throw new ArgumentNullException();         }          int b;          // Set the stream position to the beginning of the file.         fileToRead.Seek(0, SeekOrigin.Begin);          // Read each byte to the end of the file.         for (int i = 0; i &lt; fileToRead.Length; i++)         {             b = fileToRead.ReadByte();             Console.Write(b.ToString());             // Or do something else with the byte.         }     } }</pre>	

Another way to avoid exceptions is to return null (or default) for extremely common error cases instead of throwing an exception. An extremely common error case can be considered normal flow of control. By returning null (or default) in these cases, you minimize the performance impact to an app.

For value types, whether to use `Nullable<T>` or default as your error indicator is something to consider for your particular app. By using `Nullable<Guid>`, default becomes `null` instead of `Guid.Empty`. Some times, adding `Nullable<T>` can make it clearer when a value is present or absent. Other times, adding `Nullable<T>` can create extra cases to check that aren't necessary, and only serve to create potential sources of errors.

# Throw exceptions instead of returning an error code

Exceptions ensure that failures do not go unnoticed because calling code didn't check a return code.

## Use the predefined .NET exception types

Introduce a new exception class only when a predefined one doesn't apply. For example:

- Throw an [InvalidOperationException](#) exception if a property set or method call is not appropriate given the object's current state.
- Throw an [ArgumentException](#) exception or one of the predefined classes that derive from [ArgumentException](#) if invalid parameters are passed.

## End exception class names with the word **Exception**

When a custom exception is necessary, name it appropriately and derive it from the [Exception](#) class. For example:

C#	 Copy
<pre>public class MyFileNotFoundException : Exception { }</pre>	

## Include three constructors in custom exception classes

Use at least the three common constructors when creating your own exception classes: the parameterless constructor, a constructor that takes a string message, and a constructor that takes a string message and an inner exception.

- [Exception\(\)](#), which uses default values.
- [Exception\(String\)](#), which accepts a string message.

- [Exception\(String, Exception\)](#), which accepts a string message and an inner exception.

For an example, see [How to: Create User-Defined Exceptions](#).

## Ensure that exception data is available when code executes remotely

When you create user-defined exceptions, ensure that the metadata for the exceptions is available to code that is executing remotely.

For example, on .NET implementations that support App Domains, exceptions may occur across App domains. Suppose App Domain A creates App Domain B, which executes code that throws an exception. For App Domain A to properly catch and handle the exception, it must be able to find the assembly that contains the exception thrown by App Domain B. If App Domain B throws an exception that is contained in an assembly under its application base, but not under App Domain A's application base, App Domain A will not be able to find the exception, and the common language runtime will throw a [FileNotFoundException](#) exception. To avoid this situation, you can deploy the assembly that contains the exception information in two ways:

- Put the assembly into a common application base shared by both app domains.
- or -
- If the domains do not share a common application base, sign the assembly that contains the exception information with a strong name and deploy the assembly into the global assembly cache.

## Use grammatically correct error messages

Write clear sentences and include ending punctuation. Each sentence in the string assigned to the [Exception.Message](#) property should end in a period. For example, "The log table has overflowed." would be an appropriate message string.

## Include a localized string message in every exception

The error message that the user sees is derived from the [Exception.Message](#) property of the exception that was thrown, and not from the name of the exception class. Typically,

you assign a value to the [Exception.Message](#) property by passing the message string to the `message` argument of an [Exception constructor](#).

For localized applications, you should provide a localized message string for every exception that your application can throw. You use resource files to provide localized error messages. For information on localizing applications and retrieving localized strings, see the following articles:

- [How to: create user-defined exceptions with localized exception messages](#)
- [Resources in Desktop Apps](#)
- [System.Resources.ResourceManager](#)

## In custom exceptions, provide additional properties as needed


Provide additional properties for an exception (in addition to the custom message string) only when there's a programmatic scenario where the additional information is useful. For example, the [FileNotFoundException](#) provides the [FileName](#) property.

## Place throw statements so that the stack trace will be helpful

The stack trace begins at the statement where the exception is thrown and ends at the catch statement that catches the exception.

## Use exception builder methods

It is common for a class to throw the same exception from different places in its implementation. To avoid excessive code, use helper methods that create the exception and return it. For example:

C#	 Copy
<pre>class FileReader {     private string fileName;      public FileReader(string path)     {         fileName = path;     }      public byte[] Read(int bytes)</pre>	

```

{
    byte[] results = FileUtils.ReadFile(fileName, bytes);
    if (results == null)
    {
        throw NewFileIOException();
    }
    return results;
}

FileReaderException NewFileIOException()
{
    string description = "My NewFileIOException Description";

    return new FileReaderException(description);
}
}

```

In some cases, it's more appropriate to use the exception's constructor to build the exception. An example is a global exception class such as [ArgumentException](#).

## Restore state when methods don't complete due to exceptions

Callers should be able to assume that there are no side effects when an exception is thrown from a method. For example, if you have code that transfers money by withdrawing from one account and depositing in another account, and an exception is thrown while executing the deposit, you don't want the withdrawal to remain in effect.

C#

 Copy

```

public void TransferFunds(Account from, Account to, decimal amount)
{
    from.Withdrawal(amount);
    // If the deposit fails, the withdrawal shouldn't remain in effect.
    to.Deposit(amount);
}

```

The method above does not directly throw any exceptions, but must be written defensively so that if the deposit operation fails, the withdrawal is reversed.

One way to handle this situation is to catch any exceptions thrown by the deposit transaction and roll back the withdrawal.

C#

 Copy

```

private static void TransferFunds(Account from, Account to, decimal amount)
{
    string withdrawalTrxID = from.Withdrawal(amount);
}

```

```
try
{
    to.Deposit(amount);
}
catch
{
    from.RollbackTransaction(withdrawalTrxID);
    throw;
}
```

This example illustrates the use of `throw` to re-throw the original exception, which can make it easier for callers to see the real cause of the problem without having to examine the [InnerException](#) property. An alternative is to throw a new exception and include the original exception as the inner exception:

C#

 Copy

```
catch (Exception ex)
{
    from.RollbackTransaction(withdrawalTrxID);
    throw new TransferFundsException("Withdrawal failed.", innerException:
ex)
    {
        From = from,
        To = to,
        Amount = amount
    };
}
```

## See also

- [Exceptions](#)

## Is this page helpful?

 Yes  No

## Recommended content

[How to create user-defined exceptions with localized exception messages](#)

Learn how to create user-defined exceptions with localized exception messages



## Static Constructors - C# Programming Guide

A static constructor in C# initializes static data or performs an action done only once. It runs before the first instance is created or static members are referenced.

## How to: Create User-Defined Exceptions

Learn how to create user-defined exceptions, which are an alternative to the hierarchy of exception classes derived from the Exception base class in .NET.

## Extension Methods - C# Programming Guide

Extension methods in C# enable you to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

Show more 