Exception Handling

Objectives

- Errors
- What is an Exception?
- Exception Hierarchy
  - Important Methods and Properties of Exception class
- Exception Handling Constructs
- Using try and catch Blocks
- Multiple catch Blocks
- An important note about multiple catch block
- The Finally Clause
- Why Use Exception?
- Traditional Approach vs Structured Exception Handling
- Custom Exception Class
- How to Create Custom Exception Class?
- How to Throw Custom Exception?
- How to Catch Custom Exception?
- The Throw Statement
- Point of Discussion

TIME FOR CASE STUDY

Case Study - Continued

- As a **Customer** of Shopon, I should be get custom error message when customer is trying to register his/her self with duplicate email id.

Thought

- Customer should be allowed to register in the system. System should allow customer to register his/her filling registration form using email id as login id, password and confirm password as fields.

Knowledge Byte

Error

Exception

Errors

Errors are part of any application, that we make knowingly or unknowingly. There are many types of errors that occur in our program, such as

- Compile-time error:
  - **Syntax errors:** Design-time errors. Occurs due to incorrect syntaxes. Cannot be compiled and run. Such as, forgetting to place semi-colon (;) at the end of line.
- Runtime-error:
  - **Logical errors:** Occurs during run-time. Difficult to track down. Occurs when desired output is not obtained. Such as, trying to calculate salary of an employee by adding different salary structure parameters, but forgot to include one of them and getting unexpected less amount rather than expected result.
  - **System errors:** Occurs when the program is compiled and run. Occurs because code is syntactically correct but cannot be executed due to some unexpected state of the computer. Such as, trying to access an element from an index position of an array where that index does not exist in that array.

What is an Exception?

Exception is not an error. Rather, Exception is an abnormal condition that arises due to system error while executing a program

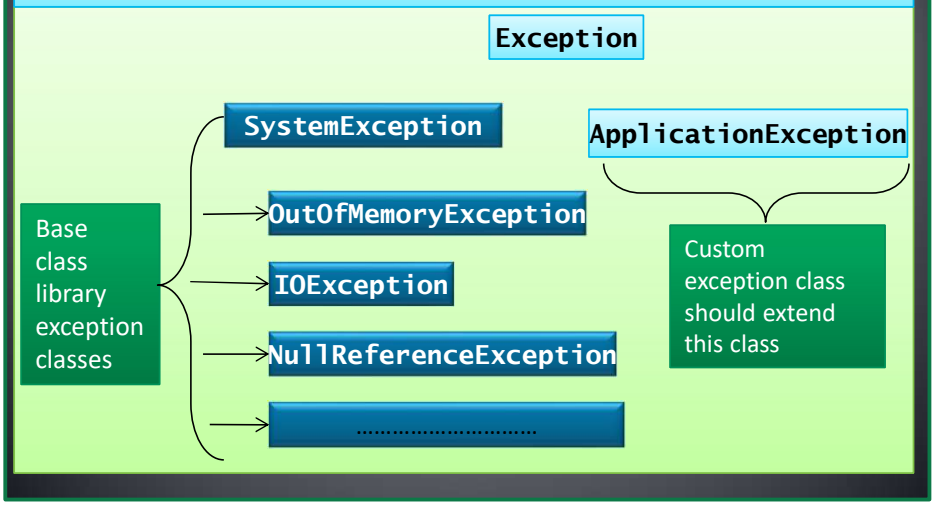
- In .NET an exception is an object that describes an exceptional condition (run-time system error) that has occurred when executing a program.
- Effective exception handling will make your programs more robust and easier to debug. They help answer these three questions:
  - What went wrong?
    - Answered by the type of exception thrown.
  - Where did it go wrong?
    - Answered by exception stack trace.
  - Why did it go wrong?
    - Answered by exception message

How is an Exception handled?

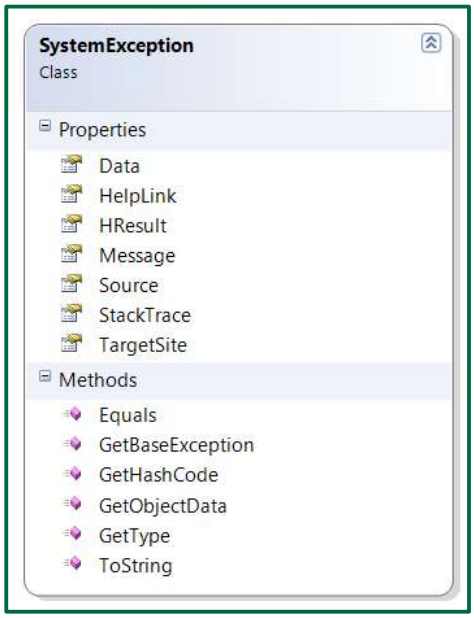
- Exception happens due to system's failure to execute code because of an abnormal condition
  - For example, when you write a code to divide zero or some other value by zero and then try to execute, system will not be able to perform that job
- CLR then creates an object to represent the unexpected error and then throws it to the method which caused it
  - That method may choose to handle the exception itself or pass it on
- Either way, at some point, the exception is caught and processed
- Sources for exceptions could be
  - Generated by CLR
  - Manually generated by programmer's code.

Exception Hierarchy

The base class for all the exception related classes is Exception. Two child classes that inherit from it are SystemException and ApplicationException class.



System.SystemException class



- Defines the base class for predefined exceptions in the System namespace.
- This class is provided as a means to differentiate between exceptions defined by the system versus exceptions defined by applications

Important Methods and Properties of Exception class

- Message property:
  - This property gets a message that describes the current exception. It returns the error message that explains the reason for the exception, or an empty string("").
- Source Property:
  - This property gets or sets the name of the application or the object that causes the error. It returns the name of the application or the object that causes the error.
- TargetSite Property:
  - Gets the method that throws the current exception. It returns the instance of MethodBase class, present in System.Reflection namespace, representing information of the method that threw the current exception
- StackTrace Property:
  - it gets a string representation of the frames on the call stack at the time the current exception was thrown. It returns a string that describes the contents of the call stack, with the most recent method call appearing first.

Exception Handling Constructs

Four constructs are used in exception handling

try	a block surrounding program statements to monitor for exceptions
catch	together with try, catches specific kinds of exceptions and handles them in some way
finally	specifies any code that absolutely must be executed whether or not an exception occurs
throw	used to throw a specific exception from the program

```
try {
    /*
     * some codes to test here
     */
} catch (SqlException sx) {
    /*
     * handle Exception1 here
     */
} catch (FileNotFoundException fx) {
    /*
     * handle Exception2 here
     */
} catch (Exception ex) {
    /*
     * handle Exception3 here
     */
} finally {
    /*
     * always execute codes here
     */
}
```

`try` block encloses the context where a possible exception can be thrown

each `catch()` block is an exception handler and can appear several times

An optional `finally` block is always executed before exiting the `try` statement.

Using try and catch Blocks

- Object-oriented solution to error handling
  - Put the normal code in a **try** block
  - Handle the exceptions in a separate **catch** block

```
try
{
        int x=0; int y=0; int z;
        z = x/y;
}
catch (DivideByzeroException caught)
{
        Console.WriteLine(caught);
}
```

Program logic

Error handling

Multiple catch Blocks

- Each catch block catches one class of exception
- A try block can have one general catch block

```
try
{
        Console.WriteLine("Enter first number");
        int i = int.Parse(Console.ReadLine());
        Console.WriteLine("Enter second number");
        int j = int.Parse(Console.ReadLine());
        int k = i / j;
}
catch (OverflowException caught) {…}
catch (DivideByZeroException caught) {…}
```

An important note about multiple catch block

- A catch block which catches all exceptions (catch block accepting Exception class object) should be placed as the last one if you are using multiple catch blocks.

```
try
{
    Console.WriteLine("Enter first number");
    int i = int.Parse(Console.ReadLine());
    Console.WriteLine("Enter second number");
    int j = int.Parse(Console.ReadLine());
    int k = i / j;

}
catch (OverflowException caught) {...}
catch (DivideByZeroException caught) {...}
catch (Exception caught) {...}
```

Correct approach

```
try
{
    Console.WriteLine("Enter first number");
    int i = int.Parse(Console.ReadLine());
    Console.WriteLine("Enter second number");
    int j = int.Parse(Console.ReadLine());
    int k = i / j;

}
catch (Exception caught) {...}
catch (DivideByZeroException caught) {...}
catch (OverFlowException caught) {...}
```

Wrong approach

The Finally Clause

- All of the statements in a finally block are always executed

```
Monitor.Enter(x);
try
{
   ...
}
catch(Exception ex)
{
//code
}
finally
{
   Monitor.Exit(x);
}
```

Any catch blocks are optional. Try can be followed by either catch or finally

Finally blocks are mainly used to clean up resources, such as if you have opened database connection or file connection in try block then close them in finally block, because due to some exception if the following codes are not executed, they are bound to get executed at least in finally block

Why Use Exception?

Exception Handling: Traditional approach

- Method returns error code.
  - Problem: Forget to check for error code
    - Failure notification may go undetected
- Problem: Calling method may not be able to do anything about failure
  - Program must fail too and let its caller worry about it
  - Many method calls would need to be checked
- Instead of programming for success object.doSomething()
  you would always be programming for failure:

```
if (!object.doSomething())
    return false;
```

Traditional Approach vs Structured Exception Handling

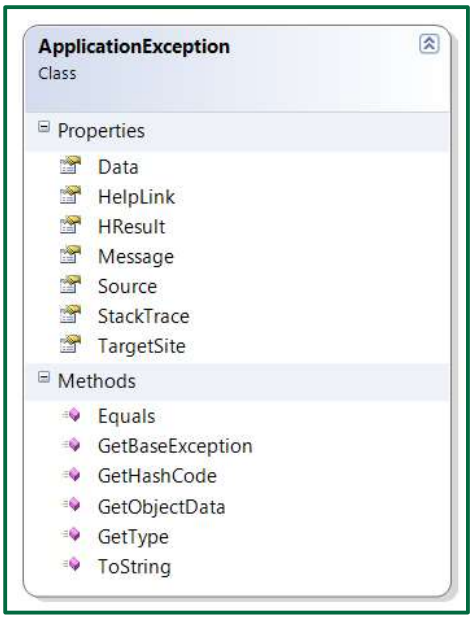
- Traditional procedural error handling is cumbersome. Actual code is not separate from exception code.

```
namespace ExceptionHandlingDemo
{
   class Program
   {
      static void Main(string[] args)
      {
            int errorCode = 0;
            //programming logic
            FileInfo source = new FileInfo("code.cs");
           //error detection
            if (errorCode == -1)
                goto Failed;
            //programming logic
            int length = (int)source.Length;
            //error detection
            if (errorCode == -2)
                goto Failed;
            //programming logic
            char[] contents = new char[length];
            //error detection
            if (errorCode == -3)
                goto Failed;
           //handling error
            Failed:
                Console.WriteLine("failure..");
      }
   }
}
```

- Structured Exception Handling makes it easy to separate exception code from actual code

```
namespace ExceptionHandlingDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         try
         {
            FileInfo source = new FileInfo("code.cs");
            int length = (int)source.Length;
            char[] contents = new char[length];
         }
         catch (IOException ex)
         {
            Console.WriteLine(ex.Message);
         }
      }
   }
}
```

System.ApplicationException class



- The exception that is thrown when a non-fatal application error occurs.
- User applications, not the common language runtime, throw custom exceptions derived from the ApplicationException class.
- The ApplicationException class differentiates between exceptions defined by applications versus exceptions defined by the system.

Custom Exception Class

- Exception class can be created by user.
- It is needed whenever you need to tackle a situation for which there is no system exception available
  - Such as, you are writing an application through which an user is entering his/her age while filling up details for online insurance policy application form and you want an exception to be thrown when the user enters age which is less than permitted minimum age for the policy. But, there is no system exception available for this purpose.
  - In this situation you need to create a custom exception class, such as AgeLessThanFiveException

How to Create Custom Exception Class?

- Create a custom exception class by extending from ApplicationException class, which inherits from base class Exception.
- Provide user-defined (overloaded constructors) which will accept error message as string data type and pass to base class using base keyword

```
namespace CustomExceptionHandlingDemo
{
    class AgeLessThanFiveException:ApplicationException
    {
        public AgeLessThanFiveException()
        {

        }

        public AgeLessThanFiveException(string errorMessage)
            : base(errorMessage)
        {

        }
    }
}
```

How to Throw Custom Exception?

User has to throw custom exception, since runtime is unaware about custom exception class

- Create an object of custom exception class wherever necessary.
- Use 'throw' keyword to throw the exception object

```
namespace CustomExceptionHandlingDemo
{
   class Applicant
   {
      //other fields

      private int age;

      public int Age
      {
         get { return age; }
         set
         {
            if (value <= 5)
               throw AgeLessThanFiveException("Error: Age is less than 5. Applicant whose age is more than 5 can apply for the insurance.");
            else
               age = value;
         }
      }
   }
}
```

How to Create Custom Exception Class?

- Create a custom exception class by extending from ApplicationException class, which inherits from base class Exception.
- Override virtual, read-only 'Message' property from base class and return custom message from that property

```
namespace CustomExceptionHandlingDemo
{
   class AgeLessThanFiveException:ApplicationException
   {
      public AgeLessThanFiveException()
      {

      }

      public override string Message
      {
         get
         {
            return "Error: Age is less than 5. Applicant whose age is more than 5 can apply
for the insurance.";
         }
      }
   }
}
```

How to Throw Custom Exception?

- User has to throw custom exception, since runtime is unaware about custom exception class

- Create an object of custom exception class where ever necessary.
- Use 'throw' keyword to throw the exception object

```
namespace CustomExceptionHandlingDemo
{
   class Applicant
   {
      //other fields

      private int age;

      public int Age
      {
         get { return age; }
         set
         {
            if (value <= 5)
               throw AgeLessThanFiveException();
            else
               age = value;
         }
      }
   }
}
```

How to Catch Custom Exception?

- Catching custom exception is in no way different from catching any system exception

- Put the suspected code in try block
- Use catch block with custom exception class variable to catch the custom exception
- Display necessary information

```
namespace CustomExceptionHandlingDemo
{
   class Program
   {
      static void Main(string[] args)
      {
         Applicant applicantobject = new Applicant();
         Console.Write("Enter age of applicant: ");
         try
         {
            applicantobject.Age = Convert.ToInt32(Console.ReadLine());
         }
         catch (AgeLessThanFiveException ex)
         {
            Console.WriteLine("Message: "+ex.Message);
            Console.WindowHeight("Source Application: " + ex.Source);
            Console.WriteLine("Source Method: "+ex.TargetSite);
         }
      }
   }
}
```

The Throw Statement

- Use ‘throw’ statement to throw an appropriate exception
- Generally used to throw custom exceptions
- Give the exception a meaningful message

throw expression ;

Custom Exception class

```
if (minute < 1 || minute >= 60) {
  throw new InvalidTimeException(minute +
                    " is not a valid minute“);
  // !! Not reached !!
}
```

Solution

DuplicateCustomerException.cs

```
public class DuplicateCustomerException : ApplicationException
{
    /// <summary> Default constructor
    0 references
    public DuplicateCustomerException()
    {   }

    /// <summary> Constructor with error message
    0 references
    public DuplicateCustomerException(string errorMsg)
        : base(errorMsg)
    {   }

    /// <summary> Constructor with error message and exception
    0 references
    public DuplicateCustomerException(string errorMsg, Exception exception)
        : base(errorMsg, exception)
    {   }
}
```



```
public class DuplicateCustomerException : ApplicationException
{
/// <summary>
/// Default constructor
/// </summary>
public DuplicateCustomerException()
{}

/// <summary>
/// Constructor with error message
/// </summary>
/// <param name="errorMsg"></param>
public DuplicateCustomerException(string errorMsg)
: base(errorMsg)
{}

/// <summary>
/// Constructor with error message and exception
/// </summary>
/// <param name="errorMsg"></param>
/// <param name="exception"></param>
public DuplicateCustomerException(string errorMsg, Exception exception)
: base(errorMsg, exception)
{}
}
```

Solution

RegisterUser.cs

```
public class RegisterUser
{
    7 references
    public string EmailId { get; set; }
    4 references
    public string Password { get; set; }
    3 references
    public string ConfirmPassword { get; set; }

    2 references
    public override bool Equals(object obj)
    {
        return EmailId.Equals(((RegisterUser)obj).EmailId);
    }

    1 reference
    public override int GetHashCode()...
}
```



```
public class RegisterUser
{
public string EmailId { get; set; }
public string Password { get; set; }
public string ConfirmPassword { get; set; }

public override bool Equals(object obj)
{
return EmailId.Equals(((RegisterUser)obj).EmailId);
}

public override int GetHashCode()
{
return EmailId.GetHashCode();
}
}
```

Solution

AccountRepo.cs

```
public class AccountRepo
{
    private List<RegisterUser> registerUsers =
        new List<RegisterUser>();

    3 references
    public void RegisterUser(RegisterUser registerUser)
    {
        if (registerUsers.Contains(registerUser))
        {
            throw new DuplicateCustomerException
                ("User with email id already exists.");
        }
        registerUsers.Add(registerUser);
    }

    1 reference
    public IEnumerable<RegisterUser> GetRegisterUsers()
    {
        return registerUsers;
    }
}
```



```
public class AccountRepo
{
private List<RegisterUser> registerUsers =
new List<RegisterUser>();

public void RegisterUser(RegisterUser registerUser)
{
if (registerUsers.Contains(registerUser))
{
throw new DuplicateCustomerException
("User with email id already exists.");
}
registerUsers.Add(registerUser);
}

public IEnumerable<RegisterUser> GetRegisterUsers()
{
return registerUsers;
}
}
```

Solution

RegisterUserMain.cs

```
class RegisterUserMain
{
    1 reference
    public void Main()
    {
        AccountRepo accountRepo = new AccountRepo();
        //Add users
        RegisterUsers(accountRepo);
        //Display users
        DisplayRegisteredUsers(accountRepo);
    }

    1 reference
    private void DisplayRegisteredUsers
        (AccountRepo accountRepo)...

    1 reference
    private void RegisterUsers
        (AccountRepo accountRepo)...
}
```

Will we get all the register user details?

```
private void DisplayRegisteredUsers
    (AccountRepo accountRepo)
{
    Console.WriteLine("User ID\tPassword");
    Console.WriteLine("----------------------------------");
    foreach (var item in accountRepo.GetRegisterUsers())
    {
        Console.WriteLine($"{item.EmailId}\t{item.Password}");
    }
}
private void RegisterUsers
    (AccountRepo accountRepo)
{
    RegisterUser user1 = new RegisterUser()
    {
        EmailId = "email1@gamil.com",
        ConfirmPassword = "password123",
        Password = "password123"
    };
    RegisterUser user2 = new RegisterUser()
    {
        EmailId = "email2@gamil.com",
        ConfirmPassword = "password123",
        Password = "password123"
    };
    RegisterUser user3 = new RegisterUser()
    {
        EmailId = "email1@gamil.com",
        ConfirmPassword = "password123",
        Password = "password123"
    };
    accountRepo.RegisterUser(user1);
    accountRepo.RegisterUser(user2);
    accountRepo.RegisterUser(user3);
}
```



```
class RegisterUserMain
{
public void Main()
{
AccountRepo accountRepo = new AccountRepo();
//Add users
RegisterUsers(accountRepo);
//Display users
DisplayRegisteredUsers(accountRepo);
}

private void DisplayRegisteredUsers
(AccountRepo accountRepo)
{
Console.WriteLine("User ID\tPassword");
Console.WriteLine("----------------------------------");
foreach (var item in accountRepo.GetRegisterUsers())
{
Console.WriteLine($"{item.EmailId}\t{item.Password}");
}
}

private void RegisterUsers
(AccountRepo accountRepo)
{
RegisterUser user1 = new RegisterUser()
{
EmailId = "email1@gamil.com",
ConfirmPassword = "password123",
Password = "password123"
};
RegisterUser user2 = new RegisterUser()
{
EmailId = "email2@gamil.com",
ConfirmPassword = "password123",
Password = "password123"
};
RegisterUser user3 = new RegisterUser()
{
EmailId = "email1@gamil.com",
ConfirmPassword = "password123",
Password = "password123"
};
accountRepo.RegisterUser(user1);
accountRepo.RegisterUser(user2);
accountRepo.RegisterUser(user3);
}
}
```

Point of Discussion

- When we run the application, it will throw following error and will not display the output, why?
- Solution – We have to surround the **accountRepo.RegisterUser(user1);** line of code with try…catch in **RegisterUserMain** > **RegisterUsers** method

```
try
{
    accountRepo.RegisterUser(user1);
}
catch(Exception e)
{
    Console.WriteLine(e.Message);
}
```

NOTE: We have **e.StackTrace** method. Please check on this.

Next Step

Recap

Useful links

Thank you

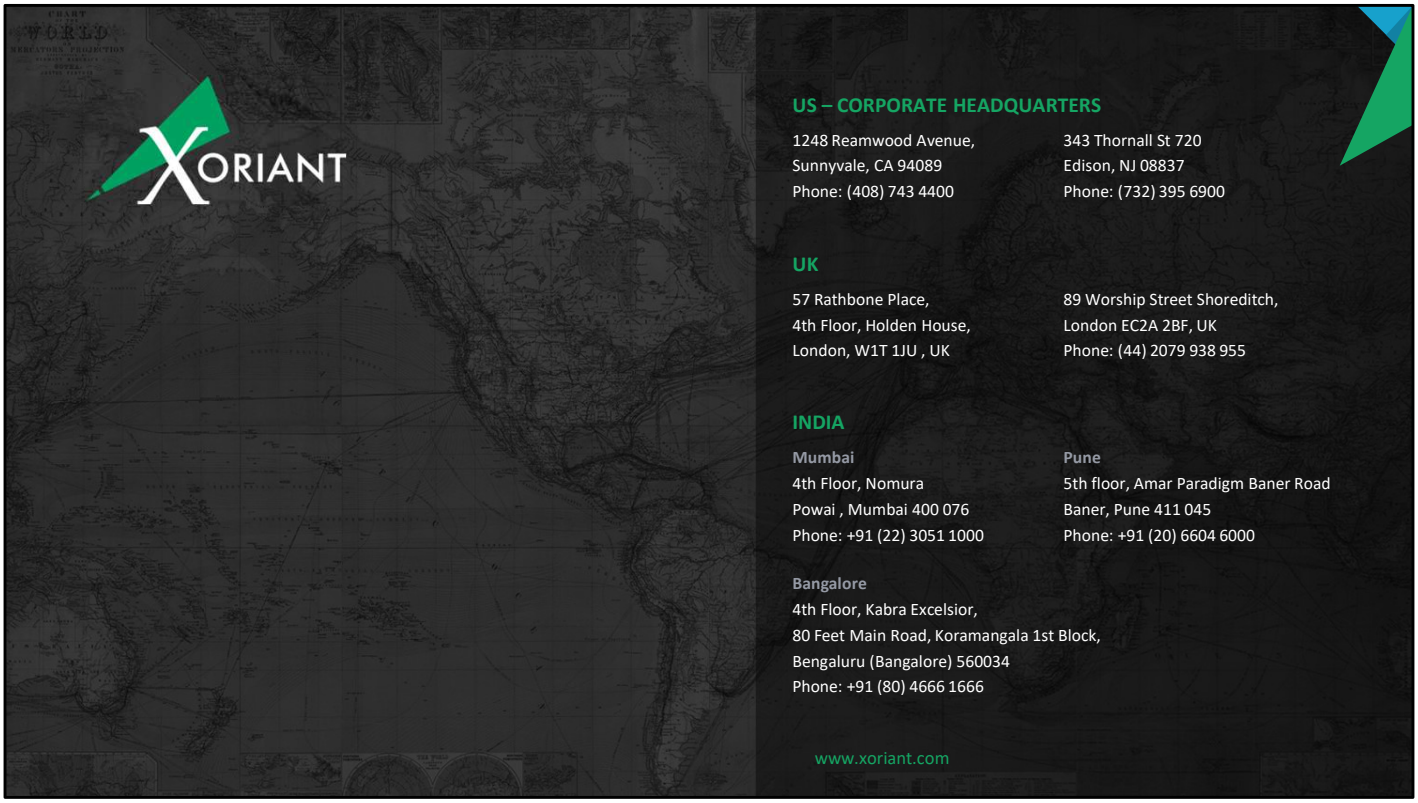
Version 1.0

Recap

- What is Error
- What is Exception
- Types of Exception
- Use of try…catch…finally
- Using multiple catch
- Creating custom exception

Useful Links

- https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/exception-handling
- https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/using-exceptions
- https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/creating-and-throwing-exceptions
- https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/compiler-generated-exceptions
- https://docs.microsoft.com/en-us/dotnet/standard/exceptions/how-to-create-localized-exception-messages
- https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions


XORIANT
US – CORPORATE HEADQUARTERS
1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400
343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900
UK
57 Rathbone Place,
4th Floor, Holden House,
London, W1T 1JU , UK
89 Worship Street Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955
INDIA
Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076
Phone: +91 (22) 3051 1000
Pune
5th floor, Amar Paradigm Baner Road
Baner, Pune 411 045
Phone: +91 (20) 6604 6000
Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road, Koramangala 1st Block,
Bengaluru (Bangalore) 560034
Phone: +91 (80) 4666 1666
www.xoriant.com

Thank YOU

- Any Questions?