# Exception and Error Handling

.NET

*The C# language's **exception handling** features help you deal with any unexpected or exceptional situations that occur during runtime. **Exception handling** uses the **try**, **catch**, and **finally** keywords to try actions that may not succeed, to handle failures when you decide that it is reasonable to do so, and to clean up resources afterward.*

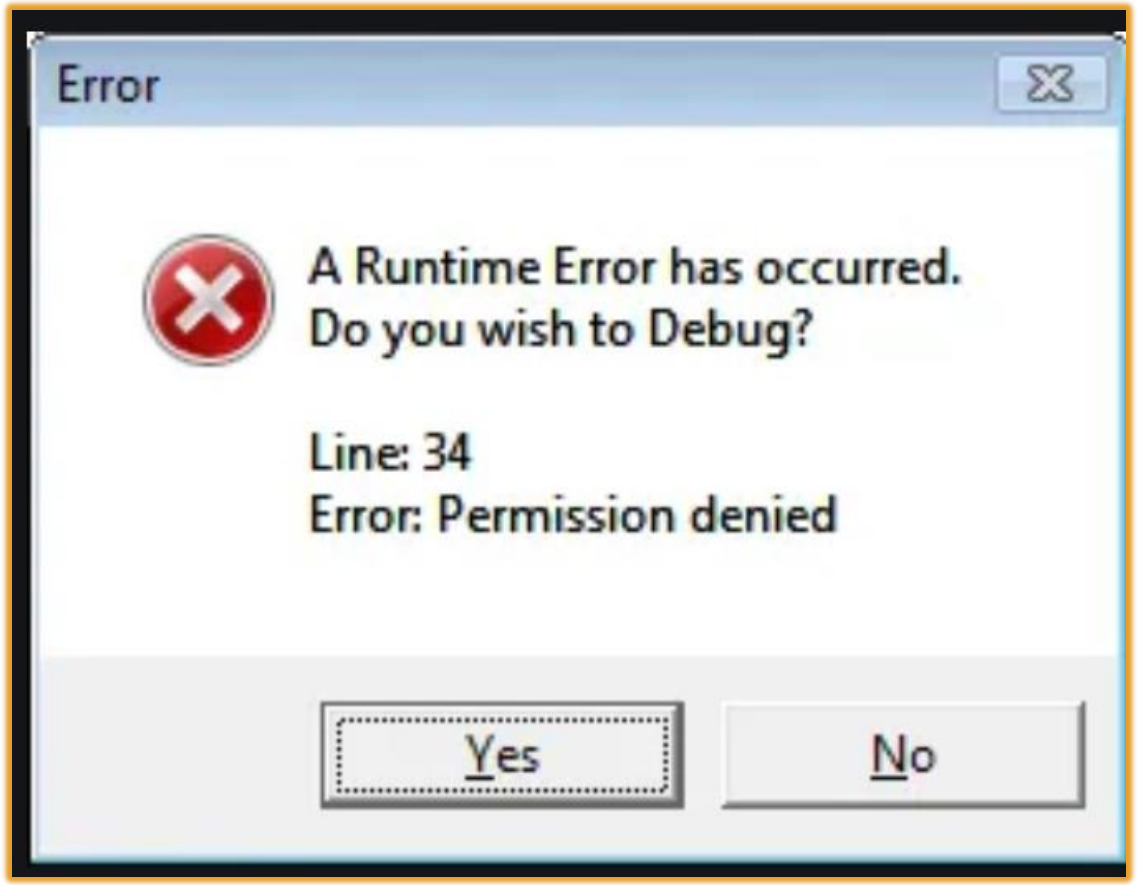
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/exceptions/exception-handling

# Exceptions and Errors – The Difference

https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=netframework-4.8#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. There are three main types of run-time errors:

- Usage Errors
  - An error in program logic that should be addressed not through exception handling but by modifying the faulty code.
- Program Errors
  - a run-time error that cannot necessarily be avoided by writing bug-free code.
- System Failures
  - a run-time error that cannot be handled programmatically in a meaningful way.



# Errors – Usage Errors

https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=netframework-4.8#errors-and-exceptions

***Usage errors*** occur due to faulty program logic and should be addressed though correction of the code rather than in handling an exception when it's thrown.

The ***override*** of the ***Object.Equals(Object)*** method in the following example assumes that the obj argument must always be non-null.

```
public override bool Equals(object obj)
{
   // This implementation contains
   // It assumes that the obj argum
   Person p = (Person) obj;
   return this.Name.Equals(p.Name);
}
```

```
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);
}
```

# Errors – Program Errors and System failures

https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=netframework-4.8#errors-and-exceptions

Program Errors –

May reflect a routine error condition. Avoid using exception handling to deal with program errors. Instead prevent the exception by trying the action first.

**USE** => ***DateTime.TryParseExact*** (returns a Boolean)

**DO NOT USE** => ***DateTime.ParseExact*** (throws a ***FormatException*** exception)

System failures –

Should not be handled by using exception handling. Any method can throw an ***OutOfMemoryException*** exception if the CLR is unable to allocate additional memory. You may be able to use an ***event*** such as ***AppDomain.UnhandledException*** and call the ***Environment.FailFast*** method notify the user before the application terminates.

# Exception Class

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/exceptions/

---

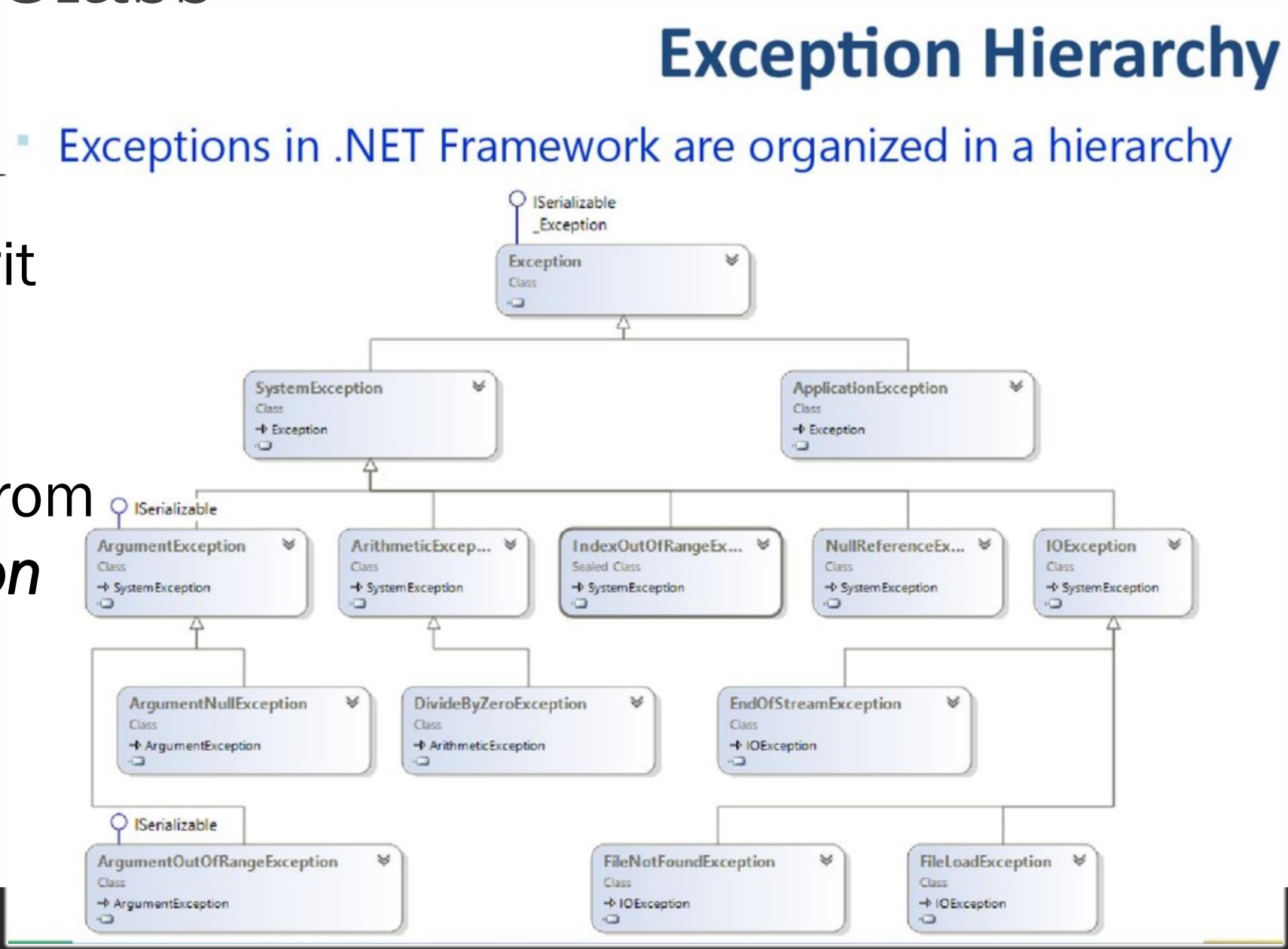
The ***Exception Class*** is the base class for all ***exceptions***. When any error occurs, the system or the application throws an ***exception*** that contains information about the error.

When an ***exception*** is thrown by a method far down the call stack, the ***CLR*** will unwind the stack, looking for a method with a ***catch*** block for that specific ***exception*** type and execute the first such ***catch*** block that it finds. If it finds no ***catch*** block in the call stack, it terminates the process and display a message to the user.

# Exception Class – Hierarchy

https://en.ppt-online.org/89884

All ***exceptions*** inherit from the ***Exception*** Class. All run-time exceptions inherit from the ***SystemException*** Class.



# Exceptions – Example

https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=netframework-4.8#examples

This catch block will handle ***ArithmeticException*** errors. The catch block also catches ***DivideByZeroException*** errors, because ***DivideByZeroException*** derives from ***ArithmeticException***. Without the exception handling, this program would terminate with a '***DivideByZeroException*** was unhandled' error.

```
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 divide by zero.
   at ExceptionTestClass.Main()

*/
```

# Exceptions – Try/Catch Block

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch

- Generally, when an ***exception*** is thrown, the CLR unwinds the stack looking for the appropriate ***catch*** statement. If no ***catch*** block is found, then the CLR displays an ***unhandled exception*** message to the user and stops execution of the program.
- The ***try/catch*** statement consists of a ***try*** block followed by one or more ***catch*** clauses, which specify handlers for different exceptions. The ***try*** block contains the guarded code that may cause the ***exception***. The block is executed until an ***exception*** is thrown or it is completed successfully.
- Using multiple ***catch*** arguments is a way to filter for the exceptions you want to handle.

```
try
{
    ProcessString(s);
}
catch (Exception e)
{
    Console.WriteLine("{0} Exception caught.", e);
}
```

# Exceptions - Try/Catch/Finally

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch-finally

Usage of ***try/catch/finally*** block is to

- obtain and use resources in a ***try*** block,
- deal with exceptional circumstances in a ***catch*** block, and
- release the resources in the ***finally*** block.

The finally block always runs.

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }

        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }
}
```

# Exceptions - Throw

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/throw
https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch

A ***throw*** statement can be used in a ***catch*** block to ***re-throw*** the ***exception*** that is caught by the ***catch*** statement. The following example extracts source information from an ***IOException exception***, and then throws the ***exception*** to the parent method.

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

You can ***catch*** one ***exception*** and ***throw*** a different ***exception***. When you do this, specify the ***exception*** that you caught as the inner ***exception***, as shown in the following example.

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

# Exceptions – Throw and Unwind The Stack

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/throw

```
using System;

public class NumberGenerator
{
   int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

   public int GetNumber(int index)
   {
      if (index < 0 || index >= numbers.Length) {
         throw new IndexOutOfRangeException();
      }
      return numbers[index];
   }
}
```

```
using System;

public class Example
{
   public static void Main()
   {
      var gen = new NumberGenerator();
      int index = 10;
      try {
          int value = gen.GetNumber(index);
          Console.WriteLine($"Retrieved {value}");
      }
      catch (IndexOutOfRangeException e)
      {
         Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
      }
   }
}
// The example displays the following output:
//       IndexOutOfRangeException: 10 is outside the bounds of the array
```

# User-Defined Exceptions

https://docs.microsoft.com/en-us/dotnet/standard/exceptions/how-to-create-user-defined-exceptions

If none of the predefined exceptions from the exception hierarchy meets your needs, you can

- create your own exception classes
- Derive(inherit) from the ***Exception*** class.
- End the class name with the word "Exception“.
- Implement the three common constructors(example).

The example defines a new exception class named ***EmployeeListNotFoundException***. The class is derived from ***Exception*** and includes three constructors.

```
using System;

public class EmployeeListNotFoundException : Exception
{
    public EmployeeListNotFoundException()
    {
    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {
    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```

# User-Defined Exceptions

https://dotnettutorials.net/lesson/create-custom-exception-csharp/#:~:text=

1. Create a new class inheriting from ***Exception***.
2. Override Exception's ***virtual property*** called message with your chosen error message.

```
namespace ExceptionHandlingDemo
{
    //Creating our own Exception Class by inheriting
    public class OddNumberException : Exception
    {
        //Overriding the Message property
        public override string Message
        {
            get
            {
                return "divisor cannot be odd number";
            }
        }
    }
```

```
class Program
{
    static void Main(string[] args)
    {
        int x, y, z;
        Console.WriteLine("ENTER TWO INTEGER NUMBERS:");
        x = int.Parse(Console.ReadLine());
        y = int.Parse(Console.ReadLine());
        try
        {
            if (y % 2 > 0)
            {
                //OddNumberException ONE = new OddNumberException();
                //throw ONE;
                throw new OddNumberException();
            }
            z = x / y;
            Console.WriteLine(z);
        }
        catch (OddNumberException one)
        {
            Console.WriteLine(one.Message);
        }

        Console.WriteLine("End of the program");
        Console.ReadKey();
    }
}
```