C# Exception Handling Best Practices

MARCH 4, 2020
https://stackify.com/csharp-exception-handling-best-practices

What is an Exception?

Exceptions are a type of error that occurs during the execution of an application. Errors are typically problems that are not expected. Whereas, **exceptions are expected to happen** within the application's code for various reasons.

Applications use exception handling logic to explicitly handle the exceptions when they happen. Exceptions can occur for a wide variety of reasons. From the infamous NullReferenceException to a database query timeout.

The Anatomy of C# Exceptions

Exceptions allow an application to transfer control from one part of the code to another. When an exception is thrown, the current flow of the code is interrupted and handed back to a parent try catch block. C# exception handling is done with the follow keywords: try, catch, finally, and throw

- **try** – A try block is used to encapsulate a region of code. If any code throws an exception within that try block, the exception will be handled by the corresponding catch.
- **catch** – When an exception occurs, the Catch block of code is executed. This is where you are able to handle the exception, log it, or ignore it.
- **finally** – The finally block allows you to execute certain code if an exception is thrown or not. For example, disposing of an object that must be disposed of.
- **throw** – The throw keyword is used to actually create a new exception that is the bubbled up to a try catch finally block.

Example #1: The Basic “try catch finally” Block

The C# try and catch keywords are used to define a try catch block. A try catch block is placed around code that could throw an exception. If an exception is thrown, this try catch block will handle the exception to ensure that the application does not cause an unhandled exception, user error, or crash the application.

Below is a simple example of a method that may throw an exception and how to properly use a try catch finally block for error handling.

```
System.Net.WebClient wc = null;
try
{
    wc = new System.Net.WebClient(); //downloading a web page
    var resultData = wc.DownloadString("http://google.com");
}
catch (ArgumentNullException ex)
{
    //code specifically for a ArgumentNullException
}
catch (System.Net.WebException ex)
{
    //code specifically for a WebException
}
catch (Exception ex)
{
    //code for any other type of exception
}
finally
{
    //call this if exception occurs or not
    //in this example, dispose the WebClient
    wc?.Dispose();
}
```

Your exception handling code can utilize **multiple C# catch statements for different types of exceptions**. This can be very useful depending on what your code is doing. In the previous example, ArgumentNullException occurs only when the website URL passed in is null. A WebException is caused by a wide array of issues. Catching specific types of exceptions can help tailor how to handle them.

Example #2: Exception Filters

One of the new features in C# 6 was exception filters. They allow you have even more control over your catch blocks and further tailor how you handle specific exceptions. This can help you fine-tune exactly how you handle exceptions and which ones you want to catch.

Before C# 6, you would have had to catch all types of a WebException and handle them. Now you could choose only to handle them in certain scenarios and let other scenarios bubble up to the code that called this method. Here is a modified example with filters:

```
System.Net.WebClient wc = null;
try
{
    wc = new System.Net.WebClient(); //downloading a web page
    var resultData = wc.DownloadString("http://google.com");
}
catch (System.Net.WebException ex) when (ex.Status ==
System.Net.WebExceptionStatus.ProtocolError)
{
    //code specifically for a WebException ProtocolError
}
catch (System.Net.WebException ex) when ((ex.Response as
System.Net.HttpWebResponse)?.StatusCode == System.Net.HttpStatusCode.NotFound)
{
    //code specifically for a WebException NotFound
}
catch (System.Net.WebException ex) when ((ex.Response as
System.Net.HttpWebResponse)?.StatusCode == System.Net.HttpStatusCode.InternalServerError)
{
    //code specifically for a WebException InternalServerError
}
finally
{
    //call this if exception occurs or not
    wc?.Dispose();
}
```

Common .NET Exceptions

Proper exception handling is critical to all application code. There are a lot of standard exceptions that are frequently used. The most common being the dreaded null reference exception. These are some of the common C# Exception types that you will see on a regular basis.

The follow is a list of common .NET exceptions:

- System.NullReferenceException – Very common exception related to calling a method on a null object reference
- System.IndexOutOfRangeException – Occurs attempting to access an array element that does not exist
- System.IO.IOException – Used around many file I/O type operations
- System.Net.WebException – Commonly thrown around any errors performing HTTP calls
- System.Data.SqlClient.SqlException – Various types of SQL Server exceptions
- System.StackOverflowException – If a method calls itself recursively, you may get this exception
- System.OutOfMemoryException – If your app runs out of memory
- System.InvalidCastException – If you try to cast an object to a type that it can't be cast to
- System.InvalidOperationException – Common generic exception in a lot of libraries
- System.ObjectDisposedException – Trying to use an object that has already been disposed

How to Create Your Own C# Custom Exception Types

C# exceptions are defined as classes, just like any other C# object. All exceptions inherit from a base System.Exception class. There are many common exceptions that you can use within your own code. Commonly, developers use the generic ApplicationException or Exception object to throw custom exceptions. You can also create your own type of exception.

Creating your own C# custom exceptions is really only helpful if you are going to catch that specific type of exception and **handle it differently**. They can also be helpful to track a very specific type of exception that you deem to extremely critical. By having a custom exception type, you can more easily **monitor your application errors and logs for it** with an error monitoring tool.

At Stackify, we have created a few custom exception types. One good example is a ClientBillingException. Billing is something we don't want to mess up, and if it does happen, we want to be very deliberate about how we handle those exceptions.

By using a custom exception type for it, we can write special code to handle that exception. We can also monitor our application for that specific type of exception and notify the on-call person when it happens.

Benefits of custom C# Exception types:

- Calling code can do custom handling of the custom exception type
- Ability to do custom monitoring around that custom exception type

Here is a simple example from our code:

```
public void DoBilling(int clientID)
{
    Client client = _clientDataAccessObject.GetById(clientID);

    if (client == null)
    {
        throw new ClientBillingException(string.Format("Unable to find a client by id
{0}", clientID));
    }
}

public class ClientBillingException : Exception
{
    public ClientBillingException(string message) : base(message) { }
}
```

How to Find Hidden .NET Exceptions

What Are First Chance Exceptions?

It is normal for a lot of exceptions to be thrown, caught, and then ignored. The internals of the .NET Framework even throws some exceptions that are discarded. One of the features of C# is something called first chance exceptions. It enables you to **get visibility into every single .NET Exception being thrown**.

It is very common for code like this below to be used within applications. This code can throw thousands of exceptions a minute and nobody would ever know it. This code is from another blog post about an app that had serious **performance problems due to bad exception handling**.

Exceptions will occur if the reader is null, columnName is null, columnName does not exist in the results, the value for the column was null, or if the value was not a proper DateTime. It is a minefield.

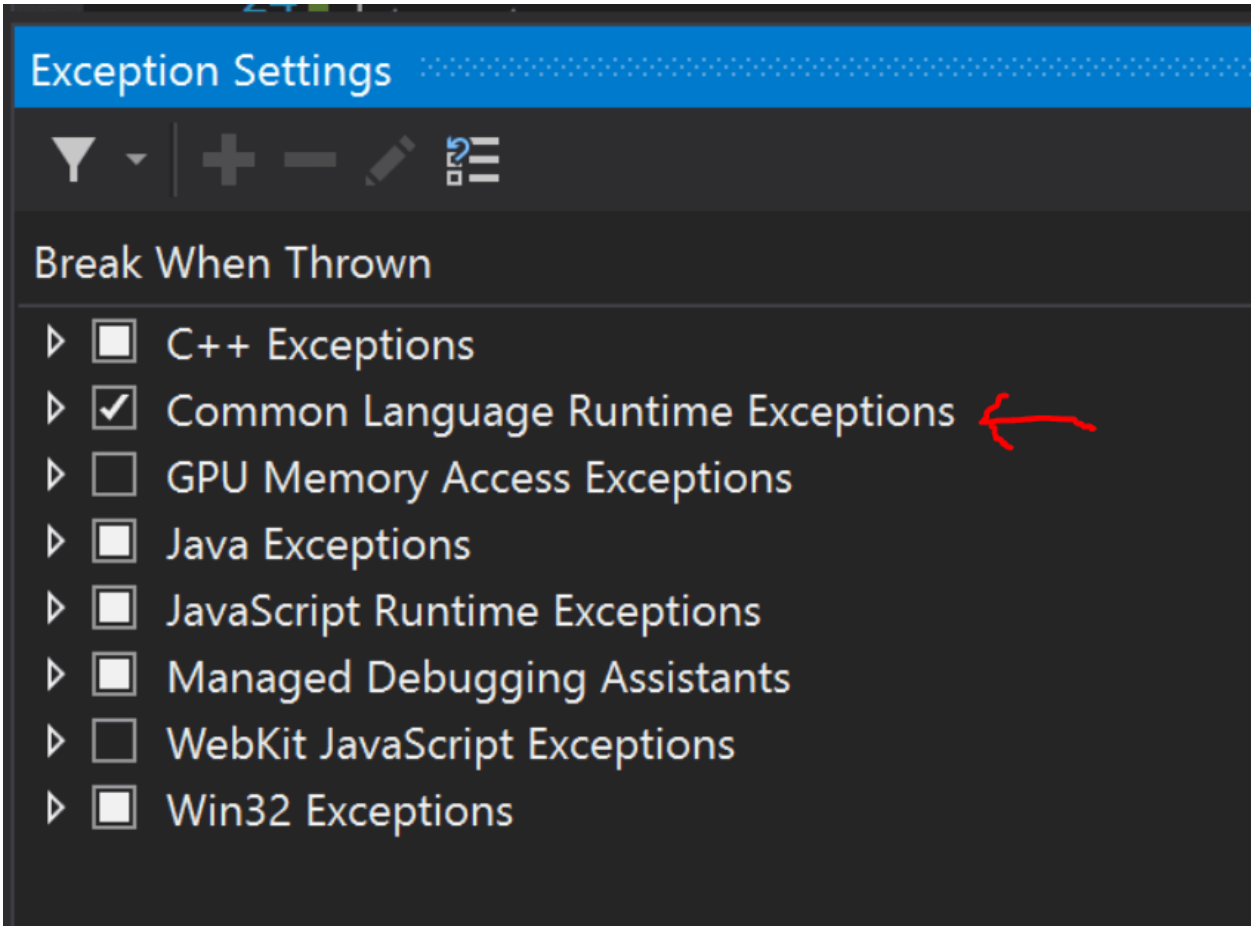
```
public DateTime? GetDate(Microsoft.Data.SqlClient.SqlDataReader reader, string
columnName)
{
    DateTime? value = null;
    try
    {
        value = DateTime.Parse(reader[columnName].ToString());
    }
    catch
    {
    }
    return value;
}
```

How to Enable First Chance Exceptions With Visual Studio

When you run your application within Visual Studio, with the debugger running, you can set Visual Studio to break anytime a C# Exception is thrown. This can help you find exceptions in your code that you did not know existed.

To access Exception Settings, go to Debug -> Windows -> Exception Settings

Under “Common Language Runtime Exceptions” you can select the types of exceptions you want the debugger to break for automatically. I would suggest just toggling the checkbox for all. Once you break on an exception, you can then tell it to ignore that particular type of exception to exclude it, if you would like.



How to Subscribe to First Chance Exceptions in Your Code

The .NET Framework provides a way to subscribe to an event to get a callback anytime an Exception occurs. You could use this to capture all of the exceptions. I would suggest potentially subscribing to them and just outputting them to your Debug window. This would give you some visibility to them without cluttering up your log files.

I would suggest potentially subscribing to them and just outputting them to your Debug window. This would give you some visibility to them without cluttering up your log files. You only want to do this once when your app first starts in the Main() method of a console app, or startup of an ASP.NET web app.

```
AppDomain.CurrentDomain.FirstChanceException += (sender, eventArgs) =>
{
    System.Diagnostics.Debug.WriteLine(eventArgs.Exception.ToString());
};
```

C# Exception Logging Best Practices

Proper exception handling is critical for any application. A key component to that is **logging the exceptions** to a logging library so that you can **record that the exceptions occurred**. Please check out our guide to C# Logging Best Practices to learn more on this subject.

We suggest logging your exceptions using NLog, Serilog, or log4net. All three frameworks give you the ability to log your exceptions to a file. They also allow you to send your logs to various other targets. These things like a database, Windows Event Viewer, email, or an error monitoring service.

Every exception in your app should be logged. They are critical to finding problems in your code!

It is also important to **log more contextual details** that can be useful for troubleshooting an exception. Things like what customer was it, key variables being used, etc.

```
try
{
    //do something
}
catch (Exception ex)
{
    //LOG IT!!!
    Log.Error(string.Format("Excellent description goes here about the exception.
Happened for client {0}", _clientContext.ClientId), ex);
    throw; //can rethrow the error to allow it to bubble up, or not, and ignore it.
}
```

To learn more about logging contextual data, read this blog post: What is structured logging and why developers need it

Why Logging Exceptions to a File Is Not Enough

Logging your exceptions to a file is a good best practice. However, this is not enough once your application is running in production. Unless you log into every one of your servers every day and review your log files, you won't know that the exceptions occurred. That file becomes a black hole.

An error monitoring service is a key tool for any development team. They allow you to collect all of your exceptions in a central location.

- Centralized exception logging
- View and search all exceptions across all servers and applications
- Uniquely identify exceptions
- Receive email alerts on new exceptions or high error rates