Exception Handling (C# Programming Guide)

Article • 03/14/2023

A try block is used by C# programmers to partition code that might be affected by an exception. Associated catch blocks are used to handle any resulting exceptions. A finally block contains code that is run whether or not an exception is thrown in the `try` block, such as releasing resources that are allocated in the `try` block. A `try` block requires one or more associated `catch` blocks, or a `finally` block, or both.

The following examples show a `try-catch` statement, a `try-finally` statement, and a `try-catch-finally` statement.

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

A `try` block without a `catch` or `finally` block causes a compiler error.

Catch Blocks

A `catch` block can specify the type of exception to catch. The type specification is called an *exception filter*. The exception type should be derived from Exception. In general, don't specify Exception as the exception filter unless either you know how to handle all exceptions that might be thrown in the `try` block, or you've included a throw statement at the end of your `catch` block.

Multiple `catch` blocks with different exception classes can be chained together. The `catch` blocks are evaluated from top to bottom in your code, but only one `catch` block is executed for each exception that is thrown. The first `catch` block that specifies the exact type or a base class of the thrown exception is executed. If no `catch` block specifies a matching exception class, a `catch` block that doesn't have any type is selected, if one is present in the statement. It's important to position `catch` blocks with the most specific (that is, the most derived) exception classes first.

Catch exceptions when the following conditions are true:

- You have a good understanding of why the exception might be thrown, and you can implement a specific recovery, such as prompting the user to enter a new file name when you catch a FileNotFoundException object.
- You can create and throw a new, more specific exception.

  C#

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index is out of range.", e);
    }
}
```

- You want to partially handle an exception before passing it on for more handling. In the following example, a `catch` block is used to add an entry to an error log before rethrowing the exception.

C#

```
try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

You can also specify *exception filters* to add a boolean expression to a catch clause. Exception filters indicate that a specific catch clause matches only when that condition is true. In the following example, both catch clauses use the same exception class, but an extra condition is checked to create a different error message:

C#

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
            throw new ArgumentOutOfRangeException(
                "Parameter index cannot be negative.", e);
        }
        catch (IndexOutOfRangeException e)
        {
            throw new ArgumentOutOfRangeException(
                "Parameter index cannot be greater than the array size.", e);
        }
    }
```

An exception filter that always returns `false` can be used to examine all exceptions but not process them. A typical use is to log exceptions:

```C#
public class ExceptionFilter
{
    public static void Main()
    {
        try
        {
            string? s = null;
            Console.WriteLine(s.Length);
        }
        catch (Exception e) when (LogException(e))
        {
        }
        Console.WriteLine("Exception must have been handled");
    }

    private static bool LogException(Exception e)
    {
        Console.WriteLine($"\tIn the log routine. Caught {e.GetType()}");
        Console.WriteLine($"\tMessage: {e.Message}");
        return false;
    }
}
```

The `LogException` method always returns `false`, no `catch` clause using this exception filter matches. The catch clause can be general, using System.Exception, and later clauses can process more specific exception classes.

Finally Blocks

A `finally` block enables you to clean up actions that are performed in a `try` block. If present, the `finally` block executes last, after the `try` block and any matched `catch` block. A `finally` block always runs, whether an exception is thrown or a `catch` block matching the exception type is found.

The `finally` block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects.

In the following example, the `finally` block is used to close a file that is opened in the `try` block. Notice that the state of the file handle is checked before the file is closed. If the `try` block can't open the file, the file handle still has the value `null` and the `finally` block doesn't try to close it. Instead, if the file is opened successfully in the `try` block, the `finally` block closes the open file.

C#

```csharp
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

C# Language Specification

For more information, see Exceptions and The try statement in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# reference
- Exception-handling statements
- using statement